UNIVERSITY OF CALIFORNIA
RIVERSIDE

Scalable spatial operations and pattern finding through algorithm paralellization

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

Copyright by
Andres Oswaldo Calderon Romero
2024

The Dissertation of Andres Oswaldo Calderon Romero is approved:

Committee Chairperson

University of California, Riverside

ABSTRACT OF THE DISSERTATION

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, November 2024
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 [58]. 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 [50], R-trees [30, 6], quadtrees [22], 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 [49, 54] 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 (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 [17, 51]) and robot motion planning (Minkowski sums [9, 16]). 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

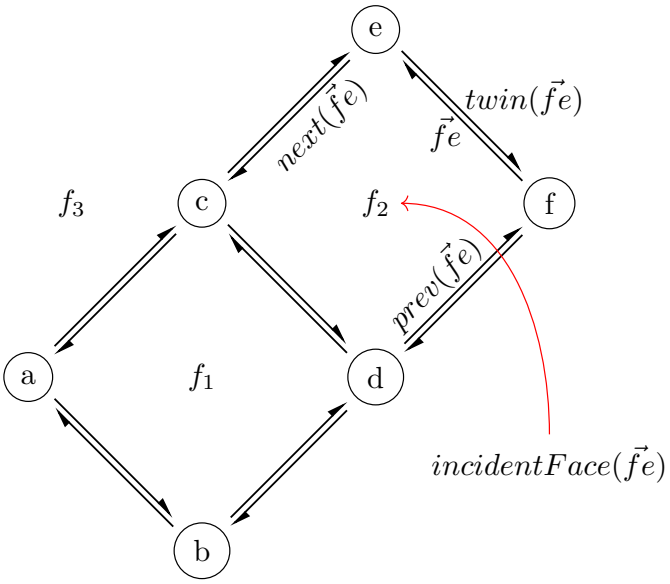


Figure 1.1: Components of the DCEL structure.

fig:dcel_example_prime⟩

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 [30, 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 oper-

ations, 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.

Chapter 2

Scalable overlay operations over DCEL polygon layers

2.1 Introduction

?⟨sec1⟩?

The use of spatial data structures is ubiquitous in many spatial applications, ranging from spatial databases to computational geometry, robotics, and geographic information systems [58]. 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 [50], R-trees [30, 6], and quadtrees [22]. *Edge-list* structures are also 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 [49, 54] is a data structure that 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 that divide adjacent faces; and the vertices are the point endings between adjacent edges (see Figure 2.1). 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 [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 [17, 51]) and robot motion planning (Minkowski sums [9, 16]). DCELs are also used to perform polygon unions (for example, on printed circuit boards to support the simplification of connected components in an efficient

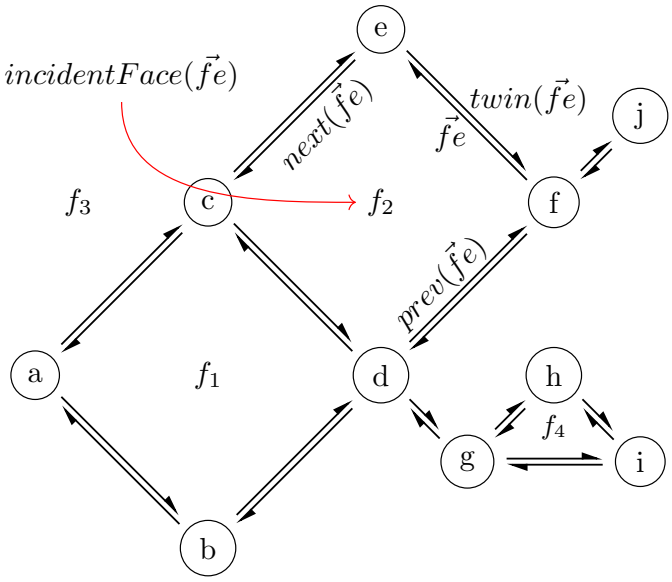


Figure 2.1: Components of the DCEL structure.

⟨fig:dcel_example⟩

manner [23]) as well as the computation of silhouettes from polyhedra [23, 8] (applied frequently in computer vision and 3D graphics modeling [10]).

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 [30, 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); 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.

In addition to the scalability issue, it is common in some applications that spatial polygons are provided in the form of scattered line segments, e.g., a set of road segments that form city blocks. Such data can be very large and appear in applications in urban planning, geo-targeted advertising, economic and demographic studies, etc. Yet, existing polygon overlay techniques cannot handle them directly at scale. In that setting, extracting the DCEL subdivision's faces (polygons) is not straightforward. To generate all of a subdivision's faces, the DCEL constructor must invoke a scalable *polygonization* procedure, which extracts all closed polygons formed by a collection of planar line segments in a subdivision.

This paper 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. Furthermore, we extend the overlay method to support input polygons in scattered line segments form by integrating a scalable and distributed polygon extraction approach. Our solutions have been implemented in a parallel framework (i.e., Apache Spark).

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. It should also consider the *dangle* and *cut edges* resulting from the polygonization process and their intersection with other polygon layers. 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.

This paper extends our previous work in [14]. The main new contributions are summarized as follows. First, we introduce a new spatial partitioner, based on the kd-tree partitioning strategy, for constructing overlay DCELs (section 2.4.1). Since it better utilizes the data distributions in optimizing DCEL partitions, it leads to noticeably improved performance. The new partitioning strategy contrasts with the original strategy that employed space-partitioning techniques based on quadtrees. Second, we enable overlay DCELs to take scattered and noisy line segments as input instead of being limited to clean polygon data.This builds on our work on scalable polygonization [1] to enable overlays of real datasets that consist of massive sets of line segments that cannot currently be handled by any existing technique. We also provide additional experiments, to quantify the benefits of the kd-tree based strategy, as well as the performance on the datasets with large volumes of line segments

The rest of this paper is organized as follows. Section 2.2 presents related work, while Section 2.3 discusses the basics of DCEL and the sequential algorithm. In Section 2.4, 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 2.5. Section 2.6 details the polygon extraction process for line input adaptation. It also extends the overlay method by supporting the overlay of dangle and cut edges. An extensive experimental evaluation appears in Section 2.7, while Section 3.6 concludes the paper.

2.2 Related Work

⟨sec:related⟩

The fundamentals of the DCEL data structure were introduced in the seminal paper by Muller and Preparata [49]. The advantages of DCELs are highlighted in [54, 9]. Examples of using DCELs for diverse applications appear in [5, 11, 26]. Our related work lies in two main areas, namely, *overlay operations* and *polygonization*, each discussed below.

Overlay operations. 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 [26]. Currently, few sequential implementations are available: LEDA [47], Holmes3D [34] 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 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), [15, 57, 44, 25, 46, 56, 55] 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.)

Polygonization. All available implementations for the polygonization procedure are built upon the JTS/GEOS implementation [39, 28]. While the JTS library is used in many modern distributed spatial analytics systems [53], including Hadoop-GIS [2], SpatialHadoop [20], GeoSpark [69], and SpatialSpark [67], the implementation of the polygonization algorithm [39] has not been extended to work in these distributed frameworks.

A data-parallel algorithm for polygonizing a collection of line segments represented by a data-parallel bucket PMR quadtree, a data-parallel R-tree, and a data-parallel R^+-tree was proposed in [32]. The algorithm starts by partitioning the data using the given data-parallel structure (i.e., the PMR quadtree, the R-tree, or the R^+-tree), beginning the polygonization at the leaf nodes. The polygonization starts by finding each line segment's left and right polygon identifiers in each node. Then children nodes are merged into their direct parent node, at which redundancy is resolved. This procedure is recursively called until the root node is reached, where all line segments have their final left and right polygon identifiers assigned. Each merging operation partitions the input data into a smaller number of partitions. At each iteration, the number of partitions decreases while the number of line segments entering and exiting each iteration remains constant. This implies that at the last iteration, the whole input line segment dataset must be processed on only one partition at the root node level. In the era of big data, where the use of commodity machines as worker nodes is common, this becomes a bottleneck when processing datasets of hundreds of millions of records on one machine. While our work and the approach in [32] rely on iterative data re-partitioning, [32] uses a constant input to each iteration while significantly decreasing the number of partitions. On the other hand, our input size decreases as the number of partitions decreases (thus avoiding processing the whole dataset on a single partition).

2.3 Preliminaries

⟨sec:prelim⟩

The DCEL [49] 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,

?⟨tab:records⟩?

Table 2.1: Vertex records.

⟨tab:vertices⟩

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

Table 2.2: Face records.

⟨tab:faces⟩

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.

⟨tab:hedges⟩

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	$\vec{bd}$	$\vec{ba}$	$\vec{fd}$
⋮	⋮	⋮	⋮	⋮	⋮

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 2.1-2.3, with the subdivision depicted in Figure 2.1.

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 2.1). 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 2.1 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). In addition, we note a couple of special half-edges. *Dangles* are the half-edges with one or both ends not incident on another half-edge endpoint. Half-edge $\overrightarrow{fj}$ and its twin are both considered dangle edges. *Cut-Edges* are the half-edges connected at both ends but do not form part of a polygon.

The half-edge $\overrightarrow{dg}$ and its twin are considered cut-edges.

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 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 2.2. 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 2.1). 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 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 2.2, this list is empty as there are no holes in any of the faces in the example of Figure 2.1.

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 $f1 \cap f2$ [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 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 2.2 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, in-

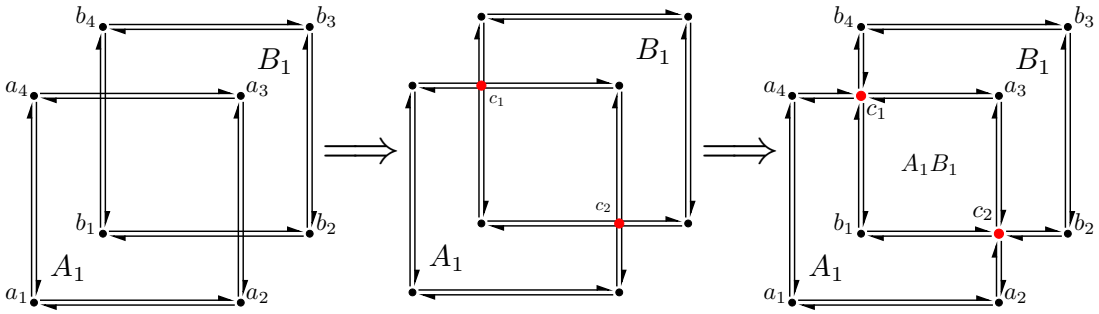


Figure 2.2: Sequential computations of an overlay of two DCEL layers.

⟨fig:dcel_seq⟩

tersection points are identified, and new vertices are created in the overlay (red vertices c_1 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 [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 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 2.3) 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.)

2.4 Scalable Overlay Construction

⟨sec:methods⟩

This section presents the construction of overlay DCELs, assuming only polygons as input without scattered line segments. 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

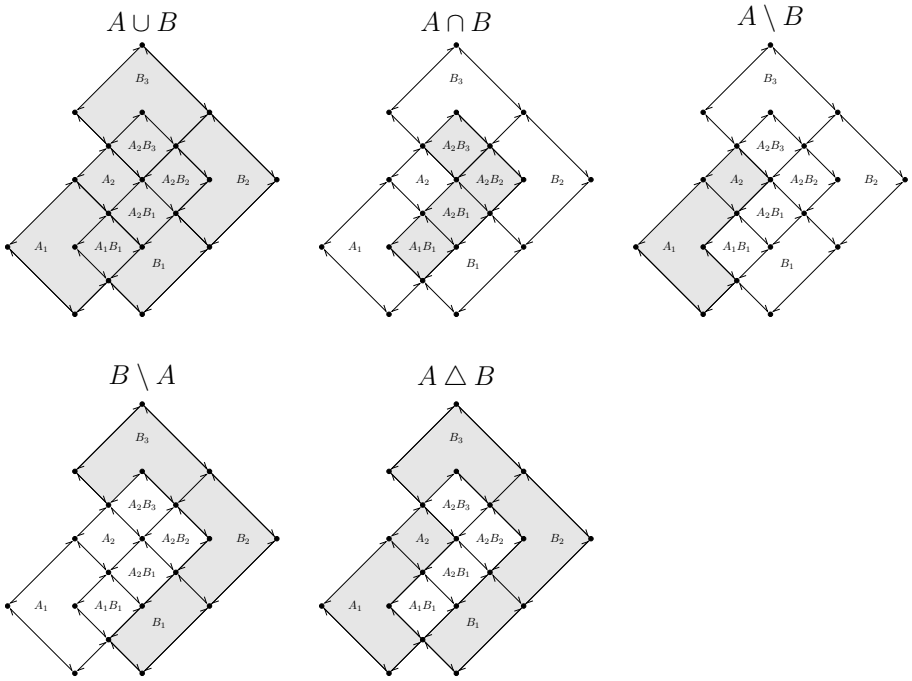


Figure 2.3: Examples of overlay operators supported by DCEL; results are shown in gray.
⟨fig:dcel_operators⟩

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 2.7). 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.

2.4.1 Partition Strategies

⟨sec:pstrategies⟩

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.

Therefore, an advanced partitioning strategy is better since it adapts to skewed spatial distributions and helps assign a similar number of edges to each cell. In particular, we used two partitioning strategies, one based on the quadtree (i.e. space-oriented) and one on the kd-tree (i.e. data-oriented) indexes.

Note that such tree-based data partitioning involves shuffling all edges; this however, happens only once. Our experimental evaluation (see Section 2.7.5) shows that the data-oriented approach leads to better performance. Nevertheless, in describing the various challenges (orphan cells and holes, overlay evaluation, and optimizations) we use the quadtree-based partition since its well-defined space-oriented partitioning makes the presentation easier.

Quadtree Partition Strategy

⟨sec: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. A quadtree data structure follows a space-oriented approach, given that it does not consider each cell's content at the moment of a possible split. 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 2.4 shows an 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 2.4). 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 2.4). 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).

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 2.4, Figure 2.5 depicts an overview of the process for creating a local overlay DCEL inside cell 2. Similarly, Figure 2.6 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

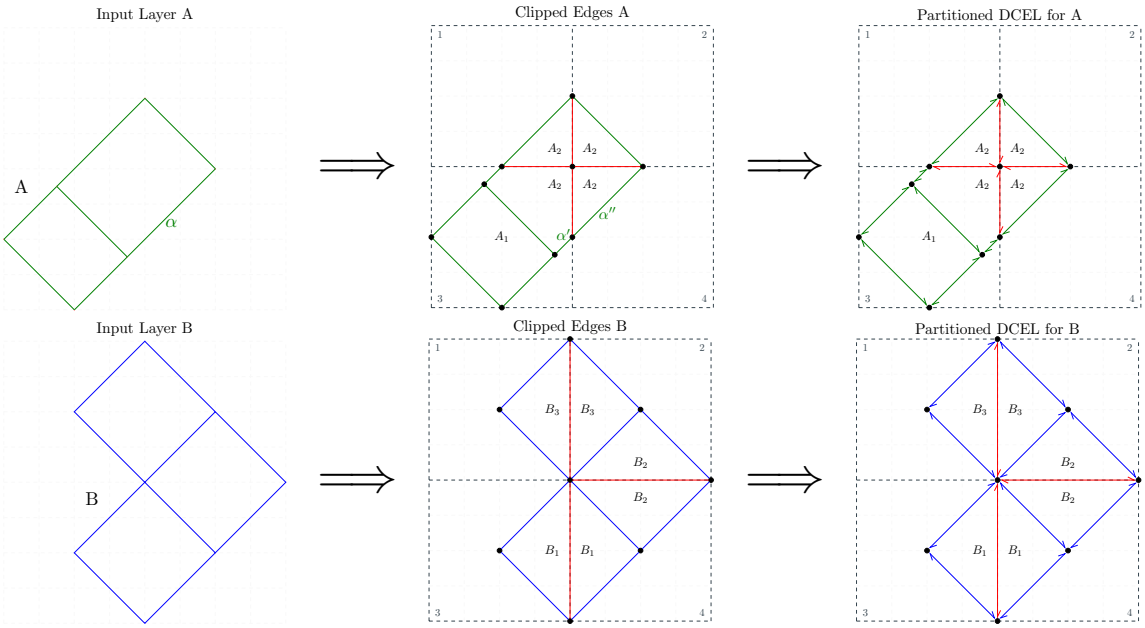


Figure 2.4: Partitioning example using input layers A and B over four cells.

⟨fig:partition_strategy⟩

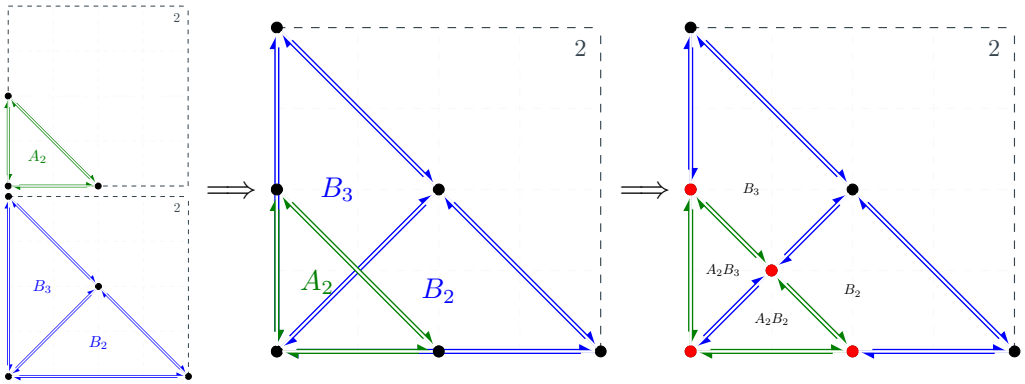


Figure 2.5: Local overlay DCEL for cell 2.

⟨fig:overlay_partition⟩

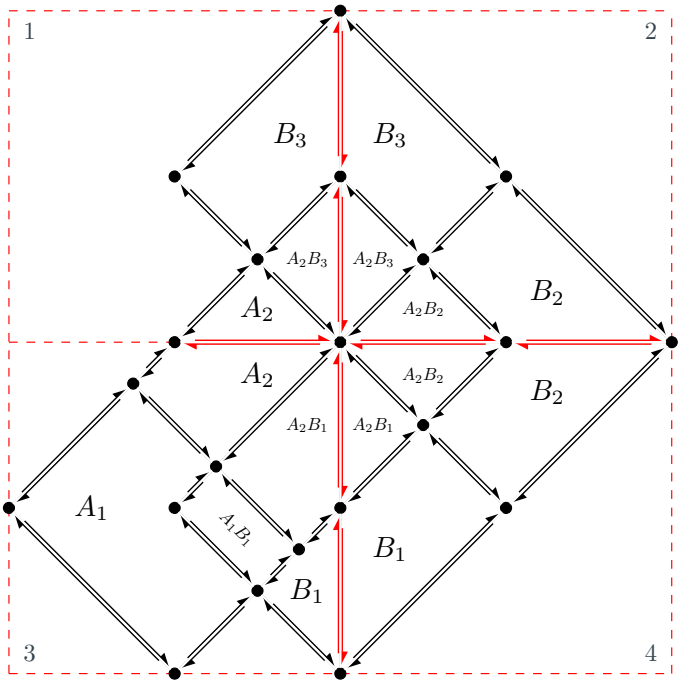


Figure 2.6: Result of the local overlay DCEL computations.

⟨fig:distributed_dcel⟩

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 2.7, 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 2.7). 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 2.4.2, 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.

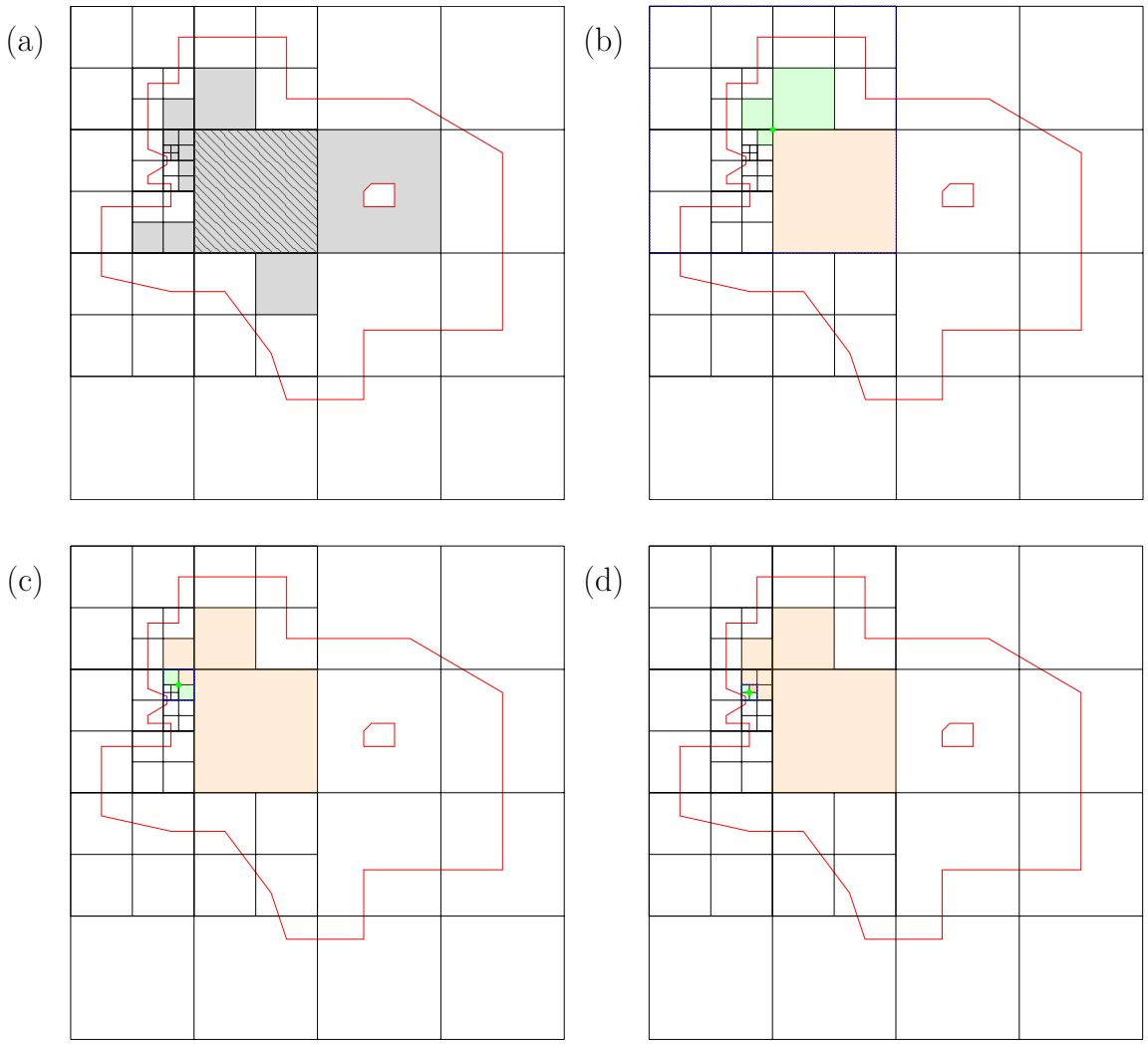


Figure 2.7: (a) Empty cell and hole examples; (b)-(c)-(d) show three iterations of the proposed solution.

⟨fig:orphan_cells⟩

Kd-tree Partition Strategy

⟨sec:kdtreestrategy⟩

The kd-tree based partitioning is a data-oriented approach because it sorts and picks the middle point inside a cell to locate the split of the future children.

Building and populating the kd-tree partitioning follows a procedure similar to that of the quadtree, by first building a kd-tree from a sample of the input data. 1% of the input data is used to build a kd-tree and extract the tree's structure. The leaves of this structure are the partition's cells. We feed the input data into the generated kd-tree structure to assign each edge to the leaf cell that has the edge within its boundaries. After the partitioning is done, the construction of the local DCELs for each layer and the overlay operation is performed in each local cell in the same fashion as described in section 2.4.1.

2.4.2 Labeling Orphan Cells and Holes

⟨sec:anomalies⟩

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 2.7(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:

?⟨lem:cells⟩?

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 2.7(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 2.7(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 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 2.7(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 2.7 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 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.

Algorithm 1 GETNEXTCELLWITHEDGES algorithm

⟨alg:one⟩ **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

⟨alg:two⟩
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

2.4.3 Answering global overlay queries

⟨sec:reduce⟩

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 2.8 shows how the intersection overlay operator ($A \cap B$) is computed, starting with the local DCELs for four cells in Figure 2.8(a). First, each cell computes the intersection using its local overlay DCEL as shown in Figure 2.8(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 2.8(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 2.8(b) were combined into one face in Figure 2.8(c). While the extra vertex was

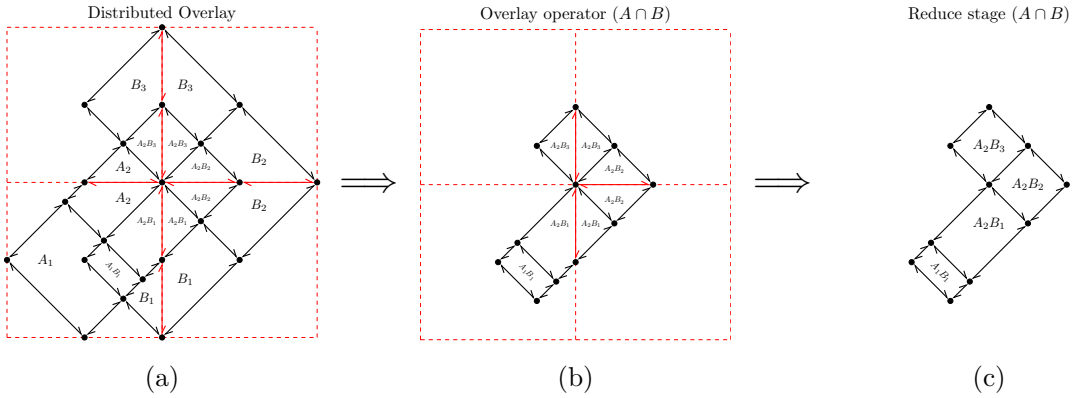


Figure 2.8: Example of an overlay operator querying the distributed DCEL.

⟨fig:overlay_operator⟩

also removed. In section 2.5.1, 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.

2.5 Overlay evaluation optimizations

ec:alternative_methods⟩

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

2.5.1 Optimizations for faces spanning multiple cells

⟨sec:optimizing⟩

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 2.7.1 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.

2.5.2 Optimizing for unbalanced layers

⟨sec:unbalance⟩

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.

2.6 Scalable Polygon Extraction for Line-based Input

⟨sec:polygonization⟩

Our discussion so far assumed the input data is a set of clean and closed polygons in the two input layers to be overlayed. However, several real-world polygons, such as city blocks formed by individual road segments represented as spatial lines, are unavailable in the polygonal form. Forming polygons in such cases at a large scale is non-trivial and takes significant computing cost. This section further extends our scalable DCEL overlay operations to handle scattered line segments as input through a scalable polygonization [1] process. Such a feature enables spatial data scientists to seamlessly exploit a rich set of publicly available datasets, e.g., spatial road networks worldwide [52, 62].

Building a DCEL data structure from an input of planar line segments extracts all closed polygons during the invocation of the *polygonization* procedure. In our work in [1], we proposed a scalable distributed framework to build a DCEL and extract polygons in parallel from the input line segments. Figure 2.9 shows an overview of the DCEL constructor.

To create a DCEL data structure from input line segments, the *DCEL constructor* undergoes a two-phase paradigm. The *Gen Phase*, detailed in Section 2.6.1, spatially partitions the input lines, generating the subdivision's vertices (V) and half-edges (H), and a subset of the subdivision's faces (F_0).

The remaining line segments that are not assigned to a face yet are passed to the

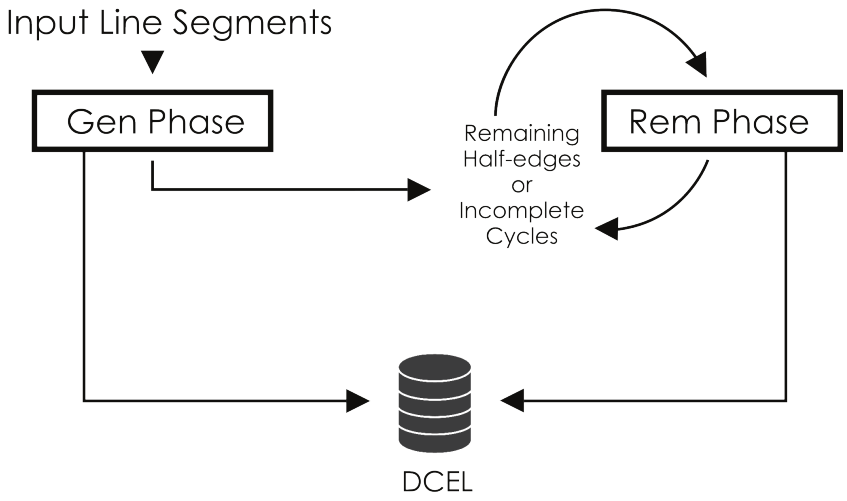


Figure 2.9: DCEL Constructor for Polygonization Overview

⟨fig:overview_ddcel⟩

subsequent phase in the form of half-edges or incomplete cycles. An incomplete cycle is a connected half-edge list that is a candidate face. The *Rem Phase*, detailed in Section 2.6.2, generates the subdivision's remaining faces, F_j, $\forall j > 0$.

Section 2.6.3 discusses different data re-partitioning schemes with a minimal number of iterations to reduce the workload of the *Rem Phase* without compromising correctness. The polygonization procedure produces two outputs: first, a set of closed polygons formed by the input planar line segments, and second, any edges that are not a part of any polygon, i.e., dangle or cut edges. Overlaying the polygons generated with any polygon layer follows the approaches discussed in sections 2.4and 2.5. In section 2.6.4, we extend the overlay approaches to handle overlaying a polygon layer with the remaining edges (the dangle and cut edges).

2.6.1 Gen Phase

⟨sec:gen⟩

The Gen phase accepts an input dataset of line segments N and starts by partitioning the input across the worker nodes in a distributed cluster using a global quadtree spatial index. Each data partition P_i covers a specific spatial area represented by its minimum bounding rectangle (MBR) B_i. Figure 2.10 shows an example of four leaf nodes of a quadtree built for input spatial line segments. Solid lines represent the line segments, and dashed lines represent the partitions' MBRs.

After spatially partitioning the input lines, each partition generates its vertices, half-edges, and faces (collectively the partition DCEL) using the subset of the dataset that intersects with the partition's MBR. The *partition vertices* are the vertices that are

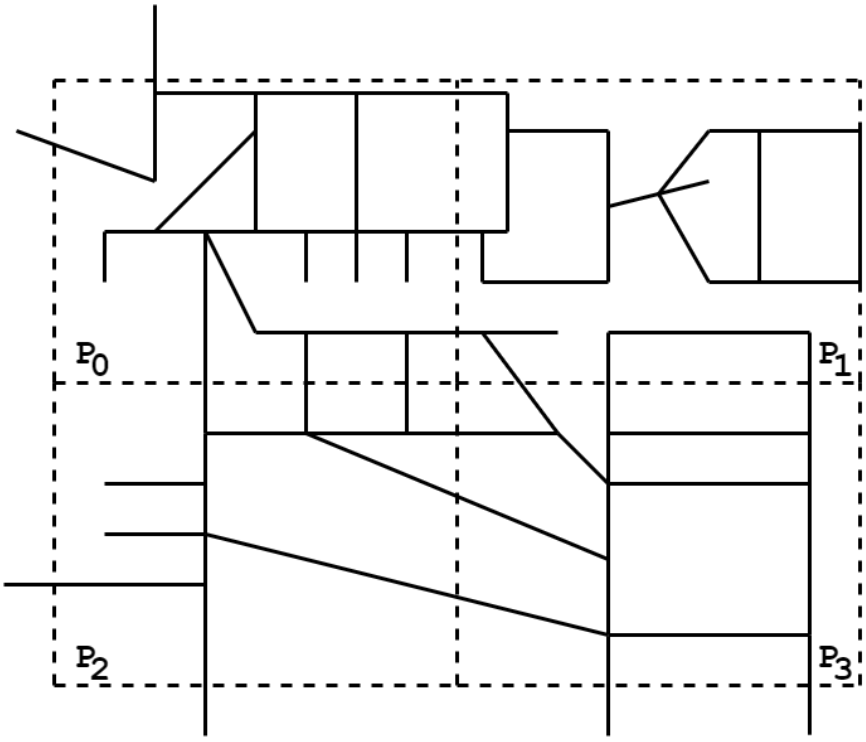


Figure 2.10: Partitioned input spatial lines.

⟨fig:ddcel:input⟩

wholly contained within the partition MBR. On the other hand, the *partition half-edges* are any half-edge that intersects with the partition MBR. *Partition faces* are the faces that are wholly contained within the MBR of the partition. On each data partition P_i, the Gen phase undergoes four main procedures; (1) first, generating the partition vertices and half-edges, (2) second, marking the dangle half-edges, (3) third, setting the next half-edge pointers for all half-edges and marking the cut edges, (4) lastly, generating the partition faces.

Step 1: Generating the Partition Vertices and Half-edges.

In the first step, the Gen Phase starts with populating the vertices and the half-edges RDDs of the DCEL data structure. Each partition P_i receives a subset of the input dataset that intersects with the partition's boundary. For every line segment object o received at partition P_i ($o \in P_i$), two vertices are generated (v_1, v_2); one for each endpoint on this line segment (p_1, p_2). These two vertices objects (v_1, v_2) are appended to the vertices RDD in the DCEL data structure. We also generate two half-edges (h_1, h_2) for every line segment. The first half-edge h_1 has its destination vertex v_1, while the other half-edge h_2 has its destination vertex v_2. These two half-edges are assigned as twins. The half-edge h_1 is appended to the incident list of the vertex v_1. Similarly, h_2 is appended to v_2's incident

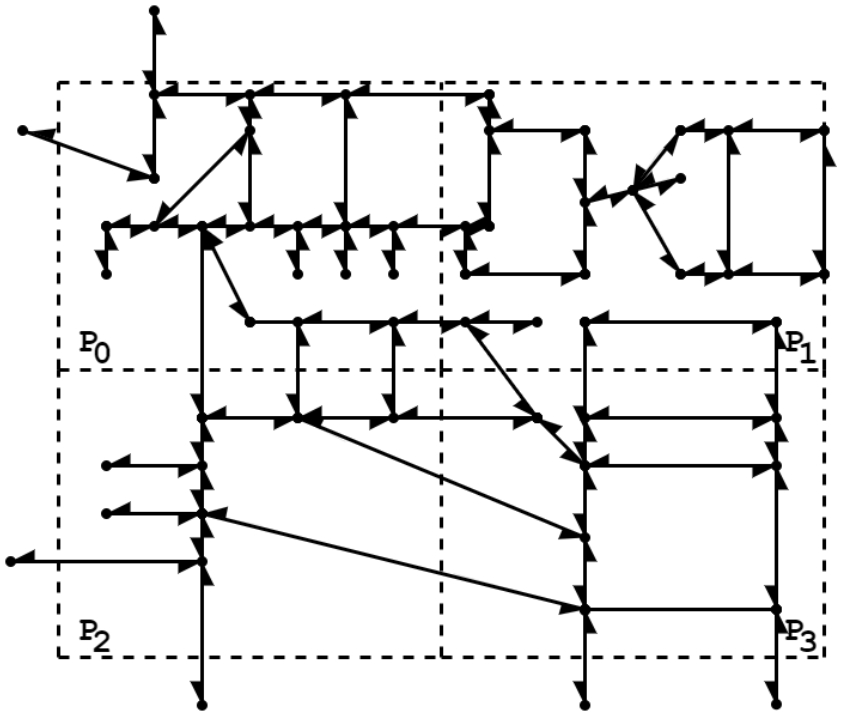


Figure 2.11: DCEL vertices and half-edges.

⟨fig:ddcel:step1⟩

list. For a half-edge to span multiple partitions, we check whether it is wholly contained within the partition MBR B_i; if not, and it is just intersecting, then this half-edge spans multiple partitions. These half-edges are duplicated on all partitions they intersect with. The remaining attributes of each half-edge object are assigned in the subsequent steps. The two generated half-edge objects (h_1, h_2) are appended to the half-edges RDD in the DCEL data structure. Figure 2.11 shows a graphical illustration of the DCEL data structure representing the input lines after generating the vertices and the half-edges on all data partitions.

Step 2: Marking the Dangle Half-edges.

Dangle half-edges are not part of any face; thus, marking them is essential to exclude them during the polygonization procedure. To find dangles in the input lines, we use previously generated information, i.e., information about the vertices and their incident half-edges. We compute the degree of each vertex $v \in V$ populated in the previous step. A vertex degree is the number of non-dangle half-edges in its incident half-edges list. If the degree of an arbitrary vertex v is less than or equal to 1 ($degree(v) \leq 1$), then all of v's incident half-edges and their twins are also dangle half-edges. Marking any new half-edge as a dangle requires

recomputing the degree of the vertices connected to it. Thus, marking the dangle half-edges is an iterative process. After the initial run over all vertices and marking the initial dangle half-edges, we reiterate over the vertices to check for newly found dangle half-edges. We keep iterating until convergence when no new dangle half-edges are detected.

Step 3: Setting the Half-edges' Next Pointers, and Marking the Cut Edges.

The third step is divided into three smaller steps: (a) setting the next half-edge pointer for each half-edge, (b) marking the cut edges, and (c) updating the next half-edges accordingly. To set the next pointer for each half-edge, we use information from the previous two steps, i.e., the vertices incident half-edges and the current dangle half-edges. For each vertex $v \in V$, we sort its incident half-edges list in clockwise order, excluding the dangle half-edges. After sorting the incident half-edges list $v.incidentH$, for every pair of half-edges $v.incidentH[t]$, $v.incidentH[t+1]$ in the sorted list, we assign $v.incidentH[t].next$ to $v.incidentH[t+1].twin$. For the last incident half-edge in the sorted list $v.incidentH[v.incidentH.len-1]$, we assign its next half-edge to $v.incidentH[0].twin$.

After the initial assignment of the next half-edge pointers, we proceed with the second sub-step, marking the cut edges. To mark the cut edges, we start our procedure at an arbitrary half-edge $h_{initial}$ and assign our $h_{current}$ half-edge pointer to it. We advance the $h_{current}$ pointer at each iteration to the $h_{current}$'s next ($h_{current} = h_{current}.next$), storing all visited half-edges in a list (current cycle). We keep advancing the $h_{current}$ pointer till we reach one of three cases. (1) We return to the initial half-edge $h_{initial}$, which means a cycle is detected and no cut edge is detected. (2) The half-edge $h_{current}.next$ is not available, which also means no cut edge is detected. (3) We find $h_{current}.twin$ in the current cycle, which means that $h_{current}$ and its twin are both cut edges. Once we reach one of these cases, we mark all visited half-edges as such and proceed with a new arbitrary half-edge to be $h_{initial}$. This process is terminated when all the partition half-edges are visited.

In the third sub-step, after marking all cut edges, we update the next pointers while excluding the cut edges. For each vertex $v \in V$, we sort its incident half-edges list in clockwise order again, now while excluding both the dangle and the cut edge half-edges. After sorting the incident half-edges list $v.incidentH$, we re-execute the same process of the first sub-step, assigning $v.incidentH[t].next$ to $v.incidentH[t+1].twin$. Figure 2.12 shows the DCEL data structure after removing the dangle and cut edges.

Step 4: Generating the Partition Faces.

Polygonization on each partition P_i starts with selecting an arbitrary half-edge as our initial half-edge $h_{initial}$. We initially assign our $h_{current}$ half-edge pointer to $h_{initial}$. We advance the $h_{current}$ pointer at each iteration to the $h_{current}$'s next ($h_{current} = h_{current}.next$), storing

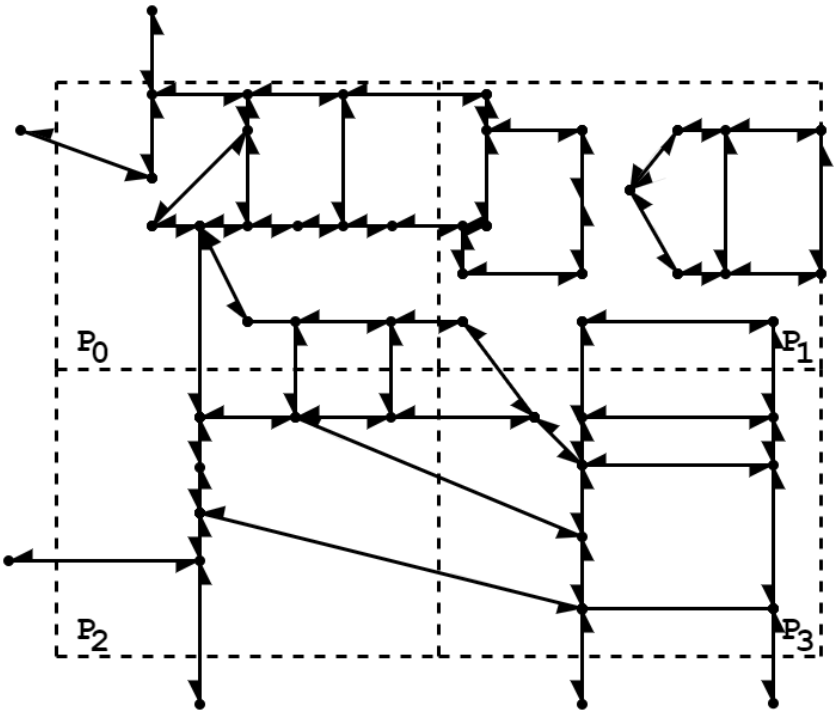


Figure 2.12: DCEL vertices and half-edges after dangle and cut edge removal.

⟨fig:ddcel:step2⟩

all visited half-edges in a list *cycle*. We keep advancing the $h_{current}$ pointer till we reach one of the following cases: (1) We return to the initial half-edge $h_{initial}$, which means that we have found a face. In this case, we add the found face f to the faces collection F_0 and assign $h.incidentF = f, \ \forall h \in cycle$. (2) The $h_{current}.next$ is not available, and $h_{current}$ is a half-edge that spans multiple partitions. In this case, the cycle needs more information from the neighboring partitions to be completed, and the current partition's data is insufficient to produce this face. To complete this cycle, we either pass the incomplete cycle into the Rem phase (the current list *cycle*), where it collects all incomplete cycles from all partitions and attempts to join them to form a face. Another approach would be passing the plain half-edges in this cycle to the next phase. Both approaches are discussed in detail in Section 2.6.2. Once we finish processing this cycle, we mark all visited half-edges as such, clear the cycle, and proceed with a new arbitrary half-edge to be $h_{initial}$. This process is terminated when all the partition half-edges are visited. In Figure 2.13, the dotted faces are the faces generated in this phase (Gen Phase).

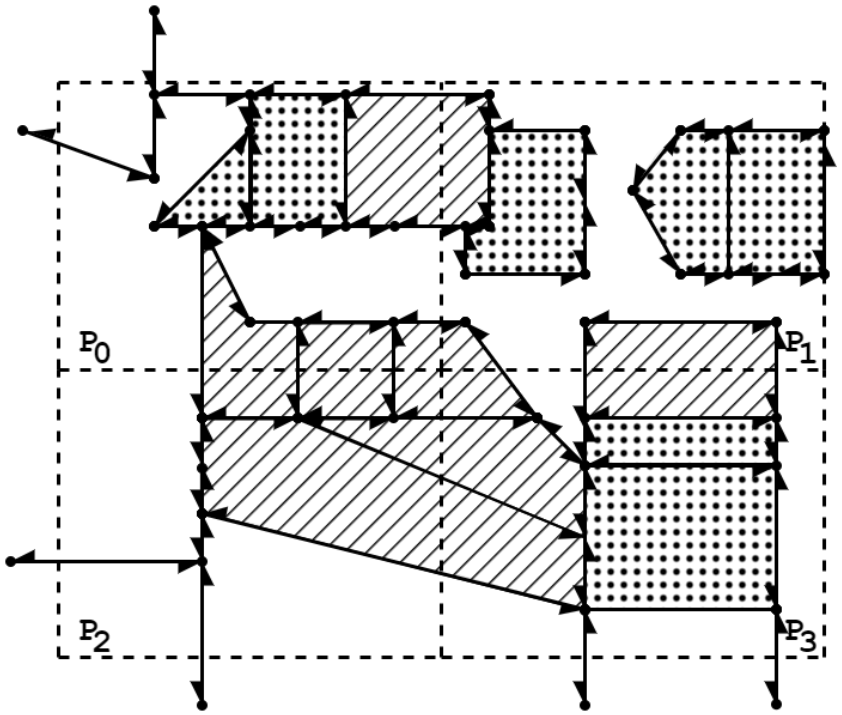


Figure 2.13: DCEL faces.

⟨fig:ddcel:faces⟩

2.6.2 Rem Phase

⟨sec:rem⟩

The Rem Phase accepts the remaining half-edges or incomplete cycles as input. To be included in the remaining half-edges set, a half-edge cannot be a dangle or a cut edge. Also, the half-edge should not have been bounded to a face yet. An incomplete cycle is a sequence of half-edges that acts as a candidate face. This incomplete cycle could not be completed since their marginal half-edges span multiple partitions.

The Rem Phase is an iterative phase, where each iteration j generates a subset of faces F_j. The unused input data at iteration j is passed to the next iteration $j+1$. Faces generated from the Gen phase and the Rem phase constitute the whole faces of the subdivision F. In each iteration, the Rem Phase starts with re-partitioning the input data across the worker nodes using a new set of partitions. This new set of partitions satisfies the convergence criteria; the new number of partitions (k_j) at iteration j must be less than the number of partitions (k_{j-1}) at iteration $j-1$. This criterion ($k_j < k_{j-1}$) ensures there is an iteration (m) at which the remaining line segments are re-partitioned to one partition only, where m is the total number of iterations of the Rem Phase, converging the problem into a sequential one and guaranteeing the termination of the procedure. After the data re-partitioning, we proceed with generating a subset of the remaining faces. Two

approaches are employed for the remaining faces generation, depending on the phase input data. The first approach assumes the phase input is a set of the Remaining Half-edges (RH Approach). While the second approach assumes the input is a set of the Incomplete Cycles (IC Approach).

RH Approach: Iterate over the Remaining Half-edges.

At each iteration j and on each new data partition, a subset of the remaining half-edges is received. Duplicate half-edges received on one new partition are merged into a single half-edge choosing the half-edge with the available next half-edge. We follow the same procedure of generating faces in the Gen Phase. Starting from an arbitrary half-edge as our initial half-edge $h_{initial}$, we assign our $h_{current}$ half-edge pointer initially to $h_{initial}$. We advance the $h_{current}$ pointer at each iteration to the $h_{current}$'s next ($h_{current} = h_{current}.next$), storing all visited half-edges in a list *cycle*. We keep advancing the $h_{current}$ pointer till we reach one of the following cases: (1) We return to the initial half-edge $h_{initial}$, which means that we have found a face. In this case, we add the found face f to the faces collection F_j and assign $h.incidentF = f, \ \forall h \in cycle$. (2) The $h_{current}.next$ is not available, and $h_{current}$ is a half-edge that is not wholly contained in the new partition MBR. Once we finish processing this cycle, we mark all visited half-edges as such, clear the cycle, and proceed with a new arbitrary half-edge to be $h_{initial}$. This iteration is terminated when all the remaining half-edges are visited. All half-edges that have not been assigned to any face yet are passed to the next iteration. The Rem Phase terminates if (1) there are no more remaining half-edges, i.e., all non-dangle non-cut edge half-edges are assigned to a face, or (2) the remaining half-edges have been processed on one partition.

IC Approach: Iterate over the Incomplete Cycles.

At each iteration j, and on each new data partition, a subset of the incomplete cycles is received. Starting from an arbitrary incomplete cycle $c_{initial}$ with first half-edge $first(c_{initial})$ and last half-edge $last(c_{initial})$, where the first and last half-edges are the incomplete cycle's terminal half-edges, we search for a match c_{match} in the remaining incomplete cycles such that the $last(c_{initial}) = first(c_{match})$. When a match is found, we merge the two cycles such that the $last(c_{initial})$ is now the $last(c_{match})$. We keep merging cycles till we reach one of the following cases: (1) The $last(c_{match}) = first(c_{initial})$, which means the cycle is now completed. In this case, we add the found face f to the faces collection F_j and remove all incomplete cycles used from the set of the incomplete cycles. (2) We can not find a match for the current last half-edge, and the last half-edge is not wholly contained within the new partition's MBR. In this case, the incomplete cycle needs more information from the neighboring partitions to be completed, and the current partition's data is insufficient to produce this face. Once we finish processing this matching process, we mark all visited incomplete cycles as such and proceed with a new arbitrary incomplete cycle to be $c_{initial}$. This itera-

tion j is terminated when all the incomplete cycles are visited. All incomplete cycles that are not completed yet are passed into the next iteration. The Rem Phase terminates if (1) there are no more remaining incomplete cycles, i.e., all cycles have been completed, or (2) the incomplete cycles have been processed on one partition. In Figure 2.13, the hatched faces are the faces generated in the first iteration ($j = 1$) of the Rem Phase.

2.6.3 Data Partitioning

sec:partitioning_ddcel⟩

The quadtree partitioner is used again to distribute the data amongst the worker nodes across the cluster. In the Gen Phase, the quadtree leaf nodes are used as the initial data partitions. The output of the Gen Phase, whether the remaining half-edges or the incomplete cycles, is iteratively re-partitioned into new sets of partitions. Each iteration set of partitions must satisfy the convergence criterion to ensure that the Rem Phase will terminate. We employ the same quadtree partitioner to generate the new partitions. Assume we have a quadtree built on the input line segments of height L. At the Gen Phase, we use nodes at the leaf level L as our initial data partitions. For each iteration j in the Rem Phase, we level up in the quadtree and choose different level nodes, aside from the leaves, to be our current data partitions. We keep leveling up in the quadtree till we reach the root ($l = 0$), which means that all data is located on only one partition (the root). Going up in the quadtree ensures that the number of partitions at iteration $j + 1$ is less than that at iteration j since the number of nodes at any arbitrary level l visited at iteration j is more than that at level l_{chosen}, $\forall l_{chosen} < l$ visited at iteration $j + 1$.

We always start with the leaf nodes level L in the Gen Phase. Choosing which levels to visit next in each iteration j is a system parameter. We offer different schemes for the visited quadtree levels:

1. Going directly to the root node at $l = 0$ after the leaf nodes, i.e., visiting only levels L in the Gen and 0 in the Rem phases. However, the experimental evaluation shows that collecting the data after the Gen phase on one node is prohibitive, and one worker node will not be able to process the Gen phase's output.

2. Going 1 Level Up (1LU) each iteration, i.e. if we visit level l at iteration j, we go to level $l - 1$ at iteration $j + 1$. This means the Rem Phase visits all the quadtree levels resulting in L iterations.

3. Going 2 Levels Up (2LU) each iteration resulting in half the number of iterations $\frac{L}{2}$ compared to 1LU.

4. Skipping to the Middle of the tree at level $\frac{L}{2}$, then continue going 1 level up for the remaining levels (M1LU), which will also result in $\frac{L}{2}$ iterations.

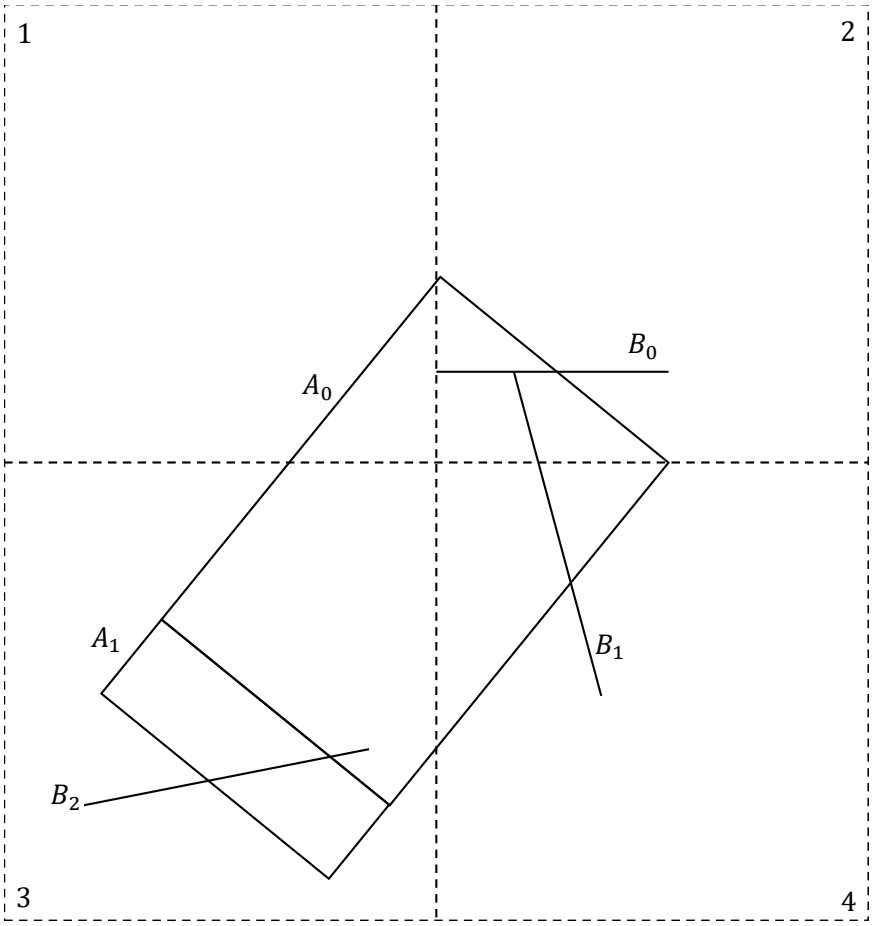


Figure 2.14: Spatial partitioning of input layers A and B

ig:dangleoverlay:input⟩

5. Skipping to the Middle of the tree every time, dividing the current level by two each iteration (MU); this will result in $\log_2(L)$ iterations.

The goal is to find a re-partitioning scheme with a minimal number of iterations, thus reducing the workload of the Rem Phase while ensuring that the worker nodes can process the chunk of the data it receives at each iteration j. The extreme case of having only one iteration at the Rem Phase will not work since the data is too big to fit one partition and be processed by only one worker node. On the other hand, the more unnecessary iterations we have, the more overhead on the system resulting in higher query latency.

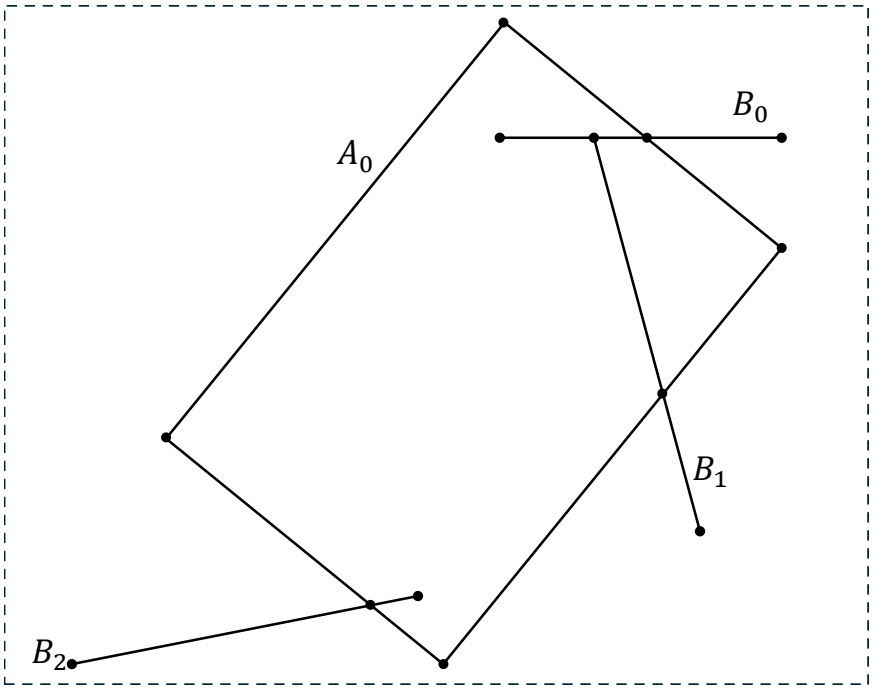


Figure 2.15: Re-Partitioning of polygon A_0 with edges it intersects with

⟨ig:dangleoverlay:inter⟩

2.6.4 Overlaying Polygons with Dangle and Cut Edges

⟨sec:over_dang⟩

The polygonization procedure produces two outputs: first, a set of closed polygons formed by the input planar line segments, and second, any edges that are not a part of any polygon, i.e., dangle or cut edges. Overlaying the polygons generated with any polygon layer follows the approaches discussed in sections 2.4 and 2.5. However, we need to modify the algorithms provided in these previous sections to overlay an input polygon layer A with the dangle and cut edges, i.e. layer B. In particular, we modify the reduce phase. Figure 2.14 illustrates the spatial partitioning of the two input layers, A and B. Layer A contains two input polygons, A_0 and A_1, while Layer B consists of three dangle edges, B_0, B_1, and B_2.

Each edge from layer B is labeled a unique label and is fed as an input to the overlay module. The local overlay is performed by finding intersections between the input polygon layer A and layer B on each data partition. If a polygon with $id = i$ from polygon layer A intersects with edges with ids $id = a, id = b, id = c$ from layer B at some data partition, we generate a label to match these intersections $A_iB_aB_bB_c$. At the reduce phase, we re-partition the data by the first label, meaning we collect all edges that intersect with the first label. If two data partitions produced the labels $A_iB_aB_bB_c$ and $A_iB_xB_y$, we repartition the data such that A_i is on one partition with all edges it is intersecting, i.e.,

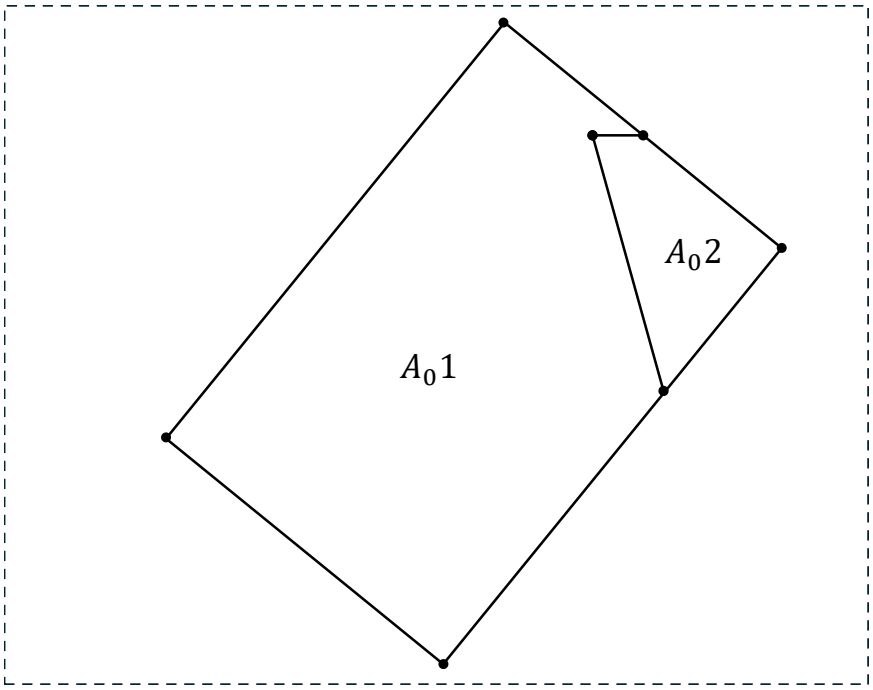


Figure 2.16: The result of polygonization of A_0 with B_0, B_1, B_2

g:dangleoverlay:result⟩

B_a, B_b, B_c, B_x, B_y. In Figure 2.15, Polygon A_0 is re-partitioned along with the edges it intersects, specifically B_0, B_1, and B_2.

After re-partitioning the data, we have all edges from both layers intersecting each other on the same partition. The next step is to find the polygons generated by these intersections. Since there is no guarantee that only one polygon is generated, we substitute the polygon concatenation proposed in Section 2.4.3 by performing *polygonization* on each partition. The polygonization procedure ensures it generates all new possible polygons. The polygonization procedure follows the algorithm mentioned in Section 2.6.1. It starts with generating the new vertices and half-edges, then marking the current dangles and cut edges, then setting the next pointers and finally generating the partition polygons. Figure 2.16 shows the result of polygonizing the edges from Polygon A_0 and B_0, B_1, and B_2, resulting in two polygons, $A_0 1$ and $A_0 2$.

Polygons from all partitions generate the overlay between the polygon layer A and layer B.

2.7 Experimental Evaluation

⟨sec:experiments⟩

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)

Table 2.4: Evaluation Datasets

⟨tab: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

and 2G memory.

Evaluation datasets. The details of the real datasets of polygons that we use are summarized in Table 3.1. 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[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 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 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 [9].

The scalable approach was implemented over the Apache Spark framework. From a Map-Reduce point of view the stages described in Section 2.4 were implemented using several transformations and actions supported by Spark. For example, the partitioning and

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

load balancing described in Section 2.4.1 and 2.6.3 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 2.4) 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 2.5.

2.7.1 Overlay face optimizations

c:overlay_optimization⟩

We first examine the optimizations in Section 2.5.1. 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 2.17 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 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 2.5 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, for the *By Label* approach, this shuffling is parallelized.

Table 2.5: Percentages of edges in incomplete faces for three states

⟨tab:percentages⟩

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

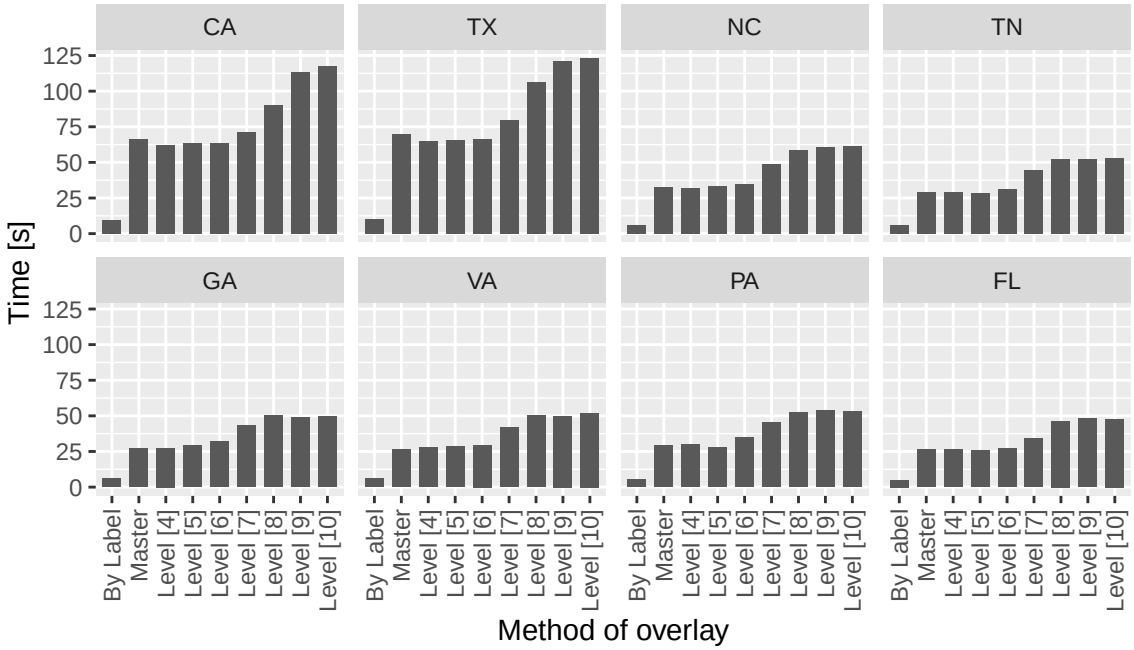
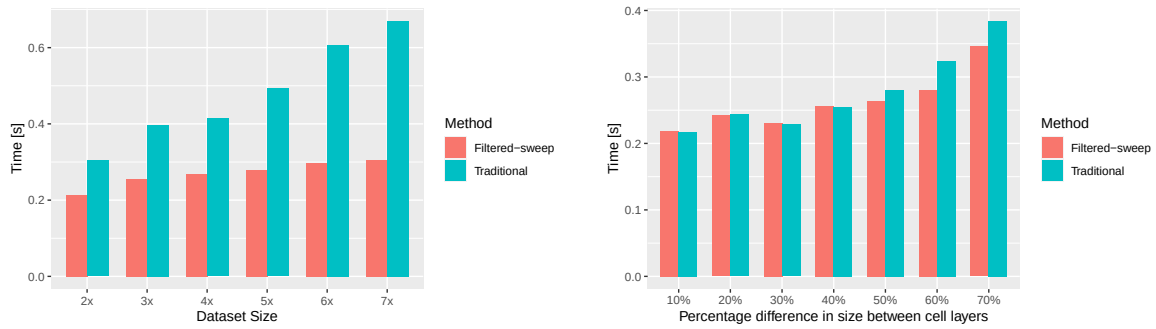


Figure 2.17: Overlay methods evaluation.

⟨fig:overlay_tester⟩



(a) (b)

Figure 2.18: Evaluation of the unbalanced layers optimization.

⟨fig:unbalance_tests⟩

2.7.2 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 2.5.2). 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 2.18(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, 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 2.18(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.

2.7.3 Varying the number of cells

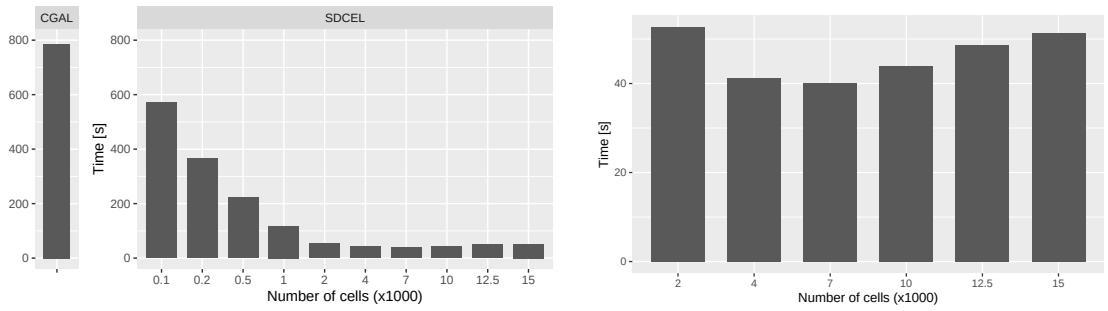
The quadtree configuration allows for performance tuning by setting the *maximum capacity* of a cell. The quadtree continues splitting until this capacity is reached. There is an inverse relationship between the capacity and the number of leaf cells: a lower capacity results in more cells, while a higher 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 2.19(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 2.19(b) focuses on that area; the best SDCEL performance was around 7K cells.

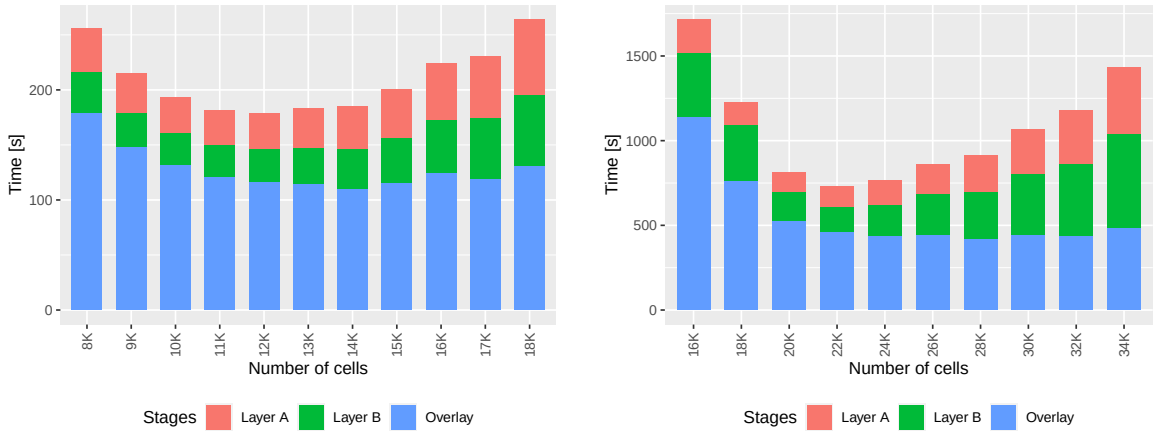
In addition, Figure 2.19(a) shows the performance of the sequential solution (CGAL library) for computing the overlay of the two layers in the CCT dataset using one of the cluster nodes. Clearly, the scalable approach is much more efficient as it takes advantage of parallelism. Note that the CGAL library would crash when processing the larger datasets (MainUS and GADM).

Figure 2.20 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,



(a) (b)

Figure 2.19: SDCEL performance while varying the number of cells in the CCT dataset.
⟨fig:ca⟩



(a) (b)

Figure 2.20: Performance with (a) MainUS and (b) GADM datasets.
⟨fig:mainus⟩

Table 2.6: Cell size statistics.

⟨tab:cell_stats⟩

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 2.7: Orphan cells and orphan holes description

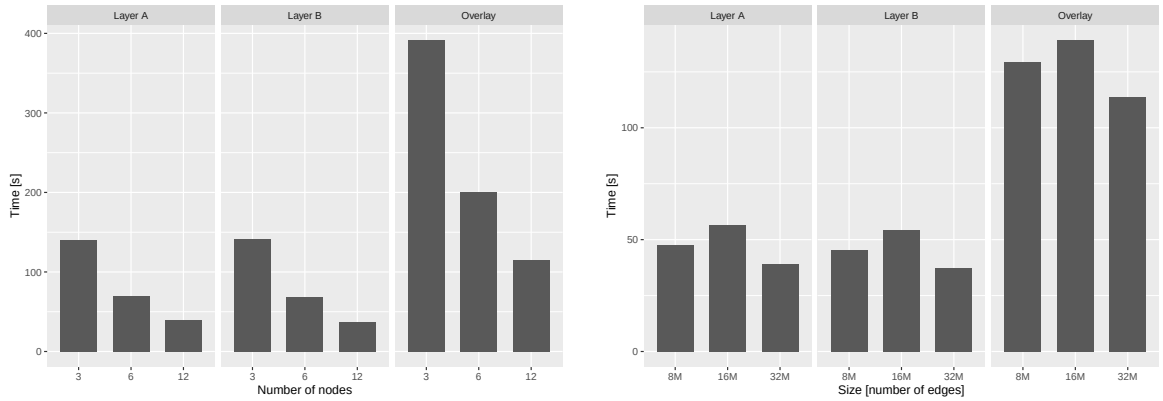
⟨tab:orphans⟩

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

the two layers have a similar number of edges; as 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 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.

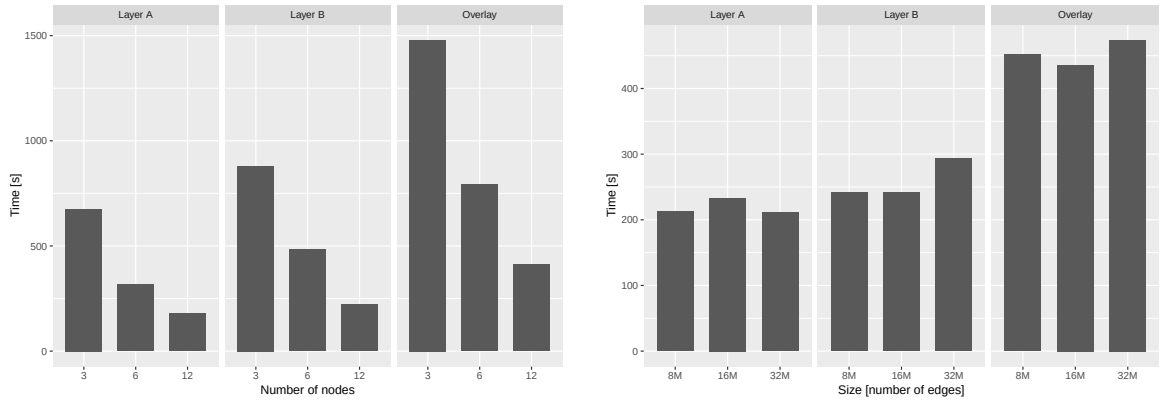
Additionally, Table 2.6 provides statistics on the cells. It shows that in larger datasets, an average cell size of approximately 3000 edges produces the best results. This cell size ensures a relatively small amount of data to transmit, which minimizes the impact on data shuffling and processing. Table 2.7 presents the number of cells, original holes, and the orphan cells and holes generated after partitioning.



(a) (b)

Figure 2.21: Speed-up and Scale-up experiments for the MainUS dataset.

⟨fig:mainus_speed_scale⟩



(a) (b)

Figure 2.22: Speed-up and Scale-up experiments for the GADM dataset.

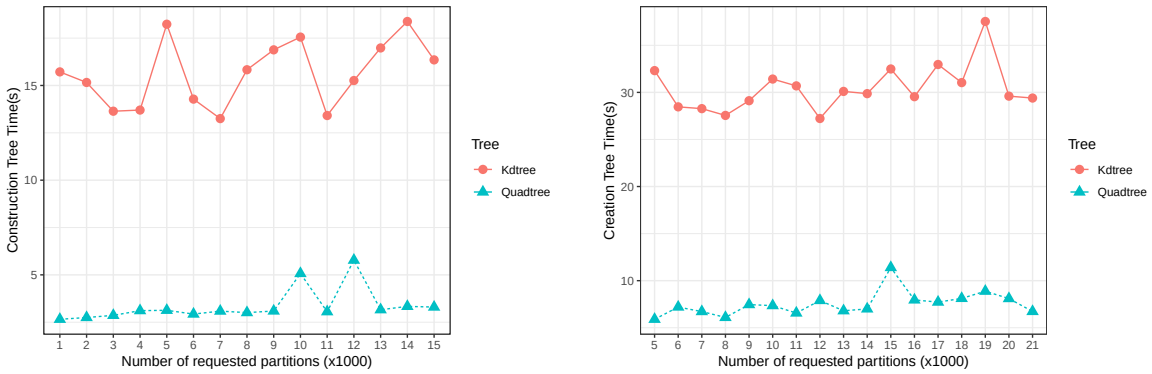
⟨fig:gadm_speed_scale⟩

2.7.4 Speed-up and Scale-up experiments

⟨sec:speed_scale⟩

The speed-up behavior of SDCEL appears in Figure 2.21(a) (for the MainUS dataset) and in Figure 2.22(a) (for the GADM dataset); in both cases, we show the performance 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



(a) (b)

Figure 2.23: Construction time for the spatial data structure in the (a) MainUS and (b) GADM datasets.

⟨fig:k_creation_us⟩

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 the scale-up experiments with the GADM dataset. The results appear in Figure 2.21(b) and Figure 2.22(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).

2.7.5 Kd-tree versus quadtree performance

⟨sec:comparison⟩

In order to compare the quadtree and the kd-tree partition strategies we analyze their performance during the construction of the spatial data structure which defines the cells that the partition will use based on the sample, the cost of partitioning; populating the cells with the full datasets, and the overall time to complete the phases of the overlay operation using each partitioning approach. We use the datasets of MainUS and GADM described in Table 3.1.

Figure 2.23 depicts the construction time during the sampling of the input layers and the generation of the partitioning cells after requesting a different number of divisions. We can see that the kd-tree takes more time, particularly because of the sorting done at each split, so as to organize the data and localize the middle point. In average, Quadtree

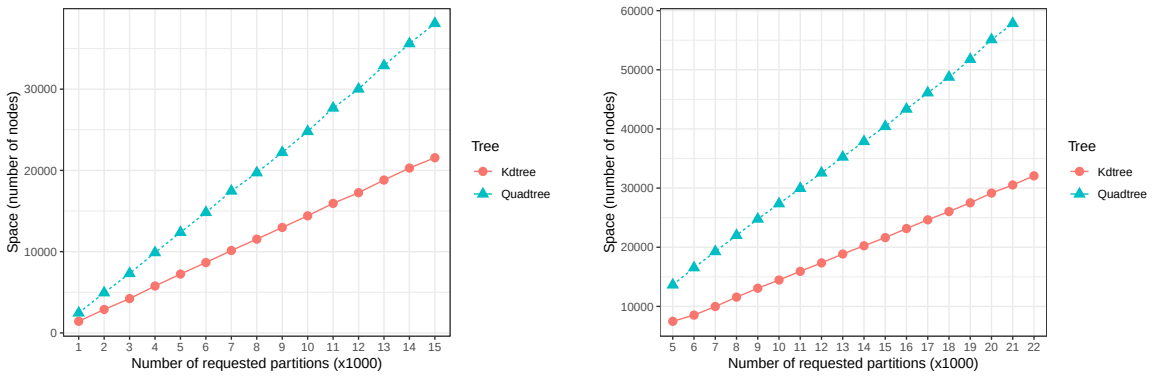


Figure 2.24: Number of cells created by each spatial data structure in the (a) MainUS and (b) GADM datasets.

⟨fig:k_space_us⟩

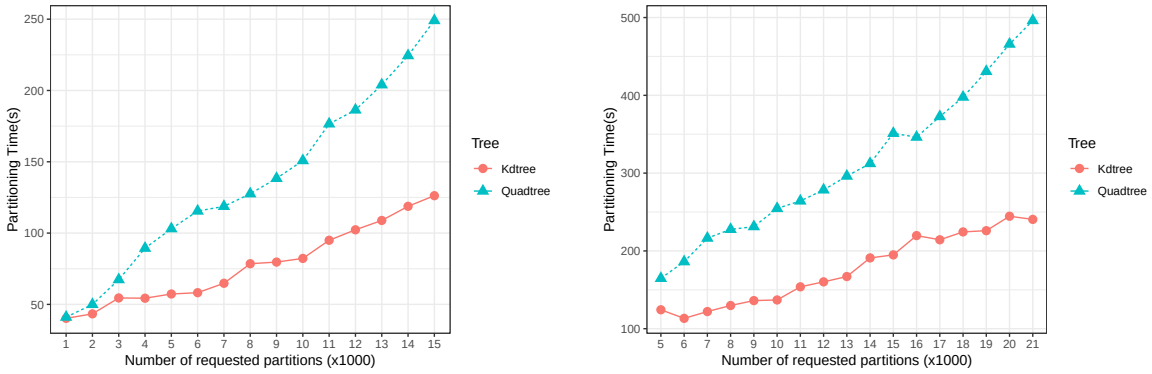
takes 23.13% the time it takes for Kdtree to be created (21.55% in MainUS and 24.72% in GADM). However, the Kdtree creation is just 5.86% of the overall time during the total DCEL construction (6.88% in MainUS and 4.87% in GADM).

An important characteristic of the behavior of each partitioning scheme is the number of cells (partitions) each sample data structure creates. Figure 2.24 depicts the number of cells created by each spatial data structure. As the quadtree follows a space-oriented technique, it creates more nodes (4 at each split) and thus generates more leaves (cells); more of them are prone to be empty compared to the kd-tree.

Figure 2.25 shows the cost to partition the full content of both layers. Given a sample tree data structure, each edge is assigned to a cell (partition) depending on which leaf the edge is located; edges are assigned (copied) to all leaves they intersect. Then, a shuffle operation is performed to move the data to the corresponding node that will handle this cell (partition). This figure shows that the quadtree partitioning takes more time. This depends largely on the number of leaves created by the sample tree and the number of edges that overlap partitions (which is expected to be larger for the quadtree since it uses more and thus smaller cells).

Once the data is assigned to their partitions, the overlay operation can be executed. Figure 2.26 shows the overlay performance under each partition strategy, for different number of cells. The Kd-tree approach performs better; as the quadtree tends to generate more and emptier cells, its performance is directly affected.

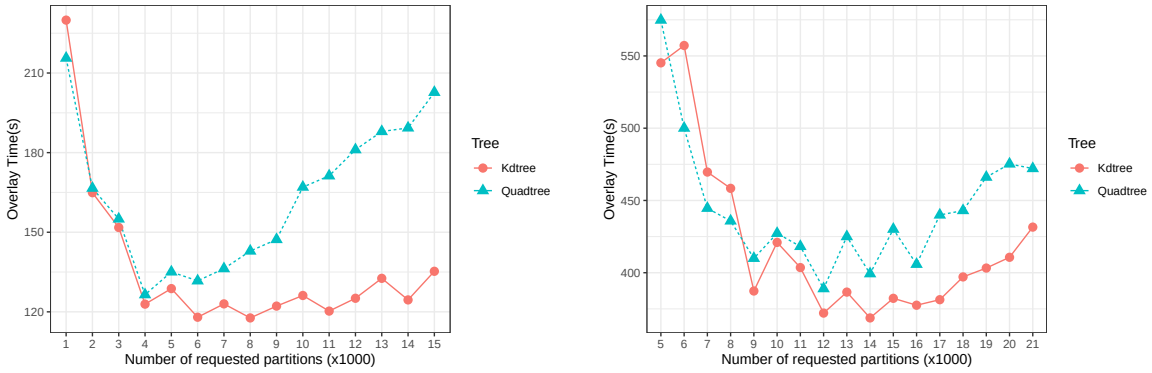
As it was said before, in particular on partitioning based on Kdtree, the smaller number of cells/partitions used in this approach give also an improvement on the impact



(a) (b)

Figure 2.25: Data partitioning time using a spatial data structure (a) in the MainUS dataset and (b) in the GADM dataset.

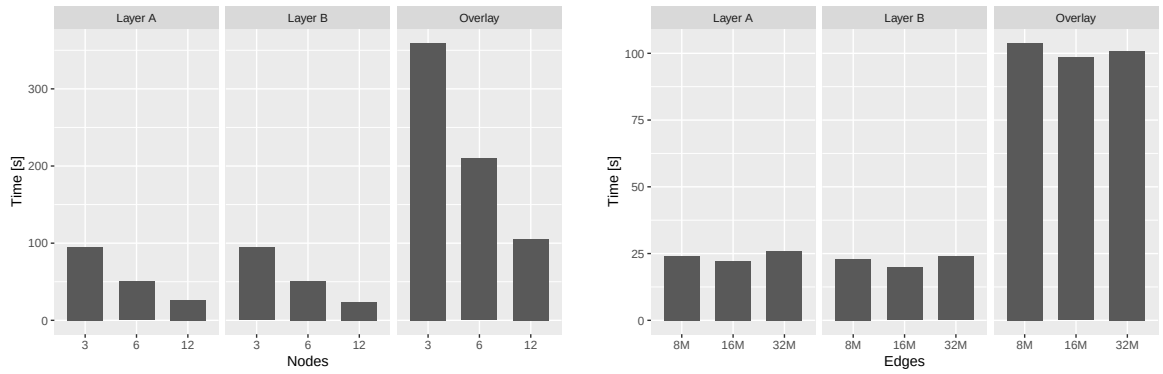
⟨fig:k_partitioning_us⟩



(a) (b)

Figure 2.26: Execution time for the overlay operation using a spatial data structure in the MainUS (a)and GADM (b) dataset.

⟨fig:k_overlay_us⟩



(a) (b)

Figure 2.27: (a)Speed Up and (b) Scale Up performance of the Kdtree partitioning using the MainUS dataset.

⟨fig:k_scale_speed_us⟩

of shuffling during the partition strategy because the number and size of the resulting partitions have a lower impact into the communication cost.

Finally, we consider the speed-up and scale-up performance using the kd-tree partitioning. Figure 2.27(a) shows the speed-up performance using the MainUS dataset (36M edges) while varying the number of nodes (for 3, 6, and 12 nodes). Similar to the quadtree partitioning strategy, the kd-tree partitioning shows good speed-up performance. As resources duplicate the execution time improves almost by a half.

Figure 2.27(b) shows the scale-up performance of the kd-tree partitioning approach. We followed the same procedure described in Section 2.7.4 to generate datasets for 8M, 16M, and 32M edges from the MainUS dataset and ran the kd-tree partitioning strategy with 3, 6, and 12 nodes, respectively. Again the kd-tree partitioning shows good speed-up performance, which remains flat as the load per node is almost equal.

2.7.6 Polygonization Scalability

?⟨sec:expr:query⟩?

Figure 2.28 evaluates the scalability of the polygonization approach using the different evaluation datasets summarized in Table 2.8. We implemented our polygonization framework on Apache Sedona [69]. The experiment is based on a Java 8 implementation and utilizes a Spark 2.3 cluster with two driver nodes and 12 worker nodes. All nodes run Linux CentOS 8.2 (64-bit). Each driver node is equipped with 128GB of RAM, while each worker node has 64GB of RAM. To increase parallelism, we divided the 12 worker nodes into 84 worker executors. Each executor is a separate JVM process with dedicated

Table 2.8: Polygonization Evaluation Dataset

⟨olygonization:datasets⟩

Dataset	Area	Number of Line Segments	Faces
USA	9.83 Mkm^2	152M	5M
South America	17.8 Mkm^2	155M	7M
North America	24.7 Mkm^2	240M	10M
Africa	30.4 Mkm^2	288M	10M
Europe	10.2 Mkm^2	563M	25M
Asia	44.6 Mkm^2	557M	23M

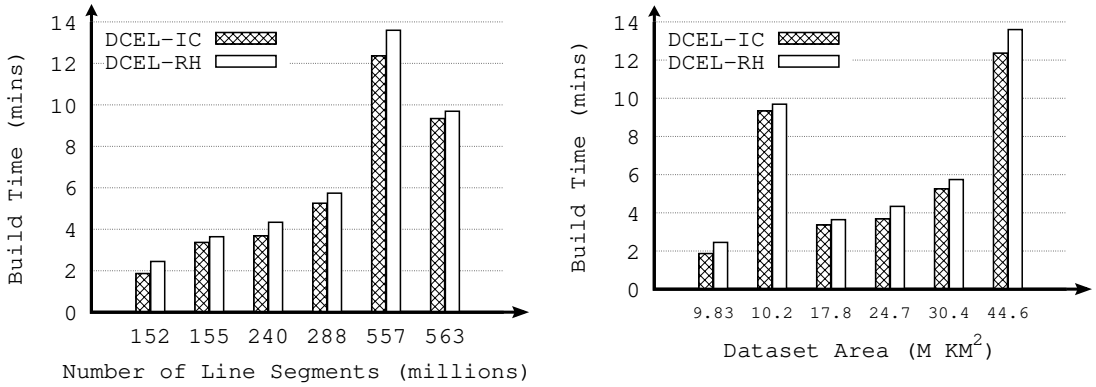


Figure 2.28: Polygonization Performance on Real Road Networks.

⟨fig:exp:query⟩

resources, such as memory and CPU cores. The distribution of these executors across the nodes is managed by the resource negotiator (YARN), which allocates resources for Spark jobs based on the availability of cores and memory. YARN typically balances resources across the cluster, so executors are likely to be evenly distributed, though some variation may occur due to resource availability at runtime. Assuming an even distribution, each worker node would run approximately 7 executors, as calculated by $\frac{84}{12} = 7$.

As discussed in Section 2.6.2, the Rem Phase has two different approaches depending on the input data received from the Gen Phase. The first approach is to process the remaining half-edges iteratively, denoted as *DCEL-RH*. In comparison, the second approach processes the incomplete cycles generated from the first phase iteratively denoted as *DCEL-IC*.

From Figure 2.28, we draw three conclusions; (1) first, the cardinality of the input dataset has a positive correlation with the build time; as the number of line segments increases, the build time also increases, as shown in Figure 2.28(a). However, we see that we have close cardinality for Asia (557M) and Europe (563M) datasets, but there is a noticeable difference in the build time; moreover, the build time for the Europe dataset is less than that of the Asia dataset. This drives us to the second conclusion; (2) for datasets with close or similar cardinalities, the area of the dataset has a positive correlation with the build time shown in Figure 2.28(b). Hence the build time of the Europe dataset (10.2 Mkm^2) is less than that of the Asia dataset (44.6 Mkm^2), even though Europe has a slightly larger dataset. (3) The third conclusion is that for all evaluated datasets, the *DCEL-IC* beats *DCEL-RH*.

2.7.7 Polygonization Speed Up Evaluation

?⟨sec:expr:speedup⟩?

Figure 2.29 shows the effect of increasing the number of executors on the build time for the USA dataset. At each step in the figure, we add 7 more executors, which is approximately equivalent to adding one additional node. Overall, our approach has good speedup performance. As the number of executors is doubled from 7 executors to 14 executors, the build time is almost halved. This trend goes on as we double the number of executors. As we increase the number of executors from 7 to 84, the build time is decreased by a factor of 8.

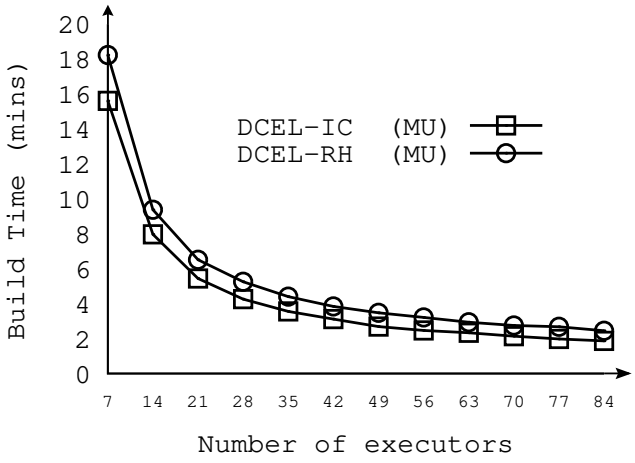


Figure 2.29: Polygonization speed up evaluation using the USA dataset

⟨fig:exp:speedup⟩

Table 2.9: Overlaying Polygons with Dangle and Cut Edges Dataset

⟨tab:dangles⟩

Dataset	Number Layer A of Polygons	Number of Layer B Edges	Result Polygons
TN	1,272	3,380,780	41,761
GA	1,633	4,647,171	49,125
NC	1,272	7,212,604	22,413
TX	4,399	8,682,950	98,635
VA	1,554	8,977,361	38,941
CA	7,038	9,103,610	96,916

2.7.8 Overlaying Polygons with Dangle and Cut Edges

In this section, we examine the performance of overlaying polygons with dangle and cut edges resulting from the polygonization as detailed in Section 2.6.4. Table 2.9 shows the number of polygons for each state for the first layer of the overlay. It also shows the number of dangle and cut edges per state for the second layer of the overlay. Finally, it shows the number of resultant polygons per state. From Figure 2.30, we conclude that the running time is affected by the number of dangle and cut edges and the number of intersections between the two layers (represented by the number of generated polygons). TN and GA have a relatively smaller number of dangle and cut edges, so they have lower execution times compared to VA, TX, and CA. However, since the intersections in NC are significantly less than those of TN and GA, NC has the lowest execution time. TX, VA, and CA have a comparable number of edges; however, VA has the least number of intersections, resulting in lower execution time compared to TX and CA.

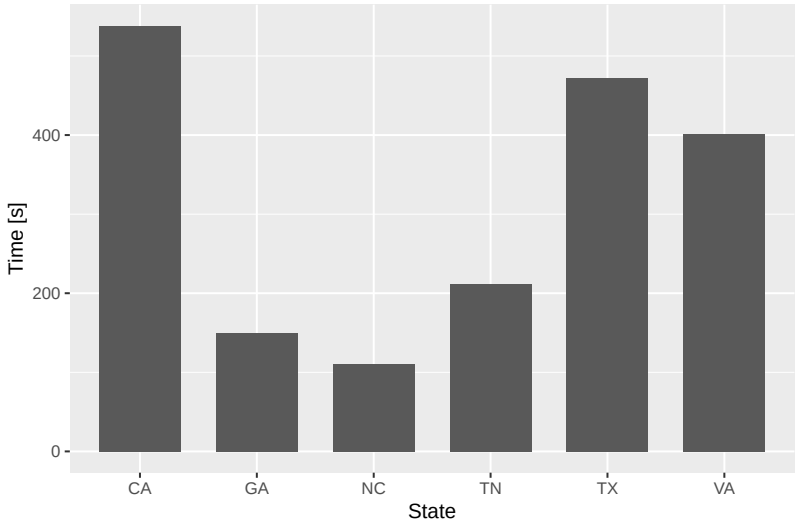


Figure 2.30: Overlaying State polygons with dangle and cut edges.

⟨fig:dangle⟩

2.8 Conclusions

⟨sec: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. We support input polygons in clean polygon format and polygons represented by scattered line segments through scalable polygonization. Existing sequential DCEL overlay implementations fail for large datasets. We first presented two partition strategies that guarantee 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 a few seconds.

Chapter 3

Scalable Processing of Moving Flock Patterns

3.1 Introduction

Technological advances in the past few decades have triggered an explosion in the collection of spatio-temporal data. The increasing popularity of GPS devices and smartphones, along with the emergence of new disciplines such as the Internet of Things (IoT) and high-resolution Satellite/UAS imagery, has made it possible to collect vast amounts of data with spatial and temporal components.

In tandem, interest in extracting valuable information from such large databases has also grown. Spatio-temporal queries about popular places or frequent events remain useful, but there has been growing interest in more complex patterns. In particular, patterns that describe the group behavior of moving objects over significant periods. Moving cluster [40], convoys [38], flocks [29] and swarm patterns [45] reveal how entities move together over a minimum time interval.

Applications for this type of information are both diverse and intriguing, particularly when dealing with trajectory datasets [37, 35]. Case studies span various domains, including transportation system management and urban planning [18], as well as ecology [42]. For example, [61] explores the identification of complex motion patterns to discover similarities in tropical cyclone paths. Similarly, [3] investigates eye movement trajectories to understand the strategies people use during visual searches. Additionally, [33] tracks the behavior of tiger sharks along the coasts of Hawaii to gain insight into their migration patterns.

One particular pattern of interest is the moving flock pattern, which captures how objects move within close proximity for a given time period. Closeness is defined by

a disk of a specified radius within which the entities must remain. Since this disk can be positioned anywhere, detecting such patterns is a non-trivial problem. In fact, [29] highlights that finding flock patterns where the same entities stay together over time is an NP-hard problem. To address this, [63] proposed the BFE algorithm, the first approach capable of detecting flock patterns in polynomial time.

Despite the increasing availability of data, current state-of-the-art techniques for mining complex movement patterns still struggle with the performance demands of large-scale spatial data. This work introduces a scalable approach designed to detect moving flock patterns in very large trajectory databases. By leveraging emerging trends in distributed frameworks for spatial operations we aim to significantly improve the speed and efficiency of detecting these patterns.

3.2 Related work

The recent increased use of location-aware devices (such as GPS, smartphones, and RFID tags) has enabled the collection of vast amounts of data with spatial and temporal components. Several studies have focused on discovering and analyzing these types of datasets [43, 48]. In this area, trajectory datasets have emerged as an interesting field where diverse kind of patterns can be identified [70, 64]. For instance, researchers have proposed techniques to discover spatial motion patterns such as moving clusters [40], convoys [38] and flocks [7, 29]. Specifically, [63] introduced BFE (Basic Flock Evaluation), an innovative algorithm designed to efficiently identify moving flock patterns in polynomial time across large spatio-temporal datasets.

A flock pattern is defined as a group of entities that move together over a specified time period [7]. The applications of such patterns are broad and diverse. For instance, [13] identifies moving flock patterns in iceberg trajectories to analyze their movement behavior and their relationship with changes in ocean currents.

The BFE algorithm provides an initial approach for detecting flock patterns. It begins by identifying disks with a predefined diameter (ε) where moving entities are sufficiently close at specific time instants. This operation is computationally expensive due to the large number of points and time instances to be analyzed, with a complexity of $\mathcal{O}(2n^2)$ per time. Although the algorithm leverages a grid-based index and a stencil to accelerate this process, the overall complexity remains high.

Both [13] and [61] adopt a frequent pattern mining approach to enhance performance when combining disks across time instants. Similarly, [59] utilize plane sweeping techniques, binary signatures, and inverted indexes to further accelerate this process. However, these methods retain the core strategy of BFE for detecting disks at each time instant.

In contrast, [4] and [27] employ depth-first algorithms to analyze the time intervals of individual trajectories and report maximal duration flocks. However, these methods are less effective for dense datasets or those that involve large numbers of entities per time step, as they struggle to scale efficiently in such conditions.

Given the high computational demands of flock pattern detection, it is not surprising that parallelism has been employed to improve performance. For example, [24] use extreme and intersection sets to report maximal, longest, and largest flocks on GPUs, albeit with limitations imposed by the GPU's memory model.

Despite the increasing adoption of cluster computing frameworks, particularly those with spatial data capabilities [19, 68, 36, 66], significant advancements in this area remain limited. To the best of our knowledge, this work is the first to explore the detection of moving flock patterns in a scalable approach.

3.3 Background

Before discussing the details of our contributions, we first provide an overview of the current state-of-the-art. This will help highlighting their challenges and limitations in handling large spatio-temporal datasets, as described in the next Section.

3.3.1 The BFE sequential algorithm

The alternative approach we will discuss closely follows the steps outlined in [63]. In that work, the authors introduced the Basic Flock Evaluation (BFE) algorithm, designed to identify flock patterns in trajectory databases. While the full details of the algorithm can be found in the source, we will provide a general overview of the key aspects. It is important to note that the BFE algorithm operates in two phases: first, it identifies maximal disks at the current time step; second, it extends and reports previous flocks by combining them with the newly discovered disks.

The input for the BFE algorithm consists of a set of points, a minimum distance ε (which defines the diameter of the disks where the moving entities must lie), a minimum number of entities μ per disk, and a minimum duration δ, representing the required time units that entities must remain together to be considered a flock. Based on this input, Figure 3.1 illustrates the workflow of the process in four general steps. The primary goal of this phase is to identify a set of disks at each time step, enabling the combination with future disks to form flocks.

The main steps in phase 1 follow:

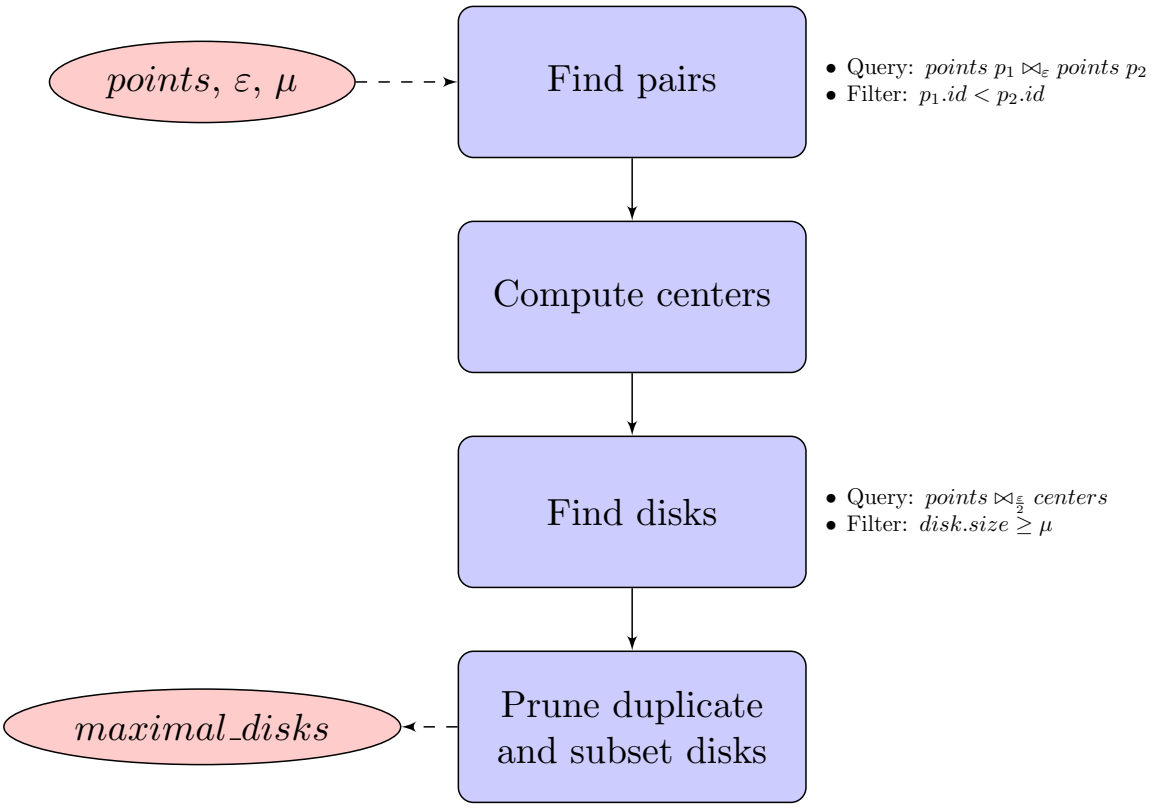


Figure 3.1: General steps in phase 1 of the sequential algorithm.

⟨fig:MF_flowchart⟩

1. Pair finding: The algorithm uses the parameter ε to identify pairs of points that are within a maximum distance of ε units from each other. This is achieved through a distance self-join operation on the set of points, using ε as the distance threshold. To avoid redundancy, duplicate pairs are eliminated; for example, the pair (p_1, p_2) is considered identical to the pair (p_2, p_1), so only one instance is retained. Point IDs are used to filter out these duplicates efficiently.

2. Center computation: From the set of pairs obtained, each pair is used to compute the centers of two circles, each with a radius of $\frac{\varepsilon}{2}$, whose circumferences pass through the two points in the pair. The pseudo-code for this procedure is provided in Appendix 3.

3. Disk finding: Once the centers have been identified, a query is executed to gather the points within a distance of ε units from each center. This is accomplished by performing a distance join between the set of points and the set of centers, using $\frac{\varepsilon}{2}$ as the distance parameter. As a result, each disk is defined by its center and the IDs of the surrounding points. At this stage, a filter is applied to discard any disks that contain fewer than μ entities.

4. Disk pruning: It is possible for a disk to contain the same set, or a subset, of points as another disk. In such cases, the algorithm reports only that one disk which contains the other(s), referred to as the *maximal* disk. The procedure for identifying maximal disks is explained in Appendix 4.

It is important to note that BFE also employs a grid structure in this phase to optimize spatial operations. The algorithm divides the space into a grid, where each cell has a side length of ε (see Figure 3.2 from [63]). This structure allows BFE to limit its processing to each grid cell and its eight neighboring cells. There is no need to query cells beyond this neighborhood, as points in more distant cells are too far away to influence the results.

Figure 3.3 explains schematically phase 2. This phase performs a recursion using the set of disks found at time i and the set of partial flocks computed at the previous time instant $i-1$. As we do not know where and how far a group of points can move in the next time instant, this step performs a (temporal) join between both sets (partial flocks computed at time $i-1$ and maximal disks found in time i). When a join is performed, we check that the number of common points remains greater than μ, in which case the partial flock extends in time. A flock is reported in the answer if its duration has reached the minimum duration δ; otherwise, it remains as partial flock and it will be further evaluated during the next iteration at the next time instant.

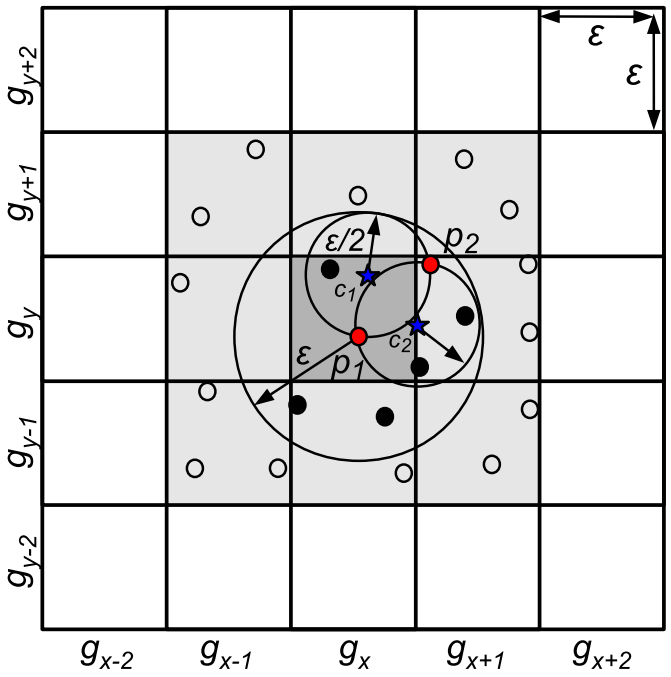


Figure 3.2: The grid-based structure proposed in [63].

⟨fig:grid⟩

Figure 3.3 provides a schematic representation of phase 2. In this phase, the algorithm proceeds recursively, utilizing the set of disks identified at time step i in conjunction with the partial flocks that were computed at the previous time step $i-1$. Since the movement and displacement of a group of points in the next time instant is unpredictable, this step performs a temporal join between the two sets—the partial flocks from time $i-1$ and the maximal disks found at time i—to determine possible continuations. During this join operation, the algorithm checks whether the number of common points between a partial flock and a new disk exceeds the threshold μ; if this condition is met, the partial flock is extended in time. A flock is only reported in the final results once its duration meets the minimum duration δ; otherwise, it remains classified as a partial flock, and it will continue to be evaluated in subsequent iterations as time progresses to the next instant.

Similarly, Figure 3.5 illustrates the recursive process and how the set of partial flocks from previous time instants feeds into the next iteration. The example assumes a δ value of 3, meaning flocks start being reported from time instant t_2. Note that time instants t_0 and t_1 are considered the initial conditions. At the start of the algorithm, maximal disks are identified at t_0, which are immediately transformed into partial flocks with a duration of 1 and then passed on to the next time instant. At t_1, a new set of maximal disks $\mathcal{D}_1$ is found and joined with the partial flocks from t_0, denoted as $\mathcal{F}_0$. The information for each

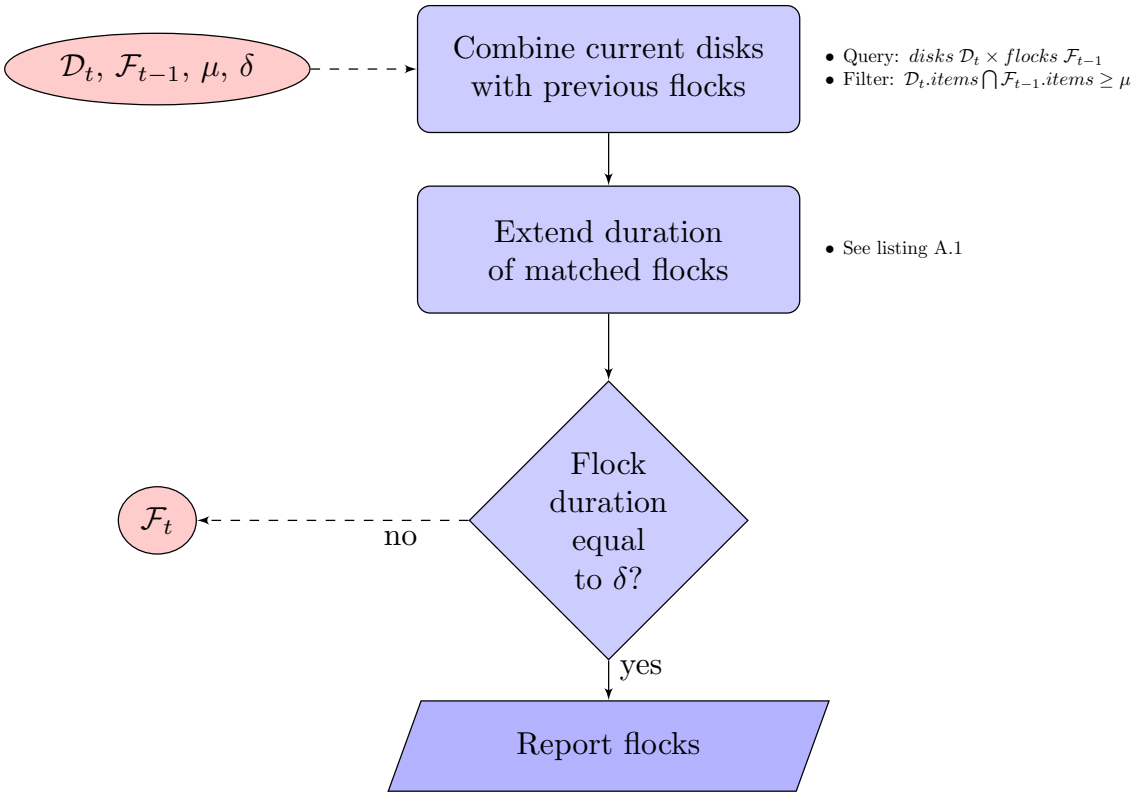


Figure 3.3: Steps in BFE phase two. Combination, extension and reporting of flocks.

⟨fig:FF_flowchart⟩

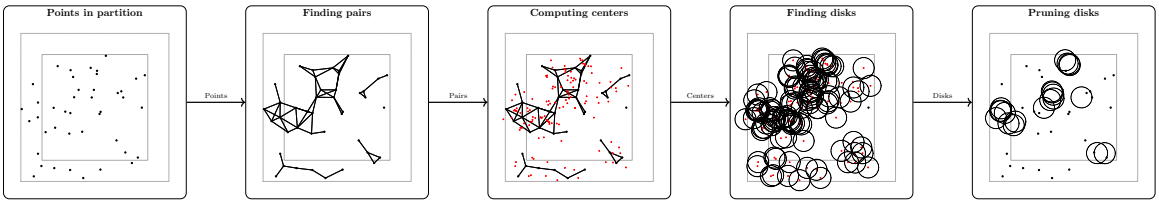


Figure 3.4: Example of BFE execution on a sample dataset.

⟨fig:example⟩

partial flock is updated accordingly, including its duration and the points it contains. From this point onward, subsequent time instants follow the exact steps outlined in Figure 3.3.

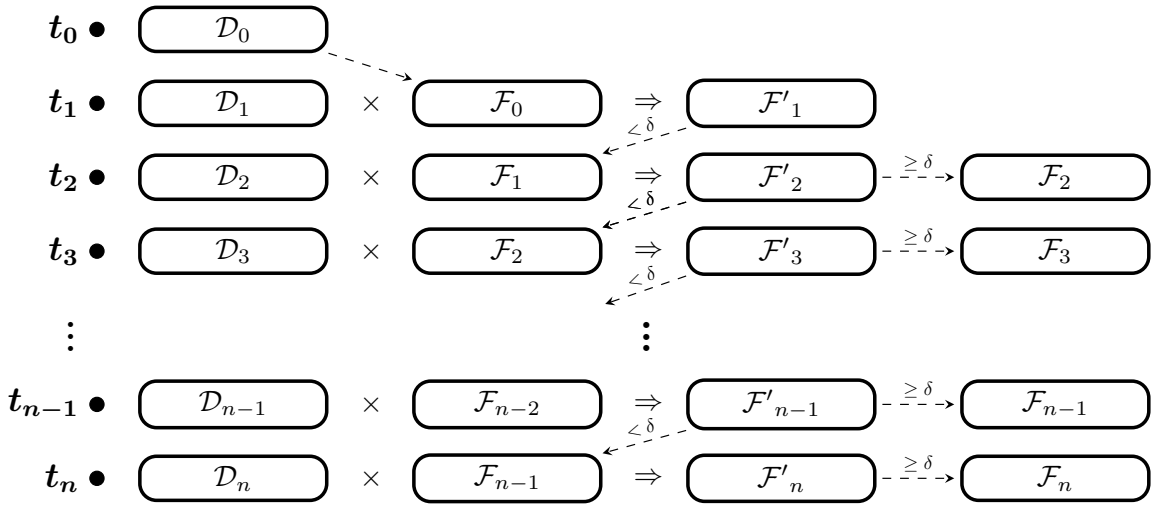


Figure 3.5: BFE phase 2 example explaining the recursion of the set of flocks along time instants and the initial conditions.

⟨fig:FF_stages⟩

3.3.2 The PSI sequential algorithm

The PSI algorithm, proposed by [59], follows a similar process to the BFE algorithm. However, instead of using a grid structure to index points within the area, PSI employs a sweep-line approach that processes points in order of their x-coordinates. For each visited point p, the algorithm considers a square of side length 2ε centered at p. It only examines the points to the right of p that lie within two half-squares of side length ε, as illustrated by the shaded regions in Figure 3.6.

While BFE processes points inside a grid cell of side length ε along with its eight neighboring cells, PSI focuses on the points in these two half-squares. As a result, PSI more efficiently identifies the points relevant for detecting candidate pairs, centers, and disks.

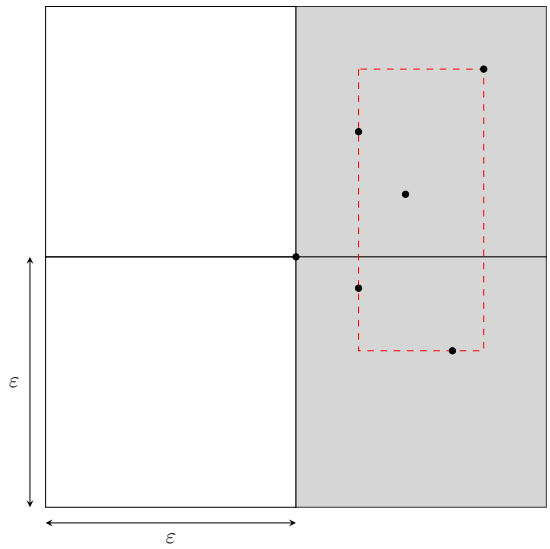


Figure 3.6: An example of the two half squares used in PSI algorithm.

⟨fig:square⟩

This indexing method has been shown to outperform BFE in most cases, with BFE offering similar or better performance only when ε values are relatively small. In such cases, the number of points to consider is smaller, and PSI still requires sorting the points for the sweep-line approach. Therefore, both approaches are considered in the following sections.

3.4 Bottlenecks in the sequential approach and proposed solutions

?⟨spatial_phase⟩?

Since both sequential approaches follow the same steps (as shown in Figures 3.1 and 3.3), we will focus on discussing their bottlenecks using the BFE as an example. Certain stages in the BFE process are notably impacted when handling very large datasets, which can significantly affect performance.

3.4.1 Phase 1: Spatial finding of maximal disks.

First, we focus on Phase 1. As illustrated in Figure 3.4, this phase's steps are demonstrated using a sample dataset. It is important to note that the final set of maximal disks is significantly smaller than the initial number of candidate disks found. Specifically, the number of candidate centers to evaluate is $2|\tau|^2$, where τ represents the number of trajectories [63]. Our experiments reveal that this issue becomes more pronounced not only

in very large datasets but also in those containing areas with a high density of moving entities.

To address this issue, we propose a partition-based strategy that divides the study area into smaller subareas, allowing for independent and parallel evaluation. The strategy consists of three key steps: first, the *partition and replication* stage, followed by the *local flock discovery* within each partition, and finally, the *filtering stage*, where we consolidate and unify the results. Each of these steps is detailed below.

- Partition and Replication: Figure 3.7 provides a brief example of the partition and replication stage. Different types of spatial indexes, such as grids, R-trees, or quadtrees, can be used to create spatial partitions of the input dataset. In the example shown in Figure 3.7.b, we use a quadtree, which generates seven partitions. To ensure each partition can locally identify flocks, it must have access to all relevant data. This is achieved by replicating points that are within a distance of ε from the border of each partition, an area referred to as the *expansion zone*, into adjacent partitions. Figure 3.7.c illustrates each partition, surrounded by a dotted line representing the expansion zone, which includes the points that need to be replicated from neighboring partitions.

- Local flock discovery: At this stage, each partition can be processed independently and in parallel, with partitions assigned to different processing nodes. Within each partition, we can execute the steps of Phase 1 of the BFE algorithm locally, as outlined in Figure 3.1.

- Filtering: While partitioning and replication facilitate parallelism, they can also lead to result duplication, as different nodes may report the same maximal disk. Specifically, if a disk's center lies within a partition, it will be reported only once by the node processing that partition. However, disks with centers located in an expansion zone will be reported by all partitions that share that zone. To address this, we propose a reporting approach that effectively prevents such duplication, which we detail below.

Disks with centers in an expansion zone are created by points that exist in both partitions due to replication. We assert that each partition should only report disks generated within its own area and not those originating in its expansion zone. Figure 3.8 illustrates the possible scenarios. Assume partitions 1 and 2 in the figure are contiguous, sharing edge AB. Consider the disks a' and a'' (each with a diameter of ε), which are generated by two points (shown in green) located in the expansion zone of partition 1 but inside partition 2. In this case, both a' and a'' will be reported by partition 2. Similarly, both c'

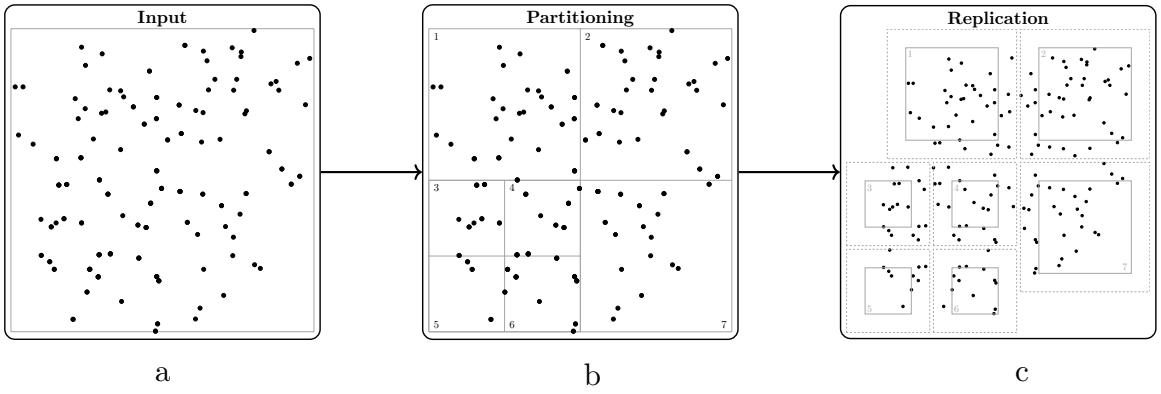


Figure 3.7: An example of partitioning and replication on a sample dataset.

⟨fig:partrep⟩

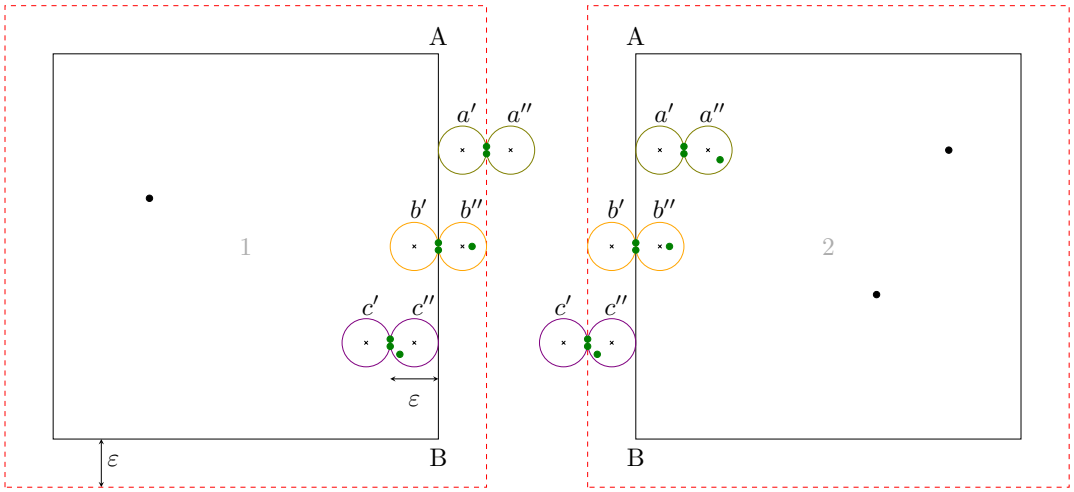


Figure 3.8: Ensuring no loss of data in safe zone and expansion area.

⟨fig:ensuring⟩

and c'' will be reported by partition 1. However, b' will be reported by partition 1, while b'' will be reported by partition 2.

3.4.2 Phase 2: Temporal join

⟨sec:temporal_join⟩

At the end of Phase 1, we have computed a set of maximal disks for a given time instant. In Phase 2, we proceed by combining these disks over time instants to form flocks. However, since Phase 1 involved partitioning the spatial domain for parallelism, Phase 2 becomes more complex as flocks can move across spatial partitions over time. Once the maximal disks are identified for a time instant i, a temporal join occurs within each partition to link these disks with partial flocks from the previous time instant $(i-1)$. However, we must account for partial flocks that may appear near the partition borders and potentially move into adjacent partitions (see Figure 3.10).

To address this issue, we introduce an additional parameter, *maxdist*, which represents the maximum distance a moving object can travel between consecutive time instants. We define the *safe area* of a partition as the internal region that is at least *maxdist* away from the partition's border (illustrated in grey in Figure 3.9). Any partial or full flocks discovered within a partition's safe area can be directly reported as results. However, flocks that start or end outside the safe area must be collected for post-processing to determine if they correspond with partial flocks from neighboring partitions. These cases, where flocks cross between partitions, are referred to as *crossing partial* flocks (CPFs).

In the post-processing stage, we evaluate four alternatives for collecting and checking crossing partial flocks (CPFs). The simplest approach is to gather all CPFs and process them sequentially on a single node (the master node). However, due to the large number of partitions and the *maxdist* parameter, the volume of CPFs requiring post-processing can become substantial, leading to a bottleneck that negatively impacts overall performance.

We also evaluate an intermediate approach where the CPFs from a given partition are sent to a middle-level node for processing, based on the quadtree structure used to create the partitions. The choice of which middle-level node to send the CPFs to is determined by a user-defined parameter called *step*. A value of *step=1* corresponds to sending CPFs to the immediate parent, *step=2* to the grandparent, and so on, until the root is reached. For example, with *step=1*, all CPFs from a partition are first sent to its parent node in the quadtree. The parent node processes its CPFs, but some flocks may still cross outside the parent's safe area. These leftover CPFs are then passed to the next parent (since *step=1*), and this process continues until all CPFs are processed, potentially reaching the root node. This approach allows for parallelism in post-processing, as moving to a parent node increases

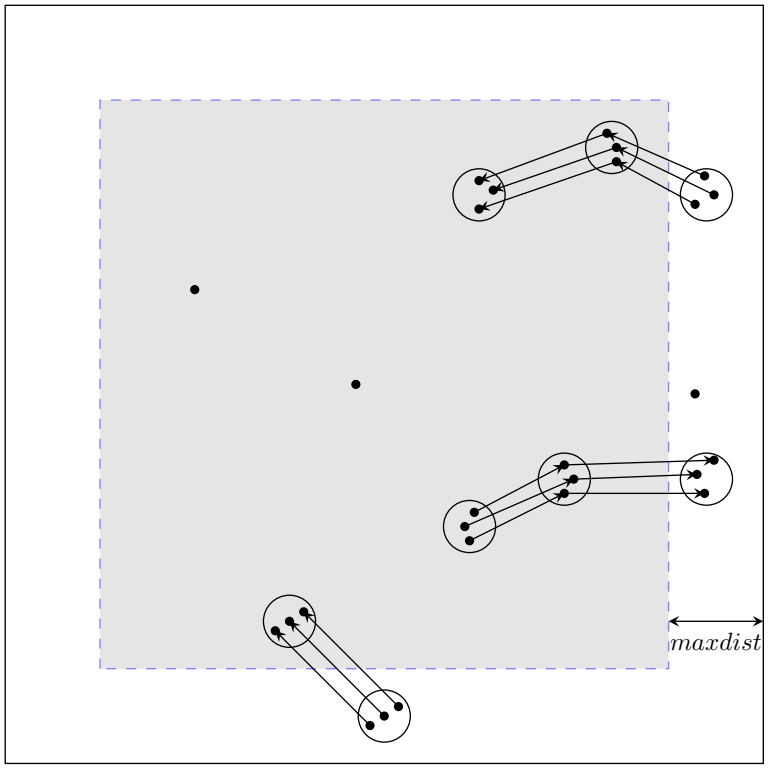


Figure 3.9: Examples of CPFs that that start or end in the border area of a partition.
⟨fig:maxdist⟩

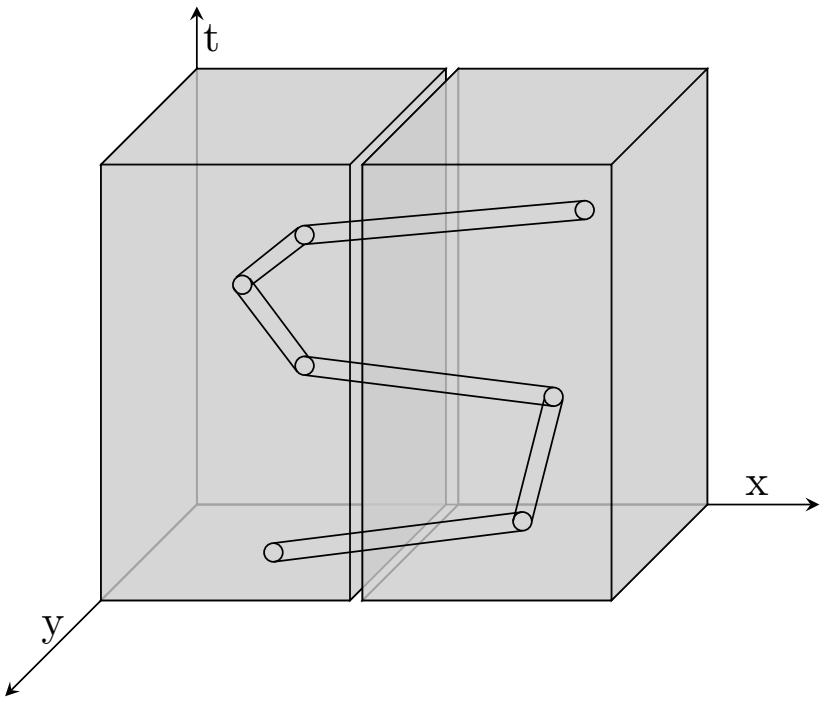


Figure 3.10: A flock that moves in different partitions along the time.

⟨fig:simple_alternative⟩

the partition's area and improves the likelihood that CPFs can be resolved at that level. In the experimental section, we test different values of the *step* parameter, such as *step=2*, where CPFs are sent to the grandparent at each stage.

Unlike the previous two approaches, which assign all CPFs from a given partition in the same way (partition-based), the third alternative assigns each CPF individually (CPF-based). For a given CPF f, we extend its most recent disk by a ring with a size of *maxdist*, identifying all overlapping partitions for this extended disk—essentially determining which neighboring partitions the objects in f could move to in the next time instant. For each overlapping partition, we retrieve the Least Common Ancestor (LCA) between that partition and f's original partition. CPF f is then sent to the node(s) corresponding to these LCAs. The benefit of this approach is that the LCA can efficiently complete the processing for f, as it exploits proximity using *mindist*. However, the downside is the increased copying overhead, as f may need to be sent to multiple nodes.

A limitation of the previous alternatives is that each spatial partition is processed by a single node, which incrementally evaluates all time instances for that partition. The fourth alternative introduces fixed divisions in the temporal domain, based on a user-defined parameter (number of divisions), as illustrated in Figure 3.11. In this approach, the spatio-temporal space is divided into temporal 'cubes,' each of which can be processed by different nodes. For simplicity, we assume that each division spans the same length of time. However, an additional validation step is required to ensure continuity of flocks across temporal divisions.

3.5 Experimental Evaluation

3.5.1 Experimental Setup

For our experiments, we utilized a 12-node cluster, each running Linux (kernel version 3.10) and Apache Spark 2.4. Each node was equipped with 8 cores, providing a total of 96 cores across the cluster. Each core operated with an Intel Xeon CPU at 1.70 GHz, and each node had 4 GB of main memory.

To evaluate the different approaches, we generated three synthetic datasets with varying characteristics, as detailed in Table 3.1. These datasets were created using the SUMO simulator [41], by importing traffic networks of Berlin and Los Angeles from OpenStreetMap [31]. We configured SUMO for pedestrian traffic and generated datasets of 10K, 25K, and 50K pedestrian trajectories. The total duration of the trajectories was set to 10, 30, and 60 minutes, respectively, with positions of pedestrians recorded at one-minute

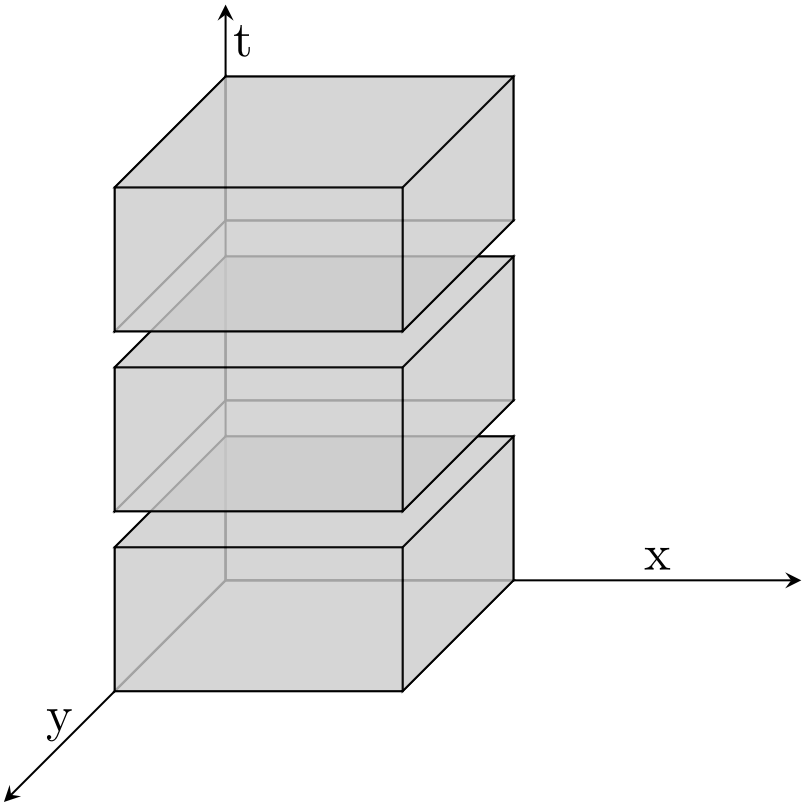


Figure 3.11: An alternative division on the time dimension to partition the data into cubes.
⟨fig:cube_alternative⟩

Table 3.1: Description of datasets

⟨tab:datasets⟩

Dataset	Number of Trajectories	Total number of points	Maximum Duration (min)
Berlin10K	10000	97526	10
LA25K	25000	1495637	30
LA50K	50000	2993517	60

intervals.

For the partitioning phase, we employed a quadtree structure, though other indexing methods could also be used. The advantage of using a quadtree is its ability to create nodes that tend to have a similar number of objects. The input to this phase is a set of points in the format *(traj-id, x, y, t)*. To construct the quadtree, we begin by sampling 1% of the input data and inserting this subset into an initially empty quadtree.

A key parameter for the quadtree is the node capacity, denoted as c. When the number of points in a node exceeds this capacity, the node splits. After all the sampled points are inserted, we use the Minimum Bounding Rectangles (MBRs) of the leaf nodes as the partitions for our approach. The remaining points are then inserted into these fixed partitions, with no further splits occurring. Each partition is assigned to a different cluster node, where a sequential version of either BFE or PSI is executed locally on the points within that partition.

3.5.2 Optimizing the number of partitions for Phase 1.

The capacity parameter c directly influences the number of partitions in the quadtree. A smaller value of c results in a higher number of partitions, which leads to many smaller tasks that can be distributed across the cluster. However, this can increase the overhead associated with data transmission and, potentially, replication, which may become a bottleneck. Conversely, a larger value of c reduces the number of partitions, resulting in fewer but larger tasks. This increases the workload of the sequential algorithm within each partition, potentially extending the response time for individual jobs.

Figure 3.12 presents the execution time (in seconds) for computing maximal disks (Phase 1) at a specific time instant, using different values of c and ε. The experiments were conducted using the LA25K dataset. For the case where $\varepsilon = 20m$, we observe that there is an optimal value of c that minimizes the execution time for finding maximal disks, which occurs at $c = 100$ (corresponding to approximately 1300 partitions). Additionally, the optimal value of c varies based on the value of ε. For instance, with a smaller $\varepsilon = 2m$, the execution time is minimized at a larger capacity $c = 500$ (around 250 partitions). When ε is large, more pairs of points need to be processed, resulting in a higher number of maximal disks to compute. In such cases, using a smaller value of c creates more partitions within the same spatial area, thereby distributing the workload more evenly across partitions and reducing the amount of work per partition.

After determining the optimal value of c for a given ε, we further analyzed the behavior of BFE and PSI on the most 'demanding' partitions, those that required the longest time to complete Phase 1. Since the partitions are processed in parallel across different

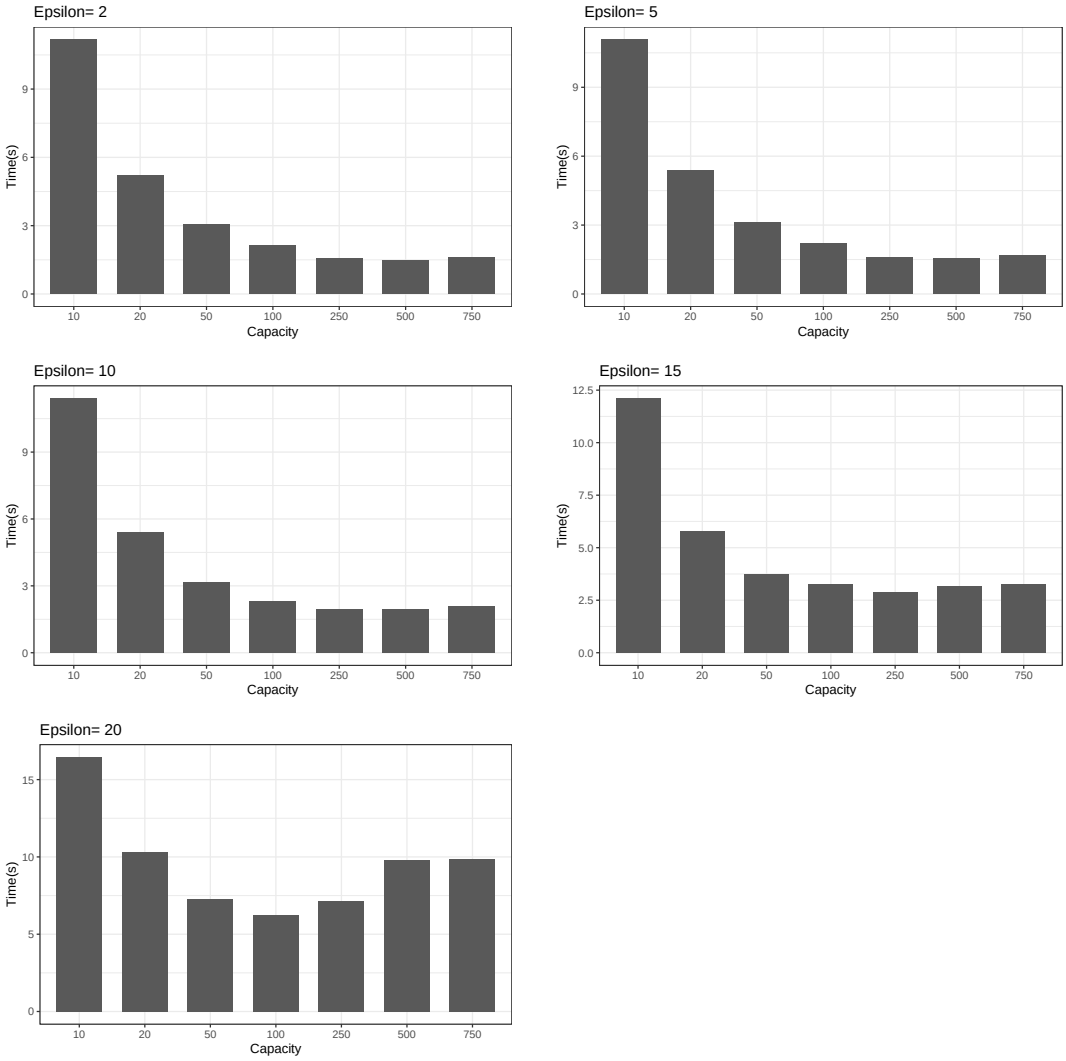


Figure 3.12: Execution time testing different values for Capacity (c) and Epsilon ε).

ig:optimal_performance⟩

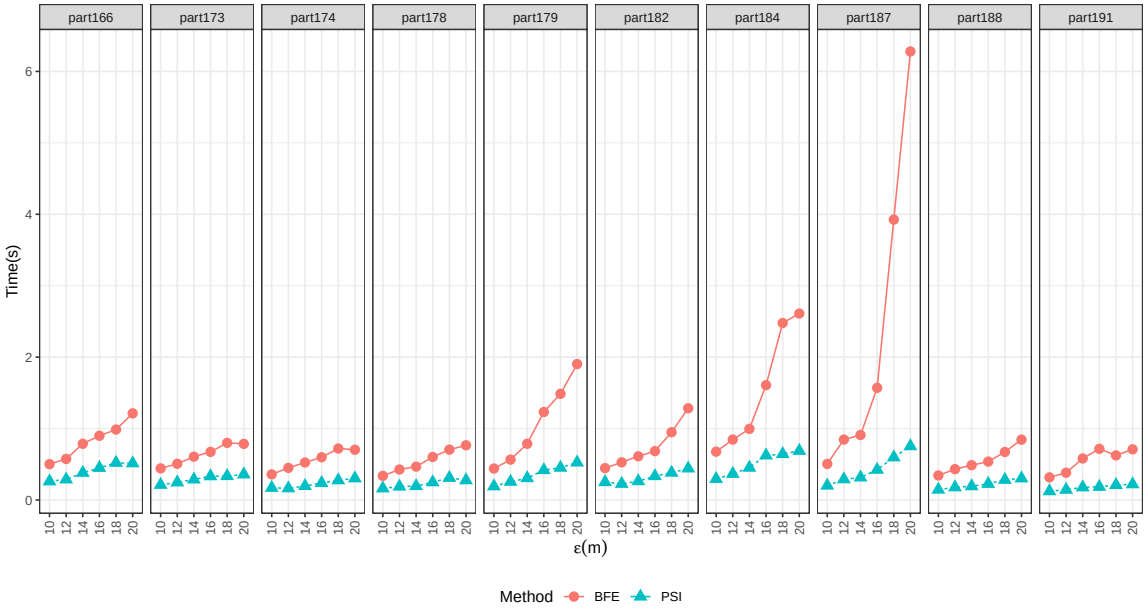


Figure 3.13: Comparing the performance of PSI and BFE for time consuming partitions.

ig:top_time_partitions⟩

cores, these demanding partitions have the greatest impact on the overall performance. By focusing on these partitions, we can better understand potential bottlenecks and further optimize the system's efficiency.

3.5.3 Analyzing most costly partitions.

We began by identifying the top 10 partitions that required the most time to execute the BFE algorithm with $\varepsilon = 20$ meters. For these specific partitions, we ran both BFE and PSI while varying ε from 10 to 20 meters. The Phase 1 execution times are shown in Figure 3.13, where it is evident that PSI consistently outperformed BFE across all values of ε.

We further investigated the reasons behind some partitions taking longer to compute. Figure 3.14 shows the Phase 1 execution times per partition while varying ε from 10m to 20m, with partitions ordered by the number of pairs they contain. One key observation is that as ε increases, the number of pairs also increases, since a larger ε allows for more maximal disks. For instance, with $\varepsilon = 10m$, the maximum number of pairs in a partition is around 1800, whereas for $\varepsilon = 20m$, some partitions contain nearly 4000 pairs.

Another notable observation is that BFE is more sensitive to the density of pairs within a partition than PSI, a difference that becomes more pronounced at higher values

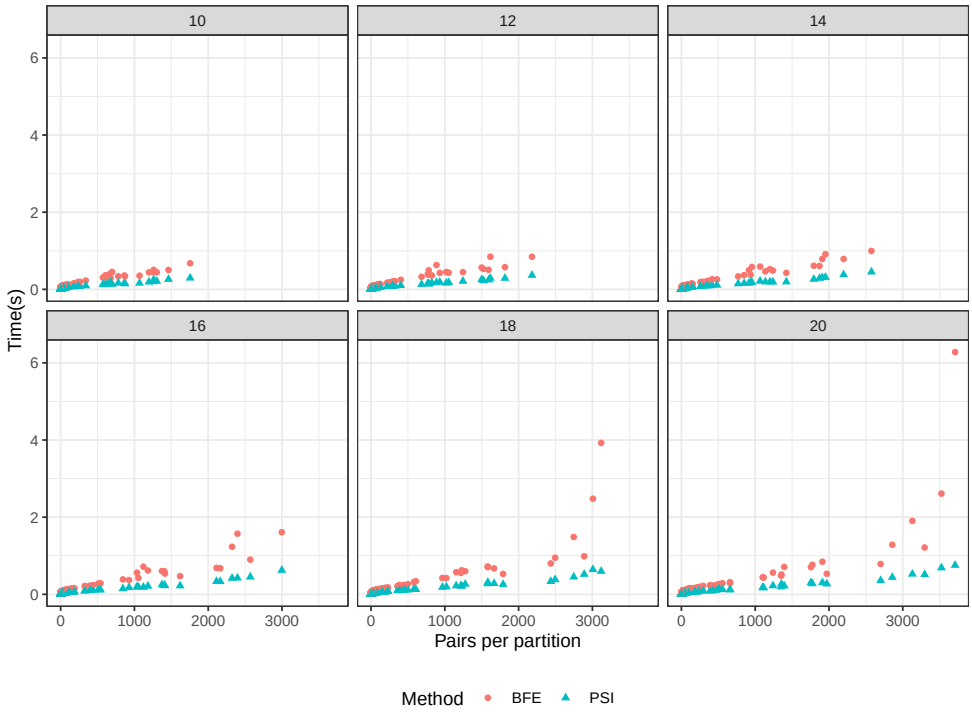


Figure 3.14: Execution time for pairs/disks finding in the dense partition.

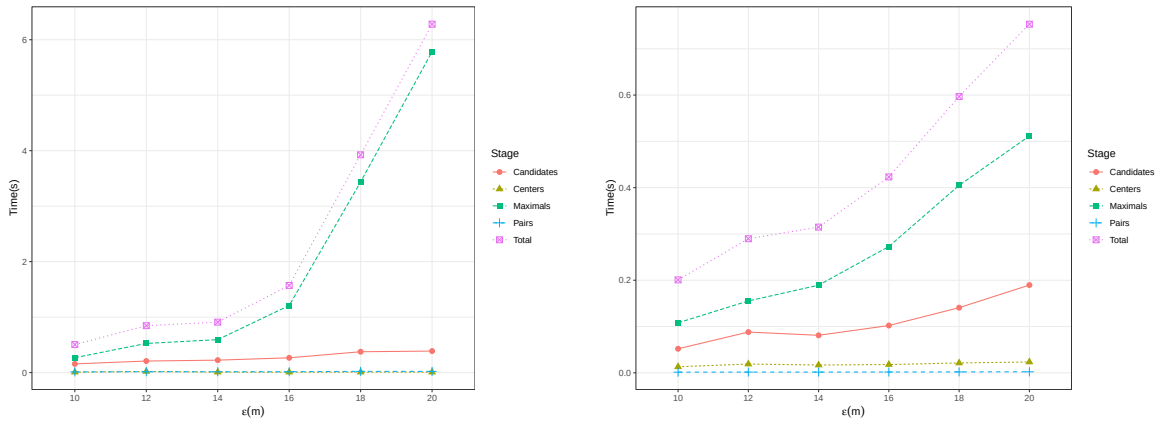
⟨fig:pairs_performance⟩

of ε (e.g., 18m or 20m). As mentioned earlier, the flexible bounding boxes used by PSI (illustrated in Figure 3.6) more effectively isolate the relevant points for computing pairs, whereas BFE relies on a fixed grid cell, which makes it less efficient in denser partitions.

A final observation is that a few partitions take significantly more time than others, particularly those with a higher density of pairs. This is directly related to the number of maximal disks that need to be computed and subsequently pruned. For example, the partition that takes the longest time when $\varepsilon = 20m$ is the one with the highest number of pairs, which corresponds to partition 187 in Figure 3.13.

We further analyzed how Phase 1 processing is distributed within the most demanding partition. Figure 3.15.a (for BFE) and Figure 3.15.b (for PSI) display the time taken by each Phase 1 stage (refer to Figure 3.4) for partition 187. The most resource-intensive stage in both cases is the final step of filtering the disks, where disks whose points are contained within others are removed—this stage identifies the *maximal* disks (labeled as 'Maximals' in the figure).

This stage is particularly costly because both BFE and PSI must scan a large set of candidate disks, identifying and removing those that are redundant. As ε increases, this processing becomes even more time-consuming, as the number of pairs and candidate disks



(a) (b)

Figure 3.15: Processing time for the stages of Phase 1, in (a) BFE, (b) PSI.

⟨fig:dense_stages⟩

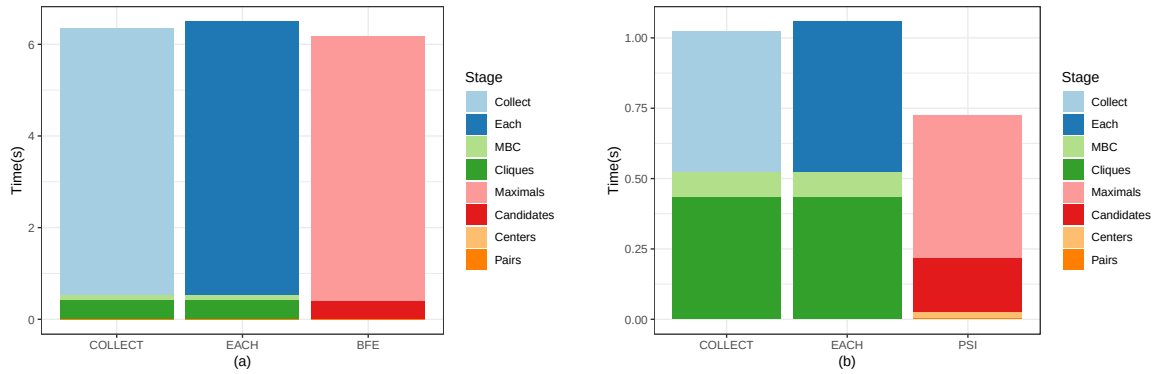
grows along with ε.

3.5.4 Can we reduce pruning time?

Dense areas pose challenges for pruning, as they are highly sensitive to increases in the value of ε, leading to an exponential growth in the number of pairs. To address this, we explored alternative strategies that could enable more effective grouping of points. It is important to note that density-based spatial clustering methods, such as DBSCAN [21], are not suitable for this problem. In very dense regions, these approaches often produce a single large cluster, which does not resolve the issue. Additionally, clustering algorithms do not enforce the strict relationships required for a flock, where all points must be within a distance of ε from each other.

Instead, we explored graph-oriented clustering, focusing on the concept of *maximal cliques*. In an undirected graph, a maximal clique is a subset of vertices where each vertex is directly connected to every other vertex in the subset. Additionally, the clique is maximal in the sense that it cannot be extended by adding more vertices [60, 12].

In this context, the points within a partition can be treated as the vertices of an undirected graph, where edges are created between pairs of points that are within a distance of ε. By finding the set of maximal cliques in this graph, we identify subsets of points where each point is connected to all others in the subset. This means that all points in the clique are at most ε apart, and no additional points can be added to the subset. However, not every maximal clique qualifies as a maximal disk. A maximal clique becomes a maximal disk only if it contains at least μ points and can be enclosed by a disk with a radius of $\frac{\varepsilon}{2}$.



(a) (b)

Figure 3.16: Execution time of the Cliques approach compared to (a) standard BFE and (b) standard PSI.

⟨fig:cmbc_variants⟩

To verify whether a maximal clique qualifies as a maximal disk, we introduce the concept of the *Minimum Bounding Circle* (MBC) [65]. Given a set of points in Euclidean space, the MBC is the smallest circle that can enclose all the points. For each maximal clique identified within a partition, we can quickly check if all points in the clique fit within an MBC with a diameter of ε. If they do, we can immediately report the set of points and their MBC as a maximal disk. However, cliques that do not satisfy this condition must be evaluated using the traditional method. This involves computing the potential disk centers, identifying candidate disks, and pruning them, as outlined in Figure 3.1.

To evaluate the cliques that do not meet the above condition, we implemented two variants. The first variant, termed *COLLECT*, gathers the points from all cliques that are not reported as maximal disks, removes duplicates (since points may appear in multiple cliques), and then applies the traditional pruning method to the entire set. In the second variant, *EACH*, we apply the pruning procedure independently for each clique that does not qualify as a maximal disk.

Figure 3.16 compares the performance of these variants against the time taken by BFE (a) and PSI (b) for the same stage. Surprisingly, neither variant improves execution time. A closer examination reveals that while identifying the cliques and their MBCs is relatively fast, few cliques actually qualify as maximal disks. As a result, the overhead involved in processing the remaining cliques is significant, making the original approach more efficient for both BFE and PSI.

Table 3.2: Number of partitions by capacity and number of points in synthetic uniform datasets.

⟨tab:uniform_ncells⟩

	25K	50K	75K	100K
c=100	544	1024	1024	2185
c=200	256	514	1024	1024
c=300	256	514	481	1024

3.5.5 Relative performance of BFE and PSI Phase 1 using synthetic datasets.

To further examine the relative performance of the scalable BFE and PSI approaches for Phase 1, we also conducted experiments using a synthetic dataset where we could control the values of c, ε, and point density. We used a fixed square area of 1000m x 1000m, within which we uniformly distributed 25K, 50K, 75K, and 100K points.

We experimented with different quadtree capacities (c values of 100, 200, and 300), which resulted in varying numbers of partitions (as shown in Table 3.2). Both BFE and PSI were tested for phase 1, where maximal disks are identified, using ε values ranging from 1m to 5m. The results are presented in Figure 3.17.

Overall, PSI demonstrated better performance than BFE, though there were cases (particularly with smaller ε values) where BFE outperformed PSI. In these cases, the smaller ε generates fewer pairs, and the additional ordering step required by PSI becomes an overhead. However, in the subsequent experiments focusing on temporal joins (phase 2, flock creation), we concentrate on the scalable performance of PSI.

3.5.6 Evaluation of Phase 2: Temporal join.

Phase 2 focuses on joining maximal disks across time instants to form flocks. In Section 3.4.2, we discussed four alternatives: Master, By-Level, LCA, and Cube-based. For these experiments, we used the scalable PSI approach due to its robust performance. First, we compared the Master and By-Level alternatives while varying ε from 20m to 40m using the Berlin10K dataset (see Figure 3.18). For the By-Level approach, we tested different step values ranging from 1 to 6. The Master approach proved to be the slowest, due to the overhead of sending all CPFs to the root node. The performance of the By-Level approach depends on the step size. A smaller step value (e.g., step 1) introduces overhead because CPFs may need to be evaluated at more intermediate nodes before completion. On the other hand, a larger step value reduces parallelism by sending more CPFs to intermediate

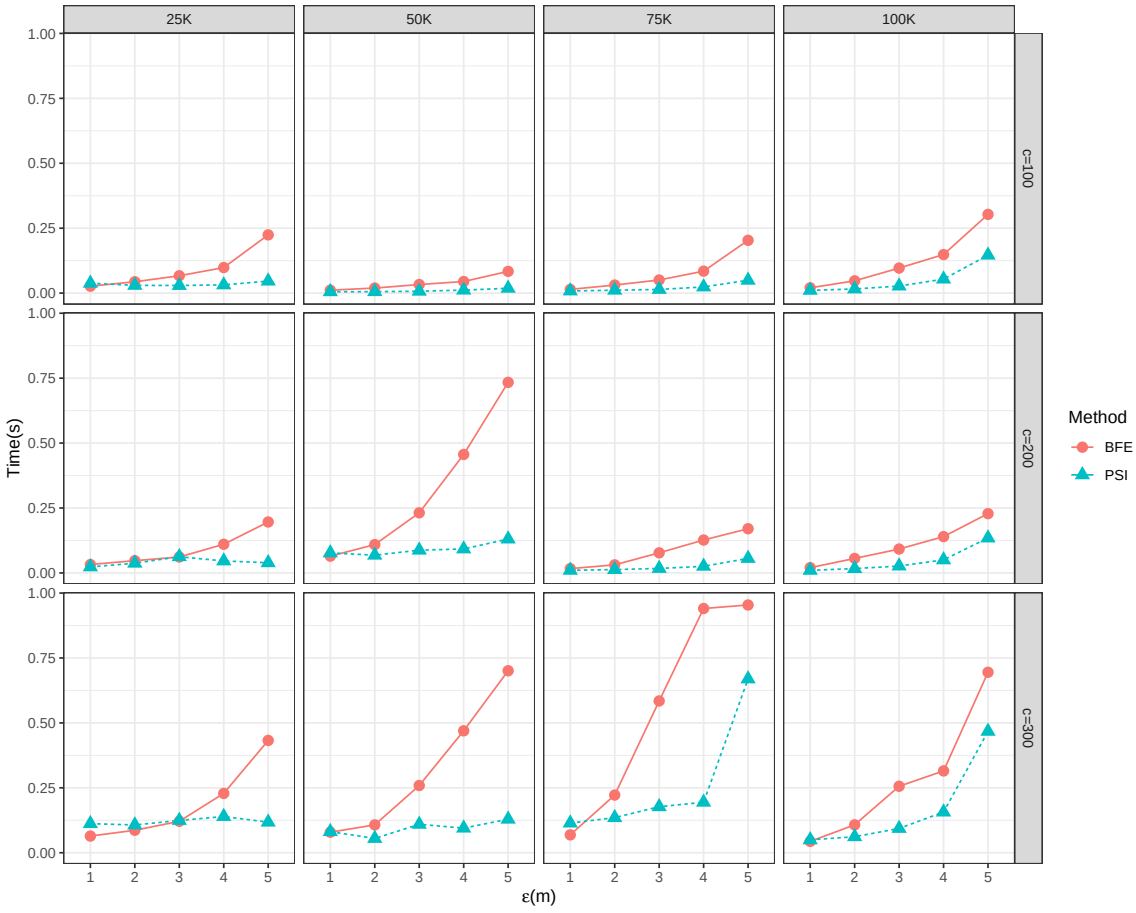


Figure 3.17: Performance in an uniform dataset analysing density and capacity with diverse values for epsilon.

ig:uniform_performance⟩

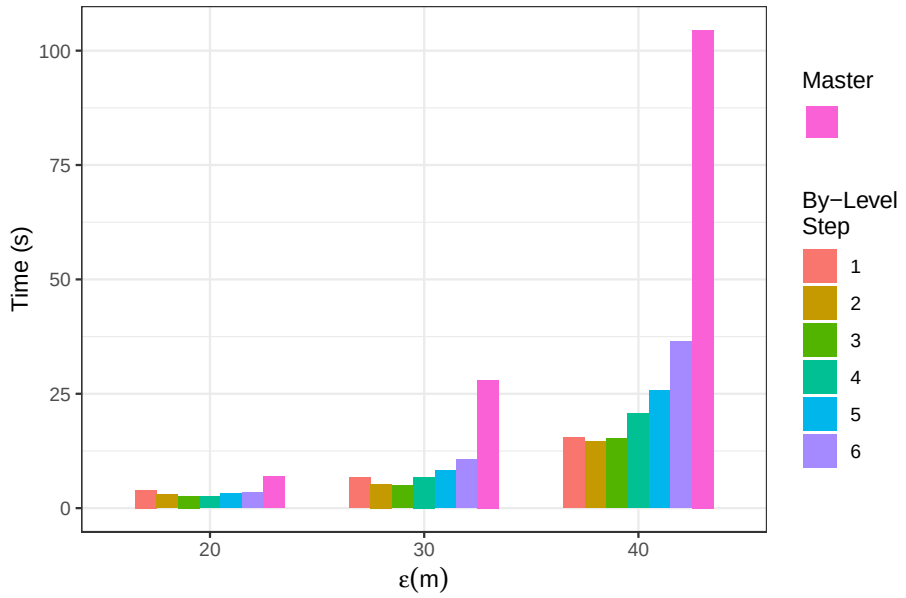


Figure 3.18: Root and step alternative for temporal join using the Berlin dataset (step=7 is root).

⟨fig:step_performance⟩

nodes. Based on these experiments, we determined that Step=3 offers the best balance.

We also evaluated the optimal value for the *interval* parameter in the Cube-based approach. Using the LA25K dataset with $\varepsilon = 30m$, we tested various interval values, ranging from 2 to 12 time instants. This dataset contains 30 time instants in total. The results, shown in Figure 3.19, illustrate the trade-offs involved. Lower interval values result in higher parallelism, as more cubes can be processed independently. However, this also increases the number of cube crossings for CPFs that need to be checked, which adds to the execution time. Conversely, larger interval values reduce parallelism but also decrease the number of CPF crossings. Based on these findings, we selected $interval = 6$ as the optimal value for the Cube-based approach.

Finally, we compared the optimized versions of the By-Level and Cube-based approaches with the Master and LCA methods. Figure 3.20 shows the results, including the sequential PSI algorithm as a reference. This experiment was conducted using the LA25K dataset with ε values ranging from 5m to 30m. Clearly, all parallel approaches offer significant improvements over the sequential PSI.

To further analyze the relative performance of the scalable approaches, Figure 3.21 focuses on the parallel algorithms for the same experiment. Interestingly, for very small ε values, the Master approach performs best —primarily because the limited number of flocks

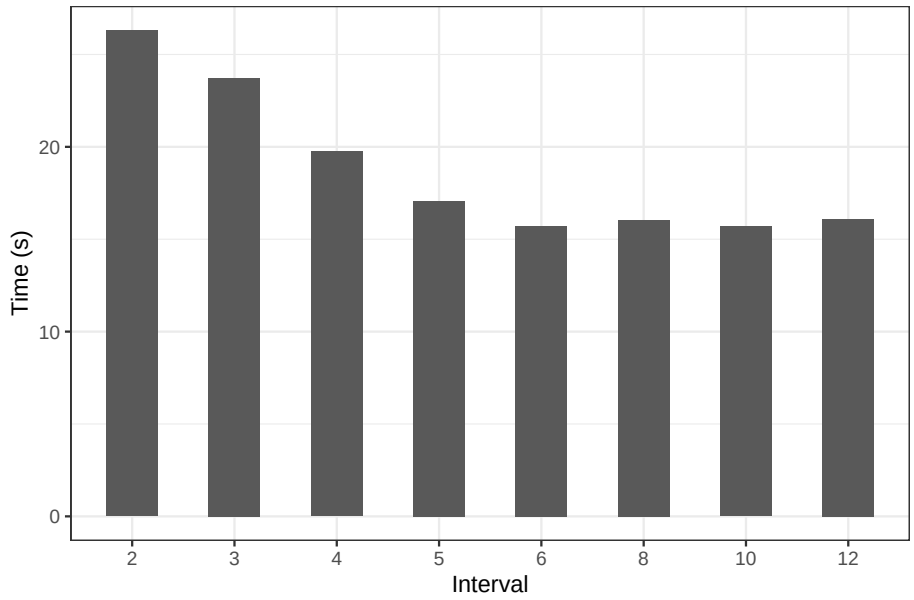


Figure 3.19: Interval optimization for the Cube-based alternative for temporal join using the LA25K dataset.

g:interval_performance⟩

makes sending the CPFs to a single node fast and efficient. However, as ε increases, the Cube-based approach becomes the most effective, leveraging greater parallelism. By-Level also improves over the Master approach as ε grows, as explained in Figure 3.18. Similarly, for larger ε values, the LCA approach outperforms By-Level because it more quickly identifies the node that can complete the CPF operations.

We repeated the same experiment with the LA50K dataset, varying ε from 4m to 20m. The results, shown in Figure 3.22, once again demonstrate that the Cube-based approach offers the best performance as ε increases.

3.6 Conclusions

⟨sec:conclusions⟩

We present a novel, scalable approach to discovering moving flock patterns in large trajectory databases. By leveraging distributed frameworks, the proposed method overcomes the limitations of sequential algorithms that struggle with large-scale spatio-temporal datasets. Through partitioning and replication, as well as improvements in pruning and temporal joins, this approach efficiently handles dense data, offering significant performance improvements over traditional methods. The evaluation results demonstrate

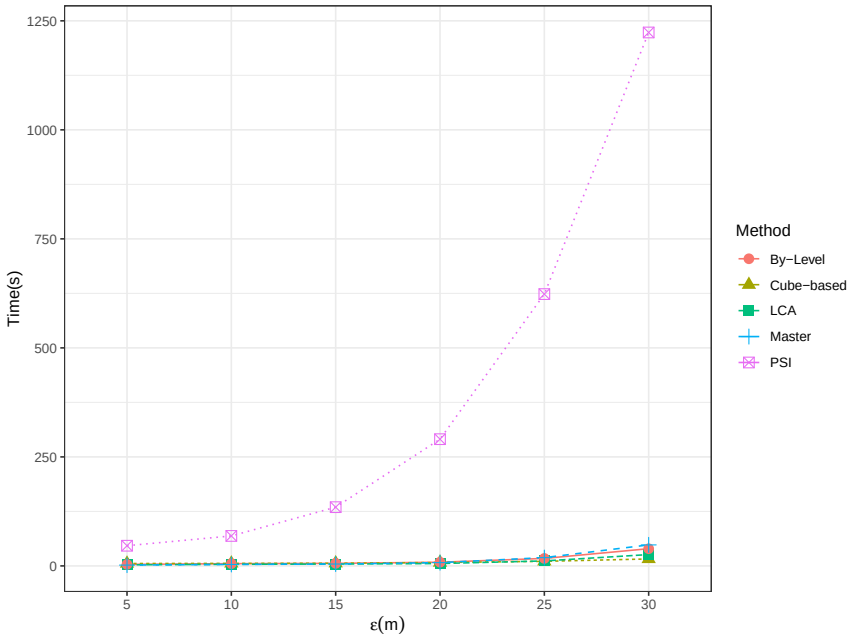


Figure 3.20: Performance comparing parallel and sequential alternatives in the LA25K dataset.

⟨fig:la25k_e_bfe_psi⟩

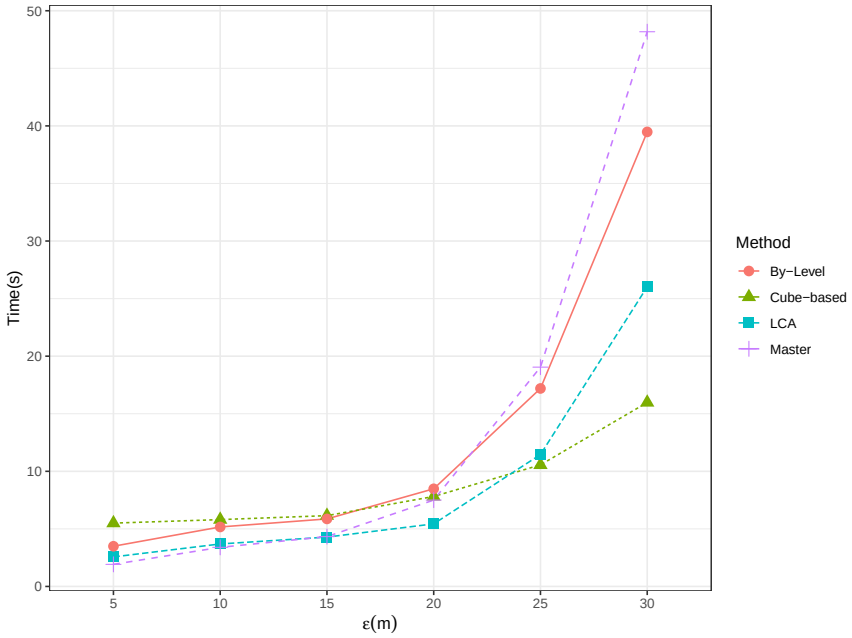


Figure 3.21: Performance of the 4 parallel alternatives in the LA25K dataset.

⟨fig:la25k_e⟩

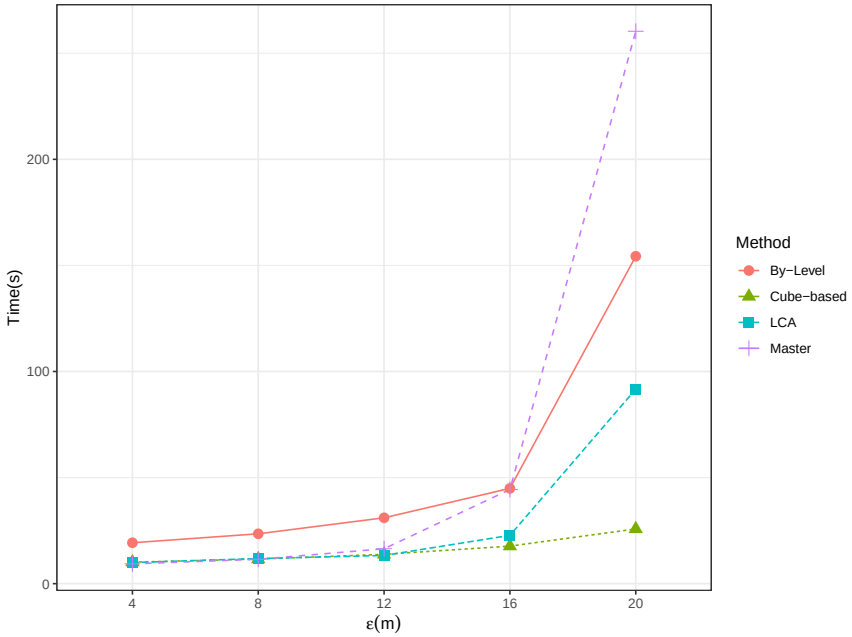


Figure 3.22: Performance of the 4 parallel alternatives in the LA50K dataset.

⟨fig:la50k_e⟩

the scalability and effectiveness of the approach, making it a valuable contribution for analyzing complex movement patterns.

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

abdelhafeez_ddcel_2023 [1] Laila Abdelhafeez, Amr Magdy, and Vassilis Tsotras. DDCEL: Efficient Distributed Doubly Connected Edge List for Large Spatial Networks. In *2023 24th IEEE International Conference on Mobile Data Management (MDM)*, pages 122–131, 2023.

aji_hadoop-gis_2013 [2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System Over Map-Reduce. In *VLDB Journal*, 2013.

amor_persistence_2016 [3] T. Amor, S. Reis, D. Campos, H. Herrmann, and J. Andrade. Persistence in Eye Movement during Visual Search. *Scientific Reports*, 6:20815, 2016.

arimura_finding_2014 [4] H. Arimura, T. Takagi, X. Geng, and T. Uno. Finding All Maximal Duration Flock Patterns in High-Dimensional Trajectories. 2014.

barequet_dcel_1998 [5] G. Barequet. DCEL - A Polyhedral Database and Programming Environment. *Ijcga*, 08(05n06):619–636, 1998.

beckmann_r-tree_1990 [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*, pages 322–331, New York, NY, USA, 1990. Association for Computing Machinery.

benkert_reporting_2008 [7] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting Flock Patterns. *Computational Geometry*, 41(3):111–125, 2008.

berich_arrangements_2010 [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.

berg_computational_2008 [9] M. Berg, O. Cheong, M. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, TU Eindhoven, P.O. Box 513, 2008.

slawski_modelling_2011 [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.

topological-based_2020 [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.

bron_algorithm_1973 [12] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

calderon_mining_2011 [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.

calderon_scalable_2023 [14] Andres Calderon, Vassilis Tsotras, and Amr Magdy. Scalable Overlay Operations over DCEL Polygon Layers. In *Proceedings of the 18th International Symposium on Spatial and Temporal Data*, SSTD '23, pages 85–95, New York, NY, USA, 2023. Association for Computing Machinery.

challa_dd-rtree_2016 [15] 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.

chew_convex_1993 [16] L. Chew and K. Kedem. A convex polygon among polygonal obstacles. *Computational Geometry*, 3(2):59–89, 1993.

tal_combinatorial_1975 [17] V. Chvátal. A combinatorial theorem in plane geometry. *Combinatorial Theory*, 18(1):39–41, 1975.

lorenzo_allaboard_2016 [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.

awy_spatialhadoop_2014 [19] A. Eldawy. SpatialHadoop: Towards Flexible and Scalable Spatial Processing Using Mapreduce. In *SIGMOD PhD Symposium*, pages 46–50, 2014.

awy_spatialhadoop_2015 [20] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A Map-Reduce Framework For Spatial Data. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2015.

dbscan [21] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

finkel_quadtrees_1974 [22] Raphael Finkel and Jon Bentley. Quadtrees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, March 1974.

fogel_cgal_2012 [23] E. Fogel, D. Halperin, and R. Wein. *CGAL Arrangements and Their Applications*. Springer Berlin, Heidelberg, 2012.

fort_parallel_2014 [24] M. Fort, J. Antoni, and N. Valladares. A Parallel GPU-Based Approach for Reporting Flock Patterns. *Ijgis*, 28(9):1877–1903, 2014.

franklin_data_2018 [25] 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.

reiseisen_colored_1998 [26] W. Freiseisen. Colored DCEL for boolean operations in 2D, 1998.

geng_enumeration_2014 [27] X. Geng, T. Takagi, H. Arimura, and T. Uno. Enumeration of Complete Set of Flock Patterns in Trajectories. In *Iwgs*, pages 53–61, 2014.

web:geos:polygonizer [28] GEOS Polygonizer Implementation. https://github.com/libgeos/geos.

undsson_computing_2006 [29] J. Gudmundsson and M. van Kreveld. Computing Longest Duration Flocks in Trajectory Data. In *Acm Sigspatial*, pages 35–42, 2006.

guttman_r-trees_1984 [30] 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.

lay_openstreetmap_2008 [31] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive computing*, 7(4):12–18, 2008.

oel_data-parallel_2003 [32] Erik G Hoel and Hanan Samet. Data-parallel polygonization. *Parallel Computing*, 2003.

holland_movements_1999 [33] 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.

holmes_dcel_2021 [34] R. Holmes. The DCEL Data Structure for 3D Graphics, 2021.

huang_mining_2015 [35] P. Huang and B. Yuan. Mining Massive-Scale Spatiotemporal Trajectories in Parallel: A Survey. In *Takddm*, volume 9441. Springer, 2015.

hughes_geomesa_2015 [36] 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.

jeung_trajectory_2011 [37] H. Jeung, M. Yiu, and C. Jensen. Trajectory Pattern Mining. In *Computing with Spatial Trajectories*, pages 143–177. Springer, 2011.

jeung_discovery_2008 [38] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. Discovery of Convoys in Trajectory Databases. *Vldb*, 1(1):1068–1080, 2008.

web:jts:polygonizer [39] JTS Polygonizer Implementation. https://github.com/locationtech/jts.

alnis_discovering_2005 [40] P. Kalnis, N. Mamoulis, and S. Bakiras. On Discovering Moving Clusters in Spatio-Temporal Data. In *Astd*, pages 364–381. Springer, 2005.

krajzewicz_recent_2012 [41] 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.

sorte_convergence_2016 [42] 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.

leung_knowledge_2010 [43] Y. Leung. *Knowledge Discovery in Spatial Data*. Springer, 2010.

li_scalable_2019 [44] 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.

li_swarm_2010 [45] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *Vldb*, 3(1-2):723–734, 2010.

magalhaes_fast_2015 [46] 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.

mehlhorn_leda_1995 [47] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

miller_geographic_2001 [48] H. Miller and J. Han. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, Inc., 2001.

muller_finding_1978 [49] D. Muller and F. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

nievergelt_grid_1984 [50] 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.

orourke_art_1987 [51] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, United States, 1987.

web:data:continents [52] OpenStreetMap Data Extracts. http://download.geofabrik.de/.

pandey_how_2021 [53] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. How Good Are Modern Spatial Libraries? *Data Science and Engineering*, 2021.

ata_computational_1985 [54] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, New York, NY, 1985.

puri_mapreduce_2013 [55] 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.

puri_efficient_2013 [56] 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.

sabek_spatial_2017 [57] 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.

samet_design_1990 [58] H. Samet. *The Design and Analysis of Spatial Data Structures*. Wesley, 75 Arlington Street, Suite 300 Boston, MA, United States, 1990.

tanaka_improved_2016 [59] P. Tanaka, M.. Vieira, and D. Kaster. An Improved Base Algorithm for Online Discovery of Flock Patterns in Trajectories. *Jidm*, 7(1), 2016.

tomita_simple_2013 [60] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments. *IEICE Transactions on Information and Systems*, E96.D(6):1286–1298, 2013.

turdukulov_visual_2014 [61] 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.

web:data:usa [62] U.S. Street Network Shapefiles, Node/Edge Lists, and GraphML Files. https://doi.org/10.7910/DVN/CUWWYJ.

vieira_-line_2009 [63] M. Vieira, P. Bakalov, and V. Tsotras. On-Line Discovery of Flock Patterns in Spatio-Temporal Data. In *Acm Sigspatial*, pages 286–295, 2009.

a_spatio-temporal_2013 [64] M. Vieira and V. Tsotras. *Spatio-Temporal Databases: Complex Motion Pattern Queries*. Springer, 2013.

welzl_smallest_1991 [65] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, pages 359–370. Springer, 1991.

xie_simba_2016 [66] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient In-Memory Spatial Analytics. In *Icmd*, pages 1071–1085, 2016.

you_large-scale_2015 [67] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale Spatial Join Query Processing In Cloud. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2015.

yu_demonstration_2016 [68] 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.

yu_spatial_2018 [69] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial Data Management in Apache Spark: the GeoSpark Perspective and Beyond. *GeoInformatica*, 2018.

zheng_computing_2011 [70] Y. Zheng and X. Zhou. *Computing with Spatial Trajectories*. Springer, 2011.

.1 Center computation.

.2 Disk pruning.

.3 Clique and MBC approach.

Algorithm 3 Find the centers of given radius which circumference laid on the two input points.

Require: Radius $\frac{\varepsilon}{2}$ and points p_1 and p_2.
Ensure: Centers c_1 and c_2.

function FINDCENTERS(p_1, p_2, $\frac{\varepsilon}{2}$)
  $r^2 \leftarrow (\frac{\varepsilon}{2})^2$
  $X \leftarrow p_1.x - p_2.x$
  $Y \leftarrow p_1.y - p_2.y$
  $d^2 \leftarrow X^2 + Y^2$
  $R \leftarrow \sqrt{|4 \times \frac{r^2}{d^2} - 1|}$
  $c_1.x \leftarrow X + \frac{Y \times R}{2} + p_2.x$
  $c_1.y \leftarrow Y - \frac{X \times R}{2} + p_2.y$
  $c_2.x \leftarrow X - \frac{Y \times R}{2} + p_2.x$
  $c_2.y \leftarrow Y + \frac{X \times R}{2} + p_2.y$
  return c_1 and c_2
end function

⟨app:centers⟩

Algorithm 4 Prune disks which are duplicate or subset of others.

Require: Set of disks D.
Ensure: Set of disks D' without duplicate or subsets.

function PRUNEDISKS(D)
  $E \leftarrow \varnothing$
  for all disk d_i in D **do**
    $N \leftarrow d_i \cap D$
    for all disk n_j in N **do**
      if d_i contains all the elements of n_j **then**
        $E \leftarrow E \cup n_j$
      end if
    end for
  end for
  $D' \leftarrow D \setminus E$
  return D'
end function

⟨app:disks⟩

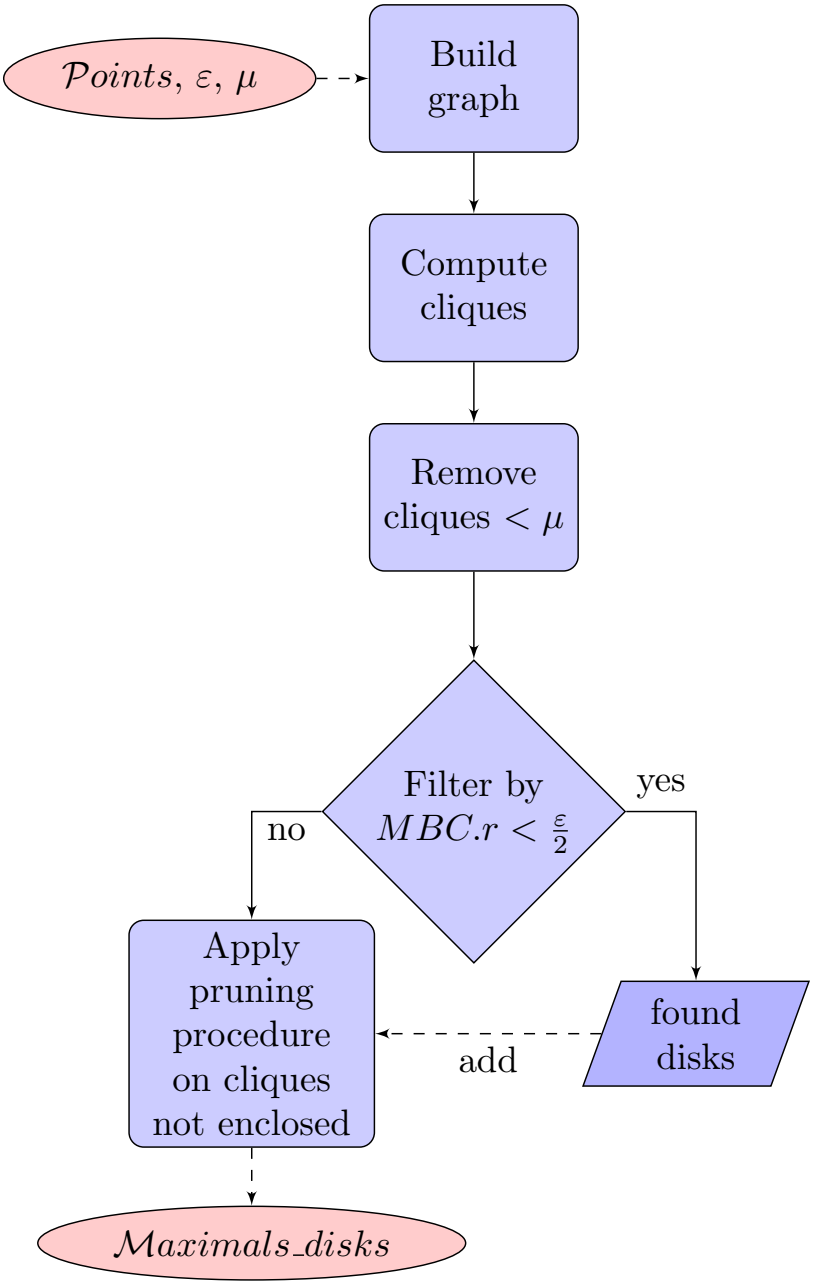


Figure .1: Schematic description of the Clique and MBC approach.

?⟨fig:cmbc_flowchart⟩?