On Scalable DCEL Overlay Operations

Andres Calderon-Romero[1*], Laila Abdelhafeez[1], Goce Trajcevski[2], Amr Magdy[1], Vassilis J. Tsotras[1]

[1]Computer Science Department, University of California, 900 University Ave, Riverside, 92521, California, US.
[2]Department of Electrical and Computer Engineering, Iowa State University, 2433 Union Dr, Ames, 50011, Iowa, US.

*Corresponding author(s). E-mail(s): acald013@ucr.edu;
Contributing authors: labde005@ucr.edu; gocet25@iastate.edu;
amr@cs.ucr.edu; tsotras@cs.ucr.edu;

Abstract

The Doubly Connected Edge List (DCEL) is an edge-list structure widely used in spatial applications, primarily for planar topological and geometric computations. However, it is also applicable to various types of data, including 3D models and geographic data. An essential operation is the *overlay operation*, which combines the DCELs of two input polygon layers and can easily support spatial queries on polygons like the intersection, union, and difference between these layers. However, existing techniques for spatial overlay operations suffer from two main limitations. First, they fail to handle many large datasets practically used in real applications. Second, they cannot handle arbitrary spatial lines that practically form polygons, e.g., city blocks, but they are given as a set of scattered lines. This work proposes a distributed and scalable way to compute the overlay operation and its related supported queries. Our operations also support arbitrary spatial lines through a scalable polygonization process. We address the issues of efficiently distributing the lines and overlay operators and offer various optimizations that improve performance. Our experiments demonstrate that the proposed scalable solution can efficiently compute the overlay of large real datasets.

Keywords: Spatial data structures, overlay operator, DCEL

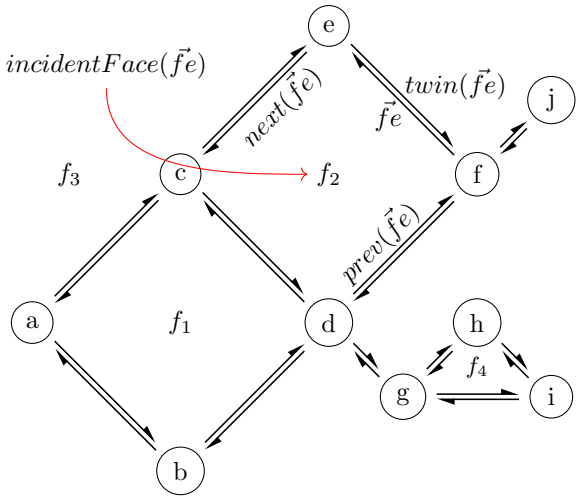


Fig. 1 Components of the DCEL structure.

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 [1]. 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 [2], R-trees [3, 4], and quadtrees [5]. *Edge-list* structures are also typically utilized in applications as topological computations in computational geometry [6].

The most commonly used data structure in the edge-list family is the *Doubly Connected Edge List (DCEL)*. A DCEL [7, 8] 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 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 [6].

The DCEL data structure has been used in various applications. For instance, the use of connected edge lists is cardinal to support polygon triangulations and their applications in surveillance (the Art Gallery Problem [9, 10]) and robot motion planning (Minkowski sums [6, 11]). 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 [12]) as well as the computation of silhouettes from polyhedra [12, 13] (applied frequently in computer vision and 3D graphics modeling [14]).

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 [3, 4]) 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 [15]. 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 4.1.2). 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 [16] 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 presents related work, while Section 3 discusses the basics of DCEL and the sequential algorithm. In Section 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 5. Section 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 7, while Section 8 concludes the paper.

2 Related Work

The fundamentals of the DCEL data structure were introduced in the seminal paper by Muller and Preparata [7]. The advantages of DCELs are highlighted in [6, 8]. Examples of using DCELs for diverse applications appear in [17–19].

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 [19]. Currently, few sequential implementations are available: LEDA [20], Holmes3D [21] and CGAL [12]. 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), [22–28] 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 [29, 30]. While the JTS library is used in many modern distributed spatial analytics systems [31], including Hadoop-GIS [32], SpatialHadoop [33], GeoSpark [34], and SpatialSpark [35], the implementation of the polygonization algorithm [29] 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 [36]. 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 [36] rely on iterative data re-partitioning, [36] 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).

3 Preliminaries

The DCEL [7] structure is used to represent an embedding of a planar subdivision in the plane. It provides efficient manipulation of the geometric and topological features

Table 1 Vertex records.

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

Table 2 Face records.

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

Table 3 Half-edge records.

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

of spatial objects (polygons, lines, and points) using *faces*, *edges*, and *vertices*, respectively. A DCEL uses three tables (relations) to store records for the faces, edges, and vertices, respectively.

An important characteristic is that all these records are defined using edges as the main component (thus termed an edge-based structure). Examples appear in Tables 1-3, with the subdivision depicted in Figure 1.

An edge corresponds to a straight line segment shared by two adjacent faces (polygons). Each of these two faces will use this edge in its description; to distinguish, each edge has two *half-edges*, one for each orientation (direction). It is important to note that half-edges are oriented counter-clockwise inside each face (Figure 1). 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 3. These references are used as keys to the tables that contain the referred attributes.

Figure 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 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.

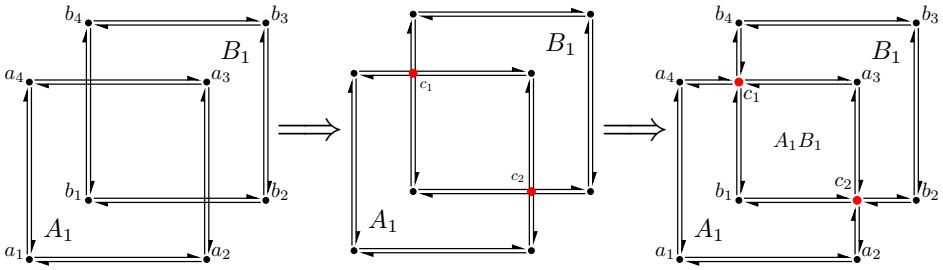


Fig. 2 Sequential computations of an overlay of two DCEL layers.

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. All other half-edges for this face's boundary can be easily retrieved following the next half-edges in orientation order. In addition to regular faces, there is one face that covers the area outside all faces; it is called the *unbounded* face (face f_3 in Figure 1). Since f_3 has no boundary, its boundary edge is set to *nil* in Table 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, this list is empty as there are no holes in any of the faces in the example of Figure 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 $f1 \cap f2$ [6]. 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 [12] 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) [6]. 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 illustrates an example of computing the overlay between two DCEL layers with one face each (A_1 and B_1 respectively) overlapping the same area. First, intersection points are identified, and new vertices are created in the overlay (red vertices c_1 and c_2). Then, new half-edges are created around these new vertices. As a result, face

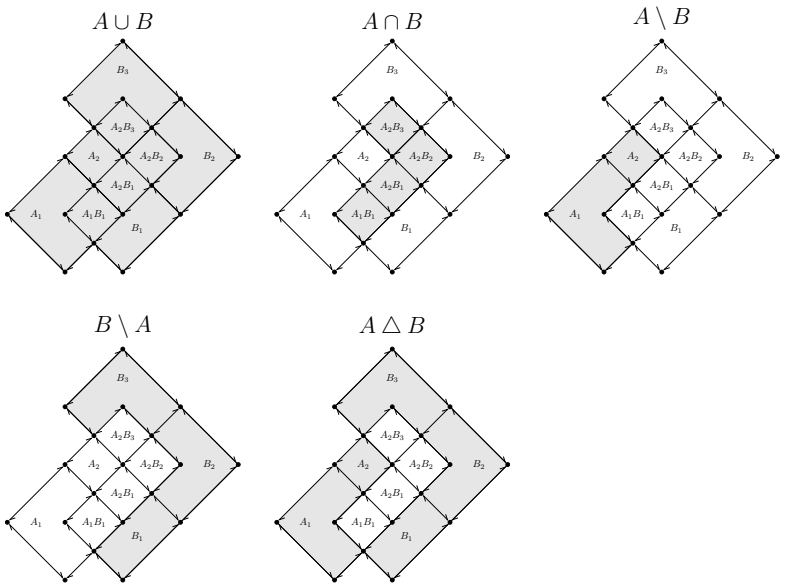


Fig. 3 Examples of overlay operators supported by DCEL; results are shown in gray.

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 [6], 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 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.)

4 Scalable Overlay Construction

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 [6].

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

4.1 Partition Strategies

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 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.

4.1.1 Quadtree Partition Strategy

The main idea of the quadtree partition strategy is to split the area covered by the input layers into non-overlapping cells, which can then be processed independently. 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 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 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 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 4, Figure 5 depicts an overview of the process for creating a local overlay DCEL inside cell 2.

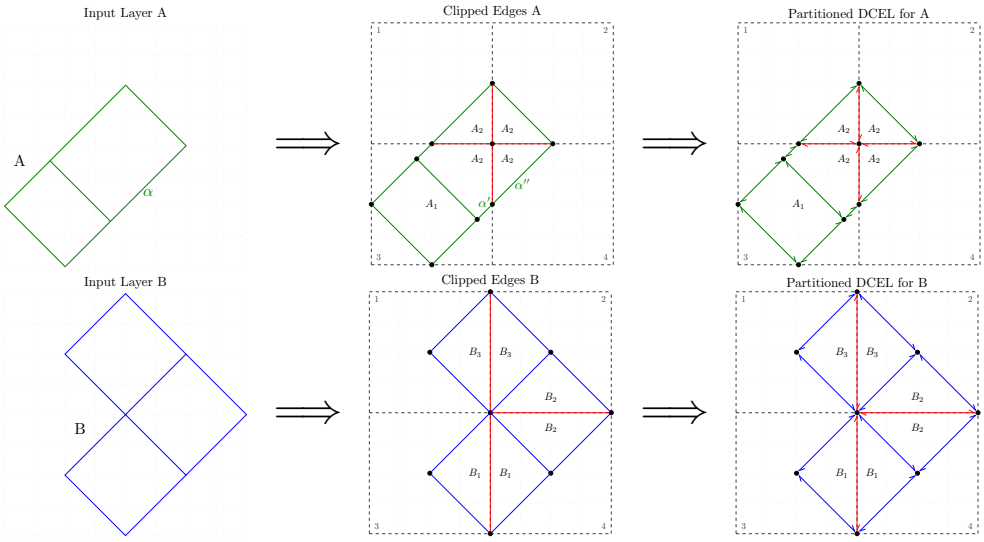


Fig. 4 Partitioning example using input layers A and B over four cells.

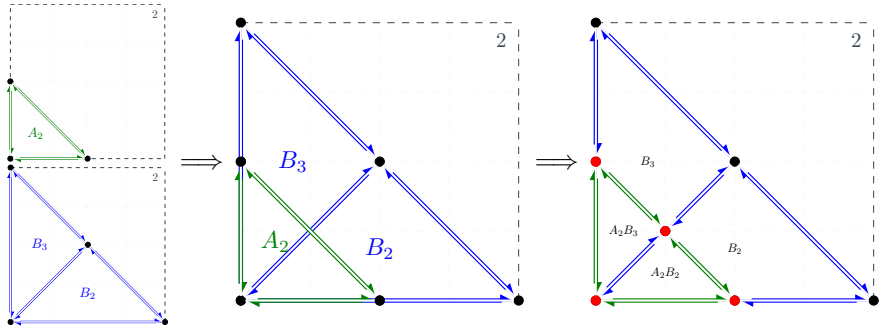


Fig. 5 Local overlay DCEL for cell 2.

Similarly, Figure 6 shows all local overlay DCELs computed at each cell (artificial edges are shown in red).

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

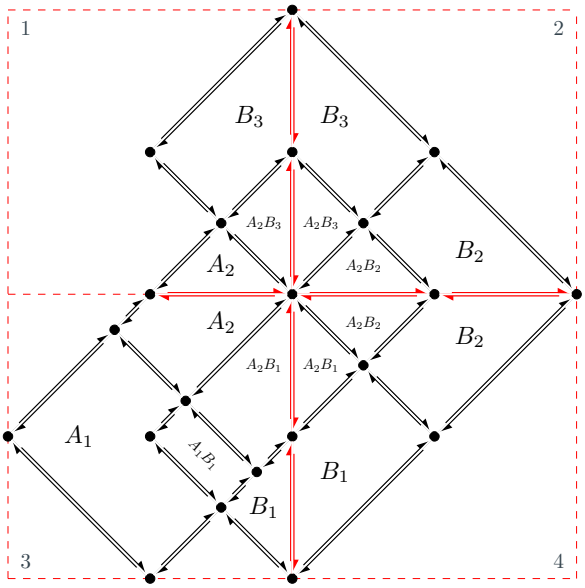


Fig. 6 Result of the local overlay DCEL computations.

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 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.

4.1.2 Kd-tree Partition Strategy

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 4.1.1.

4.2 Labeling Orphan Cells and Holes

Assuming a quadtree-based partitioning, to find the label of an orphan cell, we propose an algorithm that recursively searches the space around the orphan cell until it identifies a nearby cell that contains an edge(s) of the face that includes the orphan

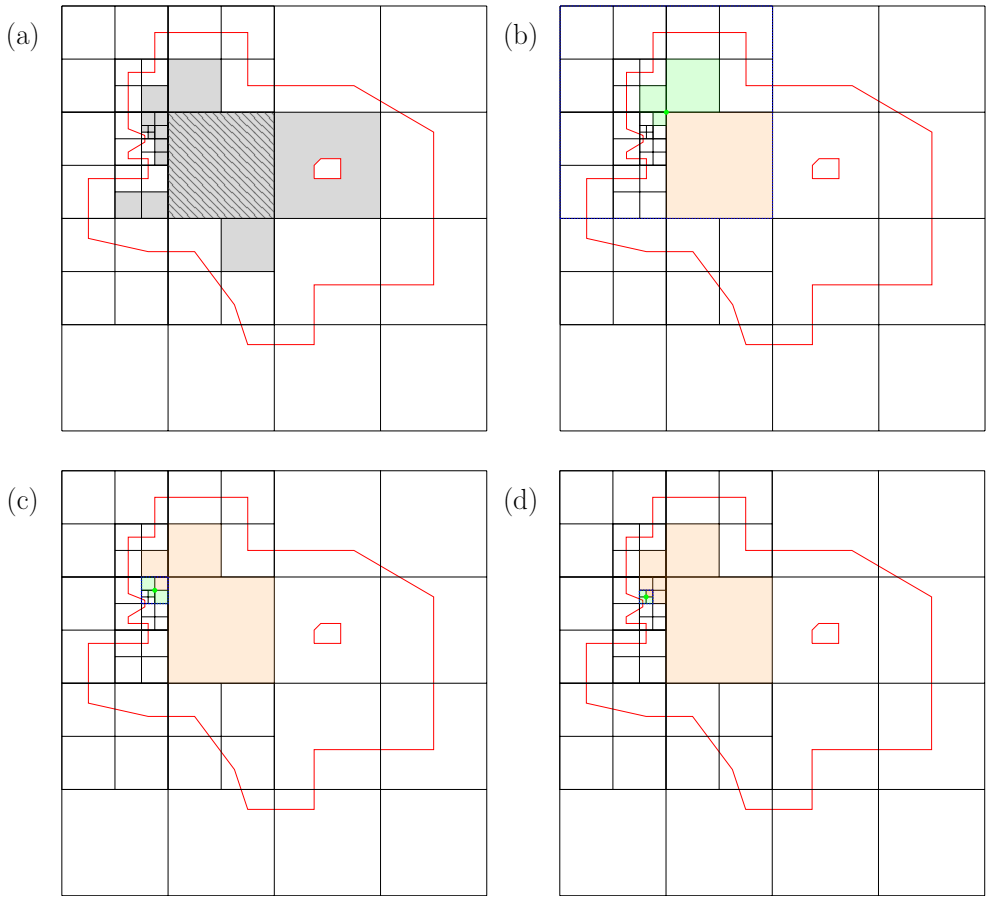


Fig. 7 (a) Empty cell and hole examples; (b)-(c)-(d) show three iterations of the proposed solution.

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 7(a) is 03. Further, note that the quadtree is an unbalanced structure, having more deep leaves where there are more edges. Thus, higher leaves correspond to larger areas, and deeper leaves correspond to smaller areas (since a cell split is created when a cell has more edges than a threshold). After identifying an orphan cell, the question is where to search for a cell containing an edge. The following Lemma applies:

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

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

For example, the parent centroid for the orphan cell 03 is the green point in Figure 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 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 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 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

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
```

4.3 Answering global overlay queries

Using the local overlay DCELs, we can easily compute the global overlay DCEL; for that, we need a reduce phase, described below, to remove artificial edges, and concatenate split edges from all the faces. Using the local overlay DCELs, we can also compute in a scalable way global operators like intersection, difference, symmetric difference, etc. For these operators, there is first a map phase that computes the specific operator on each local DCEL, followed by a reduce phase to remove artificial edges/added vertices. Figure 8 shows how the intersection overlay operator ($A \cap B$) is computed, starting with the local DCELs for four cells in Figure 8(a). First, each cell computes the intersection using its local overlay DCEL as shown in Figure 8(b). This is a map operation to identify overlay faces that contain both labels from layer A and

Algorithm 2 GETCELLSATCORNER algorithm

Require: a quadtree $\mathcal{Q}$ and a cell c.

```
function GETCELLSATCORNER ( Q, c )
    region ← quadrant region of c in c.parent
    switch region do
        case 'SW'
            corner ← left bottom corner of c.envelope
        case 'SE'
            corner ← right bottom corner of c.envelope
        case 'NW'
            corner ← left upper corner of c.envelope
        case 'NE'
            corner ← right upper corner of c.envelope
    cells ← cells which intersect corner in Q
    cells ← cells − c
    cells ← sort cells on basis of their depth
    return (cells, corner)
end function
```

layer B. Each cell can then report every such face that does not include any artificial edges, like face A_1B_1 in Figure 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 8(b) were combined into one face in Figure 8(c). While the extra vertex was also removed. In section 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.

5 Overlay evaluation optimizations

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.

5.1 Optimizations for faces spanning multiple cells

The naive reduce phase described above has the potential for a bottleneck since all faces, which can be a very large number, are sent to one worker node. From a distributed perspective, this process follows a typical MapReduce pattern. In the map

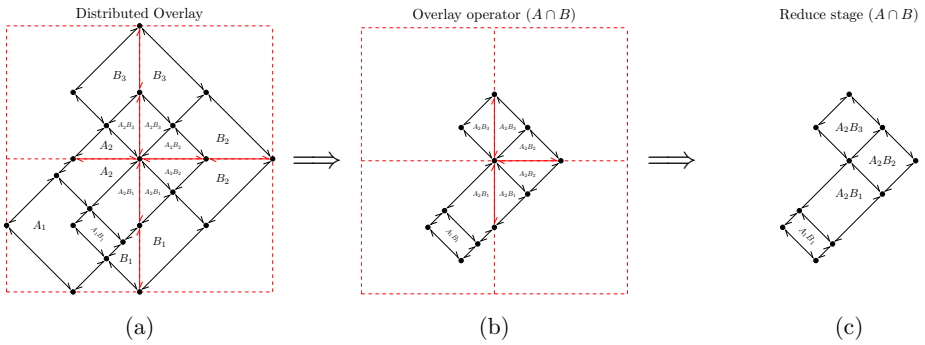


Fig. 8 Example of an overlay operator querying the distributed DCEL.

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 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.

5.2 Optimizing for unbalanced layers

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

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

An alternative approach is to scan the larger dataset only for the x-intervals where we know that there are half-edges in the smaller dataset. To do so, we order the two input sets separately. We scan the smaller dataset in x-order and identify x-intervals occupied by at least one half-edge. For each x-interval, we then scan the larger dataset using the sweep-line algorithm. This focused approach avoids unnecessary scanning of the large dataset, for example, areas with no half-edges from the smaller dataset.

6 Scalable Polygon Extraction for Line-based Input

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 [16] process. Such a feature enables spatial data scientists to seamlessly exploit a rich set of publicly available datasets, e.g., spatial road networks worldwide [37, 38].

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 [16], we proposed a scalable distributed framework to build a DCEL and extract

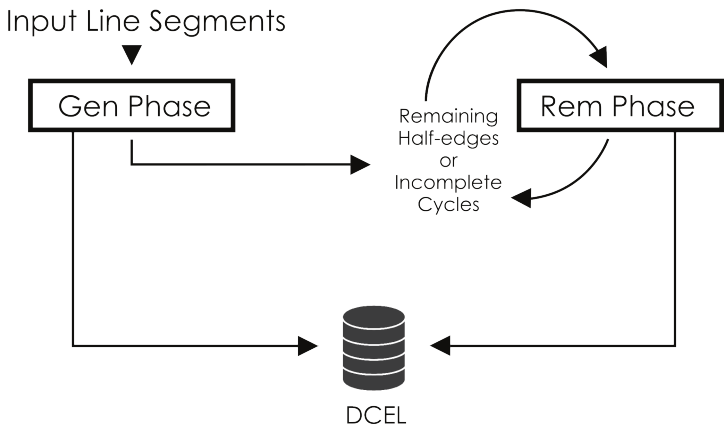


Fig. 9 DCEL Constructor for Polygonization Overview

polygons in parallel from the input line segments. Figure 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 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 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 6.2, generates the subdivision's remaining faces, F_j, $\forall j > 0$. Section 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 4 and 5. In section 6.4, we extend the overlay approaches to handle overlaying a polygon layer with the remaining edges (the dangle and cut edges).

6.1 Gen Phase

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 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

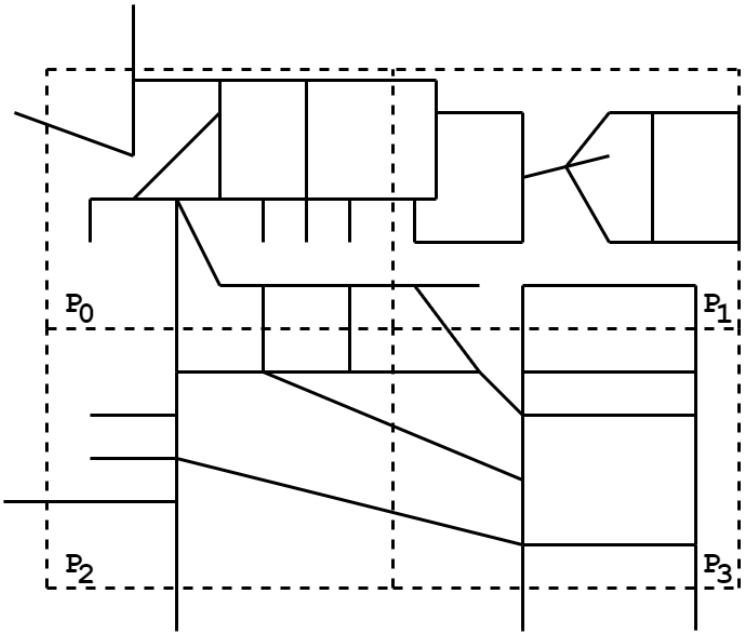


Fig. 10 Partitioned input spatial lines.

are 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 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 11

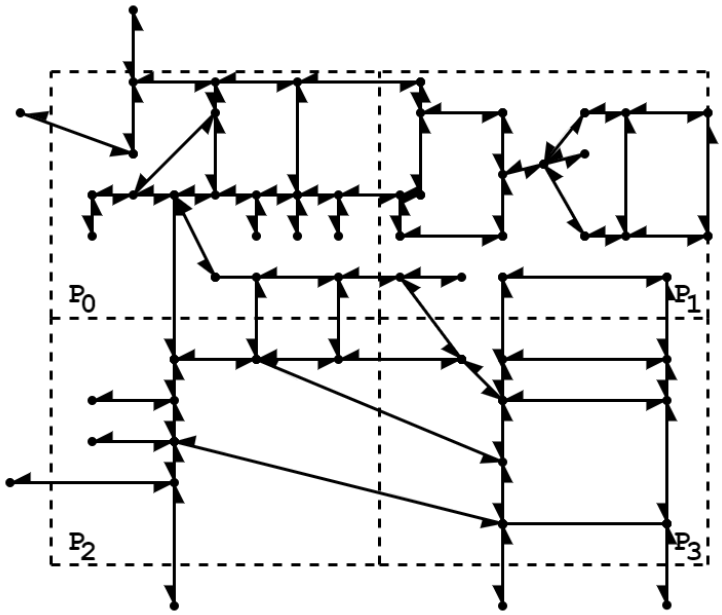


Fig. 11 DCEL vertices and half-edges.

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,

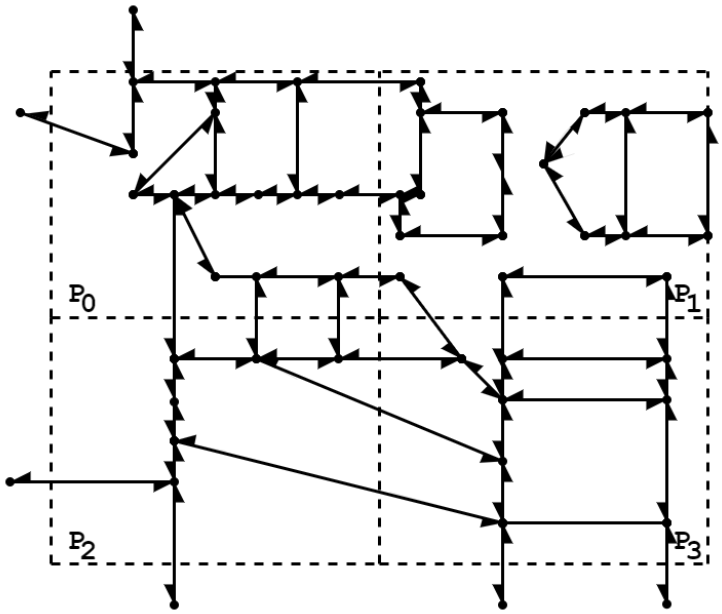


Fig. 12 DCEL vertices and half-edges after dangle and cut edge removal.

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 12 shows the DCEL data structure after removing the dangle and cut edges.

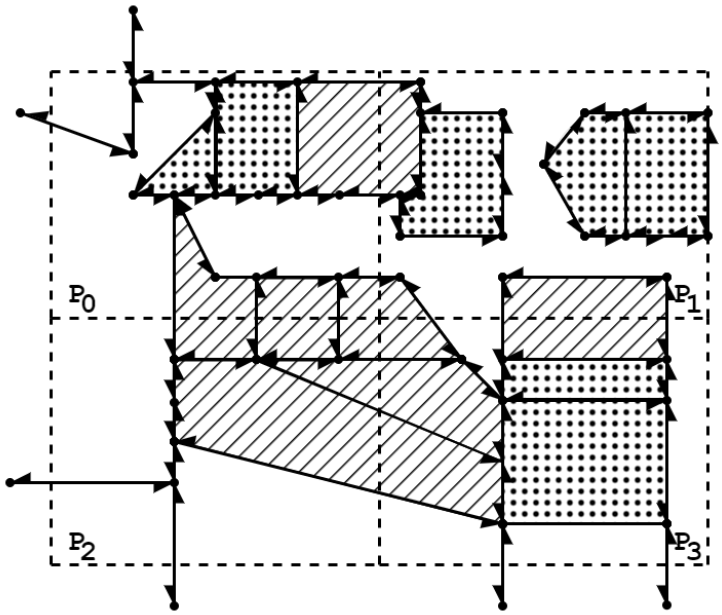


Fig. 13 DCEL faces.

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 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 $h.incidentF = f, \quad \forall h \in cycle$. (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 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 13, the dotted faces are the faces generated in this phase (Gen Phase).

6.2 Rem Phase

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 $h.incidentF = f, \quad \forall h \in cycle$. (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 iteration 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 13, the hatched faces are the faces generated in the first iteration ($j = 1$) of the Rem Phase.

6.3 Data Partitioning

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.

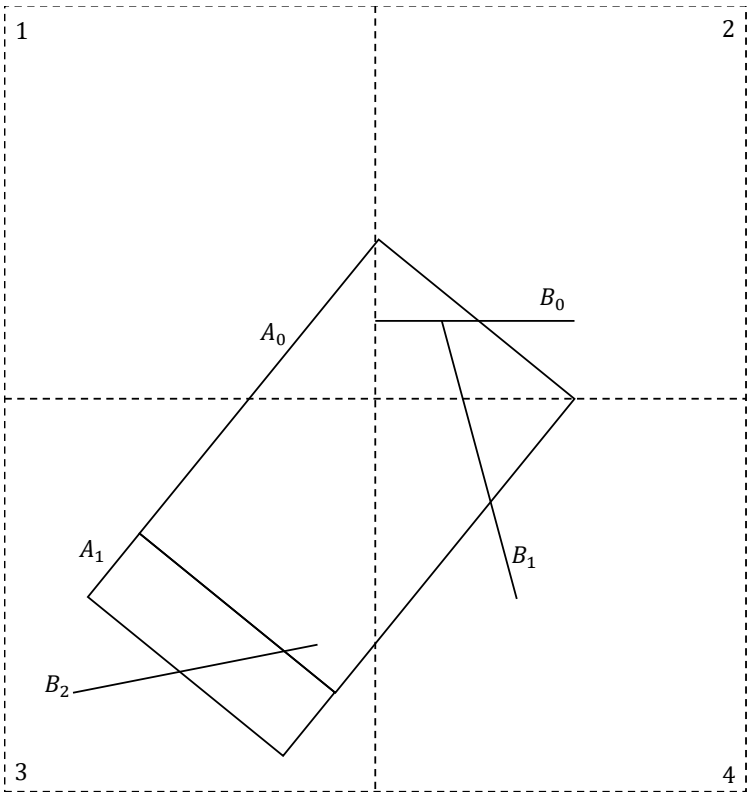


Fig. 14 Spatial partitioning of input layers A and B

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.
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.

6.4 Overlaying Polygons with Dangle and Cut Edges

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 4 and 5. However, we

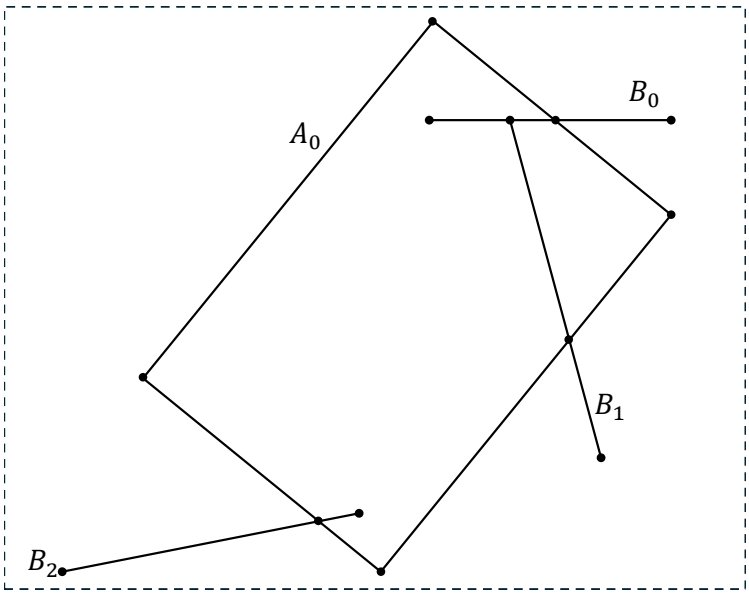


Fig. 15 Re-Partitioning of polygon A_0 with edges it intersects with

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 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., B_a, B_b, B_c, B_x, B_y. In Figure 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 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 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 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_01 and A_02.

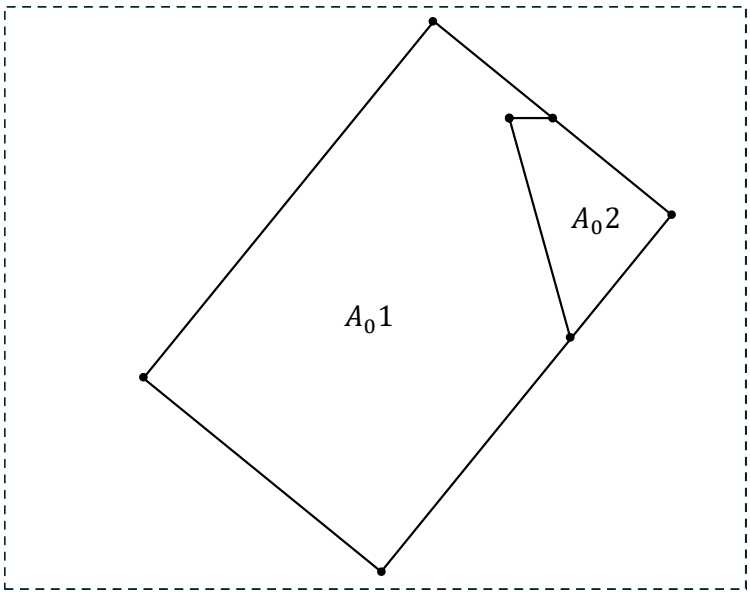


Fig. 16 The result of polygonization of A_0 with B_0, B_1, B_2

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

7 Experimental Evaluation

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

Evaluation datasets. The details of the real datasets of polygons that we use are summarized in Table 4. The first dataset (MainUS) contains the complete Census Tracts for all the states on the US mainland for the years 2000 (layer A) and 2010 (layer B). It was collected from the official website of the United States Census Bureau[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

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

Table 4 Evaluation Datasets

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

(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 [6].

The scalable approach was implemented over the Apache Spark framework. From a Map-Reduce point of view the stages described in Section 4 were implemented using several transformations and actions supported by Spark. For example, the partitioning and load balancing described in Section 4.1 and 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 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 5.

7.1 Overlay face optimizations

We first examine the optimizations in Section 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 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

Table 5 Percentages of edges in incomplete faces for three states

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

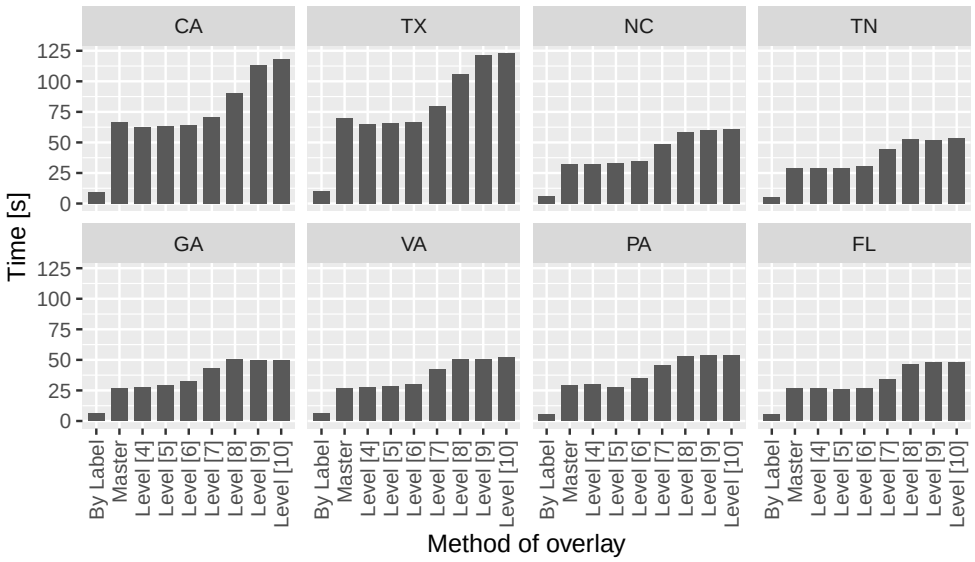


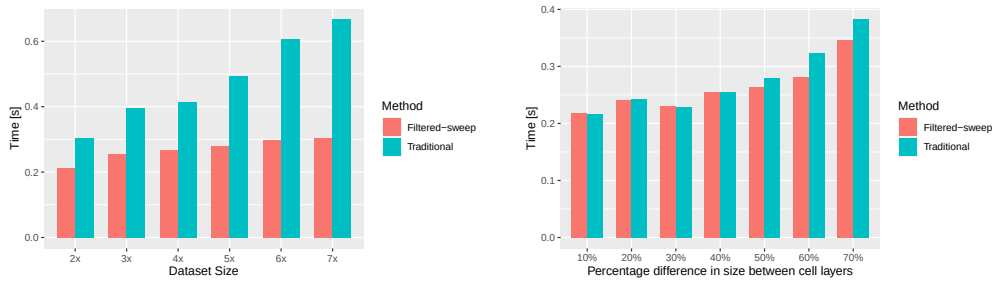
Fig. 17 Overlay methods evaluation.

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 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.

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 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



(a) (b)

Fig. 18 Evaluation of the unbalanced layers optimization.

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 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 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.

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.

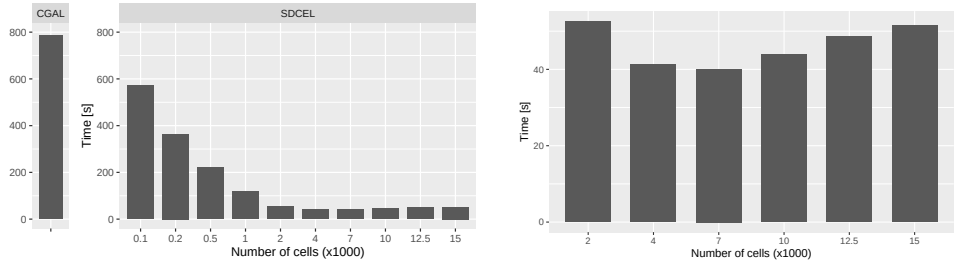


Fig. 19 SDCEL performance while varying the number of cells in the CCT dataset.

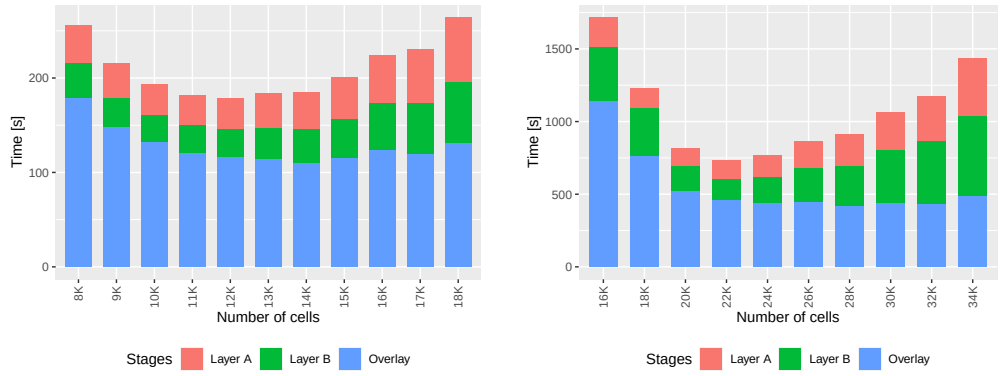
The number of cells affects the performance of our scalable overlay implementation, termed as SDCEL, since it relates to the average cell capacity given by the number of edges it could contain. As it was said before, a fewer number of cells implies larger cell capacity and thus more edges to process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

Figure 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 19(b) focuses on that area; the best SDCEL performance was around 7K cells.

In addition, Figure 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 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, 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



(a) (b)

Fig. 20 Performance with (a) MainUS and (b) GADM datasets.

Table 6 Cell size statistics.

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

Table 7 Orphan cells and orphan holes description

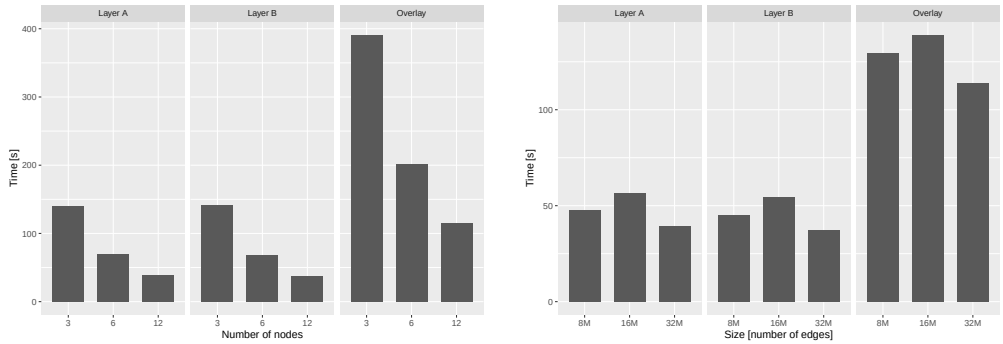
Dataset	Number of cells	Number of holes	Number of orphans (cell/holes)
GADM	21970	1999	4310
MainUS	12343	850	1069
CCT	7124	40	215

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 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 7 presents the number of cells, original holes, and the orphan cells and holes generated after partitioning.

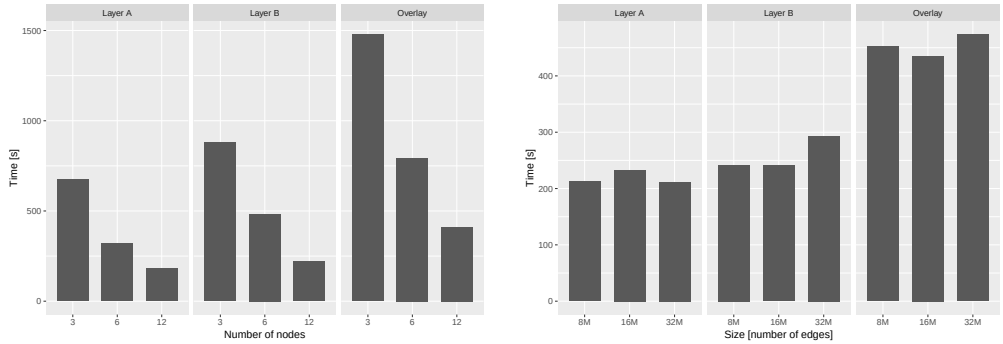
7.4 Speed-up and Scale-up experiments

The speed-up behavior of SDCEL appears in Figure 21(a) (for the MainUS dataset) and in Figure 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



(a) (b)

Fig. 21 Speed-up and Scale-up experiments for the MainUS dataset.

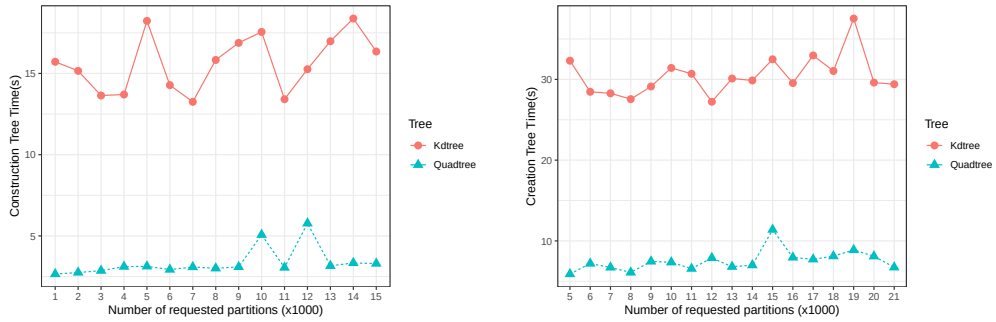


(a) (b)

Fig. 22 Speed-up and Scale-up experiments for the GADM dataset.

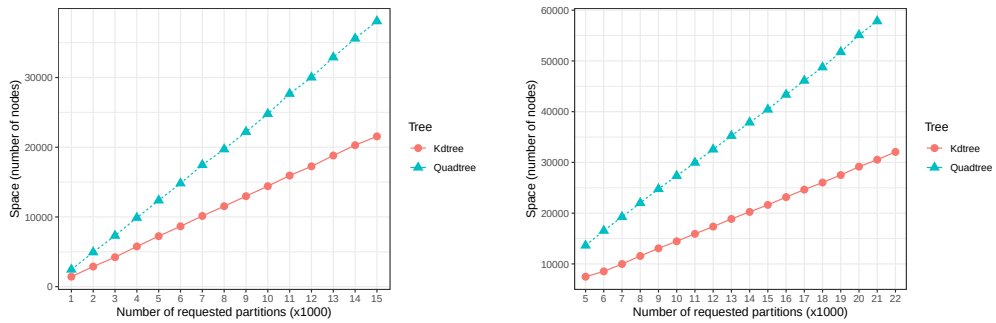
while keeping the input layers the same. Clearly, as the number of nodes increases, the performance improves. SDCEL shows good speed-up characteristics: as the number of nodes doubles from 3 to 6 and then from 6 to 12, the performance improves by almost half.

To examine the scale-up behavior, we created smaller datasets out of the MainUS and similarly out of the GADM so that we could control the number of edges. To create such a dataset, we picked a centroid and started increasing the area covered by this dataset until the number of edges was closed to a specific number. For example, from the MainUS, we created datasets of sizes 8M, 16M, and 32M edges for each layer. We then used two layers of the same size as input to a different number of nodes while keeping the input-to-node ratio fixed. That is, the layers of size 8M were processed using 3 nodes, the layers of size 16M using 6 nodes, and the 32M using 12 nodes. We used the same process for the scale-up experiments with the GADM dataset. The results appear in Figure 21(b) and Figure 22(b). Overall, SDCEL shows good scale-up performance; it remains almost constant as the work per node is similar (there are



(a) (b)

Fig. 23 Construction time for the spatial data structure in the (a) MainUS and (b) GADM datasets.



(a) (b)

Fig. 24 Number of cells created by each spatial data structure in the (a) MainUS and (b) GADM datasets.

slight variations because we could not control perfectly the number of edges and their intersection).

7.5 Kd-tree versus quadtree performance

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 4.

Figure 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 takes 23.13% the time it takes for Kdtree to be created (21.55% in MainUS

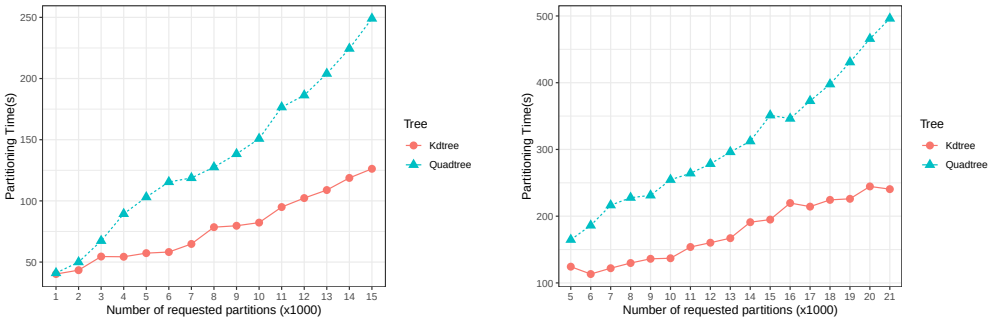


Fig. 25 Data partitioning time using a spatial data structure (a) in the MainUS dataset and (b) in the GADM dataset.

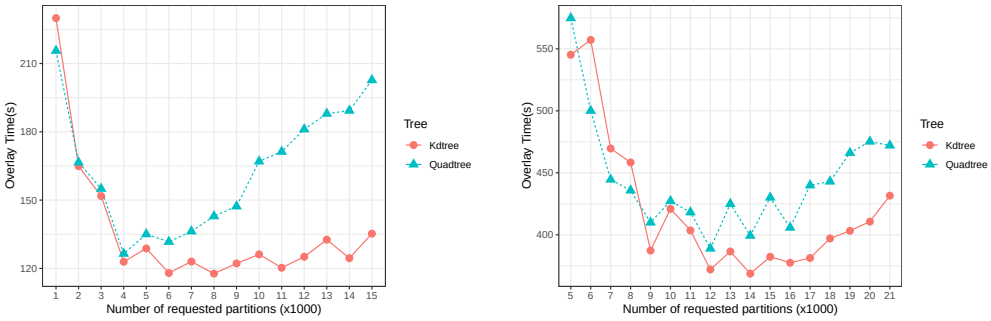


Fig. 26 Execution time for the overlay operation using a spatial data structure in the MainUS (a)and GADM (b) dataset.

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 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 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).

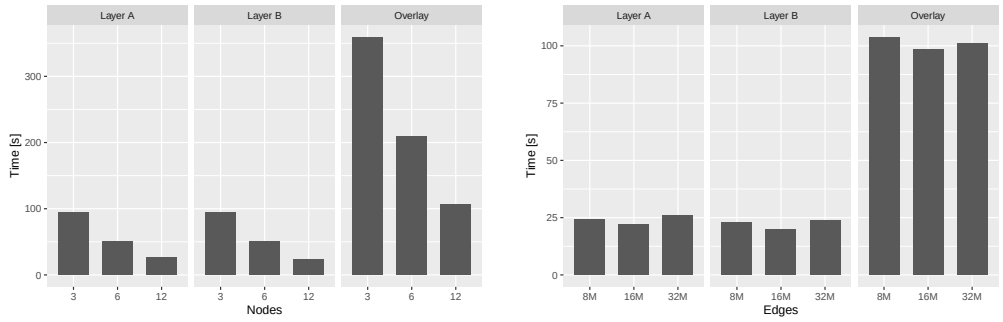


Fig. 27 (a)Speed Up and (b) Scale Up performance of the Kdtree partitioning using the MainUS dataset.

Table 8 Polygonization Evaluation Dataset

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	$557M$	$23M$

Once the data is assigned to their partitions, the overlay operation can be executed. Figure 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 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 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 27(b) shows the scale-up performance of the kd-tree partitioning approach. We followed the same procedure described in Section 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.

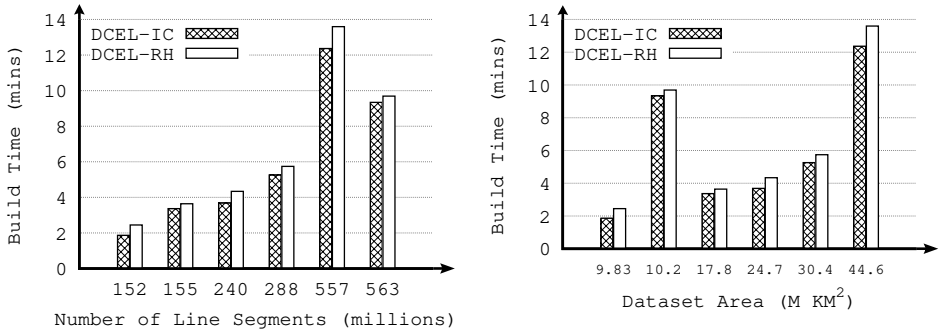


Fig. 28 Polygonization Performance on Real Road Networks.

7.6 Polygonization Scalability

Figure 28 evaluates the scalability of the polygonization approach using the different evaluation datasets summarized in Table 8. We implemented our polygonization framework on Apache Sedona [34]. 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 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 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 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 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 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*.

Fig. 29 Polygonization speed up evaluation using the USA dataset

Table 9 Overlaying Polygons with Dangle and Cut Edges Dataset

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

7.7 Polygonization Speed Up Evaluation

Figure 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.

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 6.4. Table 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 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

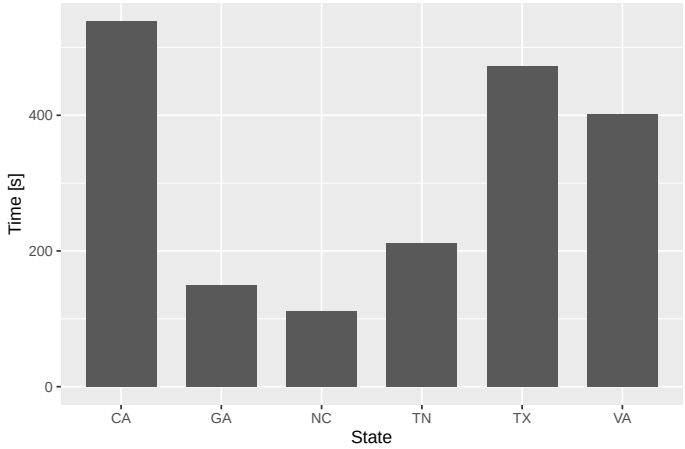


Fig. 30 Overlaying State polygons with dangle and cut edges.

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.

8 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.

Acknowledgements. We would like to thank Sergio Rey from the Center for Geospatial Sciences for introducing us to the Scalable DCEL problem.

Declarations

- Funding: This work was partially supported by the National Science Foundation under grants IIS-1901379, IIS-2237348, CNS-2031418, SES-1831615 and the Google-CAHSI research grant.

- Data availability: Data description and access is described in the experimental section. The sources of the dataset are publicly available and the references and cited accordingly.

References

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

[2] Nievergelt, J., Hinterberger, H., Sevcik, K.: The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst. **9**(1), 38–71 (1984)

[3] Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: ACM SIGMOD ICMD, pp. 47–57. Association for Computing Machinery, New York, NY, United States (1984)

[4] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: ACM SIGMOD PODS, pp. 322–331. Association for Computing Machinery, New York, NY, USA (1990)

[5] Finkel, R., Bentley, J.: Quadtrees: A data structure for retrieval on composite keys. Acta Inf. **4**, 1–9 (1974)

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

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

[8] Preparata, F., Shamos, M.: Computational Geometry: an Introduction. Springer, New York, NY (1985)

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

[10] O'Rourke, J.: Art Gallery Theorems and Algorithms. Oxford University Press, United States (1987)

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

[12] Fogel, E., Halperin, D., Wein, R.: CGAL Arrangements and Their Applications. Springer, Heidelberg (2012)

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

[14] Boguslawski, P., Gold, C., Ledoux, H.: Modelling and analysing 3D buildings with

a primal/dual data structure. ISPRS **66**(2), 188–197 (2011)

[15] Calderon, A., Tsotras, V., Magdy, A.: Scalable overlay operations over DCEL polygon layers. In: Proceedings of the 18th International Symposium on Spatial and Temporal Data. SSTD '23, pp. 85–95. Association for Computing Machinery, New York, NY, USA (2023)

[16] Abdelhafeez, L., Magdy, A., Tsotras, V.: DDCEL: Efficient distributed doubly connected edge list for large spatial networks. In: 2023 24th IEEE International Conference on Mobile Data Management (MDM), pp. 122–131 (2023)

[17] Barequet, G.: DCEL - a polyhedral database and programming environment. IJCGA **08**(05n06), 619–636 (1998)

[18] Boltcheva, D., Basselin, J., Poull, C., Barthélemy, H., Sokolov, D.: Topological-based roof modeling from 3D point clouds. In: WSCG, vol. 28, pp. 137–146. Union Agency, Science Press, CZ 301 00 Plzen (2020)

[19] Freiseisen, W.: Colored DCEL for boolean operations in 2D (1998)

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

[21] Holmes, R.: The DCEL Data Structure for 3D Graphics. University of Bremen (2021)

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

[23] Sabek, I., Mokbel, M.: On spatial joins in Map-Reduce. In: ACM SIGSPATIAL, pp. 1–10. Association for Computing Machinery, New York, NY, USA (2017)

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

[25] Franklin, W., Magalhães, S., Andrade, M.: Data structures for parallel spatial algorithms on large datasets. In: ACM BigSpatial, pp. 16–19. ACM, Seattle, WA, USA (2018)

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

[27] Puri, S., Prasad, S.: Efficient parallel and distributed algorithms for GIS polygonal overlay processing. In: IEEE IPDPS, pp. 2238–2241. IEEE Computer Society, USA (2013)

[28] Puri, S., Agarwal, D., He, X., Prasad, S.: Map-Reduce algorithms for GIS polygonal overlay processing. In: IEEE IPDPS, pp. 1009–1016. IEEE, Cambridge, MA, USA (2013)

[29] JTS Polygonizer Implementation. https://github.com/locationtech/jts

[30] GEOS Polygonizer Implementation. https://github.com/libgeos/geos

[31] Pandey, V., Renen, A., Kipf, A., Kemper, A.: How good are modern spatial libraries? Data Science and Engineering (2021)

[32] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.: Hadoop-GIS: A high performance spatial data warehousing system over Map-Reduce. In: VLDB Journal (2013)

[33] Eldawy, A., Mokbel, M.F.: Spatialhadoop: A Map-Reduce framework for spatial data. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE (2015)

[34] Yu, J., Zhang, Z., Sarwat, M.: Spatial data management in Apache Spark: the GeoSpark perspective and beyond. GeoInformatica (2018)

[35] You, S., Zhang, J., Gruenwald, L.: Large-scale spatial join query processing in cloud. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE (2015)

[36] Hoel, E.G., Samet, H.: Data-parallel polygonization. Parallel Computing (2003)

[37] OpenStreetMap Data Extracts. http://download.geofabrik.de/

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