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

Fall 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 to support other data structures, such as contiguous space, trees or graphs. In my capstone project, I have been developing contiguous Space and Tree data structure. In this report, I will mainly discuss Quad Tree and Binary Tree.

The Quad Tree improves MASS performance as it reduces number of Places by spatial decomposition. In Quad Tree Place, a two-dimensional space (which is a Place) is partitioned into four Places by recursively subdividing. In comparison to contiguous Space, the Quad Tree avoid multiple data points clustering in a single Place when data points are not evenly distributed. In this study, the Quad Tree is verified by the Closest Pair of Point and Voronoi Diagram. The execution performance is evaluated in comparison to the original Place and Space.

Another tree class derived from Place is Binary Tree, which allows user to dynamically add or delete tree node. The Binary Tree could be used to simulate divide-and-conquer problems and decision trees. In this study, the Binary Tree data structure is tested by Range Search.

2. Previous Work

In previous work, I have implemented contiguous Space in MASS library. The internal structure of SpacePlace is the same as Place. In contrast to the original Place which defines the data space as a discrete integer, the data space in SpacePlace is defined in a continuous manner. In addition, the finesse of sub-place is determined by granularity which is defined by users. A HashMap is used to trace agent and its corresponding sub-index in a place. If more than one agent reside in one sub-place, the agents collide. The contiguous Space class was tested by Closest Pair of Points and Voronoi Diagram and both applications' performance was improved.

On average, the execution time was 2.5 times faster for Closest Pair of Points and 50 times faster for Voronoi Diagram.

3. Progress

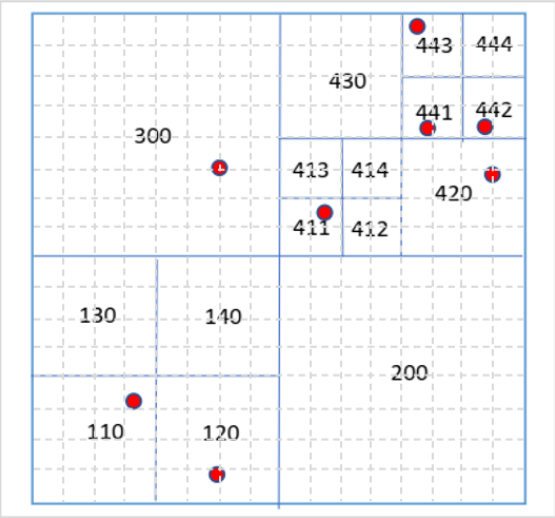
In this term, I finished the implementation of Quad Tree and Binary Tree and tested them with different applications.

	Design	Implementation	Verification
Contiguous Space	Completed	Completed	Closest Pair of Points: completed Voronoi Diagram: in progress
Quad Tree	Completed	Completed	Closest Pair of Points: completed Voronoi Diagram: in progress
Binary Tree	Completed	In progress	Range Search: in progress

4. Implementation

4.1 Quad Tree

The Quad Tree 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 Quad Tree is shows in figure 1.



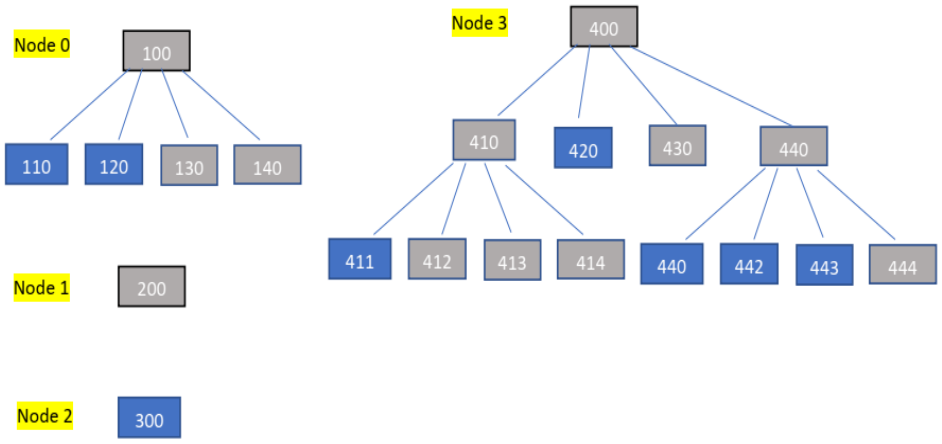


Figure 1. The spatial structure and tree structure of Quad Tree.

To distribute data among the cluster system, the Quad Tree is partitioned to different computing nodes according to its spatial coordinates. When MASS initialized, the program creates a boundary map that contains the boundary of root 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. The computing node adds data to the destination Quad Tree by insert(). Agents are initialized in the QuadTreePlace which has data point. When agent migrates, program firstly searches for the destination QuadTreePlace in local computing node. If the local node contains the destination place, agent migrates to the destination directly; if not, it searches for the destination computing node by the boundary map and sends message of migration to the destination. The details of class design are as follows.

Create QuadTreePlaces instances

```
QuadTreePlaces places = new QuadTreePlaces (int handle, String classname, int dimensions, String filename, Object args, int granularityMultiplier)
```

QuadTreePlaces manages all QuadTreePlace elements within the simulation space. QuadTreePlaces instantiates a tree in which tree nodes are QuadTreePlace objects from the “classname” class as passing an argument to the “classname” constructor by reading data from an input file. This tree is associated with a user-given handle which is unique over machines. The user is able to further sub-divide the QuadTreePlace by granularityMultiplier. In QuadTreePlaces methods, the init_master_quadTree() firstly calculates boundaries of computing nodes and

initializes places with given argument. The callAll() calls the method specified with functionId of all places and is done in parallel among multi-processes/threads.

QuadTreePlace is inherited from Place. In tree initialization, the QuadTreePlace is sub-divided to sub-places and the space of sub-place equals to the space of the most divided QuadTreePlace. The key functions includes insert(), search() and callMethodHelper(). The insert() is used to insert a node into an existing QuadTreePlace. The search() is used to locate a node in a given Quad Tree. Because QuadTreePlace is in a tree structure instead of an array, a helper function is used to recursively call callMehod() for each place. The algorithm of callMethodHelper() is shown in Listing 1.

Listing 1. Algorithm of callMethodHelper()

```
private Boolean isLeaf;
private QuadTreePlace topLeft, topRight, bottomLeft, bottomRight;
public Object callMethodHelper (int functionId, Object argument) {
   if quadTreePlace is a leaf
         Object retValue = callMethod(functionId, argument);
   if topLeft != null
         topLeft.callMethodHelper(functionId, argument);
   if topRight != null
         topRight.callMethodHelper(functionId, argument);
   if bottomLeft != null
          bottomLeft.callMethodHelper(functionId, argument);
   if bottomRight != null
          bottomRight.callMethodHelper(functionId, argument);
   return null;
}
```

4.2 Binary Tree

In Binary Tree, the BinaryTreePlace derives from Place and has two children places. The places are initialized and distributed across the cluster based on input data. The master computing node reads input, calculates boundary of computing node and sends it to each computing node. The computing node reads input in parallel and builds the binary tree according to its boundary. Although the binary tree is local, each node has boundary of all nodes and share

a global index. A two-dimension integer array is used for the indexing system. The first dimension is the rank of computing node and the second dimension is the local index that assigned sequentially. The basic structure of Binary Tree is shows in figure 2.

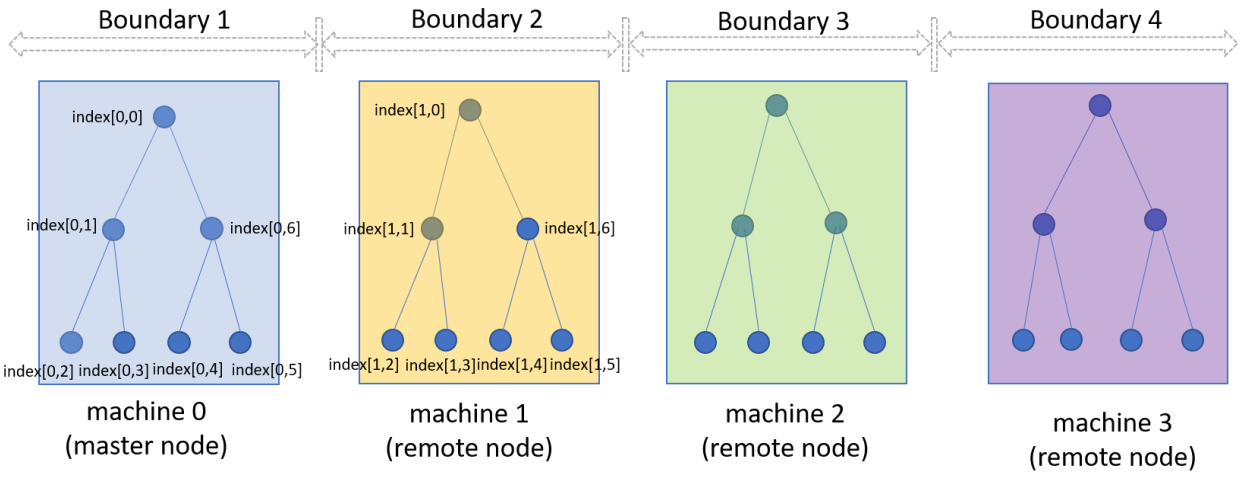


Figure 2. The structure of Binary Tree.

Agents are initialized, spawn and migrate in BinaryTreePlaces. When agent migrates, if the destination place is in local node, agent migrates to the destination directly; otherwise the migration request will be sent to the destination computing node via message based on its global index.

Create BinaryTreePlaces instances

```
BinaryTreePlaces places = new BinaryTreePlaces (int handle, String classname,
String filename, Object args)
```

The BinaryTreePlaces instantiates a tree in which tree nodes are BinaryTreePlace objects from the “classname” class as passing an argument to the “classname” constructor by reading data from an input file. This tree is associated with a user-given handle which is unique over machines. In QuadTreePlaces methods, the init_master_binaryTree() firstly calculates boundaries of computing nodes and initializes places with given argument The function addPlace(BinaryTreePlace newPlace) is used to add a new place to the corresponding tree.

5. Results

5.1 Verification of Quad Tree by Closest Pair of Points

The Quad Tree was firstly tested by the Closest Pair of Points. Figure 3 shows the execution time of Closest Pair of Points with 32768 data points by original Place, SpacePlace and QuadTreePlace. The computing time by QuadTreePlace and SpacePlace is close, which is about 1.5 times faster than the original Place in MASS library. Multiple computing nodes does not reduce execution time significantly. This may be due to the overheads of communication between computing nodes and relative fast execution by single node.

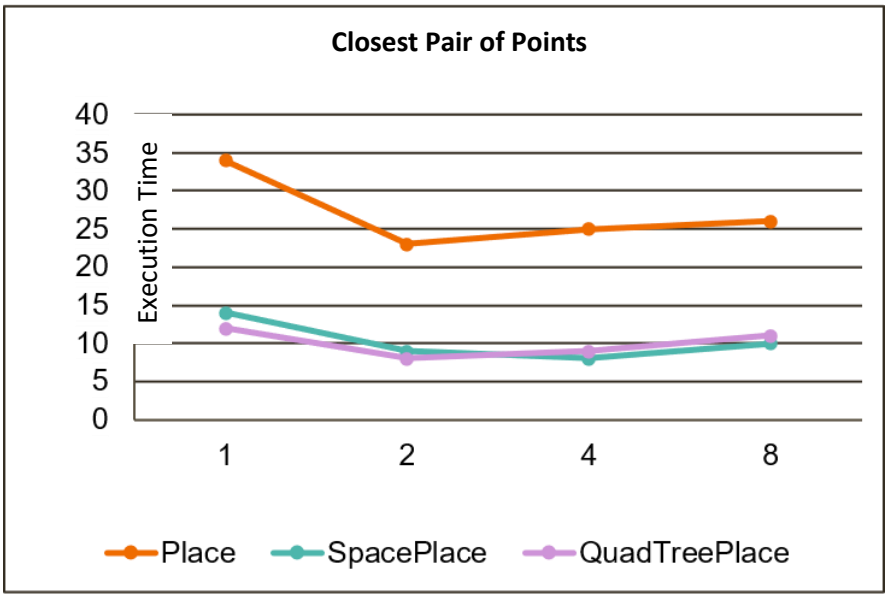
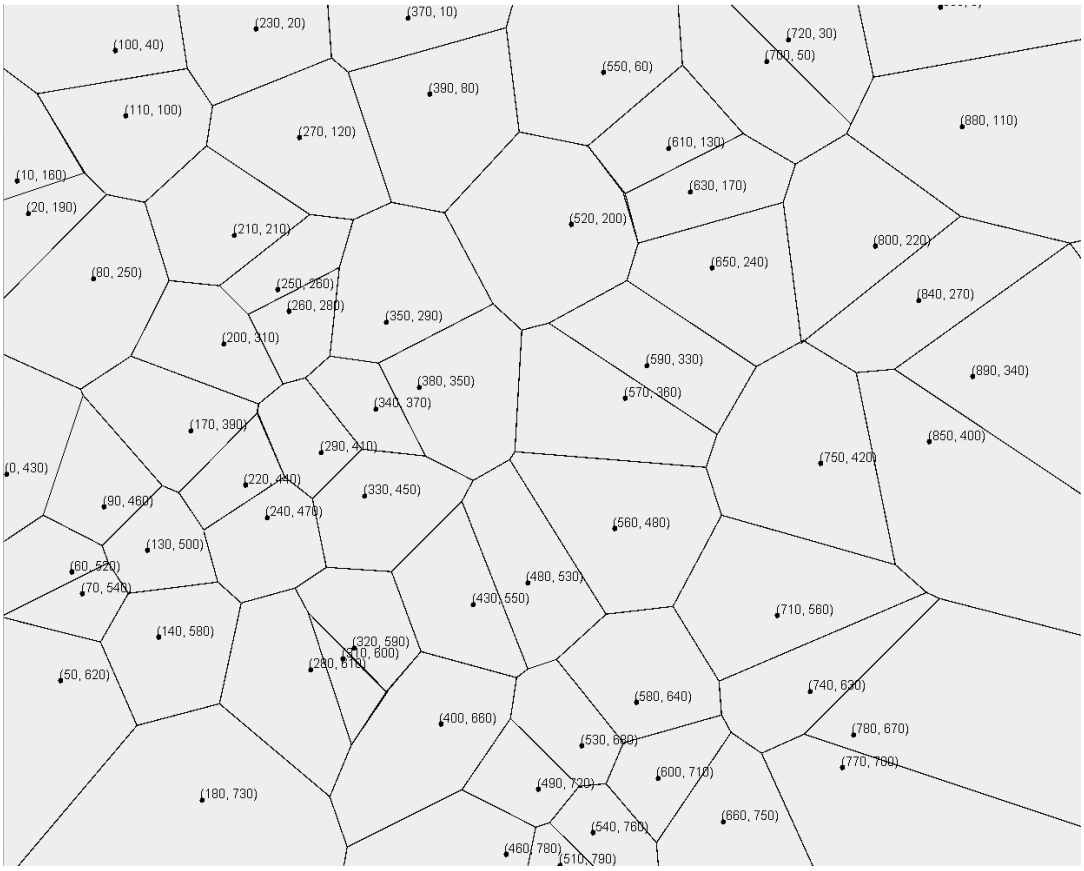


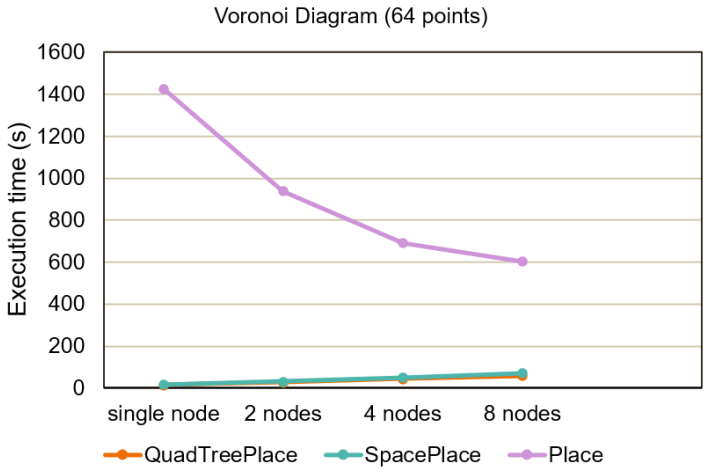
Figure 3. Execution time of Closest Pair of Points (32768 points) by QuadTreePlace in comparison to SpacePlace and original Place in MASS library.

5.2 Verification of Quad Tree by Voronoi diagram

In addition to Closest Pair of Points, the Quad Tree is also verified by the Voronoi diagram. Figure 4 shows the result of 64 points. In this experiment, the size of sub-place equals to the minmimum QuadTreePlace. The execution speed is about 90 times faster than the original Place in single node and about 10 times faster in 8 computing nodes. Similar to the Closest Pair of Points, when more computing nodes are added, the execution time increases. This is because the execution in single node is very fast (16 seconds) and building connection between computing nodes increases the total computing time.



(a)



(b)

Figure 4. (a) Voronoi diagram of 64 points. (b) Execution time of Voronoi diagram for 64 points.

However, the output of Voronoi diagram is not correct in some cases. As shown in figure 6, unnecessary Voronoi edges (pointed by arrows) are found. Using a visualization program which

prints out agents and places for every iteration, it was found that some agents collide in place which should not occur. Figure 7 shows propagation of agents from source point (630, 360), (600, 390), (650, 460) and (700, 370). Although there should not be an edge between point (600, 390) and point (700, 370), agents from those two source points propagate and collide. This is because the propagation of agents forms an octagon instead of a circle. The agent in diagonal direction migrates faster than agent in vertical or horizontal direction. To solve this problem, when an edge found by collision, we need to examine whether the edge is a valid before adding it into the output. Current I am working on developing algorithm to examine the validity of a Voronoi edge.

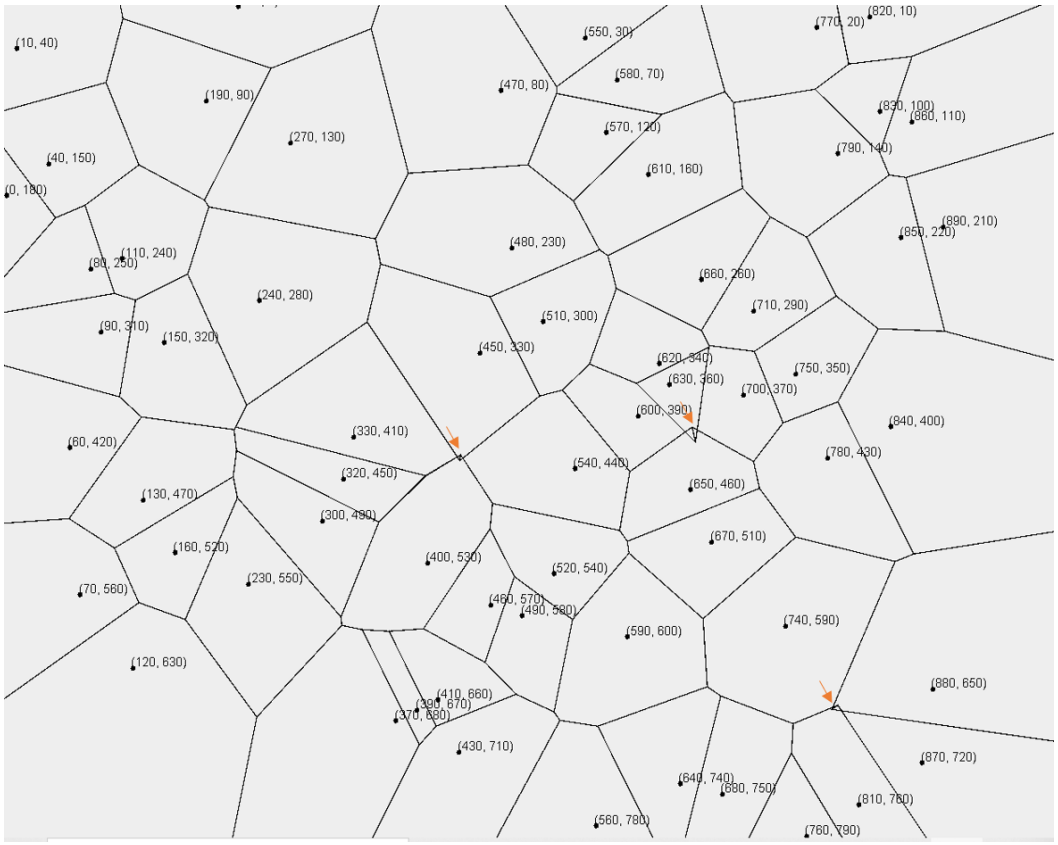


Figure 6. Voronoi diagram with different granularities.

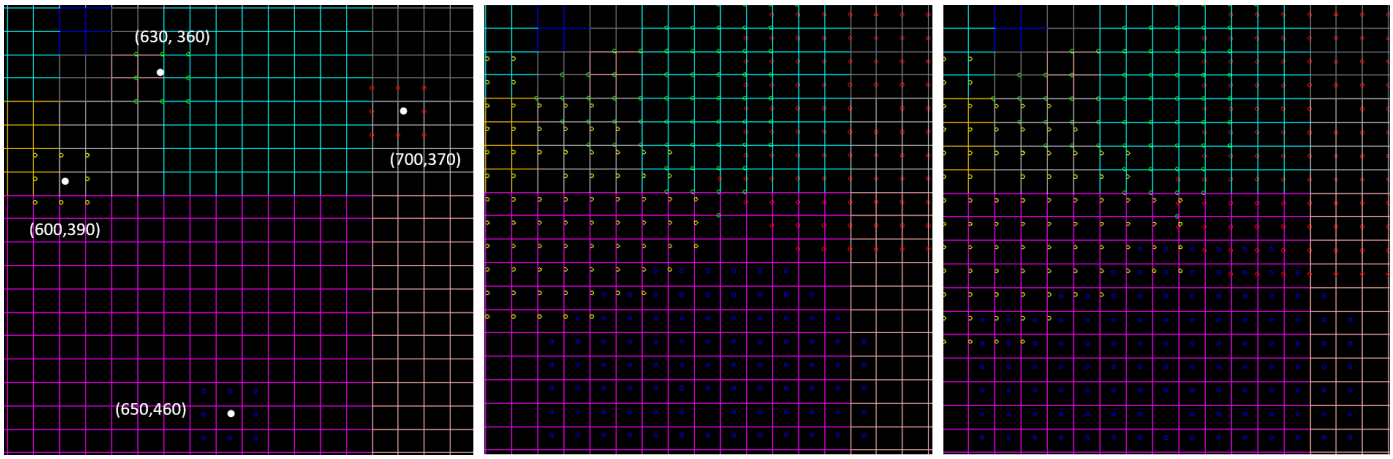


Figure 7. Visualization of agent propagation.

5.3 Verification of Binary Tree by Range Search

The Range Search application was used to test the Binary Tree, including 500,000 data point input and 5,000,000 data point input. The execution performance is shown in figure 8. With 500,000 data point, the execution time is increased when more computing nodes are added. This may be also due to the overhead of communication between nodes. With 5,000,000 data point, when more computing nodes are added, the execution time decreases. This is because multiple computing nodes read file and search for range in parallel.

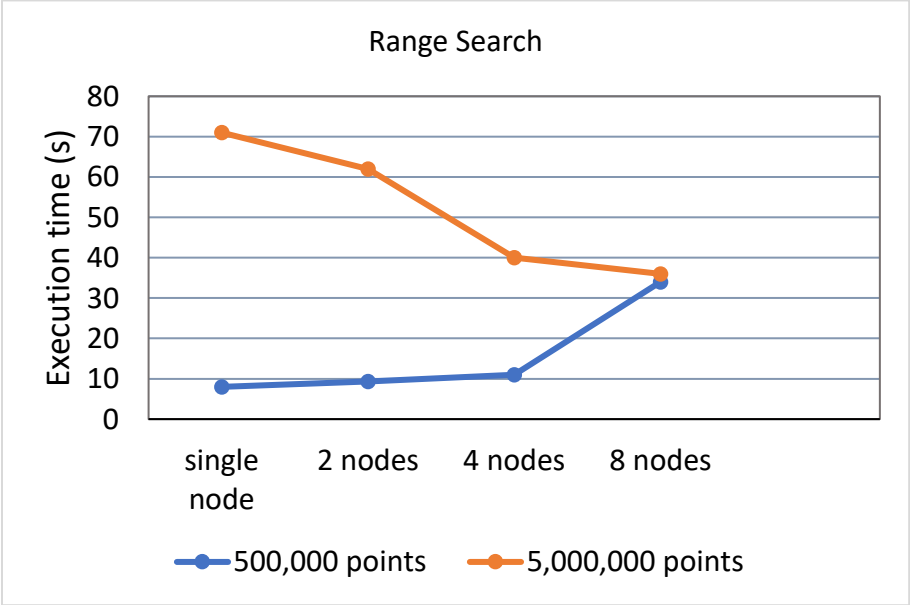


Figure 8. Execution performance of Binary Tree by the application of Range Search.

A potential issue that I found is that when the program is executed with 5,000,000 data point in multi-nodes, some computing node is not able to finish building binary tree. When I ran the program on hermes01-08, the hermes08 sends "client failed" message when it attempts to establish connection with hermes06 because hermes06 could not finish building the tree. However, this error did not occur in cssmpi machines. I am debugging this problem which might be related to Hazelcast.

6. 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

7. Conclusion

The QuadTreePlace outperforms original Place in both Closest Pair of Points and Voronoi Diagram. The execution speed is improved more significantly in Voronoi diagram. This is because the computation of Voronoi diagram is more complex. More Places and Agents are created and the Quad Tree data structure significantly reduces the number of Places and Agents.

In Range Search, the BinaryTreePlace executes successfully with large input (5,000,000 data point) and outputs correct searching result. As the number of computing node increases, the execution time reduces. For example, compared to single node, the cluster system with eight computing nodes reduces execution time from 71 seconds to 36 seconds.

8. 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
6. Wikipedia.org. Binary tree – Wikipedia
7. Wikipedia.org. Range Search – Wikipedia