2023 24th IEEE International Conference on Mobile Data Management (MDM) | 979-8-3503-4101-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/MDM58254.2023.00029

DDCEL: Efficient Distributed Doubly Connected Edge List for Large Spatial Networks

Laila Abdelhafeez
Computer Science and Engineering
Center for Geospatial Sciences
University of California, Riverside
labde005@ucr.edu

Amr Magdy
Computer Science and Engineering
Center for Geospatial Sciences
University of California, Riverside
amr@cs.ucr.edu

Vassilis J. Tsotras
Computer Science and Engineering
Center for Geospatial Sciences
University of California, Riverside
tsotras@cs.ucr.edu

***Abstract*—The Doubly Connected Edge List (DCEL) is a popular data structure for representing planar subdivisions and is used to accelerate spatial applications like map overlay, graph simplification, and subdivision traversal. Current DCEL implementations assume a standalone machine environment, which does not scale when processing the large dataset sizes that abound in today's spatial applications. This paper proposes a *Distributed* Doubly Connected Edge List (DDCEL) data structure extending the DCEL to a distributed environment. The DDCEL constructor undergoes a two-phase paradigm to generate the subdivision's vertices, half-edges, and faces. After spatially partitioning the input data, the first phase runs the sequential DCEL construction algorithm on each data partition in parallel. The second phase then iteratively merges information from multiple data partitions to generate the shared data structure. Our experimental evaluation with real data of road networks of up to 563 million line segments shows significant performance advantages of the proposed approach over the existing techniques.**

***Index Terms*—DCEL, Distributed, Polygonization**

I. Introduction

The *Doubly-Connected Edge List (DCEL)* is a popular data structure used to represent planar subdivisions. It is used in a wide variety of applications; for example, it is central in computing the overlay of planar subdivisions [9], which solves the map overlay problem. The DCEL is also used to represent Voronoi diagrams [7], [16], planar graphs [3], polyhedron [17], and TIN data [25]. It is further utilized in graph simplification [13], triangulation [4], [19], subdivision traversal [6], and topology manipulation [2].

Given an input spatial network represented by a set of line segments, the DCEL constructor generates and stores a record for each subdivision's vertex, half-edge, and face. A vertex in a subdivision is a node where two or more line segments meet, corresponding to a graph vertex of the spatial network. A half-edge is a line segment split along its length and has a directional component: an origin vertex and a destination vertex. Two opposite-direction half-edges (twin half-edges), where the origin of the first is the destination of the second and vice versa, represent each undirected line segment. So, each half-edge corresponds to a directed graph edge of the spatial network. A face of a subdivision is a polygonal region whose boundary is formed by the subdivision's vertices and half-edges with the same direction. The subdivision's vertices and half-edges can be extracted directly from the input network; the two endpoints of a line segment represent its two vertices, and each line segment (undirected edge) is represented by two half-edges (directed edges). Unlike extracting vertices and half-edges, extracting the subdivision's faces is not straightforward. To generate all of a subdivision's faces, the DCEL constructor invokes the *polygonization* procedure, which extracts all closed polygons formed by a collection of planar line segments in a subdivision. Figure 1 shows an illustrative example of a DCEL data structure. In this example, we have an input network of 10 line segments. The DCEL stores this input as a collection of 20 half-edges, 2 half-edges for each edge, a collection of 9 vertices, and a collection of 2 faces.

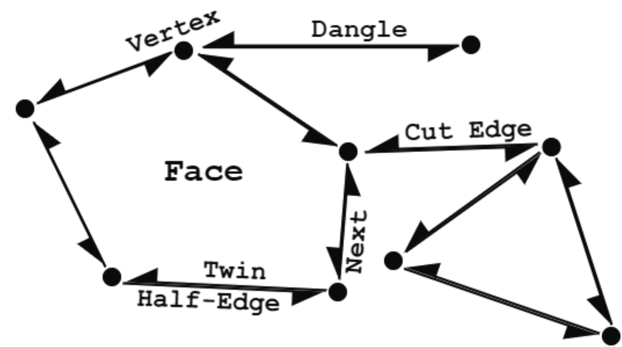


Fig. 1. DCEL Data Structure

The DCEL data structure has a great potential to facilitate efficient operations on mobility data that operate on spatial networks, e.g., road networks or river networks. For example, comprehensive maps are available in massive datasets thanks to the emergence of various map technologies such as Google Maps [12] and OpenStreetMap [20]. Because of the added benefits of the DCEL, storing these maps in DCEL objects will facilitate many applications, e.g., map overlay applications. Also, the polygonization of these maps provides means to generate new sets of polygons (faces of a DCEL)

This work is partially supported by the National Science Foundation, USA, under grants IIS-1901379, IIS-2237348, CNS-2031418, and SES-1831615.

2375-0324/23/$31.00 ©2023 IEEE
DOI 10.1109/MDM58254.2023.00029

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

that can be used in recommending and automating polygon queries, such as polygon range queries and spatial joins.

The size growth of modern spatial networks, e.g., fine-granular road network data, makes building the DCEL structure highly inefficient. The main performance bottleneck in DCEL construction is polygonization. The current implementations for the DCEL data structure assume a standalone machine environment [5], [18], which cannot scale up for large spatial networks. The polygonization is also supported in major GIS systems like QGIS [24], major spatial databases systems like PostGIS [23], and major python spatial libraries like Shapely [26]. However, these systems: (a) still assume a standalone machine environment that cannot scale up to large spatial networks, and (b) perform only the polygonization procedure but do not build a whole DCEL structure.

As we discuss in our experimental results, the execution of a sequential polygonization algorithm on PostGIS using the USA road network (152 million line segments) as an input breaks before producing results. As a result, sequential techniques are limited to analyzing small-scale or sparse networks. Splitting the workload into multiple batches is *non-trivial*. The polygonization procedure depends on multiple connected line segments, which means we need to split the input file in a way that maintains information about all faces found in the subdivision. To the best of our knowledge, the DCEL construction is currently not supported in distributed big-spatial data systems (such as GeoSpark, now Apache Sedona [29]).

To generate a DCEL object representing an input spatial network (set of line segments) on distributed big-spatial data systems, we face two main challenges: (1) First, the extraction of vertices and half-edges depends on one data record only, i.e., the line segment. The extraction process can be distributed directly along with the data records. However, in the case of the faces, one face depends on multiple data records, i.e., connected line segments. Such data records do not necessarily end up in the same data partition. (2) Second, to parallelize the polygonization procedure, data partitions need means to share data amongst each other. This data communication severely affects performance, especially since these data partitions do not necessarily reside in the same physical machine.

To overcome these challenges, we propose a novel *Distributed DCEL (DDCEL)* data structure extending the well-known DCEL to work in a scalable way. Like DCEL, the DDCEL data structure consists of three collections that store the subdivision's vertices, half-edges, and faces in a distributed way. Given an input spatial network, the DDCEL constructor's goal is to populate these collections. To achieve this goal, the constructor undergoes a two-phase paradigm: (1) Generate each partition DCEL (*Gen Phase*), in which the vertices and the half-edges collections are fully populated, whereas only a portion of the faces collection is generated. (2) Generate the Remaining Faces (*Rem Phase*), in which the polygonization procedure proceeds to generate the remaining faces.

During the *Gen Phase*, after spatially partitioning the input network into parallel data partitions, the constructor generates a *partition DCEL* on each data partition. The partition DCEL contains information about the vertices, the half-edges, and the faces of this given partition. Constructing the partition DCEL on each data partition generates all of the subdivision's vertices and half-edges. However, it does not generate all of the subdivision's faces; it only generates the faces that fit in each partition's minimum bounding rectangle. To generate the remaining faces, the constructor proceeds to the second phase.

The remaining faces are the faces that have a subset of their edges spanning multiple data partitions; hence the partition's subset of data is insufficient to generate them. The *Rem Phase* is iterative. Each iteration accepts a subset of the *remaining line segments*: input line segments that have not yet been assigned to a face. These line segments are re-partitioned into a new set of partitions, where data from multiple partitions are aggregated into a single partition. Using the new set of partitions, the constructor generates a new subset of faces, faces that fit in the new partition boundaries, appending them to the faces collection. Any remaining line segments are passed into the next iteration for further processing. The procedure terminates when we reach one of these cases: (1) there are no more remaining line segments, or (2) the remaining line segments are re-partitioned into one partition.

For the data partitioning and re-partitioning, we used a modified QuadTree [10] spatial partitioning found in Apache Sedona [29]. However, any non-overlapping spatial partitioning can be used in place of the QuadTree. We performed an extensive experimental evaluation with real-world-scale road networks. Our techniques have shown significant superiority over all existing techniques, supporting an order of magnitude larger datasets with an order of magnitude smaller query latency. Our contributions are summarized as follows:

1) We propose a novel Distributed Doubly-Connected Edge List (DDCEL) data structure extending the well-known DCEL data structure.
2) We implement the proposed DDCEL constructor to efficiently extract vertices, half-edges, and faces from large-scale spatial networks.
3) We perform an extensive experimental evaluation using large-scale spatial networks generated from OSM [20] ranging from 152 million to 563 million data records (line segments).

The rest of this paper is organized as follows: related work appears in Section II. The proposed data structure and the problem definition are formally defined in Section III. The constructor and the re-partitioning schemes are detailed in Sections IV and V, respectively. Sections VI and VII present the experimental evaluation and the conclusions.

II. Related work

All available implementations for the bottleneck polygonization procedure are built upon the JTS/GEOS implementation [11], [15]. While the JTS library is used in many modern distributed spatial analytics systems [22], including Hadoop-GIS [1], SpatialHadoop [8], GeoSpark [29], and SpatialSpark [28], the implementation of the polygonization algo-

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

rithm [15] 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 [14]. 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 can become a bottleneck when processing datasets of hundreds of millions of records on one machine. Our work differs in three main aspects: (a) first, the approach in [14] only works with directed line segments, whereas our approach is more generic and works with undirected edges, (b) second, our output is organized in a novel distributed DCEL data structure which is an extension of popular data structure used in a wide variety of applications, (c) third, while both approaches rely on iterative data re-partitioning, [14] 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).

III. The Data Structure and The Problem Definition

We start by introducing our novel data structure in Section III-A; while the problem's formal definition appears in Section III-B.

A. The DDCEL Data Structure

The Distributed DCEL (*DDCEL*) is a novel distributed version of the DCEL data structure introduced in Section I. The DDCEL data structure is a distributed graph. It consists of three collections (Apache Spark RDDs) analogous to the DCEL data structure, representing the subdivision's vertices, half-edges, and faces. An RDD [30] is an immutable, fault-tolerant, *distributed collection* of records partitioned across the cluster nodes that can be operated in parallel. The DDCEL consists of a vertices RDD (V), a half-edges RDD (H), and a list of faces RDDs (F).

Each vertex object $v \in V$ is defined by:

1) its spatial location; i.e. its coordinates ($v.coordinates$), and
2) its incident half-edges ($v.incidentH$); i.e., a list of the half-edges arriving at the vertex v.

A half-edge object $h \in H$ is defined by:

1) its origin vertex ($h.origin$),
2) its destination vertex ($h.destination$),
3) its twin half-edge ($h.twin$),
4) its next half-edge ($h.next$) to walk around a face in counterclockwise order,
5) the face it is bounded to ($h.incidentF$); note that an input line segment usually bounds two faces but viewing the different sides of a line segment as two distinct half-edges means that each half-edge bounds only one face; and
6) whether it spans multiple partitions ($h.spansMP$).

Dangles are the half-edges with one or both ends not incident on another half-edge endpoint. *Cut-Edges* are the half-edges connected at both ends but do not form part of a polygon. Examples of the dangles and cut-edges are shown in Figure 1. Both dangles and cut-edges are particular types of half-edges since they are not bounded to any face ($h.incidentF = null$). Two markers are added to each half-edge h:

7) $h.isDangle$
8) $h.isCutEdge$

These markers are added to exclude them from the face generation (polygonization procedure).

The list of faces contains m RDDs (F_0, F_1, ..., F_j, ..., F_m), such that each F_j $\forall j$ is an RDD and m is the total number of iterations executed to generate all of the subdivision's faces.

F_0 contains the set of faces generated at the first phase (*Gen Phase*). The remaining $F_j \in F$ $\forall j > 0$ represents the subset of faces generated at the jth iteration of the *Rem Phase*.

Each face object $f \in F$ is defined by a cycle of the half-edges forming it $List(h_1, h_2, h_3, ..., h_1)$.

B. Problem Definition

Consider an input spatial network N that consists of line segment objects. Each line segment object $o \in N$ is defined by two points (p_1, p_2); each point p is defined by $< lat, long >$, where $< lat, long >$ represents the latitude/longitude coordinates of the point's location in the two-dimensional space. Formally, constructing a Distributed Doubly-Connected Edge List (DDCEL) is defined as follows: Given an input spatial dataset N, the constructor generates and stores all of the network's vertices, half-edges, and faces in a $DDCEL$ data structure. An example would be: given the road network of the United States of America, extract all road information, i.e., road crossings (vertices), roads (half-edges), and faces (neighborhood blocks), and store them in a DDCEL data structure.

IV. The DDCEL Constructor

We proceed with Section IV-A, which presents an overview of the DDCEL constructor, followed by Sections IV-B and IV-C, which discuss the details of the constructor's two main phases; the *Gen Phase* and the *Rem Phase*.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

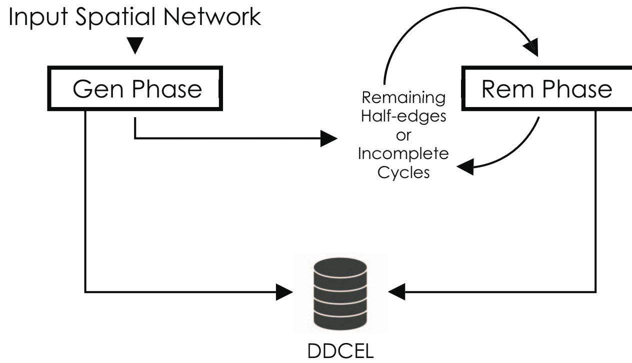


Fig. 2. DDCEL Constructor Overview

A. Overview

Figure 2 shows an overview of the DDCEL constructor. To create a DDCEL data structure from an input spatial network N, the *DDCEL constructor* undergoes a two-phase paradigm. The *Gen Phase*, detailed in Section IV-B, handles spatially partitioning the input network, 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 IV-C, handles generating the subdivision's remaining faces ($F_j,\ \forall j > 0$).

B. 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 3 shows an example of four leaf nodes of a QuadTree built for an input spatial network. Solid lines represent the network line segments, and dashed lines represent the partitions' MBRs.

After spatially partitioning the input network, each partition generates its vertices, half-edges, and faces (collectively the partition DCEL) using the subset of the dataset that intersects with the partition's MBR. The *partition vertices* are the vertices that are 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 DDCEL data structure. Each partition P_i receives a subset of the input dataset that

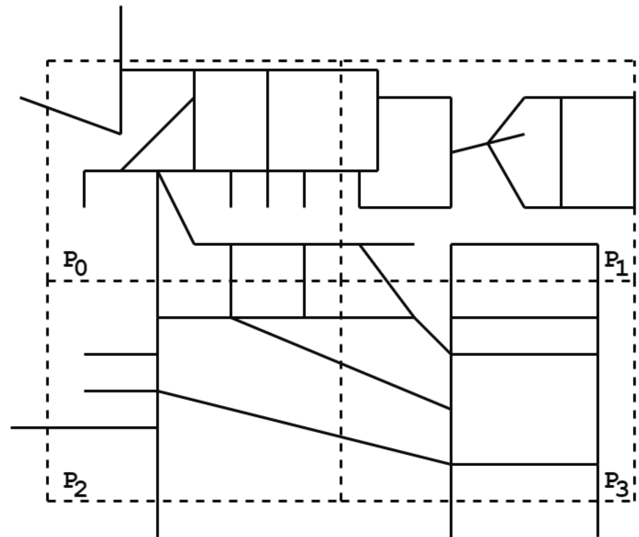


Fig. 3. Partitioned Input Spatial Network

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 DDCEL 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 DDCEL data structure. Figure 4 shows a graphical illustration of the DDCEL data structure representing the input network 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 network, 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-

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

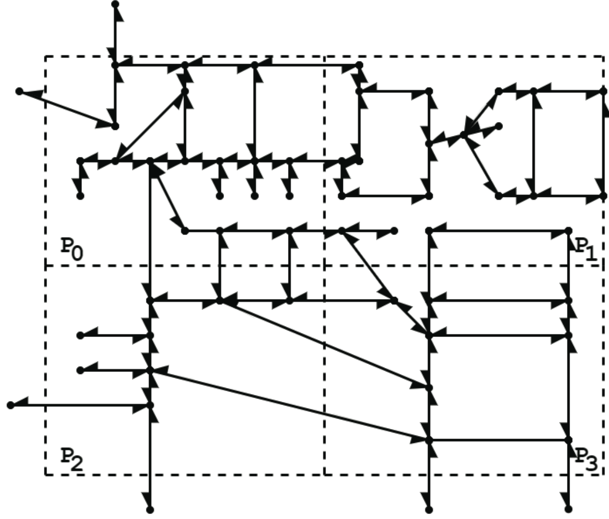


Fig. 4. DDCEL Vertices and Half-Edges

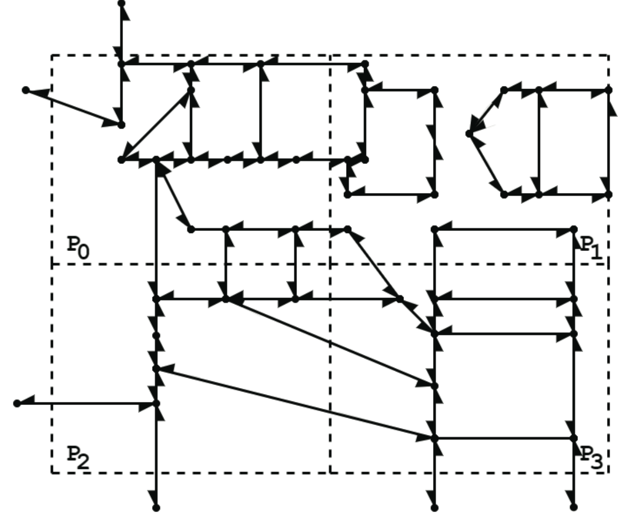


Fig. 5. DDCEL Vertices and Half-Edges After Dangle and Cut-Edge Removal

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

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

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

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

In the third sub-step, after marking all cut-edges, we update the next pointers while excluding the cut-edges. For each vertex $v \in V$, we sort its incident half-edges list in clockwise order again, now while excluding both the dangle and the cut-edge half-edges. After sorting the incident half-edges list $v.incidentH$, we re-execute the same process of the first sub-step, assigning $v.incidentH[t].next$ to $v.incidentH[t+1].twin$. Figure 5 shows the DDCEL data structure after removing the dangle and the cut-edge half-edges.

Step 4: Generating the Partition Faces.

Polygonization on each partition P_i starts with selecting an arbitrary half-edge as our initial half-edge $h_{initial}$. We initially assign our $h_{current}$ half-edge pointer to $h_{initial}$. We advance the $h_{current}$ pointer at each iteration to the $h_{current}$'s next ($h_{current} = h_{current}.next$), storing all visited half-edges in a list $cycle$. We keep advancing the $h_{current}$ pointer till we reach one of the following cases: (1) We return to the initial half-edge $h_{initial}$, which means that we have found a face. In this case, we add the found face f to the faces collection F_0 and assign $h.incidentF = f, \ \forall h \in cycle$. (2) The $h_{current}.next$ is not available, and $h_{current}$ is a half-edge that spans multiple partitions. In this case, the cycle needs more information from the neighboring partitions to be completed, and the current partition's data is insufficient to produce this face. To complete this cycle, we either pass the incomplete cycle into the Rem phase (the current list $cycle$), where it collects all incomplete cycles from all partitions and attempts to join them to form a face. Another approach would be passing the plain half-edges in this cycle to the next phase. Both approaches are discussed in detail in Section IV-C. Once we finish processing this cycle, we mark all visited half-edges as such, clear the cycle, and

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

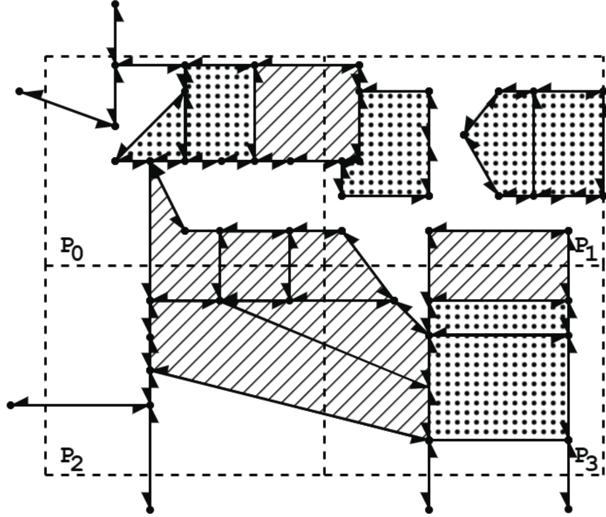


Fig. 6. DDCEL Faces

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 6, the dotted faces are the faces generated in this phase (Gen Phase).

C. 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 $h.incidentF = f, \ \forall h \in cycle$. (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 6, the hatched faces are the faces generated in the first iteration ($j = 1$) of the Rem Phase.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

V. Data Partitioning

For the input spatial network partitioning, we used a spatial QuadTree structure because of its ability to adapt to high skewness (common in several spatial networks) by adapting the tree depth in different spatial areas based on the network density. Given a parameter *capacity* that defines how many data points (line segments) are allowed within a QuadTree partition, the QuadTree partitioner starts by inserting the whole network into the root tree node. If the node capacity is exceeded, it 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 parameterized *capacity*. This standard mechanism divides spatial areas with high data densities into deeper tree levels, while sparse areas result in shallow tree depth. The optimal goal is to hold an equal data load in each partition, which balances the distributed query processing time when this data is processed for incoming queries. Nevertheless, generating a tree using all the data is expensive for enormous input data sets with hundreds of millions of records and requires a powerful master node. Apache Sedona [29] offers data sampling, where only a sample of the data is used to generate the partitions. These partitions are then used to partition the whole dataset.

The QuadTree partitioner is used 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. In the experimental evaluation section, we compare different schemes for the visited QuadTree levels. The different schemes used are:

1) Going directly to the root node at $l = 0$ after the leaf nodes, i.e., visiting only levels L in the Gen and 0 in the Rem phases. However, the experimental evaluation shows that collecting the data after the Gen phase on one node is prohibitive, and one worker node will not be able to process the Gen phase's output.
2) Going 1 Level Up (1LU) each iteration, i.e. if we visit level l at iteration j, we go to level $l-1$ at iteration $j+1$. This means the Rem Phase visits all the QuadTree levels resulting in L iterations.
3) Going 2 Levels Up (2LU) each iteration resulting in half the number of iterations $\frac{L}{2}$ compared to 1LU.
4) Skipping to the Middle of the tree at level $\frac{L}{2}$, then continue going 1 level up for the remaining levels (M1LU), which will also result in $\frac{L}{2}$ iterations.
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.

TABLE I
Evaluation Datasets

Dataset	Area	Size	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

VI. Experimental Evaluation

This section provides an extensive experimental evaluation of our work. Section VI-A describes the experimental setup. Section VI-B verifies that previous approaches do not scale for large datasets. Sections VI-C through VI-F evaluate our DDCEL constructor, including constructor scalability, re-partitioning schemes, global index tuning, and speedup.

A. Experimental Setup

We implemented our framework on Apache Sedona [29]. All DDCEL experiments are based on Java 8 implementation and use a Spark 2.3 cluster of a dual-master node and 12 worker nodes. All nodes run Linux CentOS 8.2 (64bit). Each master node is equipped with 128GB RAM, and each worker node is equipped with 64GB RAM. The total number of worker executors on the Spark cluster is 84, each with 4 GB of memory and an additional executor for the driver program. Our evaluation datasets are road networks extracted from Open Street Maps of the United States of America [27] and continents worldwide [21]. Statistics of the evaluation datasets and the number of generated DDCEL faces are shown in Table I. Unless mentioned otherwise, the default QuadTree partition capacity for USA and South America datasets is 15K line segments, for North America and Africa 17K line segments,

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

and for Asia and Europe 25K line segments. The default re-partitioning scheme is *MU* for all evaluation datasets.

B. Comparison with Previous Approaches

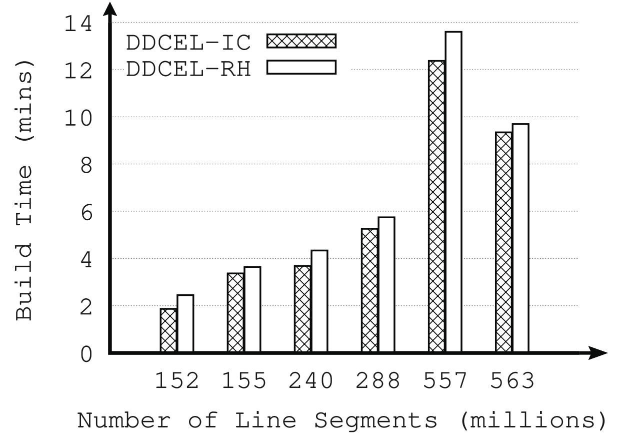
As mentioned in the introduction, sequential approaches do not scale when dealing with large network sizes. Running the sequential polygonization algorithm on a 64GB RAM machine with 1.8TB disk space using the USA road network as an input breaks before producing results. For example, PostGIS was able to process only up to 20 Million edges out of the 152 Million edges of the USA road network dataset (13% of the dataset) in around four hours of execution time.

We then tested the parallel approach introduced in [14] on a distributed environment setting (Spark cluster), using the USA road network dataset (152 Million edges). The input dataset is undirected edges, while [14] only works with directed edges. As a workaround, we defined each input undirected edge as two directed edges (similar to half-edges). The input data is spatially partitioned using a QuadTree onto around 35K partitions, with a QuadTree depth of 13. In each merging operation, the input dataset is re-partitioned into a smaller number of partitions till we reach a single partition. As we reach the 9th iteration, the processing breaks down after the data is re-partitioned into 112 partitions. This means that the original input dataset of the 152M edges is being partitioned into only 112 partitions after it was initially partitioned into 35K partitions. This means that, on average, each data partition goes from handling around 4K edges at the first iteration to handling around 1.3M edges in the 9th iteration. Processing this huge chunk of the input data on a worker node eventually becomes prohibitive.

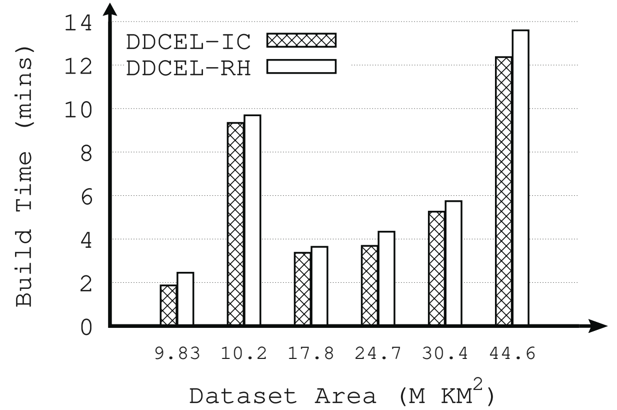
C. DDCEL Constructor Scalability

Figure 7 evaluates the scalability of *DDCEL* constructor using the different evaluation datasets. As discussed in Section IV, 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 *DDCEL-RH*. In comparison, the second approach processes the incomplete cycles generated from the first phase iteratively denoted as *DDCEL-IC*.

From Figure 7, 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 7(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 7(b). Hence the build time of the Europe dataset (10.2M 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 *DDCEL-IC* beats *DDCEL-RH*.



(a) Varying Number of Line Segments



(b) Varying Dataset Area

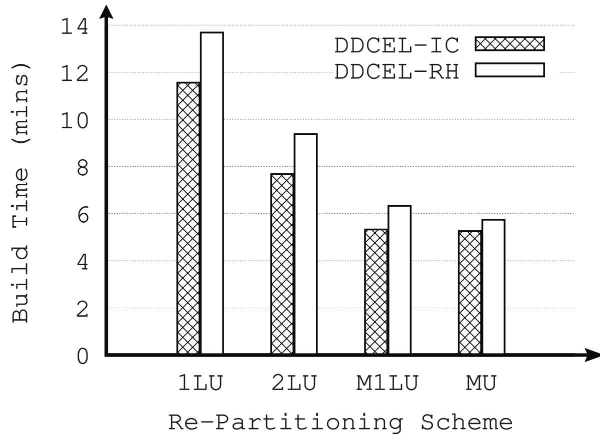
Fig. 7. DDCEL Constructor Performance on Real Road Networks.

D. Re-partitioning Schemes Evaluation

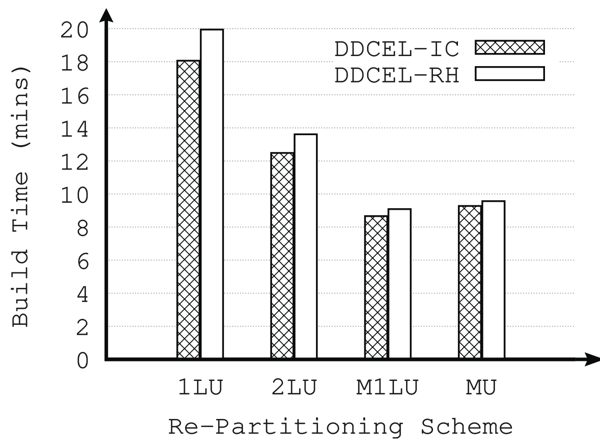
Figure 8 shows the performance of the different re-partitioning schemes discussed in Section V over two of the evaluated datasets, namely Africa and Europe road network datasets. It confirms the previous finding; *DDCEL-IC* beats *DDCEL-RH* in all evaluated re-partitioning schemes for both datasets. *1LU* is the worst performing re-partitioning scheme since there are relatively more iterations where each iteration generates only a small subset of the remaining faces. Also, the input data skew leads to an unbalanced QuadTree. Going a level up from the leaves does not guarantee that all nodes are merged to a parent node. This cause to have some idle partitions, the partitions that are already processed and not yet merged, at earlier iterations.

MU or *M1LU* are the best-performing approaches since both have a minimal number of iterations; $\frac{L}{2}$ and $\log_2(L)$ iterations, respectively, while maintaining parallelism by ensuring data is re-partitioned to enough multiple partitions. Jumping to nodes at higher levels in the QuadTree gives two major advantages;

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

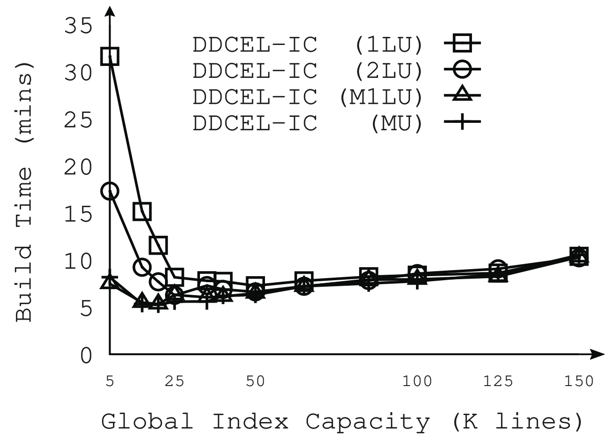


(a) Africa Dataset

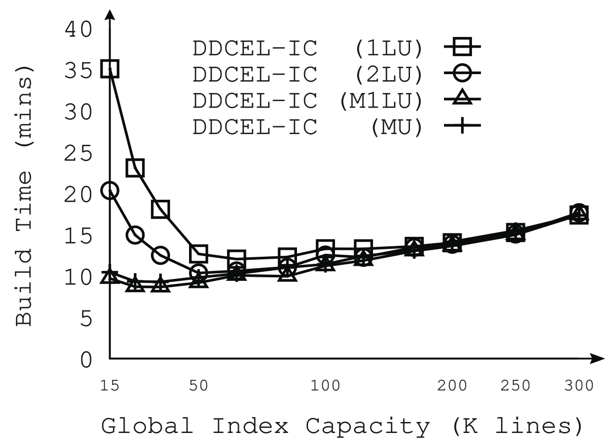


(b) Europe Dataset

Fig. 8. Varying Re-Partitioning Schemes.



(a) Africa Dataset



(b) Europe Dataset

Fig. 9. Varying the Initial Partition Capacity.

first, these nodes have larger areas that can fit larger faces, and second, it minimizes or completely eludes the idle partitions. However, the extreme approach of going straight to the root after the Gen phase decreases the parallelism and assumes that one worker node is able to generate the remaining faces, which is not the case. The experiments of this approach terminated without producing any results due to memory issues.

E. Global Index Tuning

Figure 9 shows the effect of varying the initial partition capacity on two of the evaluated datasets, Africa and Europe datasets, using the IC approach with varying partitioning schemes. The number of partitions has an inverse correlation with the partition capacity. As we increase the partition capacity, the number of partitions decreases. Parallelism is at its highest at smaller partition capacities (i.e., more data partitions are processed in parallel). However, too many partitions result: (1) too many iterations since the depth of the QuadTree is directly correlated with the number of partitions, and (2) fewer faces generated per iteration since more faces will not fit in one partition anymore. So we need to find an optimal partition capacity that gives us enough partitions to ensure parallelism but not too many partitions that become an overhead on the system. Figure 9 shows that 17K partition capacity works best for the Africa dataset, whereas 25K partition capacity works best for the Europe dataset.

Figure 9 gives also two major insights: (1) We confirm that *MU* and *M1LU* perform better than *1LU* and *2LU*. (2) For almost all techniques, the build time starts very high with low partition capacity, which confirms that too many partitions are overhead from the system's point of view. With increasing the partition capacity, the build time drops significantly for *1LU* and *2LU*. However, the drop is less noticeable for *MU* and *M1LU* because *MU* and *M1LU* already (a) have less number of iterations and (b) skip the levels with too many partitions (deeper levels in the QuadTree). The build time increases again after the drop in build time because we now have a small number of partitions, and insufficient data is processed

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.

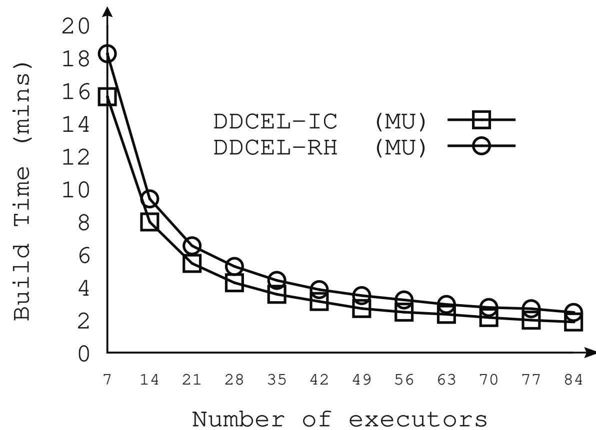


Fig. 10. Speedup Evaluation using the USA Dataset

in parallel, thus, decreasing the query parallelism.

F. Speedup Evaluation

Figure 10 shows the effect of increasing the number of executors on the build time for the USA dataset. 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.

VII. Conclusions

This paper proposes a Distributed Doubly-Connected Edge-List (DDCEL) data structure, a novel distributed extension to the traditional DCEL data structure. The DDCEL data structure is a distributed graph implemented by three RDDs corresponding to the subdivision's vertices, half-edges, and faces. To construct a DDCEL for an input spatial network of line segments, the constructor undergoes two phases: the Gen and Rem Phases. In the Gen Phase, the input data is spatially partitioned across a cluster of worker nodes. In each data partition, the Gen Phase generates the partition's subset of vertices, half-edges, and faces. The Gen Phase generates all of the input's vertices and half-edges. However, it cannot generate faces with half-edges spanning multiple partitions. Through multiple iterations, the Rem Phase generates any remaining faces by re-partitioning the remaining half-edges or the incomplete cycles into a smaller set of partitions that aggregate information from multiple older partitions. We used a QuadTree partitioning in both the initial data partitioning and in the Rem Phase iterative re-partitioning. In the initial data partitioning, we always use the leaf nodes of the QuadTree as our partition boundaries. We propose various schemes for choosing the other tree levels to visit in the Rem Phase iterations, and we compare them. The experimental evaluation shows significant superiority of our approaches compared to the existing techniques to support large-scale spatial networks.

References

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System Over MapReduce. In *VLDB Journal*, 2013.
[2] Ergun Akleman, Jianer Chen, and Vinod Srinivasan. A New Paradigm for Changing Topology Of 2-Manifold Polygonal Meshes. In *Pacific Graphics*, 2000.
[3] Tatsuya Akutsu, Colin de la Higuera, and Takeyuki Tamura. A Simple Linear-Time Algorithm for Computing the Centroid and Canonical Form of a Plane Graph and Its Applications. In *Annual Symposium on Combinatorial Pattern Matching*, 2018.
[4] David Avis. Generating Rooted Triangulations Without Repetitions. *Algorithmica*, 1996.
[5] DCEL Python Implementation. https://github.com/anglyan/dcel, 2015.
[6] Mark De Berg, René Van Oostrum, and Mark Overmars. Simple Traversal of a Subdivision without Extra Storage. *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, 1996.
[7] Mark Theodoor De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Science & Business Media, 2000.
[8] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A Mapreduce Framework For Spatial Data. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2015.
[9] Ulrich Finke and Klaus H Hinrichs. Overlaying Simply Connected Planar Subdivisions In Linear Time. In *Symposium On Computational Geometry*, 1995.
[10] Raphael Finkel and Jon Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974.
[11] GEOS Polygonizer Implementation. https://github.com/libgeos/geos.
[12] GoogleMaps. https://www.google.com/maps.
[13] Alexis Gourdon. Simplification of Irregular Surfaces Meshes in 3D Medical Images. In *Computer Vision, Virtual Reality and Robotics in Medicine*, 1995.
[14] Erik G Hoel and Hanan Samet. Data-parallel polygonization. *Parallel Computing*, 2003.
[15] JTS Polygonizer Implementation. https://github.com/locationtech/jts.
[16] Menelaos I Karavelas. Voronoi diagrams in CGAL. In *European Workshop on Computational Geometry*, 2006.
[17] Nilanjana Karmakar, Arindam Biswas, and Partha Bhowmick. Fast Slicing Of Orthogonal Covers Using DCEL. In *International Workshop on Combinatorial Image Analysis*, 2012.
[18] Lutz Kettner. Halfedge Data Structures. In *CGAL User and Reference Manual*. CGAL Editorial Board, 2022.
[19] Mingzhao Li, Zhifeng Bao, Farhana Choudhury, Hanan Samet, Matt Duckham, and Timos Sellis. AOI-shapes: An Efficient Footprint Algorithm to Support Visualization of User-defined Urban Areas of Interest. *ACM Transactions on Interactive Intelligent Systems*, 2021.
[20] OpenStreetMap. https://www.openstreetmap.org/.
[21] OpenStreetMap Data Extracts. http://download.geofabrik.de/.
[22] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. How Good Are Modern Spatial Libraries? *Data Science and Engineering*, 2021.
[23] PostGIS. https://postgis.net.
[24] QGIS Geographic Information System. https://www.qgis.org.
[25] Eric Saux, R'emy Thibaud, Ki-Joune Li, and Min-Hwan Kim. A New Approach For A Topographic Feature-Based Characterization of Digital Elevation Data. In *Proceedings of ACM GIS*, 2004.
[26] Shapely: Manipulation and Analysis of Geometric Objects. https://github.com/Toblerity/Shapely.
[27] U.S. Street Network Shapefiles, Node/Edge Lists, and GraphML Files. https://doi.org/10.7910/DVN/CUWWYJ.
[28] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale Spatial Join Query Processing In Cloud. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2015.
[29] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial Data Management in Apache Spark: the GeoSpark Perspective and Beyond. *GeoInformatica*, 2018.
[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on November 29,2023 at 23:15:30 UTC from IEEE Xplore. Restrictions apply.