UNIVERSITY OF CALIFORNIA
RIVERSIDE

Scalable spatial operations and pattern finding through algorithm paralellization

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

Doctor of Philosophy

in

Computer Science

by

Andres Oswaldo Calderon Romero

October 2023

The Proposal of Andres Oswaldo Calderon Romero is approved:

Committee Chairperson

University of California, Riverside

ABSTRACT OF THE PROPOSAL

Scalable spatial operations and pattern finding through algorithm paralellization

by

Andres Oswaldo Calderon Romero

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, October 2023
Dr. Vassilis Tsotras, Chairperson

The Doubly Connected Edge List (DCEL) is an edge-list structure that has been widely utilized in spatial applications for planar topological computations. An important operation is the *overlay* which combines the DCELs of two input layers and can easily support spatial queries like the intersection, union and difference between these layers. Here we propose a distributed and scalable way to compute the overlay operation and its related supported queries. Previous sequential implementations do not scale and fail to complete for large datasets (for example the US census tracks). We address the issues involved in distributing the overlay operator and offer various optimizations. Using real datasets, our scalable solution can compute the overlay of very large datasets (32M edges) in few minutes.

Contents

List of Figures

List of Tables

Chapter 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 [71]. Spatial data structures have been used to improve the efficiency of various spatial queries, such as spatial joins, nearest neighbors, voronoi diagrams and robot motion planning. Examples include grids [64], R-trees [34, 6], quadtrees [21], etc. There are also *edge-list* structures that have been typically utilized in applications as topological computations in computational geometry [9].

The most commonly used data structure in the edge-list family is the Doubly Connected Edge List (DCEL). A DCEL [63, 67] is a data structure which collects topological information for the edges, vertices and faces contained by a surface in the plane. The DCEL 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 which divide adjacent faces; and the vertices are the point endings between adjacent edges

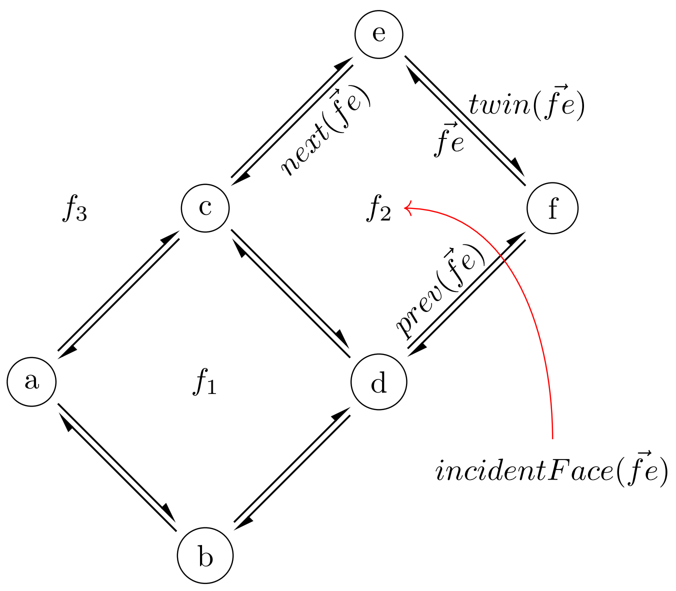


Figure 1.1: Components of the DCEL structure.

(see Figure 1.1). In addition to 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 [9].

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 [16, 65]) and robot motion planning (Minkowski sums [9, 15]). 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 [23]) as well as the computation of silhouettes from polyhedra [23, 8] (applied frequently in computer vision and 3D graphics modelling [10]).

Edge-list data structures have also been utilized for the creation of thematic *overlay maps*. In this problem, the input contains the DCELs of two polygon 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 of join the input layers and match their attributes in order to unveil patterns or anomalies in data which 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, identify their *difference* (or symmetric difference), or create their *union*.

Spatial databases have been using spatial indexes (R-tree [34, 6]) 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) that is 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 track 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 then 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 tracks contains around 72K polygons; the execution for computing the overlay over such large file crashed on a stock laptop. To the best of our knowledge there is no scalable solution to compute overlays over DCEL layers.

In this paper we describe the design and implementation of a *scalable* and *distributed* approach to compute the overlay between two DCEL layers. We first present a partition 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. Our approach has been implemented in a parallel framework (i.e., Apache Spark).

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

The rest of this paper is organized as follows. Section 2.1 presents related work while Section 2.2 discusses the basics of DCEL and the sequential algorithm. In Section 2.3 we present a partitioning scheme that enables 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 2.4. An extensive experimental evaluation appears in Section 2.5.

Chapter 2

Scalable overlay operations over DCEL polygon layers

2.1 Related Work

The fundamentals of the DCEL data structure were introduced in the seminal paper by Muller and Preparata [63]. The advantages of DCELs are highlighted in [67, 9]. Examples of using DCELs for diverse applications appear in[4, 11, 28]. 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 [28].

Currently, few sequential implementations are available: LEDA [60], Holmes3D [39] and CGAL [23]. 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 overlay DCEL.

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

Table 2.1: Vertex records.

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

Table 2.2: Face records.

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

Table 2.3: 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	$\vec{bd}$	$\vec{ba}$	$\vec{fd}$
⋮	⋮	⋮	⋮	⋮	⋮

joins, intersections, unions etc. in a parallel way (using clusters, multicores or GPUs), [14, 70, 55, 27, 58, 69, 68] 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.)

2.2 Preliminaries

The DCEL [63] 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 as an edge-based structure). Examples appear in Tables 2.1-2.3 below, following 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, 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 2.3. 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 2.1) 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 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 2.2. All other half-edges for this face's boundary can be easily retrieved following 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 2.2. 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 itself 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 describe the hole. Note that in Table 2.2 this list is empty as there are no holes in any of the faces in the example of Figure 1.1.

An important advantage with the DCEL structure is that a user can combine two DCELs from different layers over the same area (e.g. the census tracks 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 $f1 \cap f2$ [9]. 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 [23] 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) [9]. 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

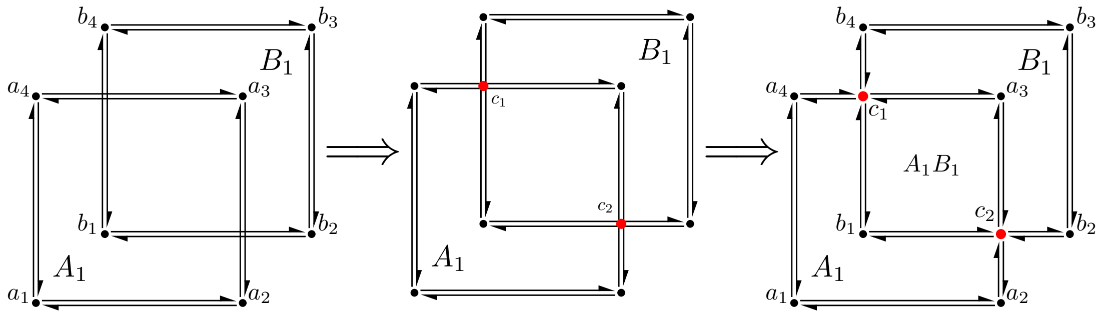


Figure 2.1: Sequential computations of an overlay of two DCEL layers.

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, records for faces and vertices tables are also updated with the new information.

Figure 2.1 illustrates an example for computing the overlay between two DCEL layers with one face each (A_1 and B_1 respectively), that overlap over the same area. First, intersection points are found and create new vertices in the overlay (red vertices c_1 and c_2). Finally, 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 [9], 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 the creation of the overlay (and the operators it supports) as they are

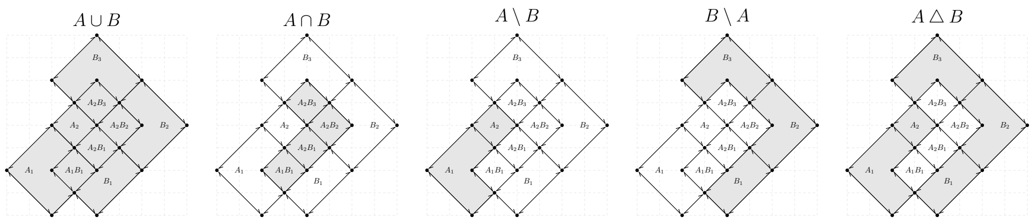


Figure 2.2: Examples of overlay operators supported by DCEL; results are shown in gray.

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 2.2) 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.)

2.3 Scalable Overlay Construction

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 [9].

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 tracks (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 2.5). 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 using a spatial index (or grid) 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.

2.3.1 Partition Strategy

The main idea of the partition strategy is to split the area covered by the input layers into non-overlapping cells which could then be processed independently. One could use a simple grid to divide the area but our early experiments showed that such approach would result in unbalanced cells (in number of edges) which affects performance. In the rest we assume that the partitioning is performed using a quadtree index which adapts to skewed spatial distributions and helps to assign a similar 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 then 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 not overlapping or stacked polygons in the dataset. To quickly build the partitioning quadtree structure we take a sample from the edges of each layer (1% of the total number of edges in that layer). After the quadtree is created, we use its leaf nodes as the partitioning cells for each layer. Each input layer file is then read from disk and *all* of its edges are inserted to the appropriate cells of the partitioning structure.

For this approach to work, it is important that each cell can compute its two DCELs independently. Note that 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 that it intersects, but within each cell, we use the part of the edge that lies fully inside the cell. Figure 2.3 shows an example, where there are 4 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 2.3). 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 2.3). Such artificial edges are shown in red in the figure. This allows to create a smaller polygon that is contained within each cell. For example polygon A_2 is clipped into four

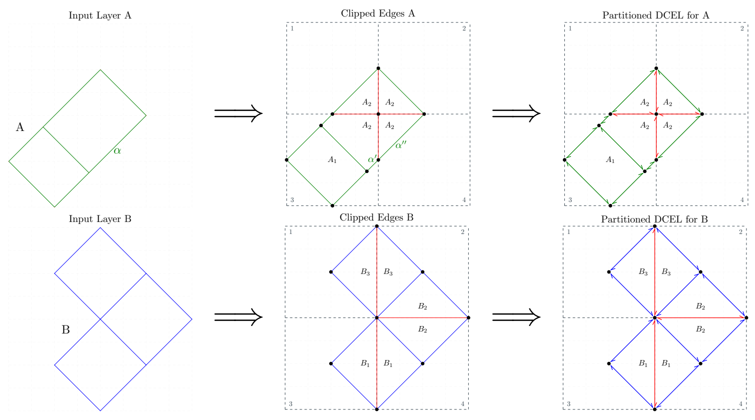


Figure 2.3: Partitioning example using input layers A and B over four cells.

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

Once a cell is assigned to a 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 2.3, Figure 2.4 depicts an overview of the process for creating a local overlay DCEL inside cell 2. Similarly, Figure 2.5 shows all local overlay DCELs computed at each cell (again, artificial edges are shown in red).

Nevertheless, the partitioning can create two problems (not present in the sequential environment) that need to be addressed. The first is the case where a cell is empty, that

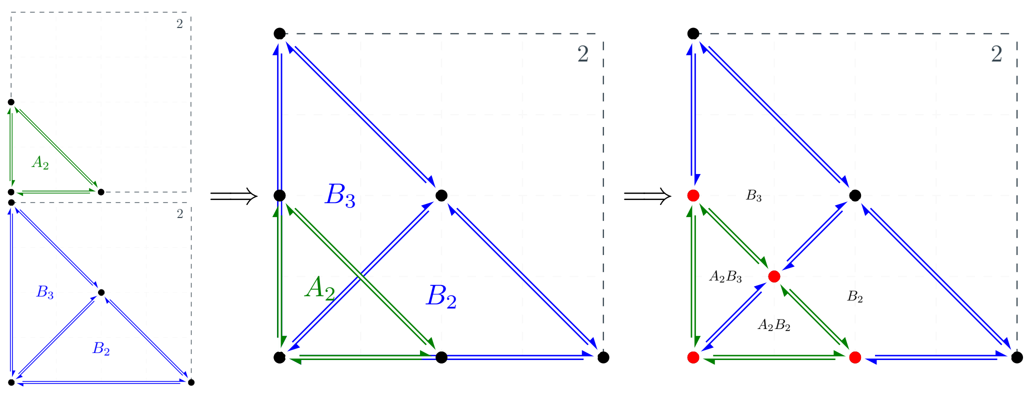


Figure 2.4: Local overlay DCEL for cell 2.

is, it does not intersect with (or contain) any regular edge from neither layer. A regular edge is one that is not part of a hole. This empty cell does not contain any label and thus we do not know which face it may belongs to. We term this as the ***orphan cell*** problem. An example is shown in Figure 2.6 which depicts a face (from one of the input layers) whose boundary goes over many quadtree cells; orphan cells are shown in grey.

Note that an orphan cell may contain a hole (see Figure 2.6). 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. While for simplicity we focus in the case where a hole is within one orphan cell, in the general case, a hole can split among many such cells.

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

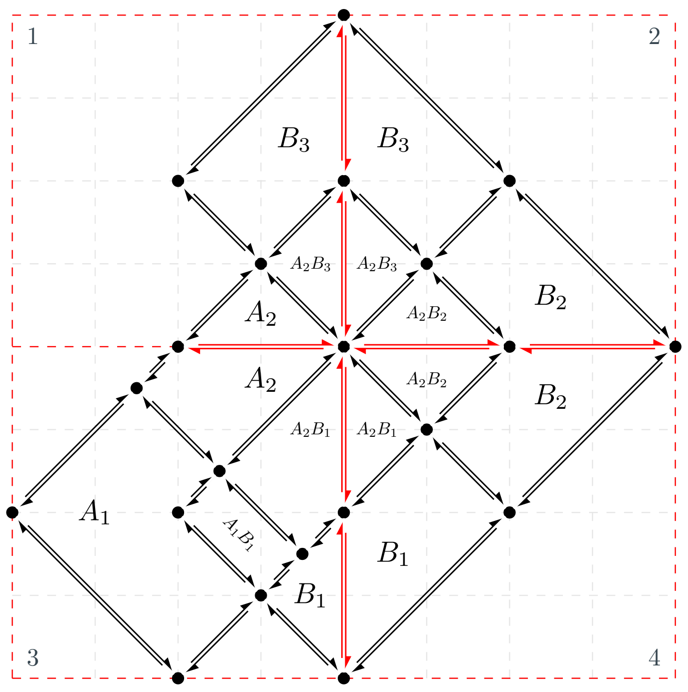


Figure 2.5: Result of the local overlay DCEL computations.

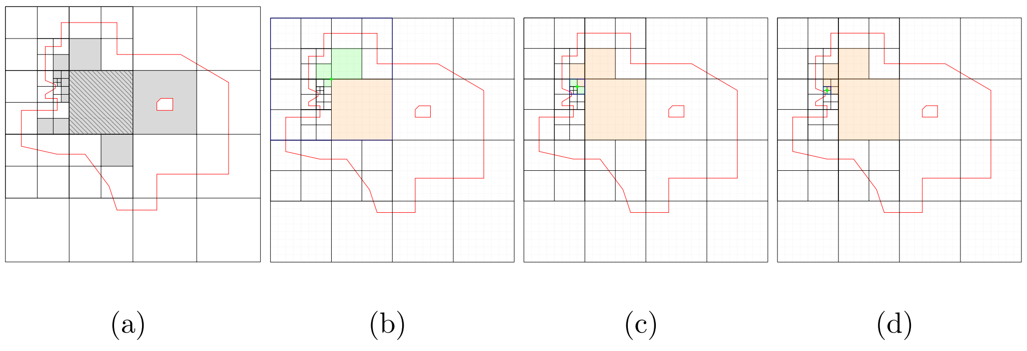


Figure 2.6: (a) Empty cell and hole examples; (b)-(c)-(d) show three iterations of the proposed solution.

2.3.2 Labeling Orphan Cells and Holes

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 which contains edge(s) of the face that contains the orphan cell and thus acquire the appropriate label information. This search is accommodated by the quadtree index. 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 2.6a 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 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 that will contain 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 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 2.6b. We then query the quadtree to identify which cells (leaves; one from each sibling) contain this point. We check these cells if they 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. This is the case in Figure 2.6b, where all three cells (green in the figure) are also orphan.

The algorithm has to pick one of them as the current orphan cell and repeat the process recursively. One can use different heuristics. Below we consider the case where we use the deepest cell (i.e. the one with the longest lineage) among the three. This is because, we expect that this will lead us to the denser areas of the quadtree index, where there is more chance to find cells with edges. Figure 2.6 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 pseudocode of the search process can be seen at Algorithms 1 and 2.

Another heuristic we used (not described here) is to follow the highest among the three orphan cells (the one with the shorter lineage) since this has larger area and will thus help us cover more empty space and possibly reach the border of the face faster.

Algorithm 1 getNextCellWithEdges algorithm

```
Require: a quadtree $\mathcal{Q}$ and a list of orphan cells $\mathcal{M}$.
function getNextCellWithEdges ( $\mathcal{Q}$, $\mathcal{M}$ )
    $\mathcal{C} \leftarrow$ list of orphan cells in $\mathcal{M}$
    for each $orphanCell$ in $\mathcal{C}$ do
        initialize $cellList$ with $orphanCell$
        $nextCellWithEdges \leftarrow null$
        $referenceCorner \leftarrow null$
        $done \leftarrow false$
        while not $done$ do
            $c \leftarrow$ last cell in $cellList$
            $cells, corner \leftarrow$ getCellsAtCorner($\mathcal{Q}, c$)
            for each $cell$ in $cells$ do
                $nedges \leftarrow$ get edge count of $cell$ in $\mathcal{M}$
                if $nedges > 0$ then
                    $nextCellWithEdges \leftarrow cell$
                    $referenceCorner \leftarrow corner$
                    $done \leftarrow true$
                else
                    add $cell$ to $cellList$
                end if
            end for
        end while
        for each $cell$ in $cellList$ do
            output($cell$,
                    $nextCellWithEdges$, $referenceCorner$)
            remove $cell$ from $\mathcal{C}$
        end for
    end for
end function
```

2.3.3 Answering global overlay queries

Using the local overlay DCELs we can easily compute the global overlay DCEL; for that we simply 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

Algorithm 2 getCellsAtCorner algorithm

```
Require: a quadtree with cell envelopes $\mathcal{Q}$ and a cell $c$.
 1: function getCellsInCorner ( $\mathcal{Q}$, $c$ )
 2:     $region \leftarrow$ quadrant region of $c$ in $c.parent$
 3:     switch $region$ do
 4:         case ‘SW’
 5:             $corner \leftarrow$ left bottom corner of $c.envelope$
 6:         case ‘SE’
 7:             $corner \leftarrow$ right bottom corner of $c.envelope$
 8:         case ‘NW’
 9:             $corner \leftarrow$ left upper corner of $c.envelope$
10:         case ‘NE’
11:             $corner \leftarrow$ right upper corner of $c.envelope$
12:     $cells \leftarrow$ cells which intersect $corner$ in $\mathcal{Q}$
13:     $cells \leftarrow cells - c$
14:     $cells \leftarrow$ sort $cells$ on basis of their depth
15:     return ($cells$, $corner$)
16: end function
```

on each local DCEL, followed by a reduce phase to remove artificial edges/added vertices. Figure 2.7 shows how the intersection overlay operator ($A \cap B$) is computed, starting with the local DCELs for 4 cells (Figure 2.7a). First each cell computes the intersection using its local overlay DCEL (Figure 2.7b). This is a simple map operation as we just need 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 2.7b); 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 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

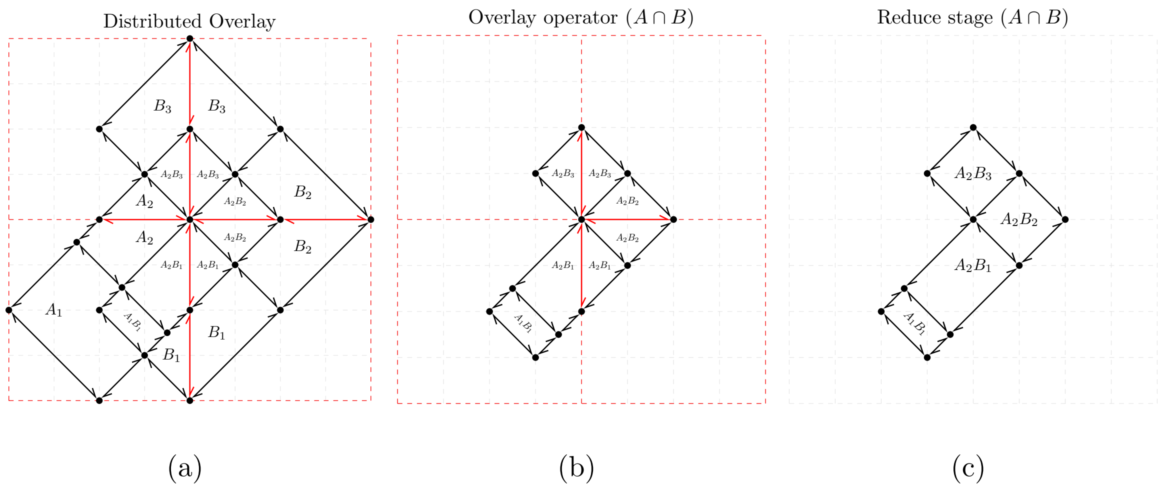


Figure 2.7: Example of an overlay operator querying the distributed DCEL.

2.7b were combined into one face in Figure 2.7c while the extra vertex was also removed). In section 2.4.1 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.

2.4 Overlay evaluation optimizations

2.4.1 Optimizations for faces expanding 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 node. One observation is that faces from different cells that are concatenated 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. 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 nodes. Nevertheless, there may be faces (typically few) 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.

Clearly, picking the appropriate level is important. Choosing a large level i (i.e., going to nodes lower in the quadtree structure) implies larger number of intermediate reducers and thus higher parallelism. However, at the same time, 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 (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.

2.4.2 Optimizing for unbalanced layers

During the overlay computation, the most critical task is finding the intersections between the half-edges. 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 the current approach, the input sets of half-edges within a cell are combined into one dataset which is first ordered by the x-origin of each half-edge and 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 set 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 with the sweep-line algorithm. This focused approach avoids unnecessary scanning of the large dataset (for example, areas where there are no half-edges present from the smaller dataset).

2.5 Experimental Evaluation

This section presents our experimental evaluation using 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.

Evaluation datasets. The details of the real datasets of polygons that we use are summarized in Table 2.4. The first dataset (MainUS) contains the complete Census Tracts for all the states in the US mainland for years 2000 (layer A) and 2010 (layer B). It was collected from the official website of the United States Census Bureau[1]. 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 many more intersections between the two layers thus creating many new faces for the DCEL.

The second dataset (GADM - taken from Global Administration Areas[2]) collects 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 layer has the 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) which is a subset from MainUS for the state of California; layer A corresponds to the CA census tracts from year 2000 while layer B for 2010. (Below we also use other states to create datasets with different number of faces).

2.5.1 Overlay face optimizations

We first examine the optimizations in Section 2.4.1. To consider different distributions of faces, for these experiments we used 8 states from the MainUS dataset with different

[1] https://www2.census.gov/geo/tiger/TIGER2010/TRACT/

[2] https://gadm.org/

Table 2.4: 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	116995	32789444
	Polygons for Level 3	117891	37690256
CCT	Polygons for 2000	7028	2711639
	Polygons for 2010	8047	2917450

number of tracks (faces). In particular, we used (in decreasing order of number of tracks): 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 2.8 shows the results for the distributed overlay computation stage (that is, after the local DCELs were computed at each cell). Note that for each state experiment we tried different number of leaf cells for the quadtree and present the one with the best performance. 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 level 4 and 6. The same figure also shows the optimization that re-partitions the faces by label id. This approach has actually the best performance. This is because there are few faces with the same label that can be combined independently. This results to smaller jobs that are better distributed among the cluster nodes and no reduce phase is needed. As a result, for the rest of the experiments we use the label re-partition approach to implement the overlay computation stage.

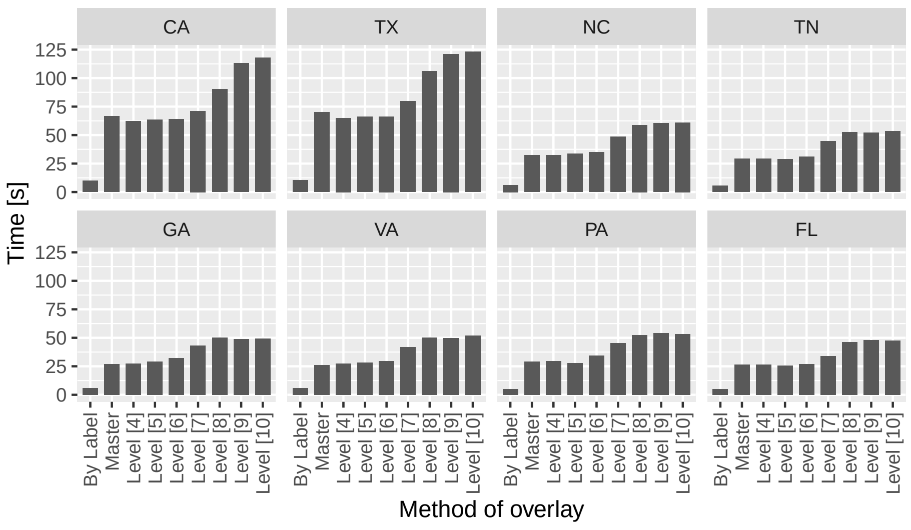
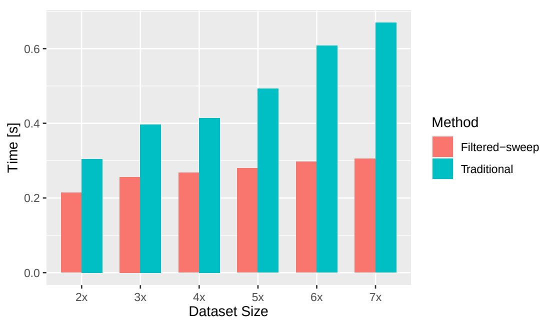


Figure 2.8: Overlay methods evaluation.

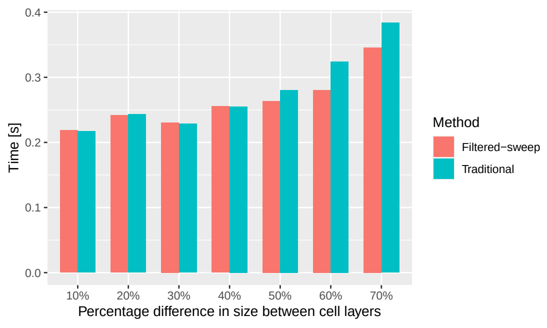
2.5.2 Unbalanced layers optimization

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 2.4.2). To create the smaller cell layer, we picked a reference point in the state of Pennsylvania (from the MainUS dataset) and started adding 2000 census tracks 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, we started adding tracks 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 2.9a shows the behaviour 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 that the other layer the



(a)



(b)

Figure 2.9: Evaluation of the unbalanced layers optimization.

filtered-sweep approach overcomes the traditional one.

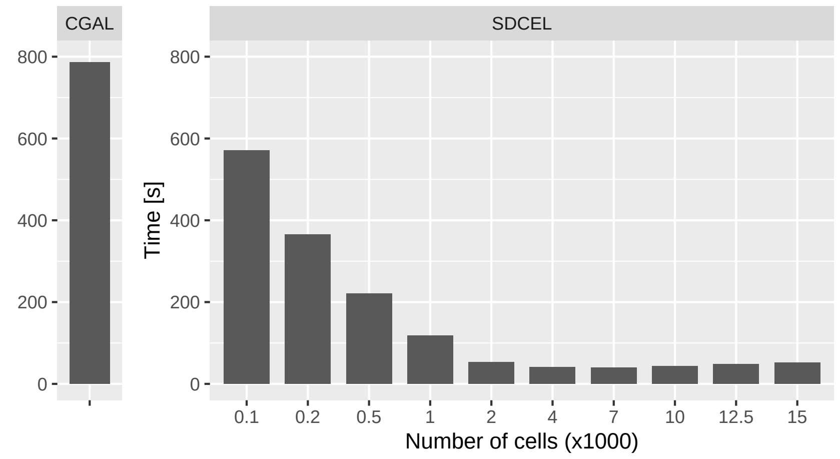
We also performed an experiment where the difference in size between the two layers varies between 10% and 70%. For this experiment we first identified cells from the GADM dataset where the smaller layer had around 3K edges. Among these cells we then identified those where the larger layer had 10%, 20%, ... up to 70% more edges. In each category we picked 10 representative cells and computed the overlay for the cells in that category. Figure 2.9b 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 again 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%).

2.5.3 Varying the number of cells

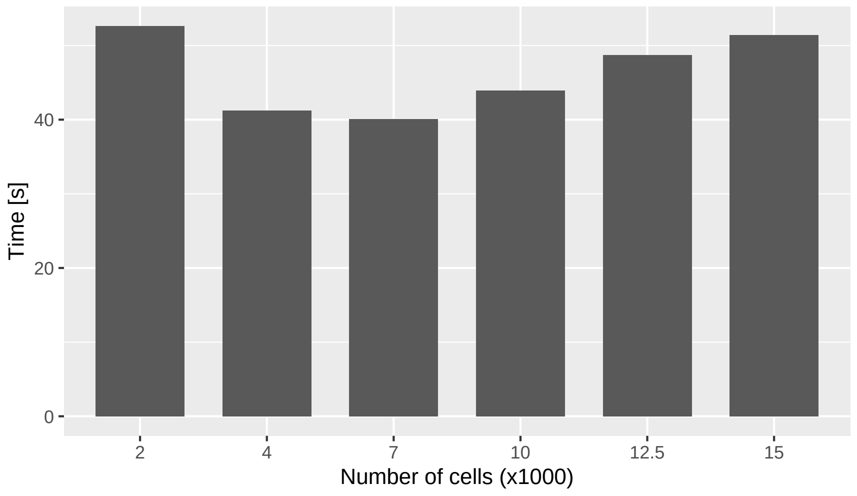
The quadtree settings allow for tuning its performance by providing the *number of leaf cells* to be created as a parameter. The quadtree then continues its splits so as to reach this capacity (approximately). The number of cells affects the performance of our scalable overlay implementation (termed as SDCEL below) since it relates to the average cell capacity (in number of edges). Fewer number of cells implies larger cell capacity (and thus more edges to process within each cell). On the other hand, creating more cells increases the number of jobs to be executed. Figure 2.10a 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 combining 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 2.10b focuses on that area; the best SDCEL performance was around 7K cells.

In addition Figure 2.10a 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 2.11 shows the results when using the larger MainUS and GADM datasets, while again varying the number of cells parameter (from 1K to 15K and from 2K to 26K respectively). In this figure we also show the time taken by each stage of the overlay computation (namely, 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 5K for the

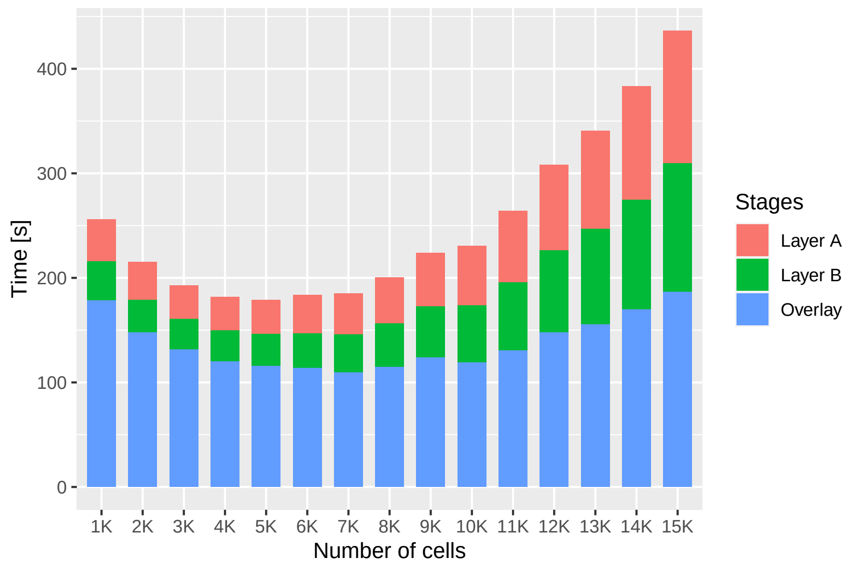


(a)

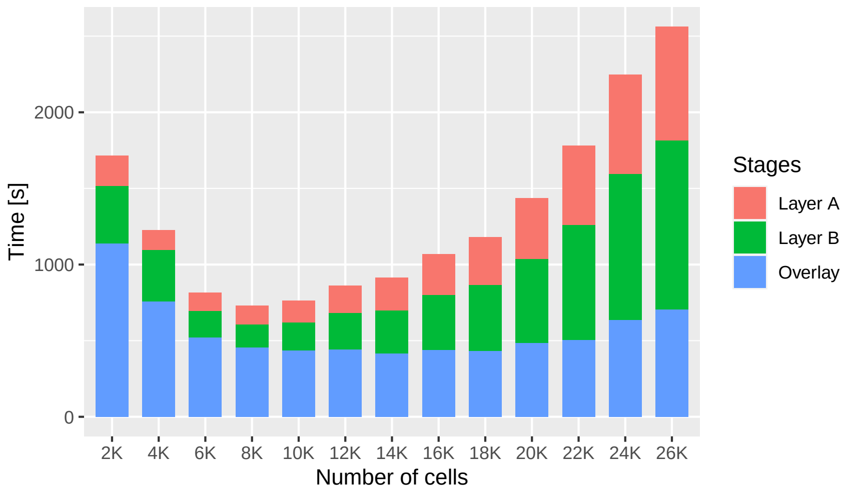


(b)

Figure 2.10: SDCEL performance while varying the number of cells in the CCT dataset.



(a)



(b)

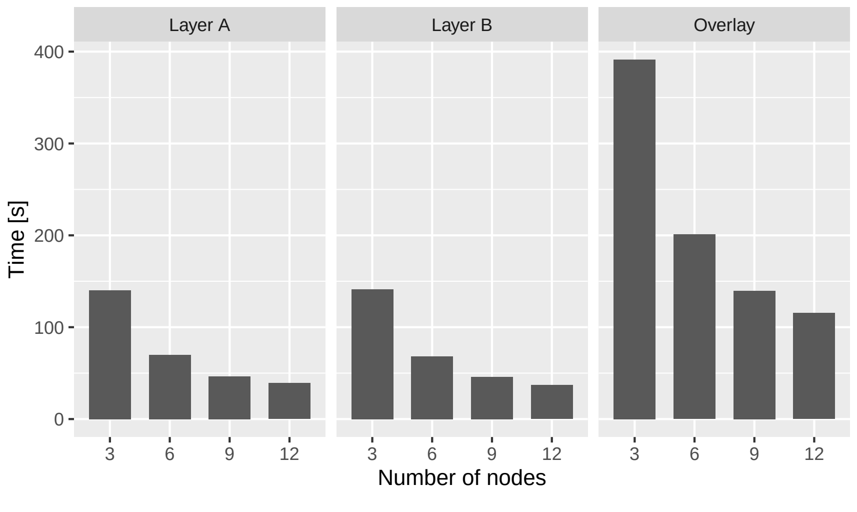
Figure 2.11: Performance with (a) MainUS and (b) GADM datasets.

MainUS and respectively 8K for the GADM dataset. Note that in the MainUS dataset the two layers have similar number of edges; as it can be seen their DCEL computations 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 the states. Moreover, county polygons are included in the (larger) state polygons. When the size of cells is small (i.e. larger number of cells like in the case of 26K 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 increase the number of intersections, resulting in higher overlay computation.

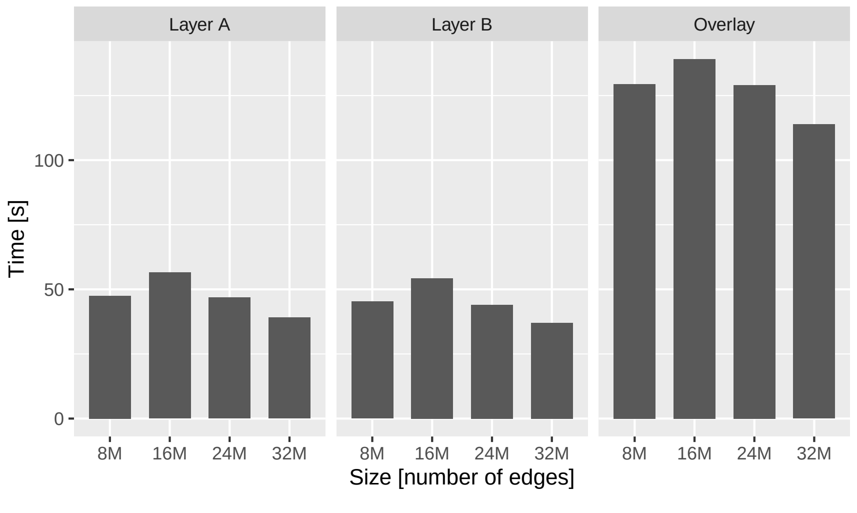
2.5.4 Speed-up and Scale-up experiments

The speed-up behavior of SDCEL appears in Figure 2.12a (for the MainUS dataset) and in Figure 2.13a (for the GADM dataset); in both cases we show the performance for each stage. For these experiments we varied the number of nodes from 3 to 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 almost by half.

To examine the scale-up behavior we created smaller datasets out of the MainUS (and similarly out of the GADM) so that we can control the number of edges. To create

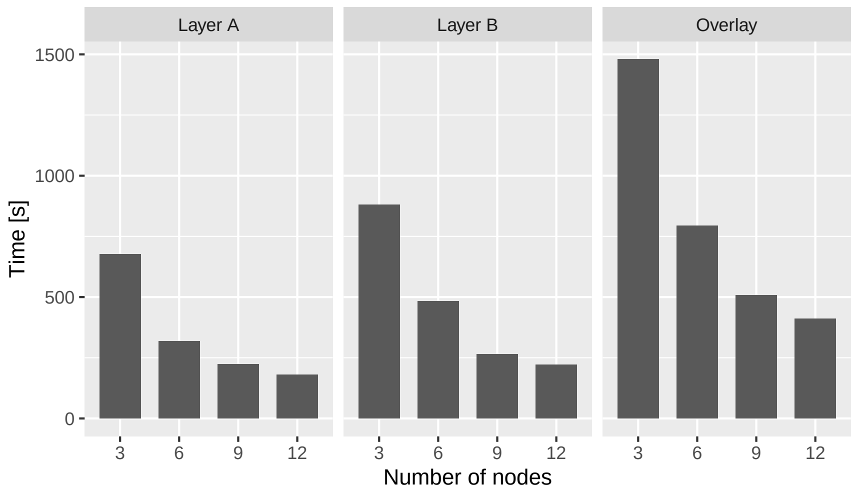


(a)

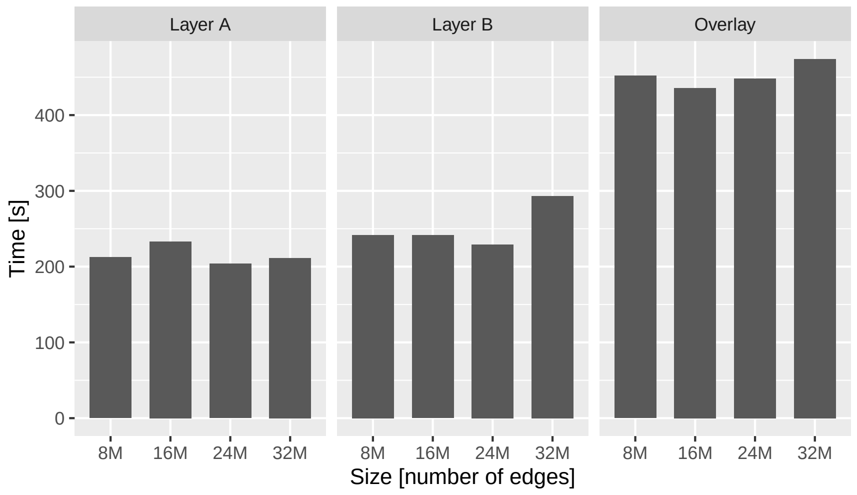


(b)

Figure 2.12: Speed-up and Scale-up experiments for the MainUS dataset.



(a)



(b)

Figure 2.13: Speed-up and Scale-up experiments for the GADM dataset.

such a dataset we picked a centroid and started increasing the area covered by this dataset until the number of edges were closed to a specific number. For example, from the MainUS we created datasets of sizes 8M, 16M, 24M and 32M edges for each layer. We then used two layers of the same size as input to 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, the 24M using 9 nodes and the 32M using 12 nodes. We did the same process for the scale-up experiments of the GADM dataset. The results appear in Figure 2.12b and Figure2.13b. 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).

Chapter 3

Towards parallel discovery of movement patterns in large spatio-temporal datasets

3.1 Introduction

Technology advances in last past decades have triggered an explosion in the capture of spatio-temporal data. The increase popularity of GPS devices and smart-phones together with the emerging of new disciplines such as the Internet of Things and Satellite/UAS high-resolution imagery have made possible to collect vast amount of data with a spatial and temporal component attached to them.

Together with this, the interest to extract valuable information from such large databases has also appeared. Spatio-temporal queries about most popular places or frequent events are still useful, but more complex patterns are recently shown an increase of interest. In particular, those what describe group behaviour of moving objects through significant periods of time. Moving cluster [48], convoys [45], flocks [32] and swarm patterns [56] are new movement patterns which unveil how entities move together during a minimum time interval.

Applications for this kind of information are diverse and interesting, in particular if they come in the way of trajectory datasets [44, 40]. Case of studies range from transportation system management and urban planning [18] to Ecology [51]. For instance, [73] explore the finding of complex motion patterns to discover similarities between tropical cyclone paths. Similarly, [2] use eye trajectories to understand which strategies people use during a visual search. Also, [38] tracks the behavior of tiger sharks in the coasts of Hawaii in order to understand their migration patterns.

In particular, a moving flock pattern show how objects move close enough during a given time period. Closeness is defined by a disk of a given radius where the entities must keep inside. Given that the disk can be situated at any location, it is not a trivial problem. In deed, [32] claims that find flock patterns where the same entities stay together during their duration is a NP-hard problem. [76] proposed the BFE algorithm which the first approach to be able to detect flock patterns in polynomial time.

Despite the fact that much more data become available, state-of-the-art techniques to mine complex movement patterns still depict poor performance for big spatial data problems. This work presents a parallel algorithm to discover moving flock patterns in large trajectory databases. It is thought that new trends in distributed in-memory frameworks for spatial operations could help to speed up the detection of this kind of patterns.

3.2 Related work

Recently increase use of location-aware devices (such as GPS, Smart phones and RFID tags) has allowed the collection of a vast amount of data with a spatial and temporal component linked to them. Different studies have focused in discovering and analyzing this kind of collections [53, 61]. In this area, trajectory datasets have emerged as an interesting field where diverse kind of patterns can be identified [84, 77]. For instance, authors have proposed techniques to discover motion spatial patterns such as moving clusters [48], convoys [45] and flocks [7, 32]. In particular, [76] proposed BFE (Basic Flock Evaluation), a novel algorithm to find moving flock patterns in polynomial time over large spatio-temporal datasets.

A flock pattern is defined as a group of entities which move together for a defined lapse of time [7]. Applications to this kind of patterns are rich and diverse. For example, [13] finds moving flock patterns in iceberg trajectories to understand their movement behavior and how they related to changes in ocean's currents.

The BFE algorithm presents an initial strategy in order to detect flock patterns. In that, first it finds disks with a predefined diameter (ε) where moving entities could be close enough at a given time interval. This is a costly operation due to the large number of points and intervals to be analyzed ($\mathcal{O}(2n^2)$ per time interval). The technique uses a grid-based index and a stencil to speed up the process, but the complexity is still high.

[13] and [73] use a frequent pattern mining approach to improve performance

during the combination of disks between time intervals. Similarly, [72] introduce the use of plane sweeping along with binary signatures and inverted indexes to speedup the same process. However, the above-mentioned methods still keep the same strategy as BFE to find the disks at each interval.

[3] and [29] use depth-first algorithms to analyze the time intervals of each trajectory to report maximal duration flocks. However, these techniques are not suitable to find patterns in an on-line fashion.

Given the high complexity of the task, it should not be surprising the use of parallelism to increase performance. [24] use extreme and intersection sets to report maximal, longest and largest flocks on the GPU with the limitations of its memory model.

Indeed, despite the popularity of cluster computing frameworks (in particular whose supporting spatial capabilities [19, 82, 42, 81]) there are not significant advances in this area. At the best of our knowledge, this work is the first to explore in-memory distributed systems towards the detection of moving flock patterns.

3.3 Methods

This section explains two alternative methodologies to find moving flock patterns in large spatio-temporal datasets using modern distributed frameworks to divide and parallelize the workload. Before to explain the details of our contributions we will explains the in a general manner the details of the current state-of-the-art to highlight the challenges and drawbacks at the moment to dealt with very large spatio-temporal datasets.

3.3.1 The BFE algorithm

The alternatives we will discuss later follows closely the steps explained at [76]. In this work, the authors proposed the Basic Flock Evaluation (BFE) algorithm to find flock patterns on trajectory databases. The details of the algorithm can be accessed at the source but we will explain the main aspects in a general view. It is important to clarify that BFE runs in two phases: firstly, it finds valid disks in the current time instant; secondly, it combines previous flocks with the recently discovered disks to extend them and report them.

The main inputs of the BFE algorithm are a set of points, a minimum distance ε which will define the diameter of the disks where the moving entities should lay, a minimum number of entities μ at each disk and a minimum duration δ which is the minimum number of time units the entities should be keep together to be considered a flock. Based on these inputs, figure 3.1 breaks down schematically the work flow of this phase where we can identify 4 general steps. The main goal of this phase is to find a set of valid disks at each time instant to allow further combinations with subsequent sets of disks coming in the future.

The main steps in phase one can are explained as follows:

1. Pair finding: Using the ε parameter, the algorithm query the set of points to get the set of pairs which laid at a maximum distance of ε units. Usually, it is a distance self-join operation over the set of points using ε as the distance parameter. The query also pays attention to do not return pair duplicates. For instance, the pair between point p_1 and p_2 is the same that pair between p_2 and p_1 and just one of them should

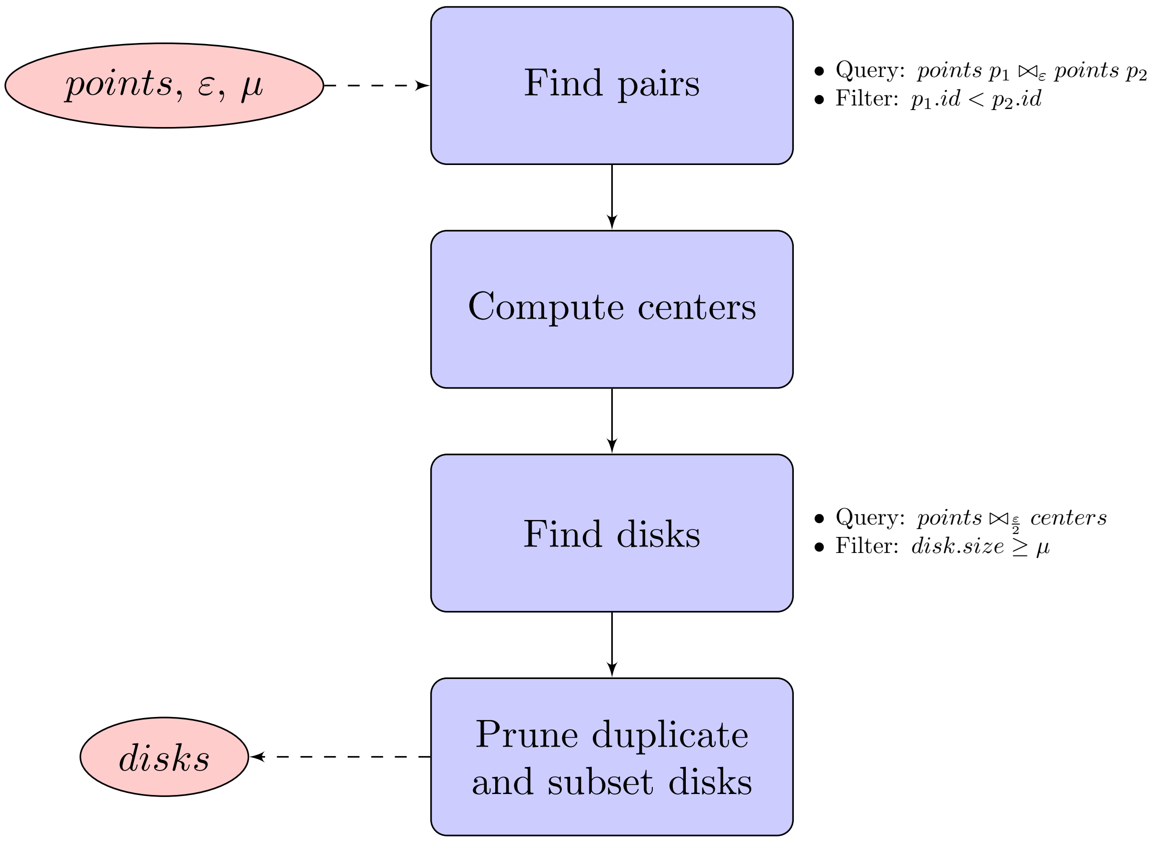

$points$, ε, μ
Find pairs
• Query: $points\ p_1 \bowtie_{\varepsilon} points\ p_2$
• Filter: $p_1.id < p_2.id$
Compute centers
Find disks
• Query: $points \bowtie_{\frac{\varepsilon}{2}} centers$
• Filter: $disk.size \geq \mu$
Prune duplicate and subset disks
$disks$

Figure 3.1: General steps in phase 1 of the BFE algorithm.

be reported (the id of each point is used to filter duplicates).

2. Center computation: From the previous set of pairs, each tuple is the input of a simple computation to locate the centers of the two circles of radius $\frac{\varepsilon}{2}$ which circumference laid on the input points. The pseudocode of the procedure can be seen in appendix **??**.

3. Disk finding: Once the centers have been identified, a query to collect the points around those centers is needed in order to group the set of points which laid ε distance units each other. This is done by running a distance join query between the set of points and the set of centers using $\frac{\varepsilon}{2}$ as the distance parameter. Therefore, a disk will be defined by its center and the IDs of the points around it. At this stage, a filter is applied to remove those disks which collect less than μ entities around it.

4. Disk pruning: It is possible than a disk collects the same set of points, or a subset, of the set of points of another disk. In such cases the algorithm should report just that one which contains the others. An explanation of the procedure can be seen in appendix **??**.

It is important to note that BFE also proposes a grid index structure in this phase to speed up spatial operations. The algorithm divides the space area in a grid of ε side (see figure 3.2 from [76]). In this way, BFE just processes each grid and its 8 neighbor grids. It does not need to query grids outside of its neighborhood given that points in other grids are far away to affect the results.

The second phase is more straightforward. Figure 3.3 explains schematically what

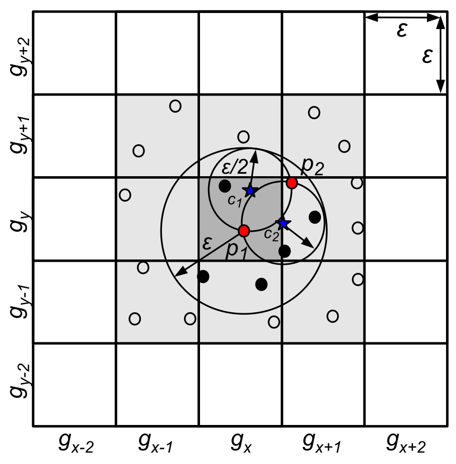


Figure 3.2: The grid-based index structure proposed at [76].

is done once the set of current disks (as explained in figure 3.1) is found at every time instant. This phase performs a recursion using the current set of disks and the previous set of flocks which comes from the previous time instant. Due to we do not know where and how far a group of entities can move in the next time instant, a cross product between both sets is required.

However, just disks which match the entities of previous flocks are kept. Indeed, only when the size of common items is greater than μ we keep the pair. Then, it updates the end time and duration of the filtered flocks. This information is used to decide if it is time to report a flock or keep it for further analysis. If the duration has reached the minimum duration δ, a flock is reported and removed from the set. The remaining flocks are sent to the next iteration for further evaluation in the next time instant.

Similarly, figure 3.4 illustrates the recursion and how the set of flocks from previous time instants feeds the next iteration. The example assumes a δ value of 3, so it starts reporting flock since time instant t_2. Note that time instants t_0 and t_1 are initial conditions.

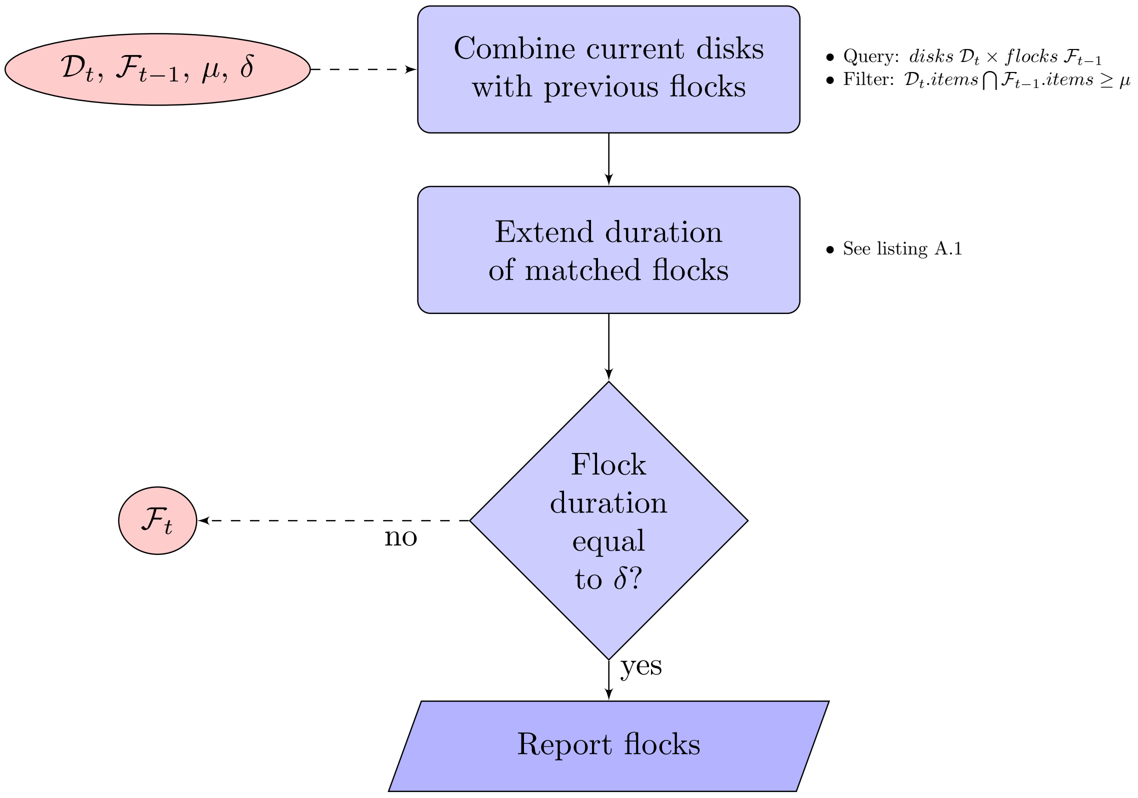
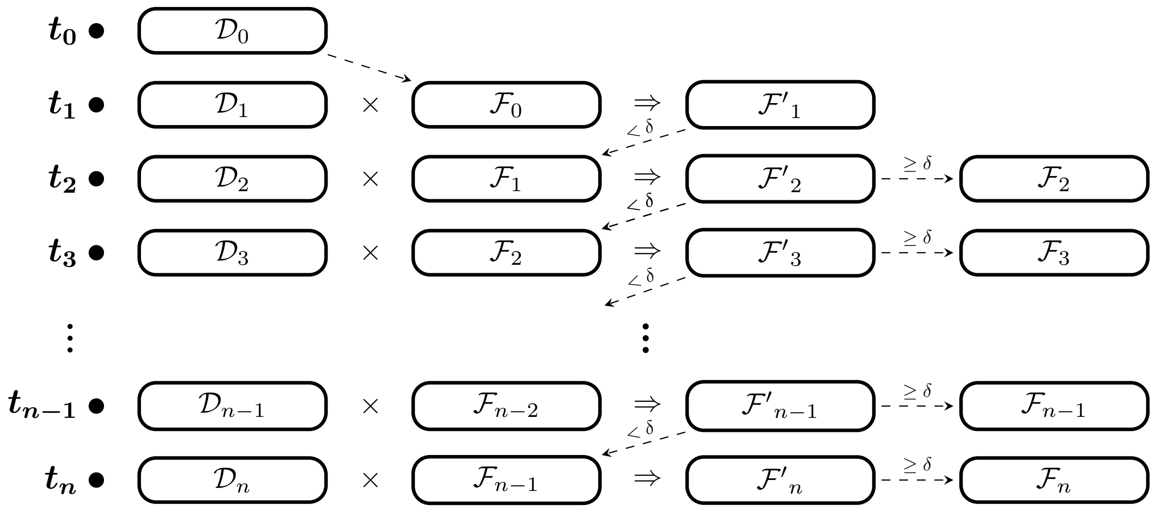

$\mathcal{D}_t$, $\mathcal{F}_{t-1}$, μ, δ
Combine current disks with previous flocks
• Query: $disks\ \mathcal{D}_t \times flocks\ \mathcal{F}_{t-1}$
• Filter: $\mathcal{D}_t.items \bigcap \mathcal{F}_{t-1}.items \geq \mu$
Extend duration of matched flocks
• See listing A.1
Flock duration equal to δ?
no
$\mathcal{F}_t$
yes
Report flocks

Figure 3.3: Steps in BFE phase two. Combination, extension and reporting of flocks.

At the very beginning of the execution, we just can find valid disks at t_0 which immediately are transformed to flocks of duration 1 and feed the next time instant. At t_1 we can find a new set of disks $\mathcal{D}_1$ which combines with the set of previous flocks $\mathcal{F}_0$. It updates the information of each flock accordingly but it does not report any flock yet. From now on, subsequent time instants follows strictly the steps summarized on figure 3.3.



t_0 ● $\mathcal{D}_0$
t_1 ● $\mathcal{D}_1$ × $\mathcal{F}_0$ ⇒ $\mathcal{F}'_1$
$< \delta$
t_2 ● $\mathcal{D}_2$ × $\mathcal{F}_1$ ⇒ $\mathcal{F}'_2$ $\geq \delta$ $\mathcal{F}_2$
$< \delta$
t_3 ● $\mathcal{D}_3$ × $\mathcal{F}_2$ ⇒ $\mathcal{F}'_3$ $\geq \delta$ $\mathcal{F}_3$
$< \delta$
t_{n-1} ● $\mathcal{D}_{n-1}$ × $\mathcal{F}_{n-2}$ ⇒ $\mathcal{F}'_{n-1}$ $\geq \delta$ $\mathcal{F}_{n-1}$
$< \delta$
t_n ● $\mathcal{D}_n$ × $\mathcal{F}_{n-1}$ ⇒ $\mathcal{F}'_n$ $\geq \delta$ $\mathcal{F}_n$

Figure 3.4: BFE phase 2 example explaining the recursion of the set of flocks along time instants and the initial conditions.

3.3.2 Bottlenecks in BFE and possible solutions

There are some steps during the execution of BFE which are particularly affected when it deals with very large datasets. Firstly, we will focus on phase 1 of BFE. In figure 3.5, the steps of this phase are illustrated for a sample dataset. You can see that the number of centers and disks found is considerable large in comparison with the final set of valid disks. Indeed, the finding of centers and the following operations grows quadratic depending on

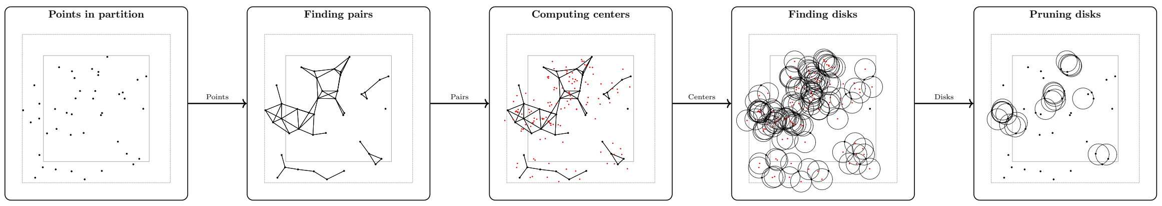


Figure 3.5: Example of BFE execution on a sample dataset.

the number of points and possible pairs (which itself depend on the ε parameter).

[76] claims that the number of centers, and consequent disks, to be evaluated is equal to $2|\tau|^2$ where τ is the number of trajectories. However, our experience show that there are a large number of duplicate and subset disks which are later pruned in the final stage. This behaviour is exacerbated no just in very large datasets but also in those with areas with high density of moving entities.

As solution for this issue, we proposed a partition strategy to divide the study area in smaller sections which can be evaluated in parallel. The strategy has three steps: first, a partition and replication stage, then the flock discovery in each local partition and finally the merge stage where we collect and unify the results. Let's explain each stage in more detail:

- Partition and Replication: Figure 3.6 shows a brief example of the partition and replication stage. Note that it is possible to use different types of spatial indexes (grids, r-tree, quadtree, etc.) to create spatial partitions over the input dataset. In the case of the example we use a quadtree which creates 7 partitions. Now, we need to ensure that each partition has access to all the required data to complete the finding of flocks locally. To accomplish this, all the points laying at ε distance of the border

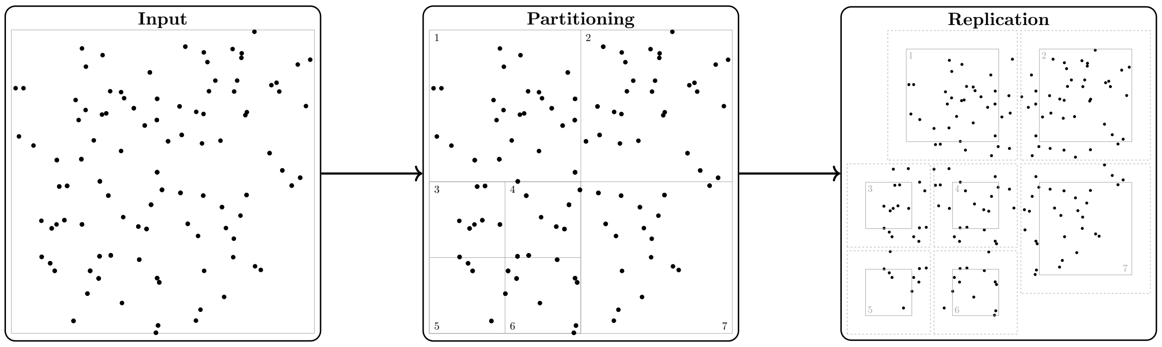


Figure 3.6: An example of partitioning and replication on a sample dataset.

of its partition are replicated to adjacent partitions. At the right of figure 3.6, it can be seen each partition surrounded by a dotted area with the points which need to be copied from its neighbor partitions. At this point, each partition is ready to be submitted to different nodes for local processing.

- Local discovery: Once we have all the data we need at each partition we can run the steps of the phase 1 of the BFE algorithm locally (as were explained on figure 3.1). Actually, you can see that the example of figure 3.5 describes the execution using the partition 2's points of figure 3.6 as sample data.

- Merging: In order to merge back the results we will have to pay special attention to disks laying close to the border of each partition. We will show that if the position of disk centers lay inside of the current partition they will be safe to operate but those located in the expansion zone or outside of it will require to be treated to avoid duplicate reports.

  Disks with centers in the expansion zone will be repeated in contiguous partitions and, therefore, they will lead to duplication. In addition, it is possible that pairs of points generates disks with centers outside of the expansion zone. For example, Fig. 3.7 illustrates the case. Disks a' and b' are generated for points in partitions 1 and 2 respectively, however both are located outside of their expansion zone boundaries and we should to avoid to report them twice. We present lemma 2 to show that we can safely remove those kind of disks.

Lemma 2 *A disk with its center laying in the expansion zone or outside of it can be discarded as they will be correctly evaluated by one of the partitions in its neighborhood.*

Proof. In order to support our proof we will define some concepts: First, we will divide the area of a partition in three zones to clarify our assumptions: we already talked about the *expansion zone* as the area beyond the border of a partition (between black line and dotted red line in figure 3.7) and a width equal to ε. The *border zone* is a strip of width equal to ε touching the interior border of a partition. In figure 3.7, it is compromised by the dotted blue and the black lines. The *safe zone* will be the remaining internal area in the partition which is not covered by the border zone. Second, we will call the contiguous partition which replicate a particular disk with the current one as the *replicated partition*. In figure 3.7, the partition 2 is the replicated partition of partition 1 and vice versa.

From here, it will be clear that there is a symmetric relation between the disks in the border and expansion zones in the current partition and the disks in its replicated partition. We can certainly said that if a disk is located in the border zone of the current partition, it will be located in the expansion zone of its replicated partition. Similarly, any disk with a center laying outside of the expansion zone of the current

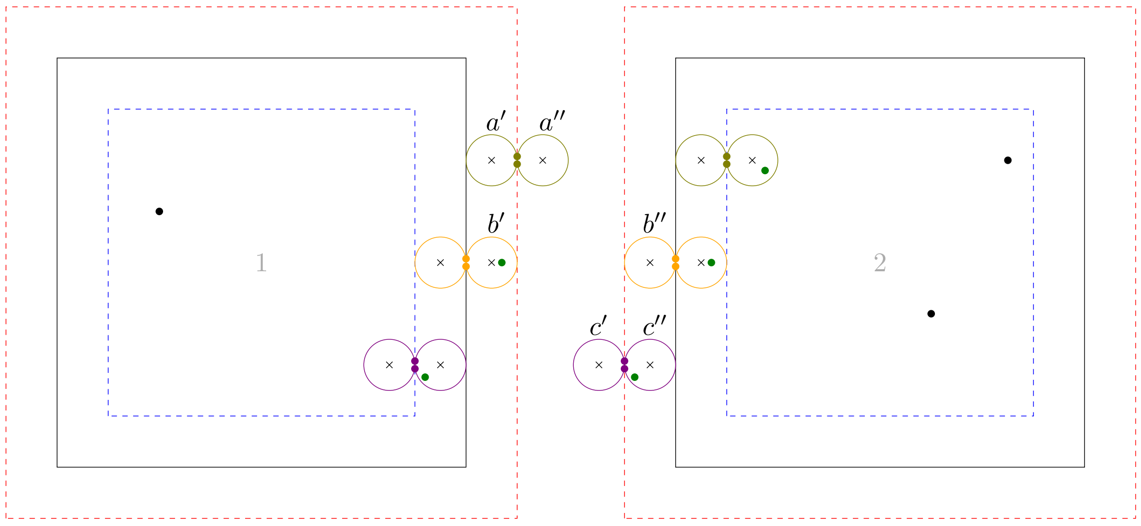


Figure 3.7: Ensuring no lost of data in safe zone and expansion area.

partition will be located in the safe zone of its replicated partition. Keeping just the disks with centers laying in the current partition (border or safe zone) will be enough to ensure no loss of information. ■

3.3.3 Additional improvements during local execution

Even the partition strategy could reduce the size of the points to be processed at each local partition , it is possible to introduce some filter techniques which allows to reduce the impact in the creation of centers and disks and subsequent pruning. This section explains the use of maximal cliques and minimum bounding circles (MBC) techniques as heuristics to improve the finding of disk at local level.

In graph theory, a clique is a subset of vertices forming a subgraph where all of their nodes share connections among them. It is said that the induced subgraph is complete.

Similarly, a maximal clique happens when no additional vertex can be included in the current clique, that is, there is not a superset of vertices which could include the current clique [12]. Although it is know that finding all maximal cliques is a problem of high complexity, studies claims that working on relatively small real-worlds graphs, they can be found in near-optimal time [20].

Now, in our approach, it can be noted that after obtaining the set of pairs at the beginning of the BFE algorithm, we can see the result as a graph where the points are the vertices and the connections between nearby points are the edges. So it is possible to run an algorithm (i.e. Bron-Kerbosch) over that graph to get a set of maximal cliques.

The advantage to find a set of cliques is that we will have access to set of points inside of each clique that by sure must generate one or more disks. Given the condition of the maximal cliques, all the points inside them are ε distance each other, and we can know that no other point must be included because that contradicts the definition of a maximal clique. That constitutes a kind of sub-partitioning technique where we could do additional processing. For example, a simple filter we can apply is to remove those cliques which number of points are less than the mu parameter given that those cliques will be unable to generate disks that fulfill that condition.

To introduce the next filter we need define the concept of minimum bounding circle (MBC). Algorithms for MBC finding aims to return the center and minimum radius possible of a circle that encloses all the items of a set of points in the plane. One of the most representative algorithm is proposed at [80] which is able to solve the problem in linear time.

The intention to use MBC algorithms at this stage is to detect maximal cliques where all their points can be enclosed in one single disk. If that is the case, we could report directly the MBC result and save the costly computation of finding centers and pruning duplicates. This can bring important improvements especially in datasets where the density of points is high.

However, we have to clarify that when an unique MBC to cover all the points is not possible, the traditional steps (BFE algorithm) should be applied over them. Finally, disks from both branches are unified and a final pruning is done to remove possible duplication. This final step should not be costly because most of the pruning is done at maximal clique level.

Figure 3.8 shows a schematic flow chart of the steps for this alternative.

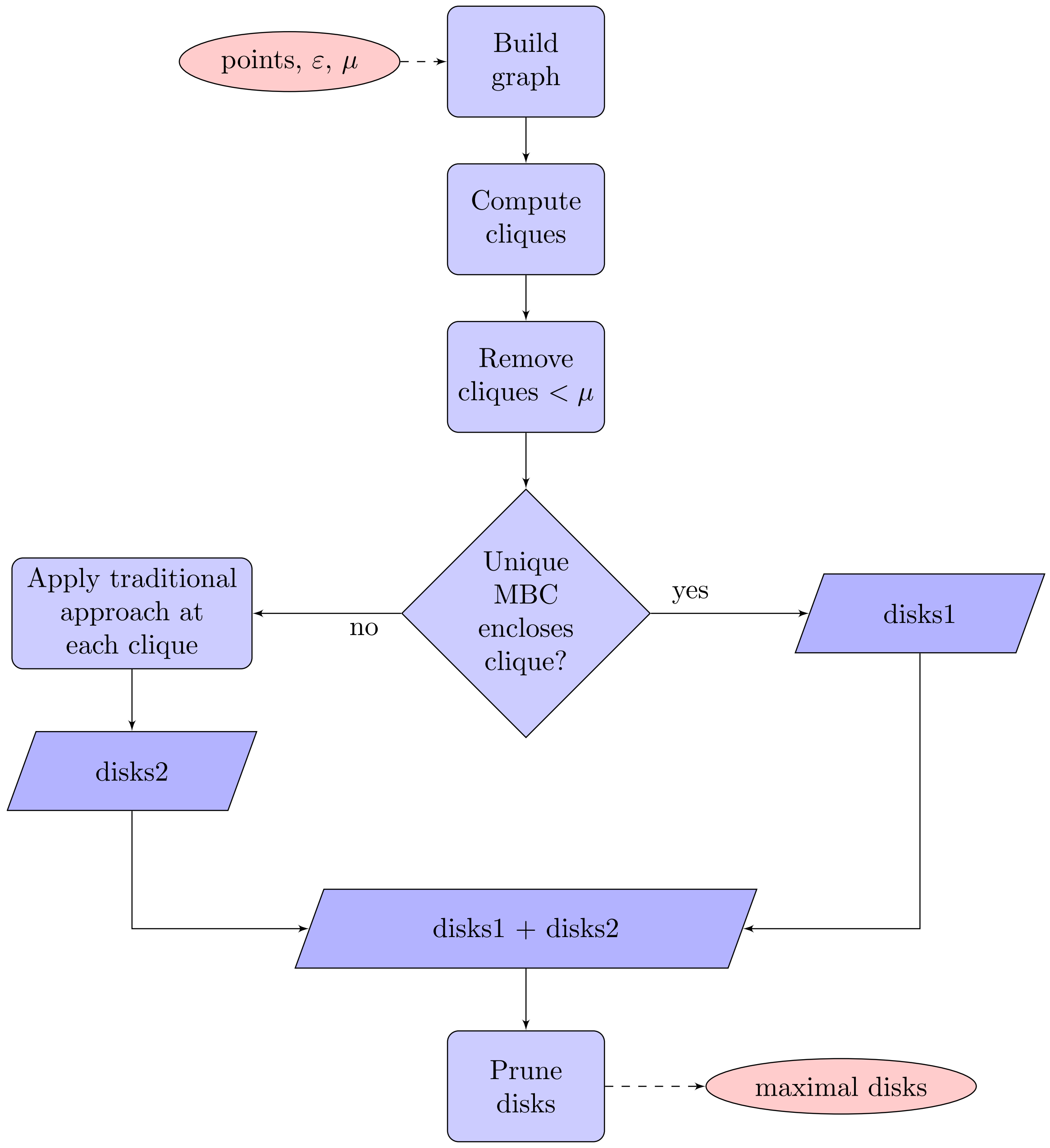


Figure 3.8: Flow chart for the maximal clique and MBC filters.

Chapter 4

Conclusions

We introduced SDCEL, a scalable approach to compute the overlay operation among two layers that represent polygons from a planar subdivision of a surface. Both input layers use the DCEL edge-list data structure to store their polygons. Existing sequential DCEL overlay implementations fail for large datasets. We first presented a partition strategy which guarantees that each partition collects the required data from each layer to work independently. We also proposed several optimizations to improve performance. Our experimental evaluation using real datasets shows that SDCEL has very good scale-up and speed-up performance and can compute the overlay over very large layers (up to 37M edges) in few seconds.

Bibliography

[1] P.K. Agarwal and M. Sharir. Arrangements and Their Applications. In *Handbook of Computational Geometry*, pages 49–119. Elsevier Science, 1998.

[2] T. Amor, S. Reis, D. Campos, H. Herrmann, and J. Andrade. Persistence in eye movement during visual search. *Scientific Reports*, 6:20815, 2016.

[3] H. Arimura, T. Takagi, X. Geng, and T. Uno. Finding all maximal duration flock patterns in high-dimensional trajectories. 2014.

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

[5] L. Becker, A. Giesen, K. Hinrichs, and J. Vahrenhold. Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets. pages 270–285. Springer, 1999.

[6] 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*, page 322–331, New York, NY, USA, 1990. Association for Computing Machinery.

[7] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111–125, 2008.

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

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

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

[11] 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.

[12] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[13] A. Calderon. Mining moving flock patterns in large spatio-temporal datasets using a frequent pattern mining approach. Master's thesis, University of Twente, 2011.

[14] 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.

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

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

[17] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 109–120. Ieee, 2010. 00000.

[18] G. Di Lorenzo, M. Sbodio, F. Calabrese, M. Berlingerio, F. Pinelli, and R. Nair. AllAboard: Visual Exploration of Cellphone Mobility Data to Optimise Public Transport. *Ieee Tvcg*, 22(2):1036–1050, 2016.

[19] A. Eldawy. Spatialhadoop: Towards flexible and scalable spatial processing using mapreduce. In *SIGMOD PhD Symposium*, page 46–50, 2014.

[20] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *Algorithms and Computation*, pages 403–414, 2010.

[21] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[22] Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan, and Eti Ezra. The design and implementation of panar maps in CGAL. *ACM Journal of Experimental Algorithmics*, 5:13–es, December 2001.

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

[24] M. Fort, J. Antoni, and N. Valladares. A parallel GPU-based approach for reporting flock patterns. *Ijgis*, 28(9):1877–1903, 2014.

[25] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh Lam. *The SPMF Open-Source Data Mining Library Version 2*, pages 36–40. Springer, 2016.

[26] Andrew Frank, Jonathan Raper, and J. P. Cheylan, editors. *Life and Motion of Socio-Economic Units.* CRC Press, London ; New York, 1 edition edition, December 2000. 00071.

[27] 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.

[28] W. Freiseisen. Colored dcel for boolean operations in 2d, 1998.

[29] X. Geng, T. Takagi, H. Arimura, and T. Uno. Enumeration of complete set of flock patterns in trajectories. In *Iwgs*, page 53–61, 2014.

[30] Xiaoliang Geng, Takeaki Uno, and Hiroki Arimura. Trajectory Pattern Mining in Practice-Algorithms for Mining Flock Patterns from Trajectories. In *Kdir/kmis*, pages 143–151, 2013. 00002.

[31] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Report of fimi'03. *ACM SIGKDD Explorations*, 6(1):109–117, June 2004.

[32] J. Gudmundsson and M. van Kreveld. Computing Longest Duration Flocks in Trajectory Data. In *Acm Sigspatial*, pages 35–42, 2006.

[33] J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *Iwgis*, pages 250–257. Acm, 2004.

[34] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Acm Sigmod Icmd*, page 47–57, New York, NY, United States, 1984. Association for Computing Machinery.

[35] D. Halperin. Arrangements. In Jacob E. G. and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. CRC Press, 2004.

[36] Jiawei Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[37] Idit Haran and Dan Halperin. An experimental study of point location in planar arrangements in CGAL, February 2009.

[38] K. Holland, B. Wetherbee, C. Lowe, and C. Meyer. Movements of tiger sharks (Galeocerdo cuvier) in coastal Hawaiian waters. *Marine Biology*, 134(4):665–673, 1999.

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

[40] P. Huang and B. Yuan. Mining Massive-Scale Spatiotemporal Trajectories in Parallel: A Survey. In *Takddm*, volume 9441. Springer, 2015.

[41] Xiaoke Huang, Ye Zhao, Chao Ma, Jing Yang, Xinyue Ye, and Chong Zhang. TrajGraph: A Graph-Based Visual Analytics Approach to Studying Urban Network Centralities Using Taxi Trajectory Data. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):160–169, January 2016. 00000.

[42] J. Hughes, A. Annex, C. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Defense + Security Symposium*, 2015.

[43] Sachiko Iwase and Hideo Saito. Tracking Soccer Player Using Multiple Views. In *Mva*, pages 102–105, 2002. 00000.

[44] H. Jeung, M. Yiu, and C. Jensen. Trajectory pattern mining. In *Computing with Spatial Trajectories*, pages 143–177. Springer, 2011.

[45] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. Discovery of Convoys in Trajectory Databases. *Vldb*, 1(1):1068–1080, 2008.

[46] Karl Johansson and Hakan Terelius. An efficiency measure for road transportation networks with application to two case studies. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 5149–5155. Ieee, 2015. 00000.

[47] Alison Johnston, Daniel Fink, Mark D. Reynolds, Wesley M. Hochachka, Brian L. Sullivan, Nicholas E. Bruns, Eric Hallstein, Matt S. Merrifield, Sandi Matsumoto, and Steve Kelling. Abundance models improve spatial and temporal prioritization of conservation resources. *Ecological Applications*, 25(7):1749–1756, 2015. 00007.

[48] P. Kalnis, N. Mamoulis, and S. Bakiras. On Discovering Moving Clusters in Spatio-temporal Data. In *Astd*, pages 364–381. Springer, 2005.

[49] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[50] Colin Kuntzsch and Alexander Bohn. A Framework for On-line Detection of Custom Group Movement Patterns. In Jukka M. Krisp, editor, *Progress in Location-Based Services*, Lecture Notes in Geoinformation and Cartography, pages 91–107. Springer Berlin Heidelberg, January 2013. 00003.

[51] F. La Sorte, D. Fink, W. Hochachka, and S. Kelling. Convergence of broad-scale migration strategies in terrestrial birds. *Royal Society: Biological Sciences*, 283(1823):2588, 2016.

[52] Hoang Thanh Lam, Ernesto Diaz-Aviles, Alessandra Pascale, Yiannis Gkoufas, and Bei Chen. (Blue) Taxi Destination and Trip Time Prediction from Partial Trajectories. *ECML/PKDD Discovery Challenge 2015*, 2015. 00000.

[53] Y. Leung. *Knowledge Discovery in Spatial Data.* Springer, 2010.

[54] X. Li, T. Cao, E. Lim, Z. Zhou, T. Ho, and D. Cheung, editors. *Trends and Applications in Knowledge Discovery and Data Mining*, volume 9441. Springer, 2015.

[55] Y. Li, A. Eldawy, J. Xue, N. Knorozova, M. Mokbel, and R. Janardan. Scalable computational geometry in MapReduce. *Vldb*, 28(1):523–548, 2019.

[56] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *Vldb*, 3(1-2):723–734, 2010.

[57] Ying Long and Jean-Claude Thill. Combining smart card data and household travel survey to analyze jobs–housing relationships in Beijing. *Computers, Environment and Urban Systems*, 53:19–35, September 2015. 00000.

[58] 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.

[59] Dimitrios Makris and Tim Ellis. Path detection in video surveillance. *Image and Vision Computing*, 20(12):895–903, October 2002. 00165.

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

[61] H. Miller and J. Han. *Geographic Data Mining and Knowledge Discovery.* Taylor & Francis, Inc., 2001.

[62] Luis Moreira-Matias, Joao Gama, Michel Ferreira, Joao Mendes-Moreira, and Luis Damas. Predicting Taxi-Passenger Demand Using Streaming Data. *Trans. Intell. Transport. Sys.*, 14(3):1393–1402, September 2013. 00048.

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

[64] 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.

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

[66] C. Piciarelli, G. L. Foresti, and L. Snidaro. Trajectory clustering and its applications for video surveillance. In *IEEE Conference on Advanced Video and Signal Based Surveillance, 2005.*, pages 40–45, September 2005. 00000.

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

[68] 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.

[69] 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.

[70] 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.

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

[72] P. Tanaka, M.. Vieira, and D. Kaster. An Improved Base Algorithm for Online Discovery of Flock Patterns in Trajectories. *Jidm*, 7(1), 2016.

[73] U. Turdukulov, A. Calderon, O. Huisman, and V. Retsios. Visual mining of moving flock patterns in large spatio-temporal data sets using a frequent pattern approach. *Ijgis*, 28(10):2013–2029, 2014.

[74] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Fimi*, volume 126, 2004.

[75] Nacho Valladares Cereceda. *GPU Parallel Algorithms for Reporting Movement Behaviour Patterns in Spatiotemporal Databases.* PhD thesis, 2013. 00000.

[76] M. Vieira, P. Bakalov, and V. Tsotras. On-line Discovery of Flock Patterns in Spatio-temporal Data. In *Acm Sigspatial*, pages 286–295, 2009.

[77] M. Vieira and V. Tsotras. *Spatio-Temporal Databases: Complex Motion Pattern Queries.* Springer, 2013.

[78] Yida Wang, Ee-Peng Lim, and San-Yih Hwang. Efficient mining of group patterns from user movement data. *Data & Knowledge Engineering*, 57(3):240–282, June 2006. 00000.

[79] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to Cgal's arrangement package. *Computational Geometry*, 38(1-2):37–63, September 2007.

[80] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, pages 359–370. Springer, 1991.

[81] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Icmd*, page 1071–1085, 2016.

[82] J. Yu, J. Wu, and M. Sarwat. A demonstration of GeoSpark: A cluster computing framework for processing big spatial data. In *Icde*, pages 1410–1413, 2016.

[83] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. GeoSpark: A cluster computing framework for processing large-scale spatial data. pages 1–4. ACM Press, 2015. 00016.

[84] Y. Zheng and X. Zhou. *Computing with Spatial Trajectories.* Springer, 2011.

[85] Yu Zheng. Trajectory Data Mining: An Overview. *ACM Transactions on Intelligent Systems and Technology*, 6(3):1–41, May 2015. 00112.

[86] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. Understanding mobility based on GPS data. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, pages 312–321. Acm, 2008. 00462.

[87] Yu Zheng, Xing Xie, and Wei-Ying Ma. GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010. 00388.

[88] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. Mining interesting locations and travel sequences from GPS trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, pages 791–800. Acm, 2009. 00959.

[89] Chen Zhou, Zhenjie Chen, and Manchun Li. A parallel method to accelerate spatial operations involving polygon intersections. *International Journal of Geographical Information Science*, 32(12):2402–2426, December 2018.