Agent-based Computational Geometry

Akbarbek Rakhmatullaev[a], Shahruz Mannan[b], Anirudh Potturi[c], and Munehiro Fukuda[d]
Division of Computing and Software Systems, University of Washington Bothell, U.S.A.
{*akbarbek,mannans1,anii,mfukuda*}*@uw.edu*

Keywords: Agent-based modeling, data streaming, message passing, computational geometry, cluster computing

Abstract: Cluster computing increases CPU and spatial scalability of computational geometry. While data-streaming tools such as GeoSpark lines up built-in GIS parallelization features, they require a shift to their programming paradigm and thus a steep learning curve. In contrast, agent-based modeling is frequently used in computational geometry as agent propagation and flocking simulate spatial problems. We aim to identify if and in which GIS applications agent-based approach demonstrates its efficient parallelizability. This paper compares MASS, GeoSpark, and MPI, each representing agent-based, data-streaming, and baseline message-passing approach to parallelizing four GIS programs. Our analysis finds that MASS demonstrates the least boilerplate percentages and Cyclomatic complexity in its programmability and yields competitive parallel performance.

1 INTRODUCTION

Cluster computing gives more CPU and spatial scalability to GIS parallelization. Actual implementations include Hadoop-GIS (Aji et al., 2013) and Spark-GIS (Baig et al., 2017), many of which maintain spatial data in distributed storage such as Hadoop[1]; process the data in batches with data-streaming tools including Spark[2]; and respond to anticipated GIS queries through a front-end interface, (e.g., HIVE[3]). However, the nature of data streaming is their major challenge: besides their unique programming paradigm, they need to flatten, stream, shuffle, and sort spatial data structures at every computational stage, all resulting in substantial overheads.

In contrast to data streaming, we consider an agent-based approach that maintains GIS data as a multi-dimensional or graph structure over distributed memory; dispatches agents as active data analyzers; and solves spatial queries through collective group behaviors among the agents, (e.g., agent propagation, swarming, and collision) over the data structure. Our research motivation is to verify the efficiency of the agent-based approach to computational geometry, as compared to the conventional data-streaming approach. We believe that this research makes two contributions to parallel computing in computational geometry: (1) a development of geometric benchmark programs demonstrates that agent code is intuitive and smoothly fits the idea of spatial cognition (Freksa et al., 2019) and (2) agent-based approach is competitive to data streaming in some geometric applications that take advantage of agent flocking in a 2D space or agent traversing over a tree, both performed over a cluster system.

The rest of the paper is organized as follows: Section 2 introduces agent-based and data-streaming GIS parallelization; Section 3 explains the MASS (Multi-Agent Spatial Simulation) library[4] as an implementation of agent-based approach; Section 4 parallelizes four GIS benchmark programs, using MASS, Apache Sedona, and MPI, each representing agent-based, data-streaming, and conventional message-passing approach; Section 5 compares their parallel performance and evaluates the strength of agent-based approach; and Section 6 concludes our work.

2 RELATED WORK

We first identify challenges in parallelizing geometric problems, second look at the conventional data streaming as a solution, third consider agent-based

[a] https://orcid.org/0009-0005-3376-0684
[b] https://orcid.org/0009-0009-4628-5316
[c] https://orcid.org/0000-0002-9270-9628
[d] https://orcid.org/0000-0001-7285-2569
[1] https://hadoop.apache.org/
[2] https://spark.apache.org
[3] https://hive.apache.org
[4] https://depts.washington.edu/dslab/MASS

approach to an intuitive GIS solution where control moves over spatial data, and finally clarify our goals to make agent-based parallelization feasible in GIS.

2.1 Challenges in Parallelizing Geometric Problems

GEOS[5] and CGAL[6] are well-known C++ libraries that implement computational-geometry algorithms as built-in functions. Their native executions with multithreading are the fastest but limited to a single machine. The problem is that they are not so worthwhile being parallelized over a cluster system that only incurs more communication overheads than their single-machine execution. JavaGeom[7] and JTS[8] are Java versions of computational-geometry libraries that intend to ease geometric computation. Due to their interpretive execution, they do not outperform C++ libraries but show competitive processing throughput if a dataset size is maximized to the underlying memory space (Zhang and Eldawy, 2020).

In general, sequential or multithreaded execution runs fastest but its spatial scalability is restricted to a single machine's memory space.

2.2 Data Streaming to Analyzing Units

As data streaming keeps receiving great popularity in big data, it is quite natural and convenient to integrate data-streaming tools into a GIS system for scalable spatial analysis. A typical architecture modifies a Lambda service layer tool, (e.g., HIVE) for a real-time GIS query interface, uses data-streaming tools such as MapReduce and Spark for preparing anticipated query responses, and maintains entire spatial datasets in a backend database including PostGreSQL.

For instance, SpatialHadoop interfaces to users through Pigeon, a SQL-like language, which relays their queries to MapReduce for geometric computation (Eldawy and Mokbel, 2015). Hadoop-GIS is integrated with HIVE to extend HiveQL for the use of GIS spatial queries, to implement a real-time spatial query engine as a shared library in HIVE, and to access spatial data through Hadoop (Aji et al., 2013). GeoSpark, (i.e., formally named Apache Sedona) receives a spatial query through its Spatial SQL API that chooses the corresponding geometric algorithm, (e.g., range search, distance joining, and KNN) in the Spatial Query Processing Layer. The selected algorithm is then carried out though operations on Spatial RDDs, an extension of Spark RDDs (Resilient Distributed Datasets) (Yu et al., 2019). SparkGIS reads spatial data from distributed or cloud storages including HDFS and Amazon S3, preprocesses the data into Spark RDD to utilize distributed memory of a given cluster system, and invokes built-in spatial functions that have been implemented with RDD transformations and actions (You et al., 2015). For graph computing, GraphX extends Spark RDDs to edge and vertex RDDs, and supports Pregel's graph API (Spark GraphX, 2018; Malewicz et al., 2010)

In general, data streaming assumes text data as its input format since texts are easily split and streamed to *map()* functions or applied to lambda expressions. Therefore, structured files, (e.g., GIS shape files) must be disassembled into texts before being streamed. Most algorithms in computational geometry are optimized in the divide-and-conquer paradigm, which results in repetitive MapReduce or RDD transformations before reaching the final results even for every single query. This repetitive series of data streaming, shuffle, and sort may slow down geometric analysis.

2.3 Migrating Analyzers over Geometric Data Space

We consider a different approach where agents solve a geometric problem by forming their emergent collective group behavior over a spatial dataset. In other words, we apply agent-based modeling (ABM) to computational geometry. This idea is not brand-new but found in the following three ABM libraries that are available in interpretive languages:

NetLogo approximates a 2D radical propagation of agents by repetitively cloning agents to von-Neumann and Moore neighborhoods in an alternative fashion (Wilensky, 2013). Using this agent propagation from each data point, NetLogo composes a Voronoi diagram of bisector lines between any pair of data points, on which agents propagated from a different point collide with each other. Repast Simphony (North et al., 2007) populates agents on all the four boundary lines of a 2D space and march them toward the center of the space. This is a simulation of wrapping data points with an elastic band, which forms the convex hull (Saadati and Razzazi, 2022). GeoMASON supports basic geometric data operations including: reading shape files; incorporating points, line segments, and polygons into its simulation space; and allowing agents to migrate on these geospatial components (Sullivan et al., 2010). It also computes the shortest path on a network of line seg-

[5]https://libgeos.org

[6]https://www.cgal.org

[7]https://geom-java.sourceforge.net

[8]https://locationtech.github.io/jts/

ments and their intersections as built-in functions.

Their biggest challenge is single-machine execution. Because of their difficulty in being extended to cluster computing, they cannot support spatial scalability nor parallelize file I/Os[9]. Since these ABM simulators put more emphasis on GUI to non-computing users, they facilitate only basic geometric computation to find line intersections and area unions, besides the Voronoi and the convex hull algorithms. This is our motivation to apply MASS, a parallel ABM library to more advanced spatial problems.

3 COMPUTATIONAL MODEL

This section summarizes the MASS library's computational model and introduces its extension to graph and geometric computing.

3.1 MASS Library

The MASS library lines up Java, C++, and CUDA versions. While each version has its own target, for the purpose of our comparative work with GeoSpark in Java, we use MASS Java, simply referred to as MASS in the following discussions.

MASS distinguishes two classes: Places and Agents. The former is a multi-dimensional array distributed over a cluster of computing nodes. Each array element is called "place" and identified with a platform-independent logical index. The latter is a collection of mobile objects, each called "agent", populated on a given place and capable of moving to a different place, thus to a different computing node.

Listing 1 shows MASS abstract code. The *main()* function serves as a simulation scenario that gets started with *MASS.init()* (line 3) to launch a multi-threaded, TCP-communicating process at each cluster node. Lines 4-6 create an $x \times y$ 2D Places and populate Agents, each respectively referenced from *map* and *crawlers*. Places has two parallel functions: *callAll()* to invoke a given function, (e.g., *update_func* on line 7) at each *place* in parallel and *exchangeAll()* to have each place initiate a remote method invocation to all its neighbors, (e.g., *diffuse_func* in line 8), thus facilitating an RMI-based inter-place communication. Agents has two parallel functions, too. One is *callAll()*, similar to but different from Places', where agents schedule their next behavior with *spawn()*, *kill()*, and *migrate()*, each spawning new children, terminating the calling agents, and moving them to a different place (line 9). The other parallel function is *manageAll()* that intends to commit their scheduled behaviors (line 10). Upon finishing all the ABM computation, *main()* needs to shutdown all MASS processes with *MASS.finish()* (line 11).

Listing 1: MASS abstract code

```
1 public class MassAppl {
2   public static void main(String args[]) {
3     MASS.init( );
4     Places map = new Places("Map", args, x, y);
5     Agents crawlers
6       = new Agents("Crawlers", args, map);
7     map.callAll(update_func, args);
8     map.exchangeAll(diffuse_func);
9     crawlers.callAll(walk_func, args);
10    crawlers.manageAll( );
11    MASS.finish( );
12 } }
```

3.2 Agent Descriptions in Graph and Geometric Problems

The original *Place* and *Agent* specification burdens model designers with manual graph emulation and agent propagation (Gordon et al., 2019). To address these deficiencies, MASS incrementally improved its programmability, execution performance, and development environment with the following five features:

1. **GraphPlaces**: is a Places sub-class that instantiates place objects as graph vertices whose emanating edges are defined in the *neighbors* list as one of their data members (Gilroy et al., 2020).
2. **BinaryTreePlaces**: is a special form of GraphPlaces to distinguish only left and right child vertices, which eases KD-tree operations in range search (Guo, 2021).
3. **SpacePlaces**: implements a 2D contiguous space, using QuadTreePlaces that reduces the number of place objects in memory as well as mitigates unnecessary agent migration. The closet pair of points, convex hull, and Voronoi problems use this class (Guo, 2021).
4. **SmartAgents**: is an Agents sub-class that automates agent propagation over a GraphPlaces, a BinaryTreePlaces, and a SpacePlaces instance, each used in the breadth-first search, the range search, and all the 2D geometric problems (Mohan et al., 2023).
5. **Interactive programming and visualization** (Blashaw and Fukuda, 2022; Yang, 2023): allows users trial-and-error coding with JShell[10] and vi-

[9]Repast HPC is a C++ version to run on a cluster system but its I/Os must be serialized via main().

[10]https://dev.java/learn/jshell-tool/

sualizes graphs, trees, and 2D spaces, using Cytoscape[11].

These features make MASS competitive to other ABM libraries in programmability as well as to data-streaming tools in execution performance.

4 PARALLELIZED ALGORITHMS

Our expectation for agent-based computational geometry is two-fold: (1) agents could identify a given geometric shape faster if their flocking converges to a small space, and (2) agents could quickly respond to geometric queries if they use the same data structure that stays in memory. From these viewpoints, we have chosen the following four geometric problems for our comparative work.

1. **Convex hull**: wraps all points with the smallest convex polygon. Agents converges from 2D space boundaries to the hull.
2. **Euclidean shortest path**: identifies the shorts path by dodging any obstacles. Agents bouncing to an obstacle are not the fastest to reach a given destination, which shrinks their population.
3. **Largest empty circle**: finds the largest circle that has no data points. This is adversarial example that propagates agents from each Voronoi vertex and thus expands their population.
4. **Range search**: shows all points residing in a given space. Agents repetitively traverses the same KD tree as different queries.

Our comparative work parallelizes the above four programs using MASS, Apache Sedona, and MPI, each representing agent-based, data-streaming, and conventional message-passing approach. The reason why we included MPI is that it is the most flexible and lowest-level parallel-programming library which, we expect, would demonstrate the baseline parallel performance.

In the following, we give explanations of how each benchmark program can be parallelized with these three libraries.

4.1 Convex Hull (CVH)

4.1.1 Agent-based approach

MASS has agents swarm inward from the outer edges of a given space until they encounter any data points,

[11]https://cytoscape.org/

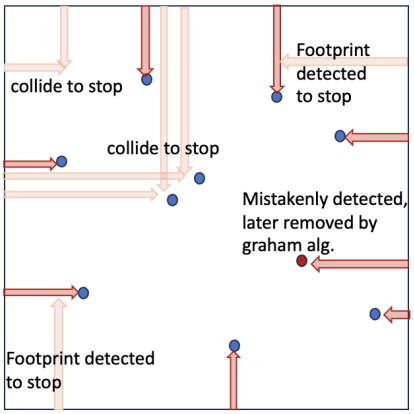

Figure 1: Agent-based convex-hull computation

which simulates wrapping all points with a rubber band. The algorithm is coded in Listing 2. It populates agents on the four boundary edges of a $size \times size$ space (lines 4-5), marches them until they hit a point (lines 10-13), excludes unvisited points (lines 15), and retrieves all data points on the final convex hull (lines 16-17). Figure 1 depicts this agent swarming. As some data points may be mistakenly detected as vertices of the convex hull, they must be removed by Andrew's monotone chain algorithm (Andrew, 1979).

Listing 2: Convex hull using MASS

```
1 public class CVH {
2   public static void main(String args[]) {
3     Places places = new Places("AreaGrid", size , size);
4     Agents agents = new Agents("RubberBandAgent",
5                                places, size * 4);
6     agents.callAll(RubberBandAgent.
7                    SET_START_POSITION);
8     agents.manageAll();
9     // March agents toward the center like a rubber band
10    while (agents.nAgents() > 0) {
11      agents.callAll(RubberBandAgent.MOVE);
12      agents.manageAll();
13    }
14    // Remove inner points and collect those on the hull
15    places.callAll(AreaGrid.CLEAR_INNER_PLACES);
16    Object[] oResults
17      = places.callAll(AreaGrid.GET_PTS, null);
18 } }
```

4.1.2 Data-streaming approach

Apache Sedona (we simply call Sedona in the rest of the paper) takes a divide-and-conquer approach that spreads out all data points to partitions, creates a per-partition convex hull, and aggregates together all the partial hulls into the final convex hull. In order to achieve this, we first create a *space* RDD (line 4) and

then partition it using Sedona's EQUALGRID type (line 7), as it shows the best execution performance among other grid types. Next, each partition creates a list of its points (lines 11-12), from which we create a multi-point object of Sedona's Geometry class (line 13-14). Then, we call Sedona's built-in *convexHull()* function on this multi-point object to create a per-partition convex hull (line 16), where each convex hull is stored as a singleton collection.

Finally, we aggregate all partial hulls into a table (line 24) and apply Sedona's SQL functions ST_ConvexHull and ST_Union_Aggr to produce the final *hull*, a single-row dataset representing the complete convex hull (line 28).

Listing 3: Convex hull using Sedona

```
1 public class CVH {
2   public static void main(String args[]) {
3     GeometryFactory geom = new GeometryFactory();
4     SpatialRDD<Point> space = new SpatialRDD<>();
5     // Read data points and partition them.
6     space.setRawSpatialRDD(getDataset());
7     space.spatialPartitioning(GridType.EQUALGRID);
8     // Compute per-partition convex hulls.
9     JavaRDD<Geometry> partialCvhRDD
10      = s.getRawSpatialRDD().mapPartitions(points->{
11          List<Point> pointList = new ArrayList<>();
12          points.forEachRemaining(pointList::add);
13          Geometry multPoint = geom.createMultiPoint
14            (poitnList.toArray(new Point[0]));
15          return Collections.singleton
16            (multipoint.convexgHull()).iterator();
17        ) }
18     // Union partial convex hulls in one dataset
19     SpatialRDD<Geometry> cvhRDD
20       = new SpatialRDD<>();
21     cvhRDD.setRawSpatialRDD(partialhRDD);
22     Dataset<Row> hulls // data from of all hulls
23       = Adapter.toDf(cvhRDD, sedona);
24     hulls.createOrReplaceTempView("hulltable");
25     // Aggregate the hulls into a single convex hull
26     Dataset<Row> hull = sedona.sql(``SELECT
27     ST_CONVEXHULL(ST_Union_Aggr(geometry))
28       as final_convex_hull FROM hullTable'');
29 } }
```

4.1.3 Message-passing approach

MPI starts similarly by reading data points from an input file. Before partitioning the data evenly for all the computing nodes, (i.e. MPI ranks), the data needs preprocessing. The points are sorted based on the x coordinate. Now, partitioning the data and distributing the subsets to each rank will result in the subsets having data points which are near each other. The data distribution can be visualized as having the plane sliced into M vertical slices where M is the number of ranks. Next, the Monotone Chain algorithm is used to compute the convex-hull points in each computing node. The algorithm constructs the upper and the lower hull separately, and thereafter combines them into a complete hull.

After creating a partial hull on every rank, *MPI_Send()* and *MPI_Recv()* are called between two neighboring ranks to merge their partial hulls into a larger hull. A typical O(N) merging algorithm is used to find the upper/lower tangent lines connecting two hulls and to remove the points between them (See Figure 2). This merging step is repeated until all hulls are combined into the final convex hull at rank 0.

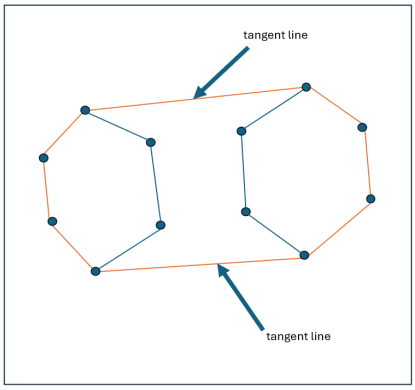

Figure 2: A convex-hull merger algorithm used in MPI

4.2 Euclidean Shortest Path (ESP)

4.2.1 Agent-based approach

Upon an initial propagation from a source, agents repeat bouncing obstacles or terminating themselves if others have visited the current grid, which eventually carries the fastest agent to a given destination. Figure 3 exemplifies agent propagation from a given source, followed by another propagation from an obstacle corner to the final destination. Listing 4 initializes a 2D space with obstacles (line 4), positions a *Rover* agent at a source point (lines 5-8), and then falls into an agent propagation loop (lines 9-19) until an agent reached the goal (line 9). Each iteration clones agents if they are the first visitor on the current grid that is not yet the destination (lines 13-16); moves all the cloned agents to non-blocking neighbors (lines 17-18); marks each grid with the first agent's footprint (line 10); and kills all slower agents (lines 11-12).

Listing 4: Euclidean shortest path using MASS

```
1 public class ESP {
2   public static void main(String args[]) {
3     Places places = new Places("Cell", sizeX, sizeY);
4     places.callAll(Cell.init_, dataset);
5     Agents agents = new Agents("Rover", places, 1);
6     agents.callAll(Rover.starting_point,
7                    (new int[]{starting_x, starting_y}));
```

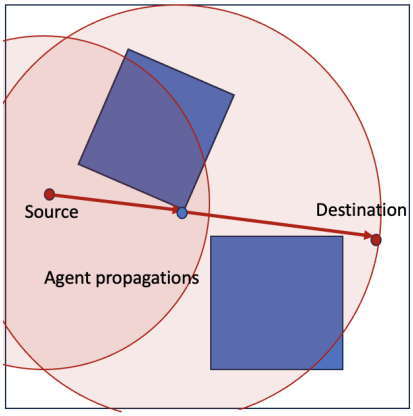

Figure 3: Agent-based Euclidean shortest path search

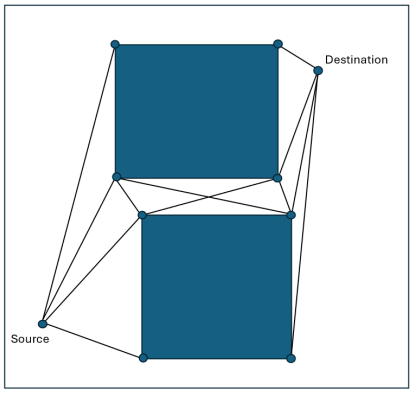

Figure 4: A visibility graph in Euclidean shortest-path search

```
8       agents.manageAll();
9       while (!foundTarget && agents.nAgents() > 0) {
10        places.callAll(Cell.update_);
11        agents.callAll(Rover.update_termination);
12        agents.manageAll();
13        Object target = agents.callAll(Rover.clone,
14                                       new Object[agents.nAgents()]);
15        agents.manageAll();
16        if ( target ) break;
17        agents.callAll(Rover.migrate_all);
18        agents.manageAll()
19} } }
```

4.2.2 Data-streaming approach

To implement the Euclidean Shortest Path in Sedona, we first define the starting and ending points (lines 5-7) and include them in a set of all points, which comprises vertices from the input obstacles (lines 8-18). Using this set, we generate a visibility graph (see Figure 4) by forming a Cartesian product of all points to get potential edges (lines 19-24). Each edge is then checked to see if it intersects any obstacles; if not, it's considered visible between the points and added to a list for later distance calculations. With all visible vertex combinations identified, we apply Dijkstra's algorithm to compute the shortest path from the start to the endpoint (lines 25-26).

Listing 5: Euclidean shortest path using Sedona

```
1 public class ESP {
2   public static void main(String args[]) {
3     GeometryFactory geom = new GeometryFactory();
4     PolygonRDD obstaclesRDD = getObstacles();
5     // Start and end points of the shortest path
6     Point org = geom.createPoint(new Coordinate(x, y));
7     Point end = geom.createPoint(new Coordinate(i, j));
8     // Collect all points to be used for visibility graph
9     List<Point> points = new ArrayList<>();
10    points.add(org);
11    points.add(end);
12    for (Polygon poly :
13      obstaclesRDD.rawSpatialRDD.collect()) {
14        points.addAll(Arrays.stream(
15          poly.getCoordinates()).
16          map(geom::createPoint).
17          collect(Collectors.toList()));
18    }
19    JavaRDD<Point> ptJavaRDD
20    = sedona.createDataset(points,
21      Encoders.kryo(Point.class)).toJavaRDD();
22    PointRDD ptsRDD = new PointRDD(ptJavaRDD);
23    JavaPairRDD<Point,Iterable<Point>> vGraphRDD
24    = generateVisibilityGraph(ptsRDD, obstaclesRDD);
25    List<Point> dijkstraShortestPath
26    = dijkstraShortestPath(org, end, vGraphRDD);
27} }
```

4.2.3 Message-passing approach

MPI constructs a visibility graph and applies Dijkstra's algorithm on it as Sedona does. Input points for this implementation are the obstacle corners as well as source and destination points. These points are then partitioned to all MPI ranks where a per-rank visibility graph is created from the subset. The simplest but greedy approach compares every pair of points from the input points whether a line segment between the pair of points intersects with obstacle edges. If a line segment does not intersect with any obstacle edge and does not go through an obstacle, the pair has a visibility edge. Once the a per-rank visibility graph is constructed, the information is saved as a Hash-Map at each rank, where the key is a vertex, and the value is a list of the vertices which can create a visibility edge with this specific vertex. Next, all the partial visibility graphs are sent back to rank 0 and combined into a complete visibility graph of all the data points. Lastly, Dijkstra's algorithm is used for finding the shortest path.

4.3 Largest Empty Circle (LEC)

Sedona, MASS, and MPI all take the same LEC algorithm - convex hull and Voronoi diagram construc-

tions followed by computing the center of LEC from all Voronoi vertices and intersections between the convex hull and the Voronoi diagram. All their parallelization strategies also take data decomposition where each RDD partition in Sedona, each place in MASS, and each rank in MPI computes reports its potential point of LEC to *main()*. This is because, if we use agent propagation in MASS, the agents exponentially diverge their population from each Voronoi vertex, thus waste memory space, and do not perform faster.

4.3.1 Agent-based approach

MASS first uses Fortune's sweep-line algorithm to create a Voronoi Diagram sequentially from input data points that are located within a space of *w* width and *h* height (line 5). It then distributes the Voronoi vertices and edges into *nP* partitions (lines 7-13). From them, MASS creates Places (line 14), each of which takes a different partition (line 15), computes the intersections between Convex Hull edges and Voronoi edges in the partition (line 16), and identifies a potential LEC center (line 18). Finally, *main()* collects potential circles from all the Places and finds the final LEC (lines 20-23).

Listing 6: Agent-based largest empty circle

```
1 public class LEC {
2   public static void main(String args[]) {
3     Point2D[] points = dataPoints();
4     // Create Voronoi Diagram
5     Voronoi diagram = new Voronoi (w, h, points);
6     // Partition vertices
7     int vSize = diagram.vertices.length;
8     int[][] v = partitionData(diagram, vSize, nP);
9     // Partition Edges
10    int eSize = diagram.edges.length;
11    int[][] e = partitionData(diagram, eSize, nP);
12    // Create subsets
13    Object[] partitions = createPartitions(v, e, nP);
14    Places places = new Places("Partitions", nP);
15    places.callAll(Parititions.Init, partitions);
16    places.callAll(Parititions.Intersections);
17    // Compute Largest Empty Circle
18    places.callAll(Partitions.LEC, points);
19    // Return all Largest Empty Circles
20    Object[] results
21      = places.callAll(Partitions.Collect);
22    // Get The largest empty circle from all the circles
23    max(results);
24 } }
```

4.3.2 Data-streaming approach

Sedona first gathers all dataset points (line 3) and constructs a convex hull (line 8), using the algorithm described in Section 4.1.2. It then generates a Voronoi Diagram from these points from its built-in *VoronoiDiagramBuilder* class (lines 11-12). Thereafter, Sedona clips the diagram along the convex hull edges to obtain Voronoi polygons (lines 13-17). These polygons, combined with the convex hull, help identify candidate points (lines 18-21). These candidate points are then converted to spatial RDD which gets partitioned using Sedona's EQUALGRID (lines 22-26). A nearest neighbor search is applied to these candidates within each partition to determine the center and radius of the largest empty circle (lines 27-29) Once it is finished, Sedona combines all the centers and radiuses from all partitions to find the one with the largest radius (lines 30-31).

Listing 7: Largest empty circle using Sedona

```
1 public class LEC {
2   public static void main(String args[]) {
3     SpatialRDD<Point> spatialRDD = getData();
4     // Partition spatialRDD
5     spatialRDD.analyze();
6     spatialRDD.spatialPartitioning(EQUALGRID);
7     Geometry convexHull = getConvexHull();
8     SpatialRDD<Geometry> allPointsRDD
9       = getPoints(spatialRDD);
10    // Build Voronoi polygons
11    VoronoiDiagramBuilder voronoiBuilder
12      = new VoronoiDiagramBuilder();
13    voronoiBuilder.setSites(coordinates);
14    voronoiBuilder.setClipEnvelope(convexHull.
15      getEnvelopeInternal());
16    List<Polygon> voronoiPolygons
17      = getPolygons(voronoiBuilder);
18    List<Coordinate> lecCenters = new ArrayList<>();
19    for (Geometry polygon : voronoiPolygons) {
20        lecCenters.addAll(polygon.getCoordinates());
21    }
22    SpatialRDD<Geometry> lecCentersGeomRDD
23      = new SpatialRDD<>(lecCenters);
24    lecCentersGeomRDD.analyze();
25    lecCentersGeomRDD.
26      spatialPartitioning(EQUALGRID);
27    JavaRDD<Tuple2<Geometry, Double>>
28      lecCentersRDD
29      = nearestNeighborSearch(lecCentersGeomRDD);
30    lecCentersRDD = lecCentersRDD.cache();
31    findLargestValue(lecCentersRDD);
32 } }
```

4.3.3 Message-passing approach

In MPI, rank 0 reads an input file and sequentially creates a Voronoi diagram from the input points, using the Fortune's sweep-line algorithm. It thereafter creates the convex hull as described in Section 4.1.3. Next, the Voronoi vertices, Voronoi edges, and the convex hull points are split into partitions and distributed to all MPI ranks. They compute the intersection points between the subsets of Voronoi Edges and the Convex Hull edges in their partition. Thereafter, all the ranks iteratively examine all the Voronoi vertices and the intersection points to calculate the radius to their closest original data point. Finally, they

report their local LECs to rank 0 that finds the largest one among them.

4.4 Range Search (RGS)

4.4.1 Agent-based approach

Listing 8 outlines agent propagation down over a KD tree from its root in search for all tree nodes in a given range. First, MASS creates a KD tree from GraphPlaces (lines 3-4), which is the slowest part of the code as the tree is recursively constructed from *main()* (line 5). Thereafter, the initial agent starts a KD tree search from its root (line 6) and repeats propagating its copies along the left/right tree branches (line 7-10). Upon every propagation down to the next tree level, agents report back to *main()* if they encounter tree nodes within a queried range (lines 8-9). Lines 6-10 can be repetitively used for responding to different queries. The strength in the MASS implementation is a global KD tree construction over distributed memory as shown in Figure 5.

Listing 8: Agent-based range search

```
1 public class RGS {
2   public static void main(String args[]) {
3     ArrayList<Point2D> points = getPoints(inputFile);
4     GraphPlaces kdTree = new GraphPlaces("KDTree");
5     constructTree(kdTree, points);
6     Agents rovers = new Agents("Rover", kdTree, 1);
7     while( rovers.nAgents() > 0 ) { // tree traverse
8       Object results[] = rovers.callAll(Rover.search);
9       Collections.addAll(results); // range identified
10      rovers.manageAll();
11 } } }
```

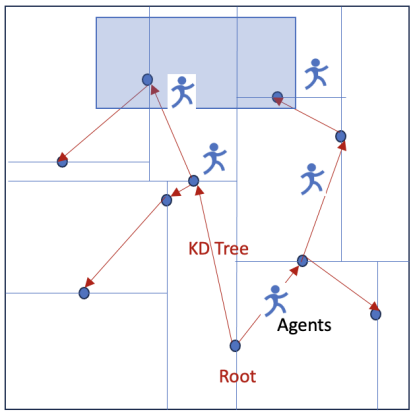

Figure 5: Agent-based range search

4.4.2 Data-streaming approach

Sedona performs a range search, using its built-in functions and classes, such as SpatialRangeQuery (line 20). This class requires only a few parameters to operate: a spatialRDD with the data points (line 6), an Envelope defining the query boundaries (line 8), a SpatialPredicate (set to COVERED_BY for this operation) (line 13), and a Boolean, (i.e., usingIndex in line 14) to specify index usage (line 21). This configuration enables Sedona to identify all points within the Envelope in spatialRDD. Before processing a query, the spatialRDD is partitioned using GridType.EQUALGRID (line 17), and results are subsequently collected (line 22).

Listing 9: Range search using Sedona

```
1 public class RGS {
2   public static void main(String args[]) {
3   GeometryFactory geom = new GeometryFactory();
4   SpatialRDD<Point> spRDD = new SpatialRDD<>();
5   // Read data and set it as raw RDD to spatialRDD
6   spRDD.setRawSpatialRDD(getDataset());
7   // Range Search coordinates
8   Envelope queryWindow
9     = new Envelope(new Coordinate(x1, y1),
10                   new Coordinate(x2, y2));
11  // Create a predicate
12  SpatialPredicate spPredicate
13    = SpatialPredicate.COVERED_BY;
14  boolean usingIndex = false;
15  // Partition spatialRDD
16  spRDD.analyze();
17  spRDD.spatialPartitioning(GridType.EQUALGRID);
18  // Query a SpatialRDD
19  JavaRDD<Point> queryResult
20    = RangeQuery.SpatialRangeQuery(spRDD,
21      queryWindow, spPredicate, usingIndex);
22  queryResult.collect();
23 } }
```

4.4.3 message-passing approach

First, data points are read from a CSV input file, equally partitioned, and distributed to all MPI ranks. Each rank constructs its local KD tree by recursively selecting dimension X or Y in turn, sorting the local points in terms of the selected dimension, splitting the smaller and the larger half in the left and right subtrees. Upon a tree completion, a query about finding points in a given range is passed to all the ranks, each traversing its own local KD tree. Once all the ranks have completed querying their trees, MPI_Gather() is called to collect into rank 0 all the points that are found in a specified range.

4.5 Programmability

Having coded the four benchmark programs with the three libraries, we summarized their programmabil-

ity in # lines of code (LoC), boilerplate percentage[12], and Cyclomatic complexity, as shown in Table 1. In general, as Sedona lines up built-in GIS functions, all its benchmark LoCs are the smallest. However, this code compactness results in increasing Sedona's boilerplate percentage even with a few additional statements that prepare distributed datasets, each fitting its corresponding built-in function. Since the MPI benchmarks are manual versions of divide-and-conquer algorithms, their LoC is three to five times larger than Sedona's. However, MPI's boilerplate percentage and Cyclomatic complexity are smaller than Sedona. This is because MPI directly accesses each data item while Sedona repetitively prepares different datasets, each using lambda expressions that handle a list of data items. In contrast, MASS programs end up in the largest LoC while demonstrating the smallest boilerplate percentage and Cyclomatic complexity, both indicating less semantically gapped and less branching code. Although MASS facilitates intuitive agent-based coding and parallelization, its current GIS supports such as Graph/Tree/SpacePlaces and SmartAgents still need to automate and to integrate more GIS features into MASS agents.

Table 1: Programmability comparison

Benchmark	Metrics	Sedona	MASS	MPI
CVH	LoC	113	710	316
	Boilerplate %	43	3.8	8.8
	Cyclomatic complexity	4.4	3.4	4.2
ESP	LoC	191	692	523
	Boilerplate %	31	5.1	4.7
	Cyclomatic complexity	3.8	4.1	3.1
LEC	LoC	210	767	612
	Boilerplate %	41	2.5	5.2
	Cyclomatic complexity	4.1	3.1	3.5
RGS	LoC	120	368	233
	Boilerplate %	47	4.1	8.5
	Cyclomatic complexity	4.0	2.6	3.1
Average	LoC	163.5	634.3	421
	Boilerplate %	40.5	3.9	6.8
	Cyclomatic complexity	4.1	3.3	3.5

5 EVALUATION

We conducted benchmark measurements on our own research cluster system at University of Washington Bothell. The system consists of 20 computing nodes, all that are 64-bit Linux servers connected to a central filesystem. Detailed information about these computing nodes is summarized in Table 2

[12]A rate of LoC needed for parallelization against the entire LoC.

Table 2: Benchmark environment

#machines #VMs	CPU cores	CPU model	Cache	Memory
3 (physical)	4	Xeon 5150 @2.66GHz	4MB	16GB
4 (physical)	4	Xeon E5410 @2.33GHz	6MB	16GB
5 VMs	4	Gold 5520R @2.20GHz	36MB	16GB
8 VMs	4	EPYC 7252 @3.10GHz	512KB	16GB

Our evaluations of the computational geometry implementations with Sedona, MASS, and MPI utilized a diverse range of GIS datasets. Table 3 shows which dataset was used for which computational geometry problem. The datasets are in CSV or text format which allowed us to feed the datasets to all the libraries. To evaluate the spatial complexity for LEC, we used a dataset of 50,000 randomized coordinates. We could not find a GIS dataset in CSV format that contained obstacles as points for benchmarking the ESP implementations. Thus, we generated a dataset containing 300 as well as 500 polygons with some randomized number of corners: ranging from 4 to 8.

Table 3: Datasets used for evaluation

Datasets	Size (points)	Benchmark Programs
National USFS fire occurrence[1]	581,541	RGS, CVH (small)
Crime locations in LA, US[2]	938,458	RGS, CVH (large)
US private school locations[3]	22,346	LEC (small)
Randomized spatial points	50,000	LEC (large)
Randomized 300 polygons[4]	1,200-1,700	ESP (small)
Randomized 500 polygons[4]	2,000-3,000	ESP (large)

[1] (U.S. Forest Service - Geospatial Data Discovery, 2024)

[2] (Los Angeles Open Data Portal, 2024)

[3] (ArcGIS Hub, 2023)

[4] Each polygon with with 4 to 6 vertices

5.1 Convex Hull (CVH)

Figures 6 and 7 compare parallel performance of Sedona, MASS, and MPI when running CVH with the small and the large dataset respectively. The trend in their execution performance does not change between the small and the large datasets. Overall, Sedona's total execution time is the slowest due to its considerable data-loading overheads. Yet even focusing on its computational time only, Sedona performs slower than MASS total execution. This is because MASS agents converge to a convex hull much faster than Sedona's repetitive data shuffle-and-sort operations. Despite that MASS needs to create a 2D Places space, its total execution time is competitive to MPI or even better than MPI as increasing the number of machines beyond eight. This is because MASS can read input data in parallel while MPI needs to distribute date from rank 0 to the other worker ranks. Using 18

or 20 machines, Sedona's shuffle-and-sort overheads diminish, which makes Sedona competitive to MASS. On the other hand, MASS agent migration over machine boundary gets increased with more computing nodes, which slows down MASS execution time beyond four machines.

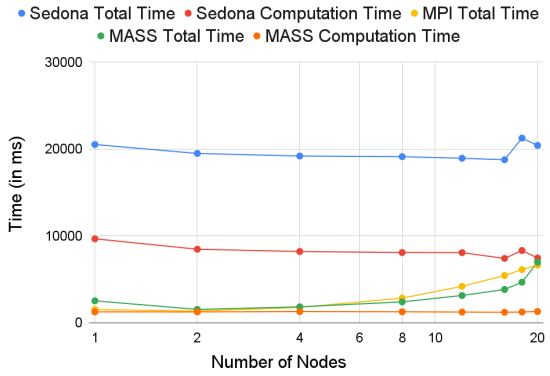

Figure 6: CVH with fire.csv

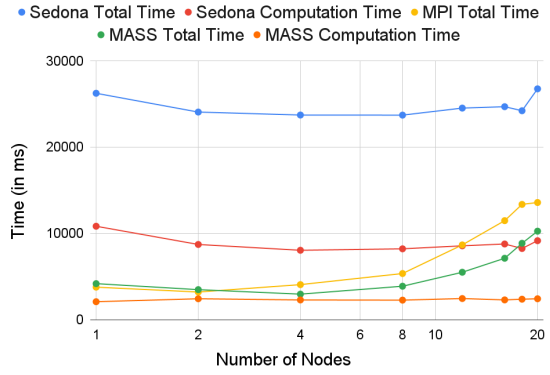

Figure 7: CVH with crime.csv

5.2 Euclidean Shortest Path (ESP)

Figures 8 and 9 show all the three libraries' parallel performance of ESP execution, each computing with 300 and 500 obstacles respectively. While the small dataset ranks MASS as the slowest execution, its parallel performance continuously improves as increasing the number of machines, which makes MASS the fastest with 20 computing nodes. The main reason is that agent propagation actually controls the agent population rather than explodes it since many agents hit obstacles to stop their propagation. As the number of computing nodes gets increased, each computing node has less agents that even alleviate their propagation. This trend is even clearer with the large dataset that includes more obstacles. On the other hand, Sedona suffers from its Cartesian product computation that is bound to $O(n^2)$. This quadratic complexity also slows down Sedona's total execution with the larger dataset, while still showing its parallel performance. MPI's visibility graph construction similarly increases quadratic to the data size, but its total execution time is the fastest until 16 computing nodes as each computation of line intersections is computationally negligible.

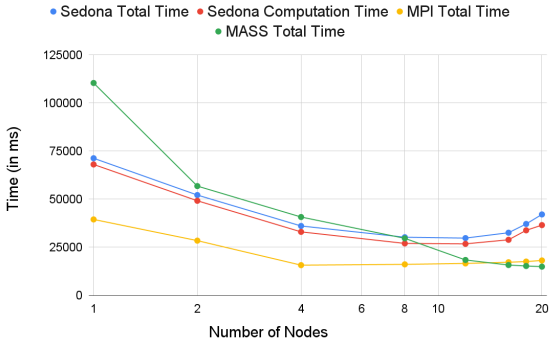

Figure 8: ESP with 300Obstacles.txt

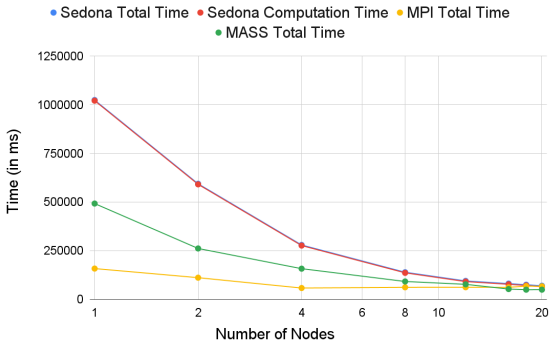

Figure 9: ESP with 500Obstacles.txt

5.3 Largest Empty Circle (LEC)

As described in Section 4.3, Sedona, MASS, and MPI take the same LEC parallelization strategy. Yet, MASS does not improve parallel performance as its main() function is the focal point that chooses the final LEC among all potential LECs, each reported from a different place element. Figures 10 and 11 show that MASS parallel performance is always bound to its *main()* function and does not change. Sedona runs the slowest with 1-12 computing nodes even with the large dataset but eventually outperforms MASS. This is because Sedona's lambda expressions repetitively

compare each pair of potential LECs, which incurs large overheads with less computing nodes. However, since Sedona has no focal point in parallelization, its performance is improved with more machines added to the computation. Finally, MPI serves as the best baseline performance as its computation is coarsely performed in each rank and a one-time reductive communication takes only at the end of the execution to find the final result among up to 20 potential LECs.

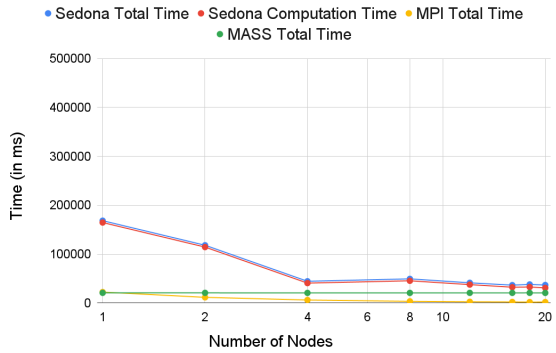

Figure 10: LEC with school.csv

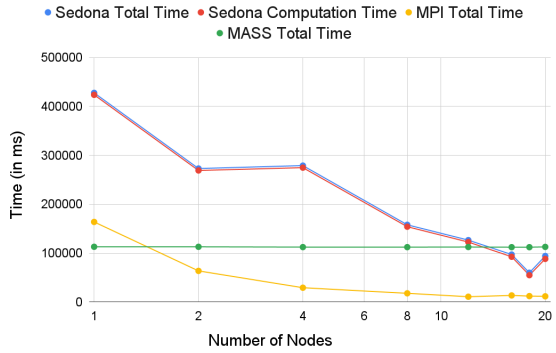

Figure 11: LEC with s.txt (random points)

5.4 Range Search (RGS)

Figures 12 and 13 measure the KD-tree construction and range-query execution time elapsed by the three libraries, as feeding the USFS fire occurrence dataset (581,541 points) and the LA crime location dataset (938,458 points). MPI runs the fastest in both cases while its query transactions, (i.e., MPI computation time) receive more communication overheads when increasing the number of machines. On the other hand, Sedona always runs the slowest. Its main overhead (which occupies 59% through 73% of the total time) is its tree construction and results from Sedona's repetitive RDD shuffle-and-sort operations. These operations also slow down query transactions in both small and large datasets, each spending 2.6-1.9 times and 2.3-1.4 times more than MASS query transactions. MASS cannot outperform MPI while its total execution time gets closer to MPI's as increasing the number of computing nodes beyond 16.

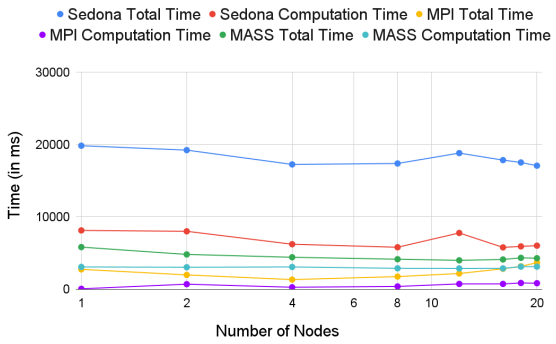

Figure 12: RGS with fire.csv

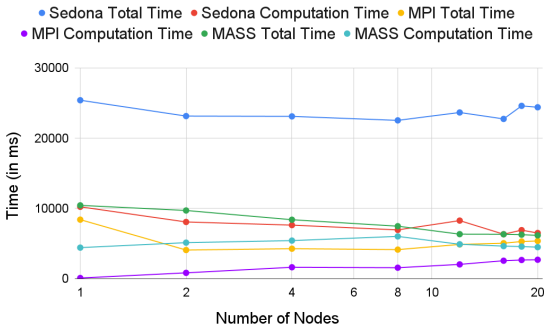

Figure 13: RGS with crime.csv

6 CONCLUSIONS

We parallelized four benchmark programs including CVH, ESP, LEC, and RGS, using MASS, Sedona, and MPI for the purpose of programmability and performance comparisons. Sedona lines up major built-in functions in computational geometry, which facilitates benchmark programming most efficiently. On the other hand, MASS allows us to code the programs from the viewpoint of spatial cognition, which makes them easier to understand than MPI. While MPI runs fastest in general due to its lowest-level parallelization, MASS outperforms Sedona in most benchmark programs. This demonstrates that agent flocking and tree traversing are effective in GIS parallel execution.

ACKNOWLEDGMENTS

This paper is dedicated to Dr. Christian Freksa, a former director of Bremen Spatial Cognition Center, who gave us valuable hints on computational geometry from the viewpoints of spatial cognition. This research was supported by IEEE CS Diversity and Inclusion Fund (IEEE CS Diversity and Inclusion, 2023), the CSS Division's graduate research funds, and the divisional RA-ship supports.

REFERENCES

Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. (2013). Hadoop GIS: A High Performance Spatial Data Warehousing System over MapReduce. In *Proc. of the 39th International Conference on Very Large Data Bases*, pages 1009–1020, Riva del Garda, Italy. VLDB Endowment.

Andrew, A. M. (1979). Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219.

ArcGIS Hub (2023). Private School Locations - Current. Accessed on: November 2, 2024. [Online]. Available: https://hub.arcgis.com/datasets/nces::private-school-locations-current/explore/.

Baig, F., Vo, H., Kurc, T., Saltz, J., and Wang, F. (2017). SparkGIS: Resource Aware Efficient In-Memory Spatial Query Processing. In *Proc. of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 28:1–28;10, Redondo Beach, CA. ACM.

Blashaw, D. and Fukuda, M. (2022). An Interactive Environment to Support Agent-based Graph Programming. In *Proc. of the 14th International Conference on Agents and Artificial Intelligence - Volume 1*, pages 148–155, Online Streaming. SCITEPRESS Digital Library.

Eldawy, A. and Mokbel, M. F. (2015). SpatialHadoop: A MapReduce Framework for Spatial Data. In *IEEE 31st International Conference on Data Engineering*, pages 1352–1363, Seoul, Korea. IEEE.

Freksa, C., Barkowsky, T., Falomir, Z., and van de Ven, J. (2019). Geometric problem solving with strings and pins. *Spatial Cognition & Computation*, 19(1):46–64.

Gilroy, J., Paronyan, S., Acoltzi, J., and Fukuda, M. (2020). Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory. In *7th Int'l Workshop on BigGraphs'20*, pages 2957–2966, Online Streaming. IEEE.

Gordon, C., Mert, U., Sell, M., and Fukuda, M. (2019). Implementation techniques to parallelize agent-based graph analysis. In *Int'l Workshops of PAAMS 2019, Highlights of Practical Applications of Survivable Agents and Multi-Agent Systems*, pages 3–14, Avila, Spain.

Guo, Y. (2021). Construction of Agent-navigable Data Structure from Input Files. MS Capstone White Paper, University of Washington Bothell, Bothell, WA 98011.

IEEE CS Diversity and Inclusion (2023). New Diversity and Inclusion Projects Powered by the IEEE Computer Society Diversity and Inclusion Fund. 15 February 2023 | D&I, DEI, Education, Focus35.

Los Angeles Open Data Portal (2024). Crime Data from 2020 to Present. Accessed on: November 2, 2024. [Online]. Available: https://data.lacity.org/Public-Safety/Crime-Data-from-2020-to-Present/2nrs-mtv8/about_data/.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, New York, NY. ACM.

Mohan, V., Potturi, A., and Fukuda, M. (2023). Automated Agent Migration over Distributed Data Structures. In *Proc. of the 15th International Conference on Agents and Artificial Intelligence*, pages 363–371, Lisbon, Portugal. SCITEPRESS Digital Library.

North, M. J., Tatara, E., Collier, N., and Ozik, J. (2007). Visual Agent-based Model Development with Repast Simphony. In *Agent 2007 Conference on Complex Interaction and Social Emergence*, Chicago, IL.

Saadati, S. and Razzazi, M. (2022). Natural Way of Solving a Convex Hull Problem. https://doi.org/10.48550/arxiv.2212.11999, arXiv.

Spark GraphX (2018). Accessed on: November 2, 2024. [Online]. Available: https://spark.apache.org/graphx/.

Sullivan, K., Coletti, M., and Luke, S. (2010). GeoMason: Geospatial Support for MASON. Technical Report GMU-CS-TR-2016-16, George Mason University.

U.S. Forest Service - Geospatial Data Discovery (2024). National USFS Fire Occurrence Point (Feature Layer). Accessed on: November 2, 2024. [Online]. Available: https://data-usfs.hub.arcgis.com/datasets/6059c1a4dca749d393e33ee5f8a0cbaf_9/about/.

Wilensky, U. (2013). The NetLogo NW Extension for Network Analysis, accessed on: October 5, 2023. [online]. available: http://ccl.northwestern.edu/netlogo/5.0/docs/nw.html.

Yang, Y. (2023). Agents Visualization and Web GUI Development in MASS Java. MS Capstone White Paper, University of Washington Bothell, Bothell, WA 98011.

You, S., Zhang, J., and Gruenwald, L. (2015). Large-scale spatial join query processing in Cloud. In *Proc. of the 31st IEEE International Conference on Data Engineering Workshops*, pages 34–41, Seoul, Korea. IEEE.

Yu, J., Zhang, Z., and Sarwat, M. (2019). Spatial data management in apache spark: the GeoSpark perspective and beyond. *Geoinformatica*, 23(1):37–78.

Zhang, Y. and Eldawy, A. (2020). Evaluating Computational Geometry Libraries for Big Spatial Data Exploration. In *GeoRich'20: Proc. of the Sixth International ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data*, Portland, OR. ACM.