Fast Proximity Graph Generation with Spark

Arjun Subramanyam Varalakshmi*
Chong WANG*
arjunsv3691@gmail.com
cwang27@uh.edu
Dept. of Computer Science, the University of Houston
Houston, Texas

Christoph F. Eick
Dept. of Computer Science, the University of Houston
Houston, Texas
ceick@central.uh.edu

ABSTRACT

Since the early 1980s, proximity graphs have served as one of the classical approaches to characterize neighborhood relationships; the most popular proximity graphs include Delaunay Triangulation (DT) and Gabriel graphs (GG). These graphs find their applications in geographical analysis, spatial analysis, pattern recognition, evolutionary biology, computer vision, cluster analysis, and visualization. The emergence of big data has created a need for scalable algorithms to generate proximity graphs for massive datasets. In this paper, we propose a novel approach for creating DT and GG by leveraging the cluster computing capabilities of Apache Spark. To compute DT, we rely on the divide and conquer paradigm. We first partition the spatial dataset into a grid; next, we compute the DT for each grid partition and separate the resulting triangles into the core and boundary triangles; next, we merge the adjacent partitions recursively and recompute boundary triangles; finally, we return the union of obtained triangles as the final result. GG generation is implemented on top of a DT, by removing edges from the DT not intersected with their corresponding Voronoi edge. We evaluate our proximity graph generation algorithms built on top of Spark to create DT and GG for a benchmark of datasets having between one and thirty million data records and compare our framework with existing Python and R libraries to create DT and GG. Our benchmark tests showcase significant performance improvements(2 times on DT and up to 24 times on GG) and the capability for creating DTs and GGs for datasets consisting of thirty million records.

*Both authors contributed equally to this research.

BigSpatial'19, November 5, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6966-4/19/11…$15.00
https://doi.org/10.1145/3356999.3365465

CCS CONCEPTS

• **Computing methodologies** → **Point-based models**; **Distributed programming languages**;

KEYWORDS

proximity graphs, distributed proximity graph generation, Delaunay Triangulation, Gabriel graph, Spark, big data

ACM Reference Format:
Arjun Subramanyam Varalakshmi, Chong WANG, and Christoph F. Eick. 2019. Fast Proximity Graph Generation with Spark. In *8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial'19), November 5, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3356999.3365465

1 INTRODUCTION

In spatial analysis, proximity or closeness measurement is a very useful operation in understanding several geographic phenomena, such as epidemic spread, geographic variation, and spatial clustering. It finds applications in hotspot detection, cluster analysis, and boundary detection. In proximity graphs, a set of data points are represented as vertices, and edges are used to describe closeness. In general, a proximity graph for a set of points, S, is a connected graph, in which two vertices a, and b are connected by an edge (a, b), if and only if no other points from the point set S are present within the exclusion region R (a, b).

The importance of measuring proximity is not limited to the field of spatial analysis; its significance extends to various domains in which extracting structural information from datasets is essential, such as computer vision, pattern recognition, social network analysis, bioinformatics, wireless sensor networks, and urban planning. Proximity graphs provide an efficient way of measuring proximity and establishing neighborhood relationships among data points. DT and GG have been successfully used for mesh generation, hotspot discovery and agglomerative clustering [5–7].

Multiple algorithms have been proposed to compute proximity graphs. Guibas et al.[1] propose a sequential algorithm to create DTs; in this approach, the dataset is split into two sets by drawing a line. Next, the DT is computed locally for each set, and these sets are merged along the splitting line. Chen et al. [2] propose a parallel partitioning approach. But

this implementation lacks in-memory processing, scalability, and fault tolerance capabilities for large datasets. Gabriel et al. [3] introduce a new proximity graph called a Gabriel graph and discuss its applications in the study of geographic variation analysis. Matula et al. [4] propose a sequential algorithm that creates the GG by removing edges from the DT of the dataset.

As datasets grow larger and larger traditional data analytics algorithms become unsuitable to cope with such data volumes. The traditional proximity graph generation approaches for DT and GG can only handle a limited amount of data. To overcome those shortcomings, this paper proposes novel, scalable algorithms to create DT and GG utilizing the high-performance computing capabilities of Apache Spark(Scala on Spark, 2.2.0) and Geotrellis package(1.2.0).

In our approach, we propose a configurable *capacity threshold* which approximates the maximum number of points for which DT can be computed with a single CPU core. If the dataset size is larger than the *capacity threshold*, we partition the dataset into a grid, assign each grid cell to a processor/core, and post-process and merge local DTs to obtain the global DT for the dataset. Our approach also makes use of the existing DT generation algorithm from the Geotrellis package—which is based on Guibas's algorithm [1]—, for computing triangles locally. For each grid cell we identify two types of triangles: core triangles and boundary triangles. In post-processing, we modify the boundary triangles and return the union of core and new boundary triangles as the final DT. Our approach creates a GG by removing edges from a DT using Matula et al. [4] algorithm.

The main contributions in this paper:

- We address these shortcomings of previous researches by leveraging the cluster computing, scalability, and fault tolerance capabilities of Apache Spark.
- We propose a novel, and grid-based algorithm that constructs the global DT from local DTs—which have been created for a simple grid cell—, by reconstructing boundary triangles of neighboring grid cells.
- We allow the user to choose the *capacity threshold* to control number of Spark partitions and partition grid size, which gives the user the flexibility to fine tune performance.
- Our proximity graph generation framework for DT and GG can handle very large amount of data with millions of data points and outperforms existing DT generation Python and R libraries. For example, our framework are able to create the DT with thirty million data points.
- Our GG algorithm leverages the parallel construction of DT to generate GG, and as far as we know, no other implementations of GG used Spark.

The remainder of the paper is organized as follows: Section 2 introduces the background. Section 3 discusses our approach to create proximity graphs in detail. In Section 4, we benchmark and evaluate our algorithms using a benchmark of massive datasets. Section 5 provides an overview for the related works. Section 6, we summarize and have a conclusion about the research.

2 BACKGROUND

Spatial Framework and Library: Apache Spark

Spark is one of the most popular open-source clustering computing frameworks which supports large scale distributed processing. Spark is much faster (up to 100 times) than other popular cluster computing frameworks such as Hadoop MapReduce, OpenMP, etc. Spark extends Map-Reduce processing model and much more processing capabilities such as in-memory processing, stream processing, interactive queries, high scalability and fault tolerance. Spark is built on top of Master-Worker architecture. The two important components of Spark are a Driver process running on the Master node and Executor processes running on multiple worker nodes, shown as Fig. 1. Spark supports horizontal scaling, as well as vertical scaling [12].

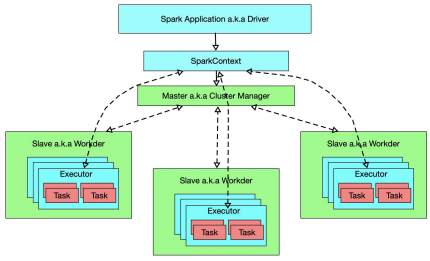

Figure 1: Spark architecture.

Spark achieves fault tolerance and fast processing through a primary abstraction namely Resilient Distributed Dataset (RDD), an immutable collection of elements distributed across multiple nodes of the cluster, which supports parallel in-memory processing. Two types of data operations can be performed on RDD's, namely Transformations and Actions [12]:

- Transformations: apply a function and return new RDD; transformations in Spark are lazily evaluated, i.e. not computed until action is called on the RDD.
- Actions: trigger computation of transformations and return a value.

The workflow of a Spark application can be summarized as follows [12]:

(1) Spark Driver runs on the Master node and groups set of tasks within the application into stages and create a

acyclic graph of stages. These stages are transformed into a set of tasks and scheduled for execution on the worker nodes.
(2) Driver requests Cluster Manager to assign executors on the worker nodes, Executors running on each worker node register with the Driver and run the task assigned to them by performing in-memory processing and computation.
(3) The number of Executor processes running on the worker node can be one or more. Similarly, a single Executor can run one or more tasks concurrently.
(4) The Cluster Manager contains information about all the resources across the compute nodes that together form the cluster. It is responsible for resource management, monitoring and scalability in the cluster.

Proximity Graph Concepts

Delaunay Triangulation. The DT, shown as in Fig. 2, is a widely studied triangulation technique that has applications in image segmentation, mesh generation, and terrain modeling. Triangulation is the process of partitioning the set of vertices into triangles, such that two triangles either share an edge or vertex; otherwise they are disjointed. Triangulations are used for determining the location of a point in a plane.

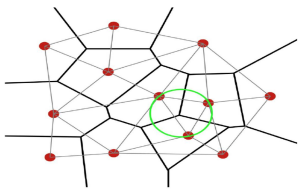

Figure 2: Delaunay triangulation.

A DT partitions the two-dimensional plane into triangles such that the circumcircle drawn for every triangle does not include any other points other than the vertices of the triangle. It maximizes the minimum angle of all the triangles' angles.

Voronoi Diagram. A Voronoi diagram of a set of 'sites' (points) is a collection of regions that divide the plane. Each region corresponds to one of the sites, and all the points in one region are closer to the corresponding site than to any other site [9]. A plane containing n points yields in the formation of n convex polygon and has n generating points enclosed within a Voronoi diagram. A Voronoi diagram is the dual of Delaunay triangulation [9], i.e. Delaunay triangulation can be obtained by joining the enclosed point within the convex polygon of a Voronoi diagram. Fig. 3 shows a Voronoi diagram in which Voronoi edges are represented using red lines, and Delaunay edges are represented using black lines.

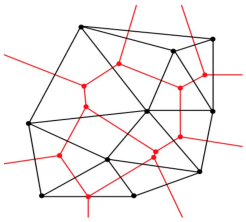

Figure 3: Voronoi Diagram.

Gabriel Graph. A GG [3] is a proximity graph that is used for geographical analysis, ecological studies, genotype analysis, and hotspot detection [3], [7]. GG is formed by connecting only those edges for which the disc, drawn with an edge as the diameter, contains no other point within the disc.

The GG is a subgraph of DT, and it can be constructed by removing edges from DT. The GG provides an efficient way to represent the neighborhood relationship with its low edge-to-vertex ratio: a GG with n vertices has at most $3n$ edges [4].

3 METHODOLOGY

Distributed Delaunay Triangulation Generation

The previous implementations of DT lack in memory processing, scalability, and fault tolerance capability for large dataset. In our implementation, we address these shortcomings by leveraging the cluster computing, scalability, and fault tolerance capabilities of Apache Spark and GeoTrellis. Our algorithm processing paradigm is decided during runtime based on a configuration parameter called *capacity threshold*. A *capacity threshold* is the number of points that belongs to a single partition and can be processed by one CPU core. If the size of a dataset is less than a *capacity threshold*, we compute DT sequentially, using a single executor process. For larger datasets whose size is greater than the *capacity threshold*, we compute DT using a distributed algorithm. Our distributed algorithm consists of the following steps: 1. preprocess point dataset; 2. partition the data into a grid; 3. compute core and boundary triangles for every grid cell; 4. merge neighboring grid-cells and post-processing boundary triangles; 5. compute final result. We will explain the five steps in the remaining of the section.

Preprocess point dataset. In this step, data is loaded into Spark, and the following data wrangling operations are performed:

(1) We define the data schema and load data into Spark using the defined schema.
(2) Data records containing null values for latitude and longitude are filtered and a Coordinate object is created for non-null values.

(3) We define a new ADT named Vertex. Every Vertex object has two fields: A Coordinate object, that contains point (x, y) value; and a unique vertex id, i.e. Vertex (ID, Coordinate [latitude, longitude]).
(4) To utilize Spark distribution feature, transformation is applied on data to create RDD as RDD [Vertex]. RDD [Vertex] is cached in memory for faster computation.

Partition the data into a grid. This is an important and novel step in our algorithm. In this step, we partition the vertex objects into a grid. The size of the grid is decided by a configuration parameter called *capacity threshold.* A *capacity threshold* is the number of data (vertex) objects that can be processed in one partition or a single CPU core. We first compute the size of the grid using the formulas(1)–(3):

$$approxNumOfGridCells = \frac{totalNumOfPoints}{capacityThresholdSize} \quad (1)$$

$$numOfRows = Maths.abs(\sqrt{approxNumOfGridCells}) \quad (2)$$

$$numOfColumns = \frac{approxNumOfGridCells}{numOfRows} \quad (3)$$

Once the size of the grid is computed, we have to compute the boundaries of each grid cell. In order to speed up this process, we compute the boundaries of the grid by taking a data sample from the whole dataset. The size of the data sample is computing using formula (4).

$$pt2Sample = \min(totalPoints, 1000 \times numOfGridCells) \quad (4)$$

We sort the sampled values and separate the x-coordinate and y-coordinate values from the sampled data and compute the x-splits and y-splits, as shown in Algorithm 1. X-splits values represent the vertical partition boundaries (partition across longitude) and y-split represent the horizontal partition boundaries (partition across latitude), we obtain these boundaries by calling SplitPoints function once for x-coordinate and y-coordinate values respectively. Steps 4 to 13 in Algorithm 1 partition both x-coordinate and y-coordinate values between the range [-180,180], as these are the minimum and maximum values for latitude and longitude. Step 15 of Algorithm 1, we return array containing x-splits and y-splits.

Once we obtain the x-split and y-split values, we compute the spatialKey for every point. SpatialKey can be defined as the grid cell to which the point should belong. It contains (row Index, column Index) of the grid. Assuming we have x-splits as (-180, 47.6, 48.5, +180) and y-splits as (-180, 85.1, 85.6, +180) from Algorithm 1, the pair of neighboring split values form the grid boundaries, Now, if we have to compute a spatialKey for a point with coordinates (48.0, 86.0), the x-coordinate value of this point is greater than 47.6 and less than 48.5, hence the row index for this point is 1, and the y-coordinate value is greater than 85.6 and less than +180. Therefore, the column index for this point is 2, and the spatialKey for this point is (1,2). To facilitate load balancing, our approach uses an irregular grid: its grid cells usually have different size.

Algorithm 1: Grid Boundary Computation

Input : data point[-180,180] lists as coordinateValues(CV), and number of splits as $numSplits$
Output : split points in the data points (x)

```
1  SplitPoints (CV[], numSplits);
2  splitArr = [ ]
3  for splitIdx := 0 to numSplits do  Find Grid
   Boundaries loop
4      arrIdx=(splitIdx / numSplits) × CV.length;
5      if arrIdx == 0 then
6          splitArr[splitIdx] = −180;
7      end
8      else if arrIdx == CV.length then
9          (splitArr[splitIdx] = +180;
10     end
11     else
12         splitArr[splitIdx] = CV[arrIdx − 1];
13     end
14 end
15 return splitArr
```

Fig. 4 summarizes the process of partitioning the point data into a grid. Next, we create an object called SpatialPoint for each point in the dataset containing the following information:

- pointID: unique id assigned to the point
- spatialKey: the grid cell (row id, col id) to which the point belongs
- coordinates: the x, y coordinates of the point

Finally, we assign all the SpatialPoint objects to their respective grid cells.

Compute core and boundary triangles for every grid cell. In this step, we compute the bounding box for every partition, such that all points within the partition are enclosed by it. In our case, the bounding box is a convex hull. To compute DT within each partition, we make use of the function provided by the GeoTrellis package. Geotrellis is a Scala library and framework built on top of Apache Spark to support faster processing of large spatial datasets [13]. The triangles computed in each partition are shown in Fig. 5. Once the DT is

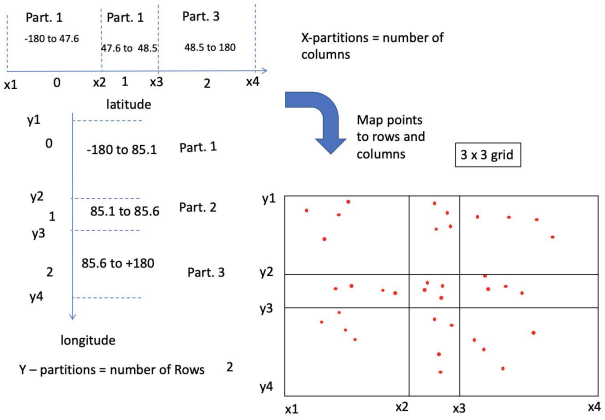

Figure 4: Partitioning data into a grid.

computed, we have to separate triangles in every partition into core triangles and boundary triangles:

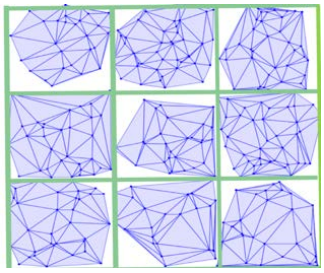

Figure 5: Delaunay triangulation in each partition.

- Boundary triangles: the triangles that either have at least one vertex on the convex hull or share a vertex with the triangles, that have a vertex on the convex hull. These triangles may change when two neighboring partitions are merged. We compute boundary triangles using the Boundary-Delaunay function in the GeoTrellis vector package.
- Core triangles: the triangles that do not change when two neighboring partitions are merged. We obtain core triangles by removing the boundary triangles from the set containing all DT triangles in the partition.

We pass the computed DT and bounding box of a grid cell to the boundary Delaunay function provided by the Geotrellis package to identify boundary triangles. This function returns the boundary triangles. The left side of Fig. 6 shows the boundary triangles, and the right side of Fig. 6 shows the core triangles obtained for a grid with four partitions.

Merge neighboring grid-cells and post-processing boundary triangles. The boundary triangles obtained from the previous step contain the following information:

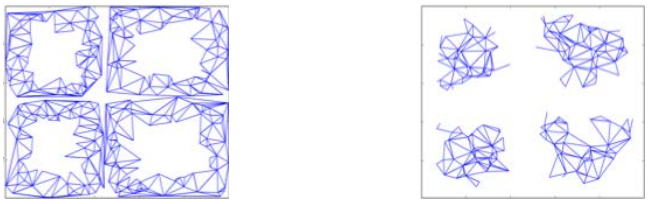

Figure 6: Left: Boundary triangles. Right: Core triangles.

- Points: a map of point id and coordinates of points forming boundary triangles
- halfEdgeTable: a list of half-edge values
- triangleMap: a map of triangle vertices and one of the half-edges of that triangle
- boundaryEdgeIndex: id of an edge that lies on the outer boundary of the structure
- isLinear: a Boolean variable, to indicate collinearity of points
- extent: a bounding box covering the grid cell

A half-edge, also popularly known as Doubly Connected Edge List, is a special data structure used to represent embedding of a planar graph in the plane. Half-Edge provides an efficient way to link vertex records, edge records, and face records together. Half-Edge stores three pieces of information: a vertex reference, a complementary half edge, and a reference to the next edge [18]. Extent is a data structure provided by the GeoTrellis vector package. It is a rectangular boundary that encloses a set of points in a 2D plane. Extents are defined using two coordinate pairs, i.e. "min" and "max" corners in the coordinate reference system [14]. In our DT algorithm, we use the extent to compute the bounding box for a grid cell.

We pass the boundary triangles and the spatial boundary details for creating the DT of unions of grid-cells from the respective DT; the code for this step is given in Algorithm 2. We merge the neighboring partitions and re-compute the boundary triangles along the boundary separating the two partitions. Assume we have 4 partitions in which boundary triangles are separated as shown in Fig. 6 left side figure, and boundary triangles for those cells are stored as follows:[(0,0), boundaryTriangles in (0,0)],[(0,1), boundaryTriangles in (0,1)] ,[(1,0), boundaryTriangles in (1,0)],[(1,1), boundaryTriangles in (1,1)].

As shown in Algorithm 2, based on the direction parameter, we decide whether we should be merging horizontally or vertically, in our algorithm we first merge horizontally and then when we have one column we merge them vertically, we merge neighboring partitions. As shown in steps 5-7 of Algorithm 2, while merging vertically, we divide the y-coordinate by 2, and two neighboring partitions will have same Spatial Key. We group two grid cells or partitions with same spatial

key, recompute and merge the boundary triangles as steps 15-16 of Algorithm 2.

Algorithm 2: REDUCE GRID PARTITIONS

Input : RDD[SpatialKey, set of boundary triangles as bdts], number of partitions, x/y direction
Output: new Delaunay triangles obtained after merge

1 ReduceGrid
  $(RDD[SpatialKey, bdts], numCells, direction)$
  while $numCells > 1$ **do**
2 **if** $direction ==' x'$ **then**
3 $(SpatialKey, bdt) = (SpatialKey(\frac{spatialKey.xi}{2}, spatialKey.yi), bdt)$;
4 **end**
5 **else**
6 $(SpatialKey, bdt) = (SpatialKey(spatialKey.xi, \frac{spatialKey.yi}{2}), bdt)$;
7 **end**
8 **if** $numCells\%2 == 0$ **then**
9 $numCells = \frac{numCells}{2}$;
10 **end**
11 **else**
12 $numCells = \frac{numCells}{2+1}$;
13 **end**
14 **end**
15 Group partitions by spatial key
16 For two gridcells with same spatial key
  RecomputeAndMerge(bdtLeft, bdtRight)
17 return new Boundary Deleaunay Triangles

[(0,0/2),boundaryTriangles was in (0,0)]=> [(0,0), boundaryTriangles now in (0,0)]

[(0,1/2),boundaryTriangles was in (0,1)]=>[(0,0), boundaryTriangles now in (0,0)]

Once we have only one column, we merge vertically.

[(0/2,0),boundaryTriangles was in (0,0)]=>[(0,0), boundaryTriangles now in (0,0)]

[(1/2,0),boundaryTriangles was in (1,0)]=>[(0,0), boundaryTriangles now in (0,0)]

We recompute the boundary triangles in the merged partitions. We combine the half-edge tables of the DTs in neighboring cells with *Geotrellis*. The right side of Fig. 7 shows the final boundary triangles obtained after merging the partitions shown in the left side of Fig. 7.

Compute final result. Once the boundary triangles are recomputed, we take a union of boundary triangles and core

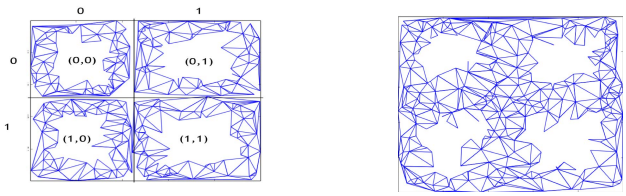

Figure 7: Left: Boundary triangles before the merge. Right: Boundary triangles after the merge.

triangles to obtain the global DT. Fig. 8 represents the final result obtained after merging the core triangles and the recomputed boundary triangles for the partitions shown in Fig. 8. In Fig. 8, the red lines represent the recomputed edges of boundary triangles, whereas the purple lines represent the original edges of the boundary triangles, and the blue lines represent edges of core triangles.

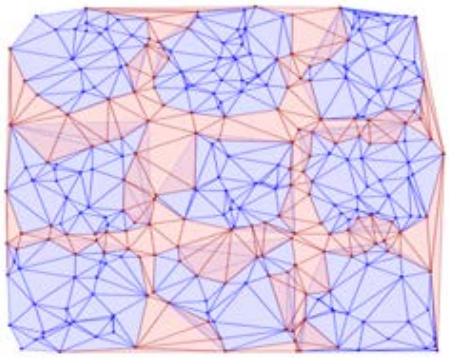

Figure 8: Delaunay triangulation after the merge.

Gabriel Graph Generation

In this section, we describe how Gabriel graphs are generated. We create the DT first, and then use Matula et al. [4] algorithm to compute the GG from the DT by removing edges from the DT. The algorithm first finds the circumcenter for each triangle. Once we determine the circumcenter of each triangle, we associate the circumcenter with the edges of the triangles to which it belongs. In DT, all non-boundary edges are shared by two neighboring triangles, hence, these edges have two circumcenters associated with them. These two circumcenters associated with an edge are its corresponding Voronoi edge coordinates. For non-boundary edges, if the Voronoi edge intersects only with its corresponding Delaunay edge, then that edge satisfies the properties of a Gabriel edge, and such edge is added to the Gabriel edges list.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate our methodology by conducting two different experiments. We first benchmark our algorithms for massive datasets. In this experiment, we use a Chicago crime dataset containing 851,223 unique records as the base dataset and generate larger datasets from this dataset by adding modified examples to the original dataset. In the second experiment, we compare our DT algorithm against a popular Python library named Scipy, and we compare GG algorithm against two popular R libraries, namely spatial dependence (Spdep) and Class Cover Catch Digraph (CCCD). We use two different datasets for this comparison: mobile activity data, and Chicago speeding violations data. We exam our new fast proximity graph generation framework performance compared with Python and R libraries on different sizes of datasets.

Our experiments are conducted using an Amazon EMR cluster consisting of one master node and 3 or 7 worker nodes, which are 4 or 8 nodes in total. All nodes are of m3-xlarge instance type, having the following configurations: four virtual CPU cores, 10 GB RAM, and 80 GB Storage.

Scalability Analysis for a Chicago Crime Dataset

The benchmark is carried out using Amazon EMR and a Chicago crime dataset [15] as a base dataset. This dataset consists of data records representing the crime incidents in Chicago from 2001 until 2018. It contains 6.58 million data records with the locations and other information regarding crime incidents. We identify the neighborhood relationship between crime locations using our DT and GG algorithms. Data points used by our algorithm are formed by the combination of latitude and longitude of a crime location. There are 851,223 unique data points in the base dataset, and we use only those in our experiments. For the purpose of benchmarking our algorithm with a larger dataset, we exposed the examples of the dataset to uniform random noise and generated three larger datasets by adding newly generated examples, such that Dataset-1⊆ Dataset-2⊆Dataset-3⊆ Dataset-4.

Benchmark results. We conducted four sets of experiments for DT and GG using the datasets listed in Table 1. Dataset-1 is our base dataset, and other datasets are generated from it to conduct the performance test. Moreover, it holds:

Tables 2-5 depict the results for the DT and GG. The semantics of columns in those tables is as follows:

- Number of Grid Cells: the number of partitions created for the given dataset.
- No. of Executors: the number of worker nodes used for computation.
- No. of Executor cores: the number of CPU cores available at one worker node.
- Capacity Threshold Size: the number of data records per grid cell, for which DT is computed locally.
- Time taken by DT (sec): the runtime of DT algorithm on AWS EMR in seconds.
- Time taken by GG (sec): the runtime of GG algorithm on AWS EMR in seconds.

The runtime of our algorithms depends on two performance variables: number of grid cells and number of executors. Executors here refer to worker nodes.

Tables 2-5 indicate that, when the threshold size is greater or equal to the number of points, then the algorithm computes on single executor as there is only one grid cell. For instance for Dataset-1, when we used threshold of 1 million points, it is slower; when we use threshold of 100k, the algorithm executes faster. However larger datasets producing too many grid cells will cause performance degradation as the step for merging neighboring cells is not completely parallel. For every iteration neighboring cells are merged, if there are n grid cells, it requires $log(n)$ steps to merge these cells. So we should choose value of *capacity threshold* in such a way that number of grid cells are less than total number of executor cores available for computation.

Table 1: dataset details

Dataset Type	Dataset Size
Dataset-1	851,223
Dataset-2	8,084,893
Dataset-3	15,943,322
Dataset-4	30,703,513

Table 2: Experiment 1: DT and GG; Dataset-1: 851,223 points; Spark: Amazon EMR

Number of Grid Cells	No. of Executors	No. of Executor Cores	Capacity Threshold	Time taken by DT (sec)	Time taken by GG (sec)
8 (2 × 4)	8	4	100,000	45	52
8 (2 × 4)	4	4	100,000	37	58
1	8	4	500,000	55	68
1	4	4	500,000	42	53
1	8	4	1,000,000	55	67
1	4	4	1,000,000	44	56

In processing larger datasets with a threshold size of 1 million, we get a better speedup as this results in less sequential operations (merge grid cells). As our approach of processing Delaunay triangles, computation of core triangles is paralleled; however the merge step requires recomputation of boundary triangles which merges n neighboring cells and will takes $log(n)$ steps. After we reduce the number of executors to half, we won't see the computation time increased by twice in proportion.

Table 3: Experiment 2: DT and GG; Dataset-2: 8,084,893 points; Spark: Amazon EMR

Number of Grid Cells	No. of Executors	No. of Executor Cores	Capacity Threshold	Time taken by DT (sec)	Time taken by GG (sec)
80 (8 x 10)	8	4	100,000	146	161
80 (8 x 10)	4	4	100,000	186	211
16 (4 x 4)	8	4	500,000	135	156
16 (4 x 4)	4	4	500,000	165	179
8 (2 x 4)	8	4	1,000,000	126	139
8 (2 x 4)	4	4	1,000,000	134	149

Table 4: Experiment 3: DT and GG; Dataset-3: 15,943,322 points; Spark: Amazon EMR

Number of Grid Cells	No. of Executors	No. of Executor Cores	Capacity Threshold	Time taken by DT (sec)	Time taken by GG (sec)
156 (12 x 13)	8	4	100,000	424	492
156 (12 x 13)	4	4	100,000	478	568
30 (5 x 6)	8	4	500,000	250	303
30 (5 x 6)	4	4	500,000	312	383
15 (3 x 5)	8	4	1,000,000	278	343
15 (3 x 5)	4	4	1,000,000	283	343

Table 5: Experiment 4: DT and GG; Dataset-4: 30,703,513 point; Spark: Amazon EMR

Number of Grid Cells	No. of Executors	No. of Executor Cores	Capacity Threshold	Time taken by DT (sec)	Time taken by GG (sec)
306 (17 x 18)	8	4	100,000	1004	1351
306 (17 x 18)	4	4	100,000	1335	1549
56 (7 x 8)	8	4	500,000	639	870
56 (7 x 8)	4	4	500,000	838	1049
30 (5 x 6)	8	4	1,000,000	439	553
30 (5 x 6)	4	4	1,000,000	645	755

Scalability Evaluation Summary for Chicago Crime Dataset experiment. Our benchmark tests showed that our DT and GG algorithms scaled well for massive datasets. Fig. 9 shows the summary of runtime of DT algorithm for our Chicago Crime Dataset scalability analysis. The runtime of our algorithms is related to the value of the *capacity threshold*. Choosing a lower value for the *capacity threshold* can result in too many grid cells; while selecting a higher value for the *capacity threshold* can result in a less number of grid cells. Both of these scenarios may increase the runtime. In order to experiment with different grid sizes for same dataset we allowed parallelism to be controlled by user.

Based on the number of the computing nodes and memory size for each node, we could make a better estimation for the *capacity thresholds*. When the number of data points is more than the *capacity threshold*, the data will be partitioned and run in parallel on different partitions. An ideal execution environment for spark parallelism could be one data partition running on one core and fit in the executor memory, so we can estimate an optimal *capacity threshold* based on the size

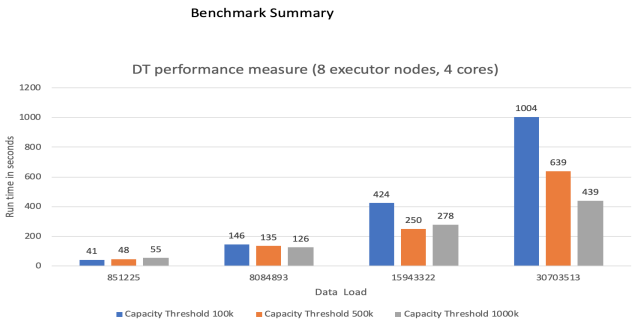

Figure 9: DT performance evaluation.

of data, the executor memory, the number of processors, and the CPU cores available per processor.

Comparison with Existing Libraries

In this experiment, we compare our algorithms results against the popular libraries for computing DT and GG. We compare DT results against the Delaunay function provided by the Scipy Python library, and we compare GG result against the functions provided by the Spdep and CCCD libraries in R.

Comparison tests for mobile activity data. A mobile activity dataset describes mobile phone activities carried out in the United States for three months. The dataset contains a total of 512,327 data records [16]. We are using the original dataset and superset of the mobile activity dataset to compare the DT algorithm. However, existing algorithms in R libraries to compute GG cannot scale for more than 30,000 points; hence, we are taking a sample of 10,000 points from the original dataset to compare the correctness of our results for GG algorithm.

Table 6 compares the DT computed using the Scipy Delaunay algorithm and our DT algorithm built on top of Spark. Table 7 compares the GG computed using Spdep and CCCD libraries against our GG algorithm using Spark. The results provided in Table 6 and Table 7 show that the number of edges and number of triangles obtained by our algorithms is same as the other popular libraries.

Looking at Table 6, we observe that the time taken by Scipy to compute DT on a dataset containing 512,327 data records is less than the time taken by Spark; this is because Spark requires bootstrapping time to initialize the master and worker nodes. However, if the dataset size increases beyond 2 million data points, our approach yields better results than Scipy.

Comparison of Spdep and Spark for speed violations data. The speeding violations dataset includes records of speed-limit violations that occurred in Chicago. The violation records in this dataset set are a cumulative collection of violations

Table 6: Comparison of DT results for mobile activity data

Algorithm Used	Library	No. of Points	No. of Triangles	Time Taken (sec)
DT	Scipy	512,327	1,024,620	11
DT	Spark	512,327	1,024,620	14
DT	Scipy	3,203,513	3,345,068	139
DT	Spark	3,203,513	3,345,068	96

Table 7: Comparison of GG result for mobile activity data

Algorithm Used	Library	No. of Points	No. of Edges	Time Taken (sec)
GG	Spdep	8214	15287	434
GG	CCCD	8214	15287	243
GG	Spark	8214	15287	18

captured by cameras and radar system. This dataset contains a total of 147 records [17].

Table 8: Comparison of DT results for speed violation data

Algorithm Used	Library	No. of Points	No. of Triangles	Time Taken (sec)
DT	Scipy	147	281	2
DT	Spark	147	281	3

Table 9: Comparison of GG results for speed violation data

Algorithm Used	Library	No. of Points	No. of Edges	Time Taken (sec)
GG	Spdep	147	206	5
GG	CCCD	147	206	4
GG	Spark	147	206	3

Table 8 compares the DT computed using the Scipy Delaunay algorithm and our DT algorithm built on top of Spark. Table 9 compares our GG algorithm against the functions provided by Spdep and CCCD. It can be observed from both the tables that the time taken by the existing libraries is almost same as our algorithms. This is because, for smaller datasets, our algorithms compute DT and GG, sequentially.

Scalability Evaluation Summary for Comparison with Existing Libraries. Our comparison tests performed on a local computer showed that the time taken by R libraries to compute GG is significantly larger than our GG algorithm. Spdep and CCCD libraries failed to compute GG for a dataset having 30,000 data records within 900 sec, whereas our GG algorithm was able to compute GG for a dataset having 5 million data records in the same time. Our DT algorithm also performed 1.5x times better than the popular Scipy library for a dataset having more than 2 million records.

5 RELATED WORK

Guibas et al. [1] present a divide and conquer algorithm. In this divide and conquer algorithm, a set of vertices P are recursively partitioned across a splitting line. A DT is computed for vertices in each partition, and the obtained triangles are merged along the splitting line. The algorithm has a complexity of $O(n \log n)$; however, this algorithm is a sequential algorithm, and it fails to take advantage of modern multicore processors.

Chen et al. [2] propose a parallel divide and conquer algorithm, and this algorithm works as follows: In the first step, the points P are partitioned into Q blocks, and these blocks are assigned to Q processors. The process of partitioning is carried out first in the x-direction and then in the y-direction. In the second step, points within each processor are sorted. In the third step, the divide and conquer algorithm is applied to every processor to triangulate within each block Q. In the fourth step, the affected zone of triangulation on the individual processor is generated. The affected zone refers to the region containing the triangles that gets changed during the merge. In the fifth step, merging of the affected zone is carried out simultaneously by modifying the triangles in the neighboring processor. However, this implementation lack in-memory processing, scalability, and fault tolerance capabilities for large datasets.

Hurtado et al. developed the Flip Algorithm [10] to compute Delaunay triangulation. In this algorithm, a triangulation is constructed for a two-dimensional data, and then the edges are flipped until all the non-Delaunay triangles are eliminated. $O(n + k^2)$ flips are sufficient to transform any triangulation of a polygon with n sides and k reflex vertices into DT [10]. For higher dimensions, this algorithm results in a disconnected graph.

Guibas et al. presented the Incremental Algorithm [11] to compute Delaunay triangulation. In this algorithm, vertices are added incrementally, and the affected parts of the graph are re-triangulated. We identify all the triangles that contain new vertices, then we flip away every triangle to construct a DT. It takes $O(n)$ time to add every new vertex and flipping the triangle edges for the formation of DT takes $O(n^2)$. The overall complexity of this algorithm is $O(n^2)$[11]. This algorithm can be extended to higher dimensions, but the run-time increases exponentially, even for smaller datasets, and it is practically not viable for larger datasets.

6 CONCLUSION

In this paper, we designed and implemented a parallel algorithm to generate DT and GG efficiently by leveraging the cluster-computing Apache Spark framework. We introduced a novel Divide and Conquer approach for parallelizing the computation of DT. In particular, we developed a unique

partitioning technique to partition the dataset into a grid. Based on a chosen *capacity threshold*, our approach creates irregular grids. The *capacity threshold* is the number of data points that a single core of a processor can process efficiently. And it supports creating an arbitrary number of grid cells for the same data load by changing the value of the *capacity threshold*. This fact distinguishes our approach from traditional Divide and Conquer DT algorithms, such as those published in [1], in which the dataset is partitioned recursively into two sets along a vertical line. In summary, our parallel algorithm computes DT locally for each grid cell and then post-processes and merges the obtained triangles to create the DT for the whole dataset.

In this paper, we first tried a different approach to parallelize DT and GG algorithms, before developing our algorithms using Spark. In the first approach, we sought to parallelize the Spdep library in R. This approach did not yield the expected results, due to lack of support for cluster computing in R.

Our benchmark tests showed significant improvement in the performance of computing DT and GG using our algorithms over existing functions present in the popular proximity graph libraries such Scipy, Spdep, and CCCD. Our DT algorithm is about 40% faster than the Scipy Delaunay function while running on a local computer and 2 times faster while running on an AWS EMR cluster. Our GG algorithm showed a 24 times better performance than Spdep and 15 times better performance than the CCCD library for computing GG for datasets having more than 5,000 data points. We were able to compute DT and GG for a dataset having thirty million records within 553 sec while running them on an AWS EMR cluster and within 1600 sec while running on the local computer.

We plan to achieve automatic grid size partitioning by considering system resources for computing capacity threshold. We also plan to integrate our algorithms with a spatial hotspot discovery framework relying on proximity graphs in our future work [7]. We are investigating alternative partitioning approaches for distributing the computation and to further improve the efficiency of our algorithms.

ACKNOWLEDGMENTS

We thank the AWS Cloud Computing, Research and Education program supporting our research project.

REFERENCES

[1] Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi," Association for Computing Machinery Transactions on Graphics, vol. 4, no. 2, pp. 74–123, Jan. 1985.

[2] M.-B. Chen, T.-R. Chuang, and J.-J. Wu, "Parallel divide-and-conquer scheme for 2D Delaunay triangulation," Concurrency and Computation: Practice and Experience, vol. 18, no. 12, pp. 1595–1612, 2006.

[3] K. R. Gabriel and R. R. Sokal, "A new statistical approach to geographic variation analysis," Systematic Zoology, vol. 18, no. 3, p. 259, 1969.

[4] D. Matula and R. Sokal, Properties of Gabriel graphs relevant to geographic variation research and the clustering of points in the plane. Dallas, Tex.: Dept. of Computer Science and Engineering, Southern Methodist University, 1979, pp. 205-220.

[5] G. Toussaint, "Geometric proximity graphs for improving nearest neighbor methods in instance-based learning and data mining", International Journal of Computational Geometry & Applications, vol. 15, no. 2, pp. 101-150, 2005.

[6] J. Choo, R. Jiamthapthaksin, C.-S. Chen, O. U. Celepcikay, C. Giusti, and C. F. Eick, "MOSAIC: A proximity graph approach for agglomerative clustering," Data Warehousing and Knowledge Discovery Lecture Notes in Computer Science, pp. 231–240.

[7] F. Akdag, J. U. Davis, and C. F. Eick, "A computational framework for finding interestingness hotspots in large spatio-temporal grids," Proceedings of the 3rd Association for Computing Machinery SIGSPATIAL International Workshop on Analytics for Big Geospatial Data - BigSpatial 14, 2014.

[8] "The circumcenter of a triangle," Triangle circumcenter definition - Math Open Reference. [Online]. Available: https://www.mathopenref.com/trianglecircum center.html. [Accessed: 2018].

[9] "Geometry in Action", University of California Irvine - ICS, 2018. [Online]. Available: https://www.ics.uci.edu/ eppstein/gina/scot.drysdale.html. [Accessed: 2018].

[10] F. Hurtado, M. Noy, and J. Urrutia, "Flipping edges in triangulations," Proceedings of the Twelfth Annual Symposium on Computational Geometry - SCG 96, 1996.

[11] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," in Automata, Languages and Programming Lecture Notes in Computer Science, pp. 414–431.

[12] J. Laskowski, "Spark Architecture · Mastering Apache Spark", Mastering Apache Spark Gitbook, 2018. [Online]. Available: https://jaceklaskowski.gitbooks.io/ mastering-apache-spark/content/spark-architecture.html. [Accessed: 2018].

[13] K. Darwin, "LocationTech GeoTrellis," Eclipse, 23-Feb-2018. [Online]. Available: https://projects.eclipse.org/projects/locationtech.geotrellis. [Accessed: 2018].

[14] "What is GeoTrellis?" GeoTrellis 1.0.0 Documentation. [Online]. Available: https://geotrellis.readthedocs.io/en/latest/. [Accessed: 2018].

[15] "Crimes - 2001 to present", City of Chicago - Data, 2018. [Online]. Available: https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2. [Accessed: 2018].

[16] "Sample Data Sources - Geovisualization - HERE Developer", HERE Developer, 2019. [Online]. Available: https://developer.here.com/documentation/geovisualization/topics/sample-data-sources.html. [Accessed: 2019].

[17] "Speed Camera Violations", City of Chicago - Data, 2018. [Online]. Available: https://data.cityofchicago.org/Transportation/Speed-Camera-Violations/hhkd-xvj4/data. [Accessed: 2018].

[18] "Core Concepts-GeoTrellis 1.0.0 documentation", GeoTrellis Docs, 2018. [Online]. Available: https://geotrellis.readthedocs.io/en/latest/guide/core-concepts.html. [Accessed: 2018].