Construction of Agent-Navigable Data Structure from Input File

Yuna Guo

Term report of work done as CSS595

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

Summer 2020

Project Committee:

Munehiro Fukuda, Ph.D., Committee Chair

Min Chen, Ph.D., Committee Member

Robert Dimpsey, Ph.D., Committee Member

1. Overview

MASS (Multi-Agent Spatial Simulation) is a parallel agent-based models (ABMs) simulator and is applied widely for data discovery in a variety of scientific fields, such as bioinformatics, climate science, space cognition and computational geometry. Compared to other parallel data analysis tools, such as MapReduce, Spark and Storm, the MASS maintains the structure of data (e.g. array, tree and graph) and has great advantages in analyzing scientific datasets with complex structures [1].

Two basic classes of MASS library are Places and Agents. The Places are simulation spaces with a multi-dimensional array that is distributed over a cluster system. Agents are entities with a set of mobile objects that can migrate over Places. MASS library performs parallel execution of Places and Agents using multi-thread communicating processes forked over cluster nodes through JSCH and connected via TCP sockets [1, 2]. The current data structure of MASS is multi-dimensional distributed array. In order to support different data structures, the MASS has been being developed and expanded to other data structure, such as contiguous space, trees or graphs. In my independent study in Spring 2020, I have developed contiguous Space class from MASS Places which optimizes data point distribution and improves performance of MASS library. In this report, the contiguous Space class was verified by two applications of computational geometry (e.g. Closest Pair of Points, Voronoi Diagram).

The contiguous Space improves MASS performance because less Places are created which saves memory and shorten execution time. In addition, the customized granularity allows user to adjust precision of Place and Agent's migration. However, there are still limitations in the Space class. For example, an evenly distributed datasets yields the best performance, but clustered data may not be suitable. To solve this problem, I have been developing a Tree class (e.g. quad/octo tree) from Place by which dynamically adds or deletes data point. In this report, I include the preliminary result of quad tree implementation. The granularity of SpacePlace is optimized which ensures that a certain number of data point in each Tree.

2. Progress

In this term, I finished the implementation of Space class and tested it with different applications in single and multiple computing nodes. Firstly, I integrated the Closest Pair of Point function to the Space MASS and the performance was evaluated and compared with original MASS library. The Voronoi Diagram function was also integrated to Space MASS and the performance was evaluated for small input points. The program is still under debugging. Besides the Space class, I have been also working on the

Tree construction. I created a Quad Tree class which distributes all data points. By this Quad Tree class, user is able to obtain an optimized granularity that only one data point resides in each sub-divided Place.

3. Methods

3.1 Voronoi Diagram

The Voronoi Diagram function is implemented in VoronoiDiagram.java and integrated into the MASS library. The VoronoiDiagram.java class reads the input points and maps them to SpacePlace which extends from Place by the coordinates of point. As shown in figure 1, agents are initialized in the input data points which represent original Voronoi site (p1, p2…). Agents propagate from the initial site following an alternating Von Neumann and Moore migration pattern. Eventually agents collide at the perpendicular bisector of corresponding source points and generate the Voronoi edges[3, 4].

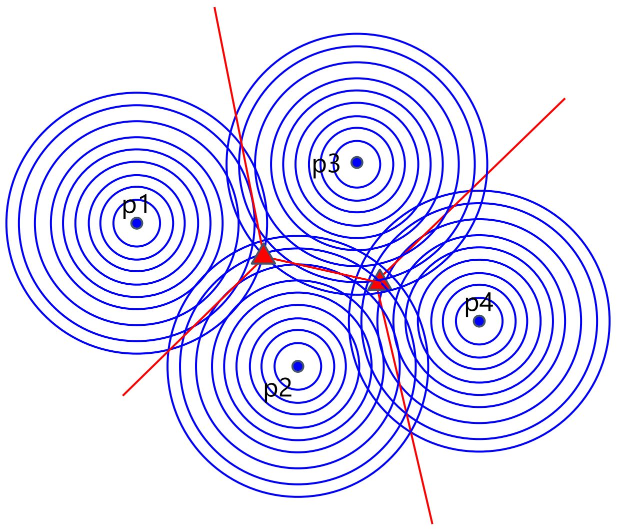


Figure 1. Agents propagate and collide at Voronoi edges.

- computeVoronoiDiagram()

The computeVoronoiDiagram() is the function which reads input, computes and outputs Voronoi diagram. The Voronoi diagram is output as a HashMap<Point, List<ParametricLine>>, in which Point is the source input data point and List<ParametricLine> is a list of all perpendicular bisectors associated with the source point. The algorithm of computeVoronoiDiagram() is shown in table 1.

Table 1. Algorithm of computeVoronoiDiagram()

```
private HashMap<Point, List<ParametricLine>> edges;
private HashMap<PairOfPoint, <ParametricLine>> allPairs;
public void computeVoronoiDiagram() {
    create Places;
    create Agents;
    while (liveAgents() > 0) {
        agents.callAll(Agent.SEARCH);
        agents.manageAll();
        agents.callAll(Agent.MIGRATION);
        agents.manageAll();
        Object[] pairs = Agents.callAll(Space.COLLECT_PAIRS);
        agents.manageAll();
        for each pair in pairs:
            if allPairs does not contain the new pair:
                add each source point to edges with ParametricLine of Slope = Integer.MAX_VALUE;
            else if allPairs contain the new pair which only show once:
                add each source point to edges;
                calculate the perpendicular bisector and add it to edges;
            else if the new pair shows > two times:
                do nothing;
 }
```

- outputVoronoiDiagram()

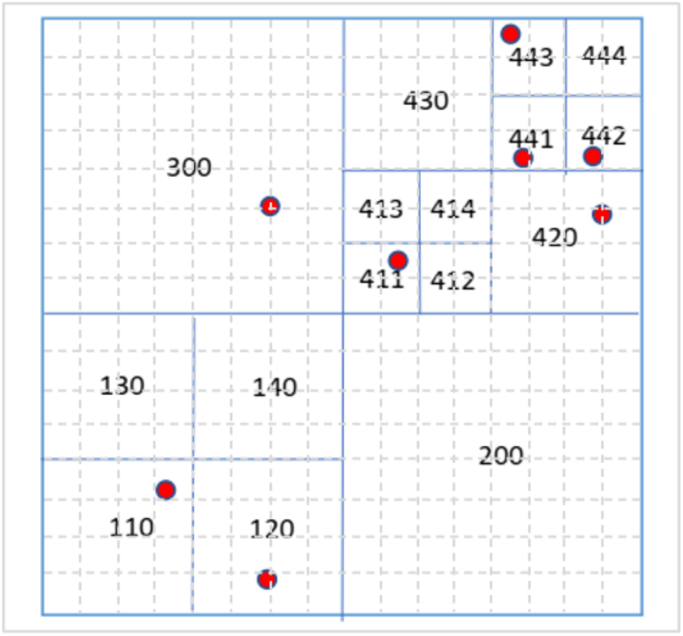
The outputVoronoiDiagram() takes HashMap<Point, List<ParametricLine>> *sortedEdges* as input, and output Voronoi diagram as HashMap<Point, List<Segment>> *vdOutput*. The sortedEdge is to sort List<ParametricLine> in order of the degree to the source point. By sorting edges, the intersection of neighbor edges would be a vertex. By calculating all vertexes, two neighbor vertexes form a segment which is a final Voronoi edge. For vdOutput, the Point is source input point and List<Segment> is a list of segments associated with the point. Furthermore, the closed Voronoi region and open Voronoi region need to be treated differently. If it is a closed Voronoi region, we need to calculate the intersection of the first and the last line which is another vertex and connect it to its neighbor vertexes. If it is an open Voronoi region, the edges have to be re-ordered and the two lines form the open area would be the first and last line. For these two lines, one endpoint is the intersection with its neighbor, the other end point is either negative infinity or positive infinity which depends on the location of the source point. The algorithm is shown in table 2.

Table 2. Algorithm of outputVoronoiDiagram()

```
HashMap<Point, List<Segment>> vdOutput;
public outputVoronoiDiagram(HashMap<Point, List<ParametricLine>> edges) {
        for each <Point, List<ParametricLine>> in edges:
            if the lines form a closed region:
                calculate the intersection of neighbor lines;
                add neighbor intersections to Segment;
                add Segment to vdOutput;
            if the lines form an open region:
                reorder edges as the first and last line are the lines in open area;
                find the endpoints of the first and last line, add them to vdOutput;
}
```

3.2 QuadTree

The QuadTree construction reads data point from file and divides a space quarterly if a new point resides on a space that includes another point [5]. The basic structure of QuadTree is shows in figure 2.



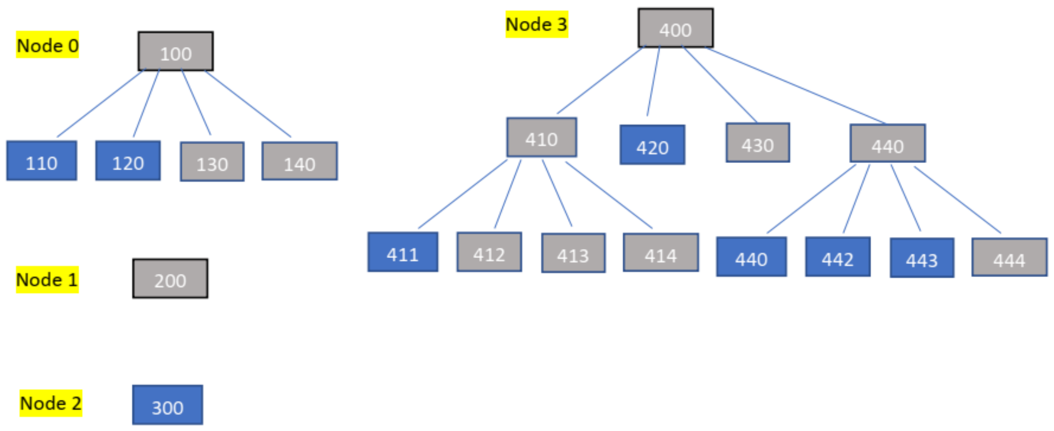


Figure 2. The spatial structure and tree structure of QuadTree.

The QuadTree class is derived from SpacePlace. The attributes include a boundary which is the range of QuadTree, index which is a unique index of the tree, level of node and its four subtrees. The functions include split() which splits the tree into four subtrees, insert() which adds data to the tree and delete() which removes data from tree .

To distribute data among the cluster system, the QuadTree is partitioned to different computing nodes according to its spatial coordinates. When MASS initialized, the program creates a boundary map that contains the range of QuadTree in every computing node and send the map to all nodes. When the program reads input data, it distributes the data point to the corresponding computing node. Then the computing node adds the data to the destination QuadTree by insert(). Agents are initialized in the QuadTree which has data point resides. When agent migrates, program firstly searches for the destination QuadTree in its current computing node. If the current node contains the destination QuadTree, agent migrates to the destination directly; if the current node doesn't contain the destination QuadTree, it searches for the destination computing node by the boundary map and send message of migration to the destination node.

QuadTree **extends SpacePlace(Object args)**
- Boundary boundary; - int[] index; - int level; - QuadTree topLeft, topRight, bottomLeft, bottomRight;

+ split();
+ insert ();
+ delete();

4. Results

4.1 Execution of Closest Pair of Points by Space MASS

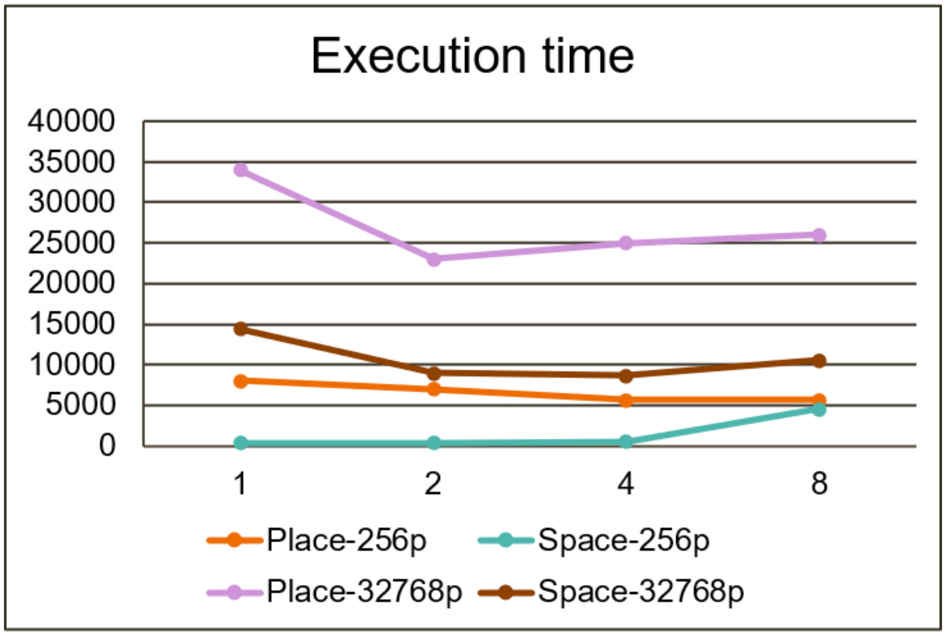
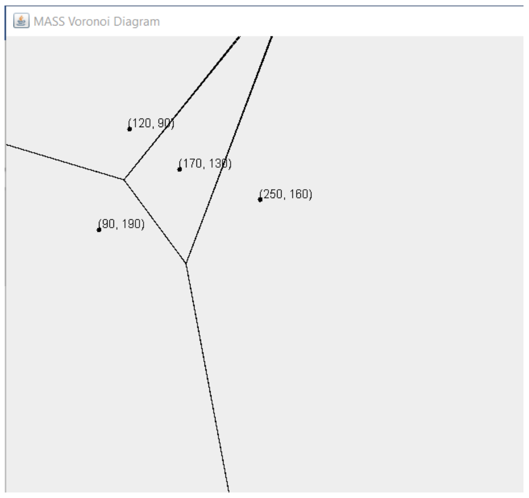


Figure 3. Comparison of Space MASS and original MASS in Closest Pair of Point

As shown in figure 3, for 256 input points, the execution of program by Space MASS is > 20 times faster than the original MASS library with 1, 2 and 4 computing nodes. With 8 nodes, the execution time is very close. For 32768 input points, the execution of program by Space MASS is about 2.5 times faster than the original MASS library with 1, 2, 4 and 8 nodes. The execution time of Space MASS is about 10 times in 8 computing nodes compared to 4 nodes, which may be due to the overheads of connection between computing nodes.

4.2 Voronoi diagram

The Space MASS was tested in Voronoi diagram with a small number of input points on single computing node. The results of 4 points input and 6 points input are shown in figure 4. In table 3, the execution speed is about 30 times faster than the original MASS.



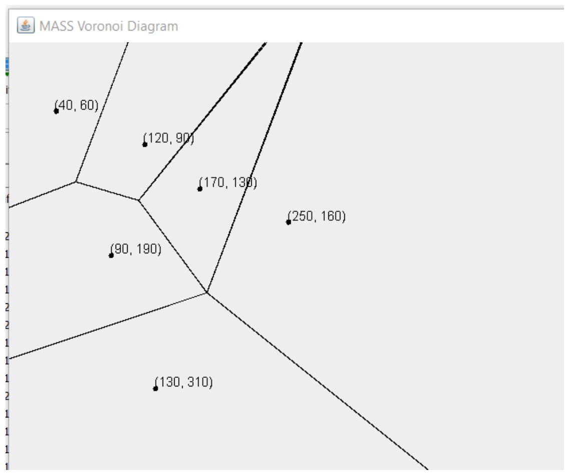


Figure 4. Voronoi diagram of 4 points and 6 points

Table 3. Execution time of Voronoi diagram in 4 and 6 points input

	4 points (ms)	6 points (ms)
Space MASS	881	1036
Original MASS	24332	34506

4.3 QuadTree

Before deriving QuadTree class from Space, a simplified QuadTree class was created to compute the optimized granularity by which at most one agent resides in sub-divided place of SpacePlace. The granularity for different size of input points is shown in table 4. For small number of input points, the granularity is mostly calculated as 2. When the input points increases, as the places are finer and the points are more evenly distributed, the granularity is calculated as 0.

Table 4. Execution time of Voronoi diagram in 4 and 6 points input

Input points	**granularity**
8 p	2
64p	2
256p	2
32768p	0

5. Project Package

All Source code can be found under the Project directory. Link to the repository: https://bitbucket.org/mass_library_developers/mass_java_core/src/75ab402bf7ad83c55b03582b971207abf51d8dce/?at=yunaguo%2Fspace

6. Conclusion & Discussion

Conclusion

The execution performance of Space MASS outperforms original MASS in both Closest Pair of Points and Voronoi diagram. The execution time is faster for different number of computing nodes and different number of input points. With the contiguous Space class, the execution speed is improved more significantly when the input data points are less. This is because the number of Place is determined by the number of input data point in Space; however, the number of Place is determined by the maximum value of input data point in original MASS. Therefore, more Places are created when the input data have large value but less quantity.

Limitations and improvement

In the Space MASS, the entire input file is read in PlacesBase class by each node and the data are passed as an ArrayList to init_all(). A giant input data may use all of memory which slows down the computing or cause algorithms crash. In addition, reading entire node in each node increases execution time. To improve the I/O, the large input file should be partitioned and each computing node should read / write the partitioned file in parallel. A divisible file (e.g. txt or CVS file) can be partitioned and stored at a different computing node's tmp which can be accessed by local node. For non-divisible file (e.g. NetCDF), the MASS can be set with a different offset and jumps to its partitioned data directly.

For Voronoi diagram application, I was only able to generate the correct Voronoi diagram for few input points. The 8 points output is shown in figure 5. I found the incorrect vertex was generated by the perpendicular bisectors that should not form. For example, the vertex (pointed by red arrow) is generated by the line (pointed by blue arrow) which is the perpendicular bisector of (30, 90) and (100, 30) which should not meet and collide. Based on the current result, the execution time is significant improved by the Space in single node computing. I will debug my program and include the result for multi-nodes in my next term report.

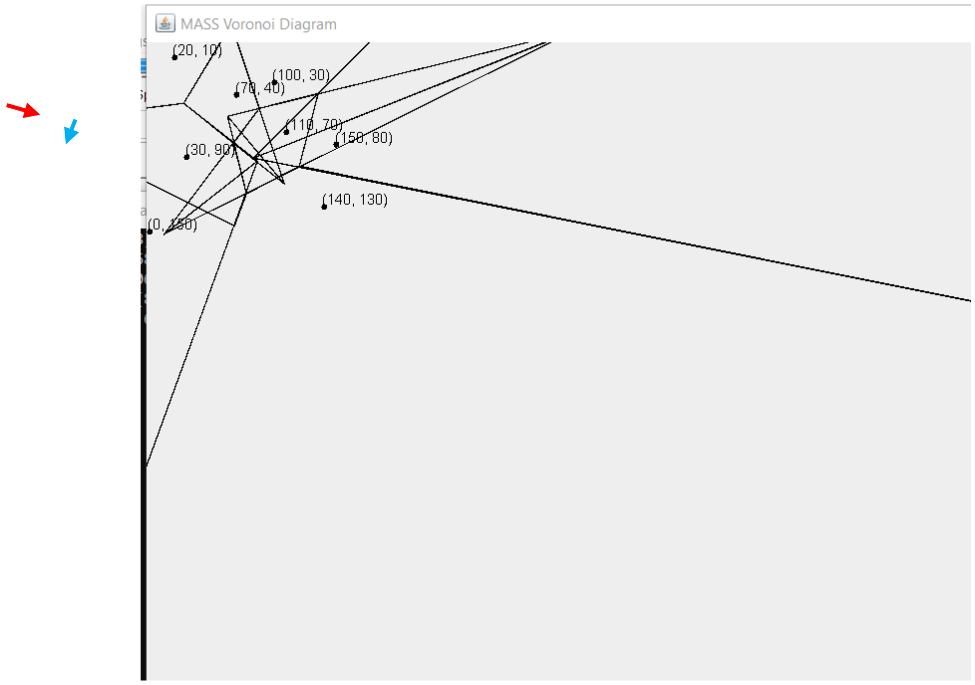


Figure 5. Voronoi diagram of 8 points.

7. References

1. Munehiro Fukuda, Collin Gordon, Utku Mert, and Matthew Sell. An agent-based computational framework for distributed data analysis. *Computer*, 53(3):16{25, 2020.
2. Munehiro Fukuda and Distributed Systems Lab. MASS Java Manual
3. Wikipedia.org. Voronoi diagram – Wikipedia
4. Saranya Gokulramkumar, Agent Based Parallelization of Computational Geometry Algorithms, Master Thesis, University of Washington, Bothell
5. Wikipedia.org. Quad tree – Wikipedia