Construction of Agent-navigable Data Structure from Input Files

Yuna Guo

A thesis submitted in partial fulfillment
of the requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

March 2021

Committee:

Munehiro Fukuda, Ph.D., Committee Chair

Min Chen, Ph.D., Committee Member

Robert Dimpsey, Ph.D., Committee Member

Abstract

Construction of Agent-navigable Data Structure from Input Files

Yuna Guo

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda
Computing and Software Systems

The multi-agent spatial simulation (MASS) library is an agent-based parallelizing library for analyzing structured datasets over a cluster of computing nodes. The current version of MASS library supports distributed multi-dimensional arrays and graphs. In this capstone project, we aim to develop three distributed data structures in MASS, including Continuous Space, Quad Tree, and Binary Tree. First, we designed and implemented these three data structures. Second, we used two geometric applications – Closest Pair of Points and Voronoi Diagram to evaluate the programmability and execution performance of Continuous Space and Quad Tree. Third, we implemented a searching application – Range Search with Binary Tree. Thereafter, we measured programmability, execution time, and memory consumption for performance evaluation. In comparison to the original MASS or MASS Graph, the programmability result shows that all three implementations reduce LOC (line of codes), the number of classes, and the number of methods. The performance evaluation shows that all the three implementations reduce execution time and memory consumption for the applications. The project successfully carried out two achievements: (1) the Continuous Space and Quad Tree facilitate users apply MASS to geometric problem and (2) the Binary Tree allows users to apply a log N search and divide-and-conquer algorithm to their applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Chapter 1. Introduction

1.1 Background and Motivation

Distributed data analysis has been widely used in big-data analytics as increasing amount of data volumes and demands in processing speeds. A variety of software tools such as MapReduce, Spark, and Storm have been used by scientist for easy parallelization of user applications. The key concept of data streaming is analyzing units (e.g., map/reduce functions in MapReduce, lambda expressions in Spark, and spouts/blots objects in Storm). To achieve the best performance, these tools read input and process data in parallel within multi-threaded computing cluster systems. However, the structure of data couldn't be maintained as these tools flatten and shuffle dataset [1-3]. Scientific datasets usually have complex structures, such as arrays, tree or graph, and they may not always fit data streaming supported by these software tools [4-5]. Therefore, it is necessary to develop tools of parallel data analysis which preserves data structures in distributed memory.

Agent-based model (ABM) is a class of computing models for simulating the actions and interactions of agents within a system [6]. The ABM has been successfully applied to many applications, including transportation, ecological, and biological simulations [7-8]. Multi-agent spatial simulation (MASS), which is a parallel ABM simulator, distributes a given dataset by creating a multidimensional array of objects (Places) over a computing cluster, and simulates entities with a set of mobile objects (Agents) that migrate over Places. The MASS performs parallel execution of Places and Agents using multi-threaded processes over a cluster of computing nodes [9-10]. The MASS has been successfully applied agent-based data analysis on a structured data set in many scientific fields, such as biological networks, computational geometry, and environmental data analysis [11-13].

Our previous work has confirmed the feasibility of MASS in agent-based programming, in which MASS shows performance advantages in parallel data analysis in comparison to other available parallel data streaming tools in some types of applications [11-14]. However, there are still performance challenges. For instance, Gokulramkumar evaluated the performance of MASS in computational geometry applications in comparison to MapReduce and Spark [14]. In Closest Pair of Points (CPP) application, the MASS implementation outperforms MapReduce and has comparable performance to Spark. In Voronoi Diagram (VD) application, the MASS execution is slower than Spark and MapReduce, which is due to the huge size of Place objects created by MASS. In MASS, the Places are discrete grids where each Place has integer coordinates with an index interval of 1 in each dimension. If the number of geometric data points is small but their distances are large, a huge size of Places are instantiated but only a small portion of them contain data points, which wastes memory and slows down the execution. One strategy to solve this problem is to develop a continuous space of Places in which each Place covers a certain range of coordinates. Furthermore, in order to optimize the Place instantiation (e.g. avoid allocating multiple agents onto a particular Place), Quad Tree data structure is implemented.

Beyond the Continuous Space and Quad Tree classes, users would like to use a more variety of data structures. To address their demands, we also implemented Binary Tree that is distributed over a cluster system. Although the Continuous Space and Quad Tree in MASS will improve the performance of the current MASS implementation by reducing the number of Places, if users would like to apply MASS on a data structure different from an array, it is still inconvenience as the users have to implement the data structure extending from Place in their own application. Therefore, it is necessary to implement other data structures in MASS which users could use directly. As tree is used in various applications, including data storage, searching, and sorting algorithm [15], we firstly focus on the simplest tree structure - Binary Tree.

1.2 Research Objective

This project aims to implement and evaluate three data structures in MASS: (1) Continuous Space, (2) Quad Tree, and (3) Binary Tree.

1. **Design and implementation of agent-navigable and distributed data structures.** We aim to derive Continuous Space, Quad Tree, and Binary Tree from Place in MASS library, which will facilitate users to apply MASS in structured data analysis.
2. **Evaluation of the programmability and execution performance.** We aim to improve the programmability and execution performance of applications, particularly testing Continuous Space and Quad Tree in geometric problems (e.g., Closest Pair of Points and Voronoi Diagram) and Binary Tree in searching problems (e.g., Range Search).

1.3 Project Overview

In order to achieve these project objectives, we modify MASS library by deriving Continuous Space, Quad Tree, and Binary Tree from Place class. In the Continuous Space class, instead of a discrete grid, the Place is a continuous geometric space with user-defined range. Similarly, in the Quad Tree, the range of each Place is optimized automatically by program based on input data distribution. The Place is divided into sub-places and agents in sub-places are recorded by an agentsMap, which is maintained using a hash table whose key/value pair is a linear index of sub-place and the corresponding agent reference. Both Continuous Space and Quad Tree classes are verified with the Closest Pair of Points and the Voronoi Diagram. In Binary Tree, the tree node is the Place that is added dynamically. The Binary Tree data structure is verified with Range Search. We evaluate the implementation of above three data structures using three metrics: programmability, execution time, and memory usage.

Chapter 2. Related Work

In this chapter, we discuss the current approaches for big-data analysis and challenges of these tools, including data-streaming approach (e.g., MapReduce, Spark, and Storm) and agent-based model (ABM) (e.g., Repast Simphony and FLAME). We also introduce our own parallel ABM simulator – MASS and its applications.

2.1 Data-streaming approach

Hadoop MapReduce is a framework for processing and generating large datasets in a parallel and distributed way. MapReduce is composed of two phases: Map and Reduce. Key-value pairs are the basic data structure in MapReduce. Before the Map operation, the master node firstly splits a dataset and distributes it across the computing nodes. The Map function is performed to <key, value> pairs and produces a set of intermediate <key, value> pairs. The results are grouped and redistributed across the cluster. The Reduce phase applies a Reduce function to each list value and produces a single output. All these processes are executed in parallel [1, 16]. Because MapReduce writes intermediate output to disk between each stage, the costly I/O operation reduces processing speed. Spark is a framework to extend MapReduce model for performing fast distributed computing by using in-memory primitives. The key element of Spark is Resilient Distributed Datasets (RDDs), which is an immutable distributed collection of objects that can be operated on in parallel. Spark allows a user program to load data into memory and query it repeatedly, which is good for online and iterative processing [2, 17]. Since creating an RDD and caching it causes memory consumption, fine tuning of RDD partitions is required to ensure that the data destined for each task fits in the memory available for that task.

Although these data streaming approaches are powerful tools for parallel data analysis and easily used by scientists who are non-computing specialists [1-3, 16-17], not all scientific data could fit into the in-memory structure of these tools and it takes substantial effort for users to apply these tools on scientific data analysis [4-5]. For example, the climate data analysis uses the NetCDF format which is a multi-dimensional array. SciHadoop could be used to partition a file into small blocks and to group them for MapReduce for processing them as an unstructured dataset. However, the original data structures could not be maintained, and the spatial relationships and patterns are lost after data flattening and shuffling [11]. Thus, it is important to develop data analysis tools which maintain a structured dataset in distributed memory and analyze unit over the structured data.

2.2 Agent-Based Model

ABM is a system that consists of a collection of autonomous decision-making entities called agents. ABM has been used successfully to simulate real-life problems in many disciplines, including biology, ecology, and economics.

The Recursive Porous Agent Simulation Toolkit (Repast) is a free and open source simulator that was originally developed by the University of Chicago in 2000, which is now able to handle large-scale agent simulation application development. Repast Simphony was built on Repast 3 with focusing on well-factored abstractions. The core concept and object in Repast Simphony is called Context, which is a simple container of any type of object. Contexts are containers for agents and projections. Agents join or leave Contexts freely and can simultaneously exist in multiple Contexts and sub-Contexts. Projection defines the relationship between agents in a given context, including multidimensional discrete grids, multidimensional continuous spaces, networks, and geographical information systems (GIS) spaces. The key concept of continuous space projection is similar to the Continuous Space implementation in MASS, in which an agent location is represented by floating point coordinates. However, the Repast Simphony does not support loading a continuous space from an input file and the continuous space could not be sub-divided into sub-spaces. Besides the continuous space, Repast Simphony also supports graphs and networks as a projection with methods, such as addVertex and addEdge. The Repast Simphony supports loading a graph from an input file with limited formats including UCINet's DL format and Excel format, but it does not support parallel I/O which is necessary for huge data inputs. Additionally, Repast Simphony systems uses Eclipse as its primary development environment and leverages Eclipse's plug-in architecture to provide a set of development options - tools, views, and perspectives for creating Repast-specific model components [18-19]. However, the major drawback of Repast Simphony is that it does not support distributing data over a cluster system.

Flexible Large-scale Agent Modeling Environment (FLAME) is another major ABM that can run on high performance computers. In FLAME, agent models are written in a combination of XML and C languages as basis. Each agent acts as a finite-state machine, whose state is changed via a set of transition functions. Although FLAME is able to run a simulation with up to 10^6 agents, agents are not capable of migrating between processes and each agent maintains the entire dataset independently. Because the FLAME uses broadcast communication for agent, agents can not directly send messages to each other, and the receiving agent must filter messages that it only needs to read [8, 20]. Due to these limitations, FLAME is not suitable for data discovery and analysis.

2.3 MASS library

MASS (Multi-Agent Spatial Simulation) is an agent-based parallelizing library that simulates real-life problems, including bioinformatics, climate science, space cognition, and environmental data science [11-14, 21]. As shown in Figure 2.1, the two main components in MASS are Places and Agents. The Places are simulation spaces with a multi-dimensional array that is distributed over a cluster system. The Agents are simulation entities with a set of mobile objects that can migrate over Places. Each computing node has multiple threads of execution for Places and Agents. MASS library performs parallel execution by distributing a multidimensional array of Places across computing nodes and migrating Agents from one Place to another. The Places and Agents are further parallelly executed by multi-thread communicating processes forked over cluster nodes through JSCH and connected via TCP sockets [10, 22].

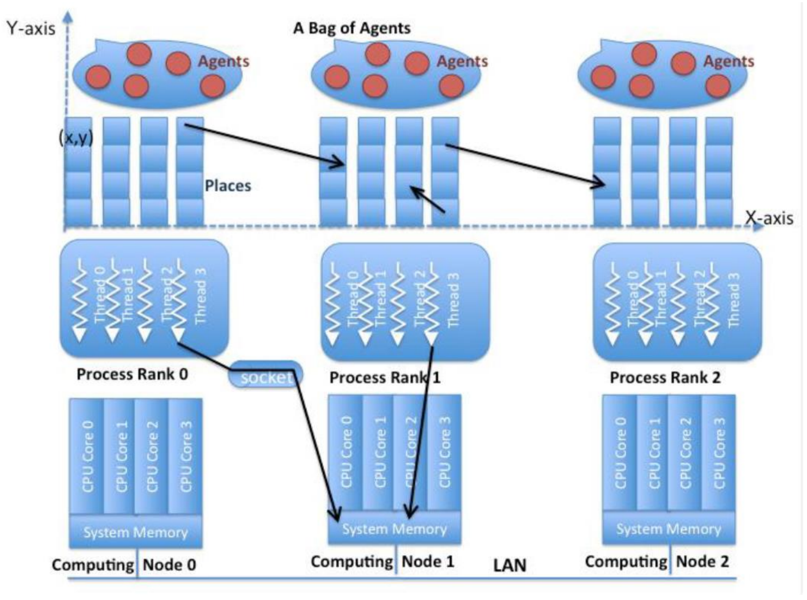


Figure 2.1 MASS library data model [10].

The performance of MASS has been evaluated in six benchmark programs, including Closest Pair of Points, Voronoi Diagram, and Range Search.

(i) The Closest Pair of Points is to find out the closest pair of points in a metric space of n points, which is the basis of many complex computational geometry problems and is used in many real-world applications, such as collision detection in air traffic [23]. The most common algorithm to solve this problem is the recursive divide and conquer, which is also used in parallelization with MapReduce and Spark [23-24]. In MASS, Places are created as a 2D array and Agents propagate across Places. Gokulramkumar implemented the Closest Pair of Points by three platforms which shows Spark as the fastest execution, MapReduce as the slowest, and MASS in the middle of them [14]. In addition, Wenger and Acoltzi found that the Repast Simphony and JCilk improve the performance of the application by using multithreading.

(ii) The Voronoi Diagram over a set of 2D points is a collection of regions that divide up the plane, in which each region corresponds to one of the points and all the points in one region are closer to the corresponding point than any other point [25]. Its application includes networking, robot navigation and computer graphics [26-28]. The basic implementation strategies in MapReduce, Spark, and MASS are the same as the Closest Pair of Points. In Gokulramkumar's work [24], the result shows Spark outperforms MapReduce and MASS in single computing node. The execution time of MASS reduces when the size of computing nodes increases [14].

(iii) The 2D Range Search problem is to search for points that reside in a querying rectangle region (range) from a set of N points in a plane [23]. The algorithm in MapReduce and Spark is divide-and-conquer. In MASS, Places are created to mimic a KD tree using the graph class and Agents traverse Places to search for the points in the range. In Paronyan's work, the execution time in MASS is slightly shorter than the execution time in MapReduce and Spark and the execution time reduces when the size of computing nodes increases [13].

The Places in the current MASS implementation is a discrete multi-dimensional array and one drawback for geometric applications (e.g. Closest Pair of Points and Voronoi Diagram) is that, when the distance between data points increases, the execution time dramatically increases as more iterations of agent's propagation is required for collision [14]. In order to develop more compact Places, we implement the Continuous Space and Quad Tree data structures in which the Places have continuous range. In addition, to improve the MASS performance in searching and sorting problems, such as Range Search, we develop Binary Tree which allows users to apply divide-and-conquer algorithms in MASS.

Chapter 3.

Implementation of Agent-navigable Data Structures

This chapter describes three agent-navigable data structures that have been implemented in MASS library, including (1) Continuous Space, (2) Quad Tree, and (3) Binary Tree. In addition, three application programs, including the Closest Pair of Points, Voronoi Diagram, and Range Search, are implemented for the verification purposes of these data structures. The Closest Pair of Points and Voronoi Diagram are used for testing the Continuous Space and Quad Tree. The Range Search is for testing the Binary Tree.

3.1 Continuous Space

3.1.1 Basic structure of Continuous Space

Continuous Space is basically structured in 2D as shown in Figure 3.1. In comparison to a discreate multi-dimensional array of Places that is supported by current MASS, the Continuous Space allows users to create more compact Places. The Place with continuous range facilitates applying MASS in solving geometric problems. The Continuous Space is a distributed matrix whose elements (SpacePlace objects) are allocated to different computing nodes. Each SpacePlace covers a continuous range of coordinates with a global index and partitioned into sub-places. As shown in Figure 3.1, the user-defined granularity is 2 and the Place is partitioned into two in each dimension (grey dash line). As a 2D space, each place contains four sub-places.

Agents are allocated to the corresponding Place according to the coordinates. For example, an agent with coordinates [62, 28] locates in the upright sub-place in Place (x:[60, 70), y: [20, 30)).

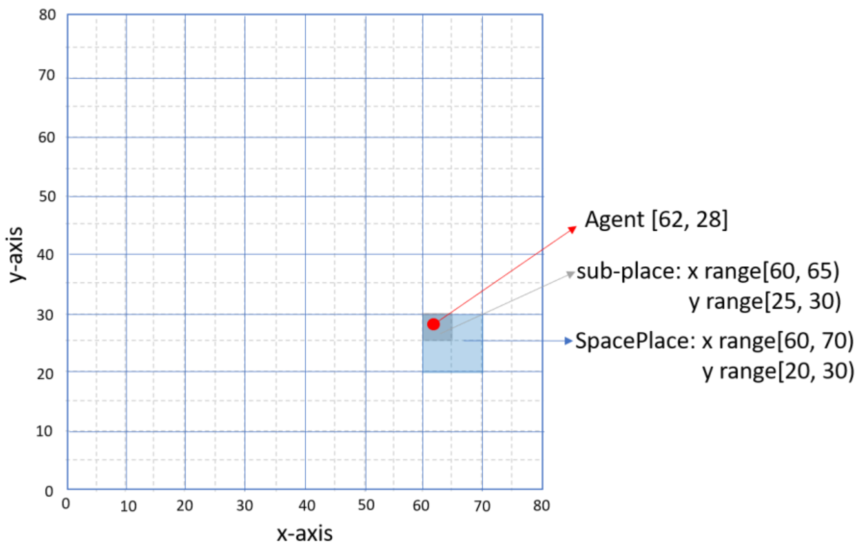


Figure 3.1 The Continuous Space model.

3.1.2 Classes in Continuous Space

The major classes in Continuous Space include (1) SpacePlaces, (2) SpacePlace, and (3) SpaceAgent. SpacePlaces is derived from Places that manages all array elements (SpacePlace object) within the space. The internal structure of SpacePlaces is the same as Places, but it identifies the data space in a continuous manner. Derived from Place, the SpacePlace is the base class from which users can derive a specific SpacePlace object in their application. Derived from Agent, the SpaceAgent is the base class from which users can define an application specific Agent object that allocates in the user-defined SpacePlace object. The key variables / constructors / methods of these classes are shown in Table 3.1.

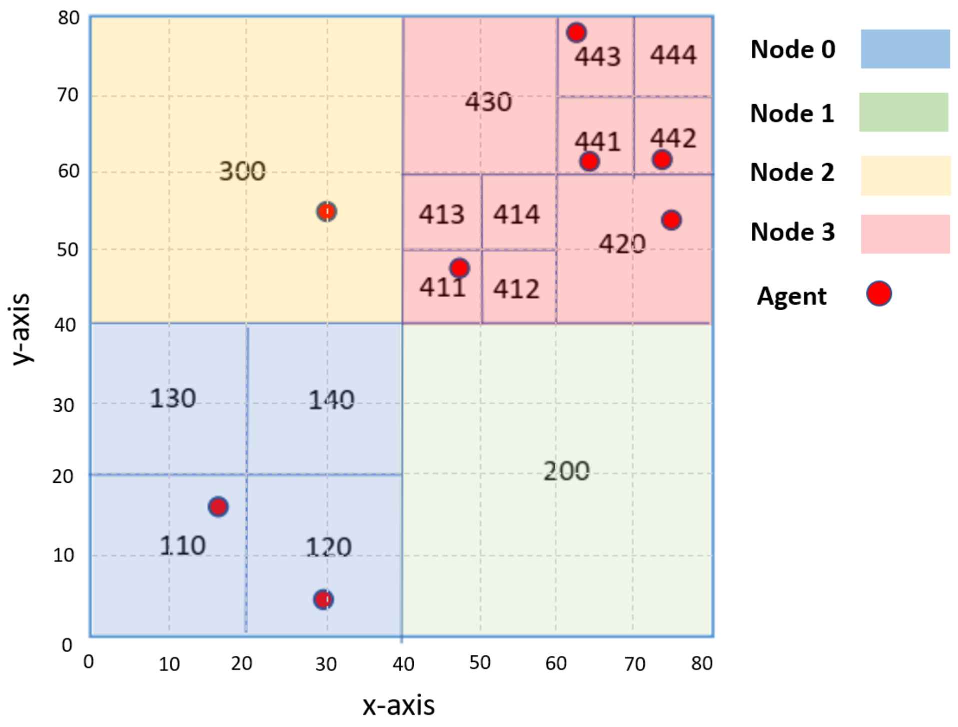
Table 3.1 Key parameters / constructors / methods in Continuous Space

Class	Variables / Constructors / methods
SpacePlaces	• `public SpacePlaces(int handle, String className, int dimensions, int granularity, String inputFile, Object argument)` An array of SpacePlace objects is instantiated by reading input file. The size of data points is automatically calculated by the program. In addition, the SpacePlaces also support creating SpacePlaces with user defined size. The SpacePlace objects are accessed and processed in parallel in multiple computing nodes.
SpacePlace	• `int granularity` The granularity is defined by the user that sub-divides Place into sub-places. • `Hashtable<Integer, Set<Agent>> agentsMap` The key/value pair are the linear index of sub-place and the agent's reference. If an agent migrates to a different Place or a different sub-place in the same Place, the hash table in the destination Place registers the new sub-place index and agent's reference.
SpaceAgent	• `double[] currentCoordinates` The coordinates where the agent locates currently. • `migrate(double[] coordinates)` Agent migrates to the corresponding coordinates • `propagate(Object argument)` Agent spawns in Moore-Neighbor (four directions: N, S, W, E) and Von Neumann-Neighbor (eight directions: N, S, W, E, NE, NW, SE and SW).

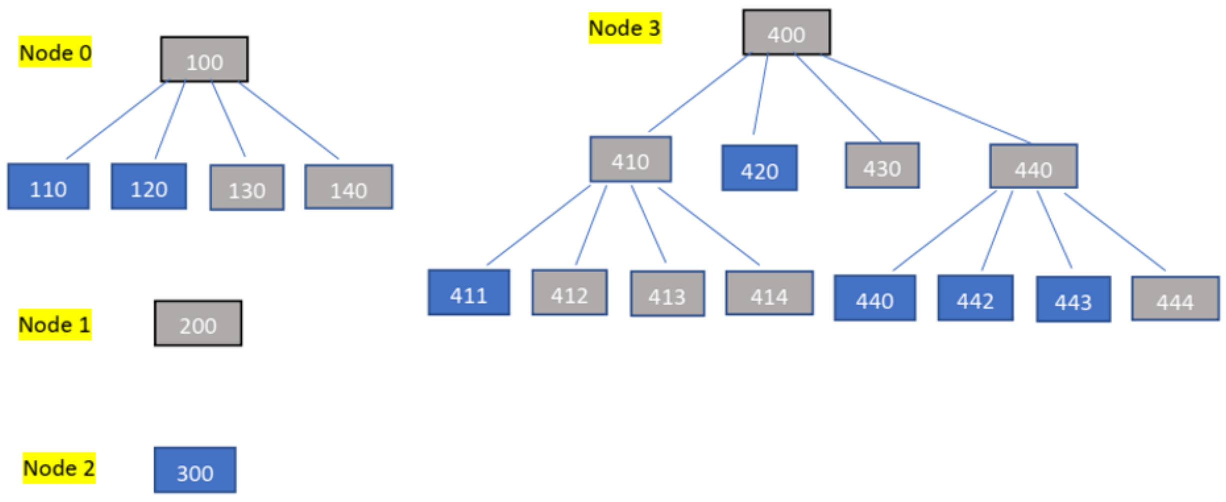
3.2 Quad Tree

3.2.1 Basic structure of Quad Tree

For Continuous Space, agents might allocate in the same sub-place when data points are not evenly distributed. To solve this problem, we design the Quad Tree class which reads data points from a file and divides a space quarterly if a new point resides on the space that includes another point [27]. The basic structure of Quad Tree is shown in Figure 3.2. To distribute data over a cluster system, the Quad Tree is partitioned into different computing nodes according to its spatial coordinates. When MASS is initialized, the program in the master node creates a boundary map that includes the boundary of Quad Tree in each computing node and send the map to all nodes. Each computing node reads input data items in parallel and build its local Quad Tree with the data items within the boundary. When an agent migrates, the program firstly searches for the destination QuadTreePlace in local computing node. If the local computing node contains the destination QuadTreePlace, the Agent migrates to the destination directly; if the local computing node does not contain the destination QuadTreePlace, it searches for the destination computing node using the boundary map and send a message of migration to the destination computing node.



(a)



(b)

Figure 3.2 The Quad Tree model (a) spatial structure (b) tree structure.

3.2.2 Classes in Quad Tree

The major classes in Quad Tree include (1) QuadTreePlaces, (2) QuadTreePlace, and (3) QuadTreeAgent. QuadTreePlaces is derived from Places which manages the local Quad Tree that is composed of tree nodes (QuadTreePlace objects) in each computing node. Derived from the Place, the QuadTreePlace is the base class from which users can derive a specific QuadTreePlace object in their application. Derived from the Agent, the QuadTreeAgent is the base class from which users can define a specific Agent object that allocates in the user-defined specific QuadTreeAgent objects in their application. The key variables / constructors / methods are shown in Table 3.2.

Table 3.2 Key parameters / constructors / methods in Quad Tree

Class	Variables / Constructors / methods
QuadTreePlaces	• `public QuadTreePlaces(int handle, String className, int dimensions, String inputFile, int granularityEnhancer, Object argument)` In tree initialization, QuadTreePlaces instantiates user-defined QuadTreePlace object by creating tree node (QuadTreePlace) using data points that are in the boundary of computing node. Only leaf node contains data point. • `QuadTreePlace root` Root of the Quad Tree in current computing node.
QuadTreePlace	• `Vector<Integer> treeIndex` **A** global index for the QuadTreePlace, in which the first integer represents the computing node rank and the following integers represent the location in the corresponding level. e.g. In Figure 3.2, for index of 414, 1) the first digit 4 means the Place is in node 3; 2) the second digit 1 means it is in the bottom left tree node in the second level; 3) third digit 4 means it is in the top right node in the third level. • `Hashtable<Integer, Set<QuadTreeAgent>> agentsMap` The key/value pair are the linear index of sub-place and the Agent's reference

	• `void subDivideTreePlace();` The place is partitioned into sub-places and the size of the sub-place is determined by the deepest tree node in all computing node, e.g. in Figure 3.2, Quad Tree is created in each computing node and the deepest tree nodes are in node 3 with index of 411-414 and 441-444 whose size is [10, 10]. Therefore, all tree nodes are partitioned into sub-place with size of [10, 10]. • `Split()` Split the tree node into four sub-tree nodes (topLeft, topRight, bottomLeft, bottomRight). • `Insert()` Inserts the data point to the current tree node. If the tree node is a leaf without data point, add data point directly; if the tree node is leaf which contains a data point, split the tree node and add data point to the corresponding sub-tree. If the tree node is not a leaf, insert the data point to the corresponding sub-tree.
QuadTreeAgent	• `double[] currentCoordinates` The coordinates where the Agent locates currently • `migrate(double[] coordinates)` Agent migrates to the corresponding coordinates • `propagate(Object argument)` Agent spawns in Moore-Neighbor (four directions: N, S, W, E) and Von Neumann-Neighbor (eight directions: N, S, W, E, NE, NW, SE and SW).

3.3 Binary Tree

3.3.1 Basic structure of Binary Tree

The Binary Tree is implemented to facilitate the user to apply a log N search and a divide-and-conquer algorithm in MASS. As shown in Figure 3.3, the tree nodes are BinaryTreePlace objects which havetwo children BinaryTreePlace objects. Reading data from an input file, the BinaryTreePlace objects are initialized and distributed among the cluster. Although each binary tree is local, all computing nodes share a boundary map and use global indices for the BinaryTreePlace object. Agents are instantiated to traverse over the tree node with user defined algorithm.

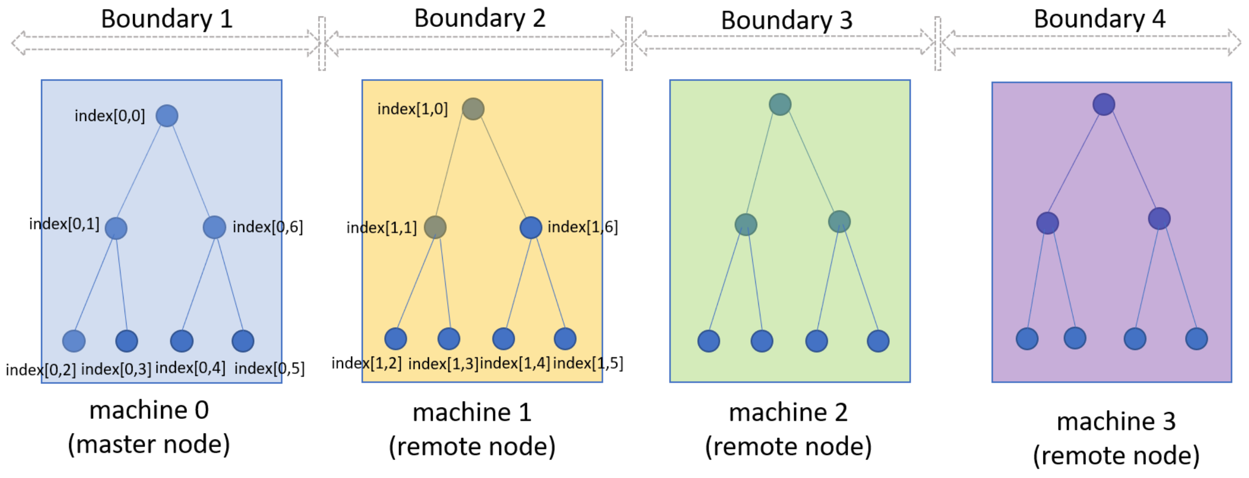


Figure 3.3 The Binary Tree Model.

3.3.2 Classes in Binary Tree

The major classes of Binary Tree include (1) BinaryTreePlaces and (2) BinaryTreePlace. Derived from the Places, the BinaryTreePlaces manages local Binary Tree that is composed of tree nodes in each computing node. Derived from the Place, the BinaryTreePlace is the base class from which users can derive a specific BinaryTreePlace object in their application. The key variables / constructors / methods are shown in Table 3.3.

Table 3.3 Key variables / constructors / methods in Binary Tree

Class	Variables / Constructors / methods
BinaryTreePlaces	• `public BinaryTreePlaces(int handle, String className, String inputFile, Object argument)` The BinaryTreePlaces instantiates user-defined BinaryTreePlace object by creating tree node using data points that are in the boundary of current computing node. • `init_master(Object argument)` The master computing node firstly reads all data and calculates boundaries of computing nodes which evenly partitions the data points. The boundary map is sent to all computing nodes. The master node constructs its local binary tree and send the message of tree initialization to remote nodes. • `init_all(Object argument)` All computing nodes read input in parallel and build the local binary tree using data points within its local boundary. .
BinaryTreePlace	• `Data data` A Data object to keep data information. • `BinaryTreePlace left, right;` The sub-trees of the current tree node. • `int[] index` A two-dimension integer array is used for the indexing system. As shown in Figure 3.3, the first dimension is the rank of the computing node and the second dimension is the local index that assigned sequentially.

3.4 Application programs

3.4.1 Closest Pair of Points

The Closest Pair of Points is implemented by the Continuous Space and Quad Tree. By reading data points from a input file, the Places are created as a 2D space and agents are initially allocated to the corresponding Places according to the coordinates of data point. Agents propagate like a water ripple over Places. The propagation behavior of the Agent is mimicked by spawning Agents in the Von Neumann and Moore neighborhood pattern alternatively. Eventually ripples collide and the source points from the first collision are identified as the closest pair of points (Figure 3.4). Listing 3.1 shows the algorithm of computing Closest Pair of Points which is implemented by Continuous Space.

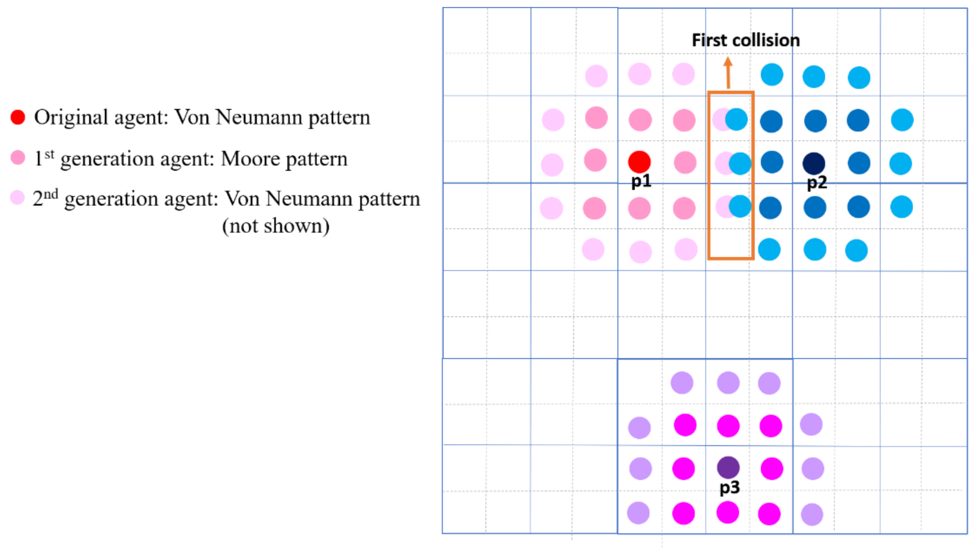

Original agent: Von Neumann pattern
1st generation agent: Moore pattern
2nd generation agent: Von Neumann pattern (not shown)
First collision
p1
p2
p3

Figure 3.4 Agents propagate and collide in Closest Pair Of Points.

Listing 3.1 Algorithm of computing Closest Pair of Points

```
1   public void computeClosestPairOfPoints() {
2           create SpacePlaces places;
3           create SpaceAgents agents;
4           while closest pair is not found {
5               agents.callAll(ClosestPairAgent.PROPAGATE);
6               agents.manageAll();
7
8               agents.callAll(ClosestPairAgent.MIGRATE);
9               agents.manageAll();

10              agents.callAll(ClosestPairAgent.KILL_DUPLICATES);
11              agents.callAll(ClosestPairAgent.KILL_PARENT);
12              agents.manageAll();
13
14              Object[] allPairs = agents.callAll(ClosestPairAgent.COLLECT_PAIRS,
null);
15              for each element in allPairs:
16                  search for the pair which has the minimum distance;
17                  if the pair is found, return the pair;
18          }
19  }

    // Note: ClosestPairAgent is the application specific SpaceAgent class
```

As shown in Listing 3.1, SpacePlaces creates a 2D array of ClosestPairPlace (application specific SpacePlace class) objects (line 1). The size of places is $\sqrt{number\ of\ data\ point} \times \sqrt{number\ of\ data\ point}$. SpaceAgents are instantiated by allocating ClosestPairAgent (application specific SpaceAgent class) objects in the corresponding Place according to the coordinates of their source data points (line 2). User defined functions in Agent and Place instances are iteratively called until the closest pair is found. In ClosestPairAgent.PROPAGATE (line 5), agents are spawned in Moore neighborhood pattern if its generation is even or in Von Neumann neighborhood pattern if its generation is odd. By calling ClosestPairAgent.MIGRATE (line 8), the spawned agents migrate to the destination place. In

ClosestPairAgent.KILL_DUPLICATES (line 10), each place contains a hash table which key/value pair is sub-index/footprints of visited Agents. If an Agent migrates to a place which has been visited by another agent from the same source point, the Agent will be killed immediately. In ClosestPairAgent.KILL_PARENT (line 11), Agents which have already spawned are killed. In ClosestPairAgent.COLLECT_PAIRS (line 14), the collision of Agents is defined as more than one Agents reside in the same sub-place. By this method call, all collision agents are returned as PairOfPoints objects. The closest pair of points is identified by calculating the distance of Agent's source point.

3.4.2 Voronoi Diagram

The Voronoi Diagram is implemented by the Continuous Space and Quad Tree. The program reads input points and creates a 2D array of VoronoiPlace which is an application specific SpacePlace. As shown in Figure 3.5, Agents are initialized at the coordinates of input data points which represent the original Voronoi sites (p1, p2…). Agents propagate from the initial site and eventually collide at the perpendicular bisector of corresponding source points which are the Voronoi edges [25, 29].

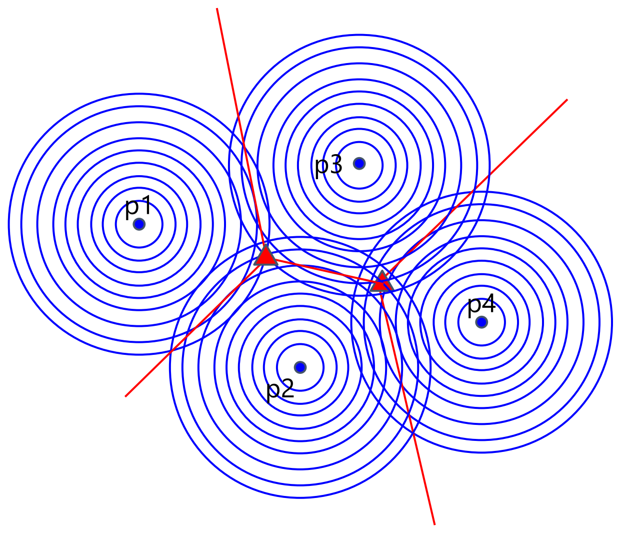


Figure 3.5 Agents propagate and collide at Voronoi edges.

- computeVoronoiDiagram()

The computeVoronoiDiagram() is the function which reads input data points, computes and outputs a Voronoi diagram. The output of Voronoi diagram is a HashMap<Point, List<ParametricLine>>, in which the Point is the source point and the List<ParametricLine> is a list of all perpendicular bisectors of that source point. The algorithm of Voronoi Diagram is shown in Listing 3.2. The PROPAGATE (line 7), MIGRATION (line 10), KILL_DUPLICATE (line 13) and KILL_PARENT (line 14) functions are the same as the functions in the Closest Pair of Points. While in COLLECT_PAIRS (line 17), the collision occurs in two cases: 1) Agents reside in the same sub-place; 2) Agents reside in the sub-places that are neighbors and Agents migrate across each other (Figure 3.6). The algorithm of collecting pairs is shown in Listing 3.3. In order to detect a collision in both cases, each place maintains a HashMap of footprint which key/value is the sub-place index/a set of source points of agents that visited the sub-place. In each iteration, Agents in collision are collected and their source points are returned as pairs to the main program. Then the collided Agents are killed immediately.

Listing 3.2 Algorithm of Voronoi Diagram

```
private HashMap<Point, List<ParametricLine>> edges;
private HashMap<PairOfPoint, <ParametricLine>> allpairs;
public void computeVoronoiDiagram() {
    create SpacePlaces places;
    create SpaceAgents agents;
    while (nAgents() > 0) {
        agents.callAll(VoronoiAgent.PROPAGATE);
        agents.manageAll();

        agents.callAll(VoronoiAgent.MIGRATION);
        agents.manageAll();

        agents.callAll(VoronoiAgent.KILL_DUPLICATES);
        agents.callAll(VoronoiAgent.KILL_PARENT);
        agents.manageAll();

        Object[] pair = agents.callAll(VoronoiAgent.COLLECT_PAIRS);
        agents.manageAll();

    for each pair in pairs:
        if allPairs does not contain the new pair:
            all each source point to edges with ParametricLine of Slope
=Integer.MAX_VALUE;
        else if allPairs contains 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 greater than two times:
            continue;

     sort edges;
     outputVoronoiEdges();
}
  // Note: ClosestPairAgent is the application specific SpaceAgent class
```

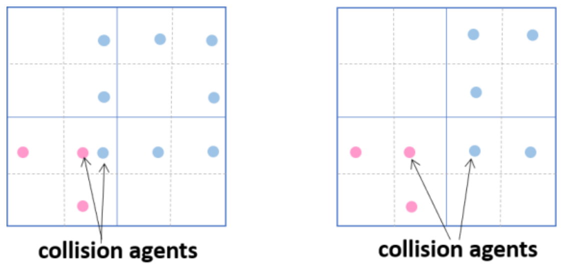


Figure 3.6. Agents collide in the same sub-place or neighbor sub-place.

Listing 3.3 Algorithm of collecting pairs

```
public void collectPairs() {
    List<PairOfPoints> pairs = new ArrayLIst<>();
    for each sub-place in current place:
        if footprint.size() <=1
            continue;
        else
            if footprint.size() <= 3
                add all pair combinations of the three source points;
            else
                search for the source points that generate valid Voronoi vertex and
                return them as pairs;

        kill all collision agents;
         return pairs;
}
```

- outputVoronoiEdges()

The algorithm of output Voronoi diagram is shown in Listing 3.4. It takes HashMap<Point, List<ParametricLine>> *sortedEdges* as an input, and an 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 the vertex. By calculating all vertexes, two neighbor vertexes form a segment which is a Voronoi edge. For vdOutput, the Point is the source input point and the List<Segment> is a list of segments associated with the point (line 1). Furthermore, the closed Voronoi region and open Voronoi region are treated differently. For a closed Voronoi region, we calculate the intersection of the first and the last line which is another vertex and connect it with its neighbor vertexes (line 4). If it is an open Voronoi region, the edges are re-ordered and the two lines form the open area are the first and last line (line 8). For these two lines, one endpoint is the intersection with its neighbor, the other end point is either negative infinity or positive infinity that depends on the location of the source point.

Listing 3.4 Algorithm of output Voronoi diagram

```
private HashMap<Point, List<Segment>> vdOutput
public void 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 lst line are the lines in an open area;
            find the endpoints of the first and last line, add them to vdOutput;
}
```

3.4.3 Range Search

The Range Search program is used to verify the implementation of Binary Tree. A KD tree is constructed to recursively partition k-dimensional space into 2 half-spaces. The application is tested by 2D data points. In the Data object, data items are compared in dimension of x- or y- coordinate alternatively for an element insertion. In each insertion, all data in the boundary of current KDtreePlace are sorted and the (data.size() / 2)th element is the key element that determines the bisector line of the KDtreePlace space. The data less than the key element are passed to the left KDtreePlace and the rest of data are passed to the right KDtreePlace. KDtreePlace are created by recursive partition until all data are inserted. Because the (data.size() / 2)th element of current data is inserted in each insertion, the initial tree is always balanced.

After the tree construction, an Agent is allocated to the root of the KD tree and traverse the branches that overlap with the searching range. As shown in Listing 3.5, Agents are called by RANGE_SEARCH (line 8), SPAWN (line 12), KILL_PARENT(line 15), and MIGRATE (line 18) repeatedly until all Agents are killed. In the RANGE_SEARCH method, if the boundary of current KDtreePlace overlaps with the searching range, we firstly check the data point in the KDtreePlace and return data point to the main program if it is in the searching range. Then the Agent spawns to its sub-trees. Otherwise, the boundary of the KDtreePlace doesn't overlap with the searching range. The Agent is killed and the branches won't be searched. In SPAWN, the Agent spawns two child Agents to its sub-trees. The identified data points are output as a text file.

Listing 3.5 Algorithm of Range Search

```
private Vector<Data> result = new Vector();
 public void rangeSearch() {

     create BinaryTreePlaces places;
     create Agents agents;
     while (number of alive agents) > 0{

         Object dataInRange = agents.callAll(RangeSearchAgent.RANGE_SEARCH, argument);
         agents.manageAll();
         add dataInRange to result;

         agents.callAll(RangeSearchAgent.SPAWN);
         agents.manageAll();

         agents.callAll(VoronoiAgent.KILL_PARENT);
         agents.manageAll();

         agents.callAll(RangeSearchAgent.MIGRATION);
         agents.manageAll();
     }

 }
// Note: RangeSearchAgent is the application specific Agent class
```

Chapter 4.

Programmability and Performance Evaluation

In this chapter, we evaluate the implementations of Continuous Space and Quad Tree using two applications, including Closest Pair of Points and Voronoi Diagram. We also evaluate the implementation of Binary Tree using the Range Search application. Programmability, execution time, and memory consumption are measured for each application.

4.1 Evaluation environment and procedures

4.1.1 Input data format

The input dataset for each application is stored in a text file in disk. The input data items are in the format of N-dimensional points. In the text file, each line represents a data point and the coordinates are integers or decimals separated by a space. The data points are generated by a generator program that randomly produces the required number of data points. All point coordinates are positive values.

4.1.2 Evaluation criteria

The evaluation for the implementation of Continuous Space, Quad Tree, and Binary Tree is conducted using three criteria: (1) programmability (2) execution time, and (3) memory consumption.

(1) Programmability: The programmability is measured by the following four metrics:

- Boilerplate code – number of lines of code required to set up the parallel programming environment, which consists of initializing MASS, setting up a debugging level, and shutting down MASS when the computation is finished.
- LOC – total number of lines of code. The LOC is calculated by using a plugin called Statistics within the IntelliJ IDE.
- Number of classes – total number of java classes in the application program
- Number of methods - the number of unique methods that launch parallel operations in the application in computing nodes.

(2) Execution time: The execution time is defined as total run time of a program, which includes reading/writing file from/to the disk.

(3) Memory consumption: The heap memory of a program is measured via valgrind.

4.2 Programmability evaluation

Programmability for the original MASS, Continuous Space MASS, and Quad Tree MASS are evaluated by implementing the Closest Pair of Points and Voronoi Diagram applications using the criteria that are described in 4.1.2. For all three MASS libraries, since the basic structure of MASS is not changed in the Continuous Space and Quad Tree classes, the boilerplate code is not changed.

For the Closest Pair of Points and Voronoi Diagram application, as shown in Table 4.1 and Table 4.2, the Continuous Space reduces LOC by 21% in the Closest Pair of Points application and 20% in the Voronoi Diagram application in comparison to the original MASS library. The Quad Tree implementation reduces LOC and number of classes for CPP and Voronoi Diagram at a similar level as the Continuous Space. This is because the Continuous

Space and Quad Tree are designed for a geometric space, the functions can be called directly by user’s application program, such as migrate (double[] coordinates) and propagate(). In addition, the original MASS instantiates Agent in every Place, the user has to implement a function to allocate Agents to a set of given data points. While in the Continuous Space and Quad Tree, an input dataset is passed as a parameter to the Agents constructor that instantiate agents at input data points.

For the number of methods, the Closest Pair of Points application invokes comparable number of methods in Continuous Space and Quad Tree compared to the original MASS. In Voronoi diagram application, the number of methods reduces by the Continuous Space and Quad Tree compared to the original MASS. This is because the Continuous Space and Quad Tree in MASS provide specific functions for geometric applications that the user can call directly in their application.

For the Range Search application, as shown in Table 4.3, the boilerplate code is not changed because the basic structure of MASS is the same in the Binary Tree. The Binary Tree reduces LOC by 39% and reduces the number of classes by 33% in comparison to the MASS Graph. This is because the Binary Tree includes functions that can be called directly in a user application. For example, in Graph a user has to build the tree using the GraphPlace in an application. In contrast, the Binary Tree builds the tree which is composed of application specific BinaryTreePlace in the BinaryTreePlaces constructor. The number of methods is not changed.

Table 4.1 Programmability of Closest Pair of Points application

Metrics	Original MASS	Continuous Space	Quad Tree
Boilerplate code (line of code)	3	3	3
LOC	585	465(↓ 21%)	467(↓ 20%)
Number of classes	10	7(↓ 30%)	7(↓ 30%)
Number of methods	7	6	6

Table 4.2 Programmability of Voronoi Diagram application

Metrics	Original MASS	Continuous Space	Quad Tree
Boilerplate code (line of code)	3	3	3
LOC	1797	1433 (↓ 20%)	1426 (↓20%)
Number of classes	14	11(↓ 21%)	11(↓ 21%)
Number of methods	15	7	7

Table 4.3 Programmability of Range Search application

Metrics	MASS Graph	Binary Tree
Boilerplate code (line of code)	3	3
LOC	490	298(↓ 39%)
Number of classes	6	4
Number of methods	4	4

4.3 Performance analysis

4.3.1 Closest Pair of Points

The Closest Pair of Points is firstly used to evaluate the performance of the three implementations. Figure 4.1 shows the execution time of Closest Pair of Points with 32,768 data points. Continuous Space and Quad Tree implementation have comparable performance, which is about 1.5 times faster than the original MASS library. In the original MASS, 10^6 Places are created. While in the Continuous Space and Quad Tree, 33,489 Places and 67,306 Place objects are created respectively. With fewer Place instantiation, the Continuous Space reduces the execution time. In addition, we also find that multiple computing nodes does not reduce execution time significantly in this test case. This may be due to the overheads of communication between computing nodes and relative fast execution by single node.

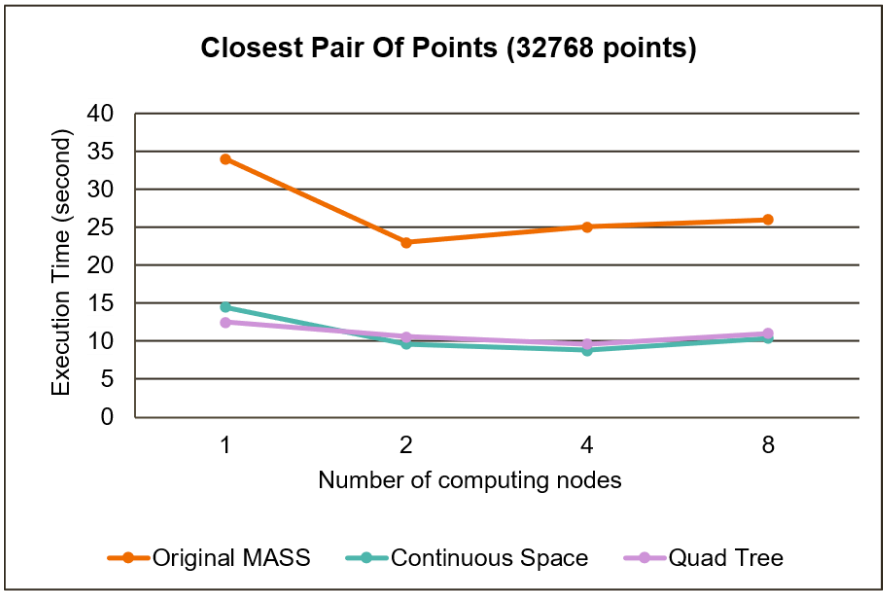


Figure 4.1 Execution time for the Closest Pair of Points application.

Figure 4.2 shows the memory consumption of the Closest Pair of Points implemented by the original MASS, Continuous Space and Quad Tree. The program is executed with 256 data points in a single node. Compared to the original MASS, the Continuous Space and Quad Tree implementations reduce the total memory usage by 70% and 22% respectively. This is because fewer Place and Agent objects are created in the Continuous Space (33,489 places) and Quad Tree (67,306 places) than the original MASS (10^6 places). More Places are instantiated in Quad Tree which explains the increased memory usage in Quad Tree compared to the Continuous Space.

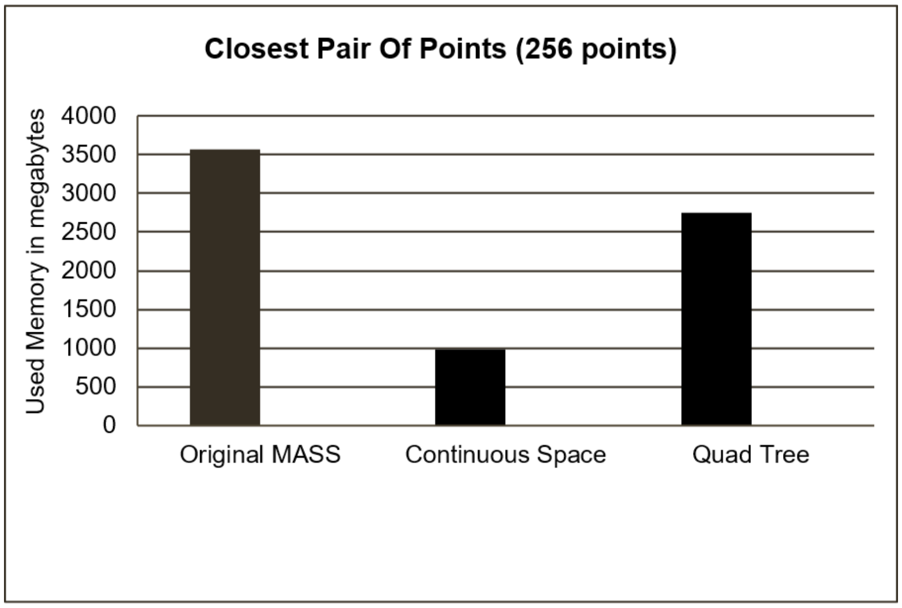


Figure 4.2 Memory consumption for the Closest Pair of Points application.

4.3.2 Voronoi Diagram

In addition to the Closest Pair of Points, the Voronoi Diagram is also used to test the performance of the three implementations. Figure 4.3 shows the execution time of the Voronoi

Diagram application with 64 data points. The Continuous Space and Quad Tree have comparable performance, which is about 90 times faster than the original MASS in single node and about 10 times faster in 8 computing nodes. The reduced execution time in Continuous Space and Quad Tree is because fewer Place objects are instantiated. We found that 81 Places are created in the Continuous Space and 133 Places are created in the Quad Tree. While in the original MASS, 676,400 Places are created. Similar to the Closest Pair of Points, when more computing nodes are added, the execution time increases. This is because the execution time with a single node is short (16 seconds) and building connection between computing nodes increases the total computing time.

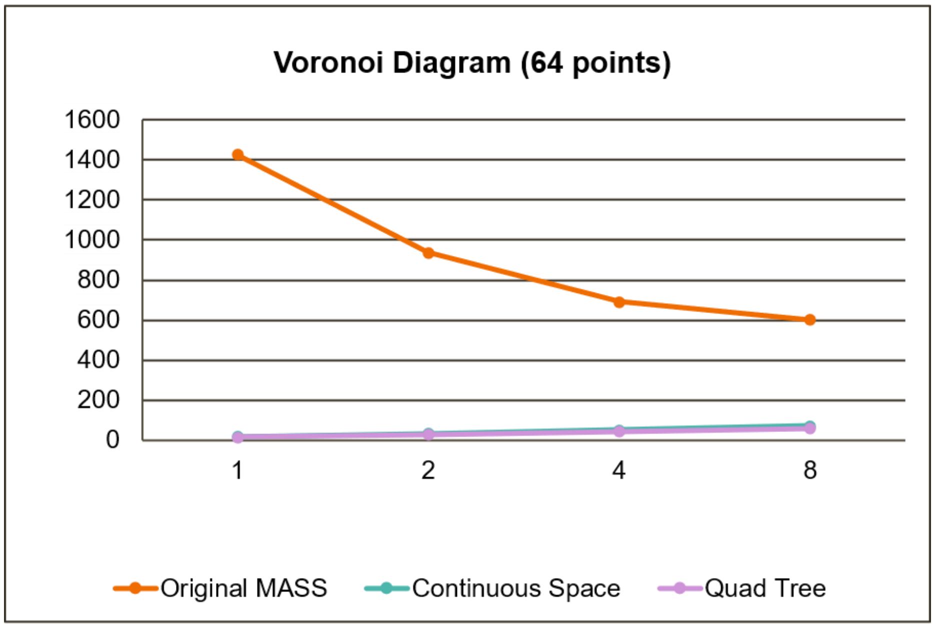


Figure 4.3 Execution time for the Voronoi Diagram application.

Figure 4.4 shows the memory consumption of Voronoi Diagram (16 points) implemented by the original MASS, Continuous Space, and Quad Tree. Compared to the original MASS, the Continuous Space and Quad Tree implementations reduce memory usage by 80% and 70%. The reduction of memory reduction is because a smaller number of Places are instantiated by the

Continuous Space (25 Places) and Quad Tree (37 Places) in comparison to the original MASS (99,200 Places). Because the Quad Tree creates more Places than the Continuous Space, the memory consumption is also higher.

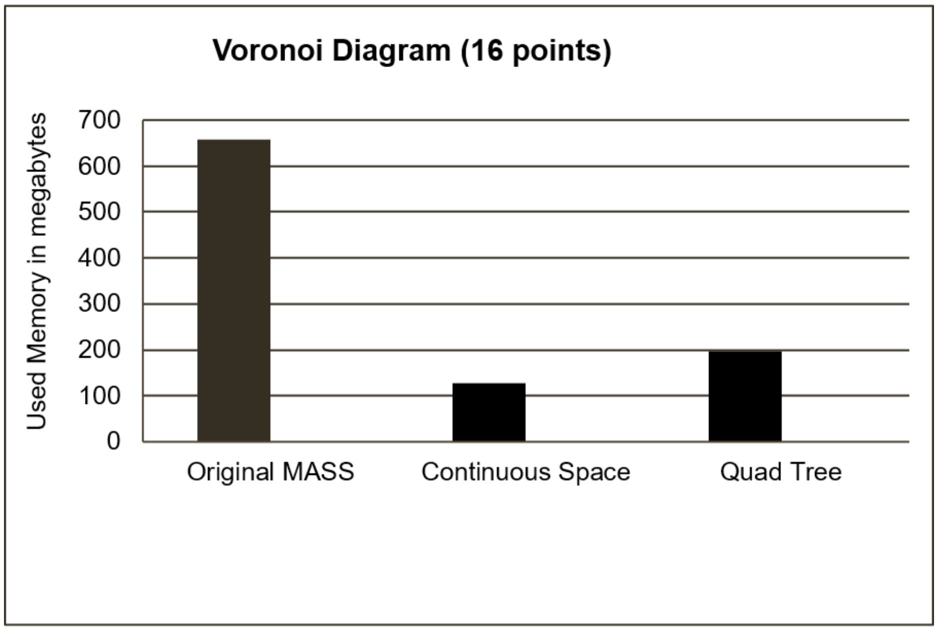


Figure 4.4 Memory consumption for the Voronoi Diagram application.

4.2.3 Range Search

The Range Search application is used to test the Binary Tree implementation. The program is executed with 500,000 data points in single, 2, 4 and 8 computing nodes. The execution performance is shown in Figure 4.5. For Graph, the data for single node is not available because the program is running out of memory. In 2, 4 and 8 computing nodes, the Binary Tree significantly reduces execution time, which is 20×, 30× and 10× faster for 2, 4 and 8 computing nodes respectively compared to the Graph. In Graph, as the graph mimics a binary tree, only one Agent is instantiated at the root. In Binary Tree, each computing node instantiates an Agent at the root and searches simultaneously. Therefore, parallel searching in each computing node speeds up the execution. In addition, because the Graph distributes data items to each computing node sequentially without any spatial preference, when Agents traverse over the tree that is mimicked by Graph, they migrate across computing nodes frequently. In contrast, in Binary Tree, as the tree is local in computing node, when the Agent traverse across branches, the overhead of communication between computing nodes is minimized.

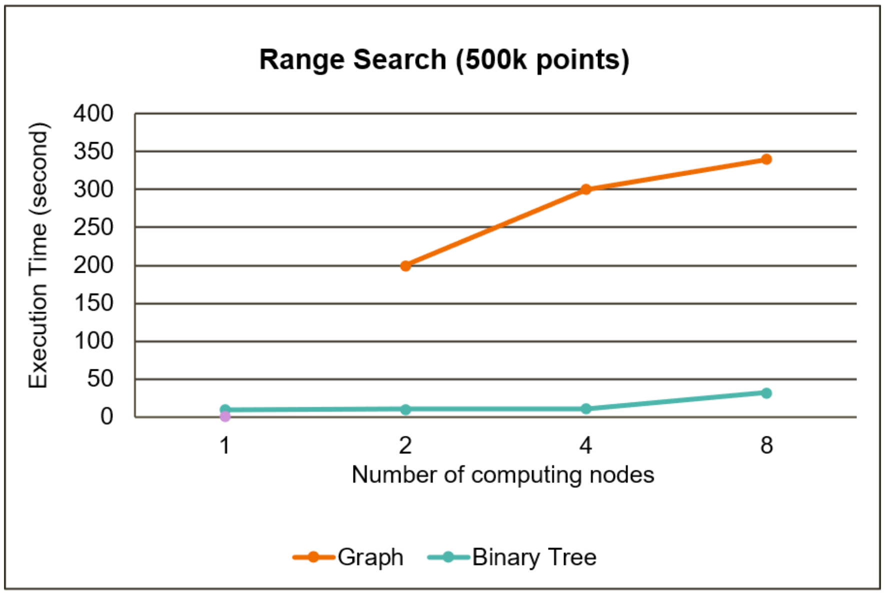


Figure 4.5 Execution time for the Range Search application.

For the memory consumption, as shown in Figure 4.6, the Binary Tree uses much less memory than the Graph. This is because in the application program for Graph, the main class includes a Map<Integer, Point2D> which contains all data items from input file. This global variable consumes big portion of heap memory if the input file is large. Therefore, the application program for the Graph needs to be modified in order to compare the memory consumption of the two implementations.

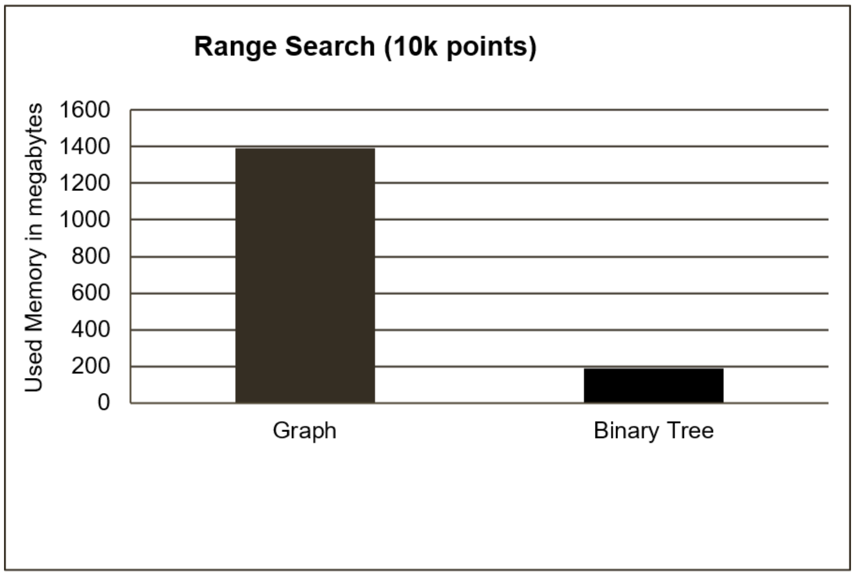


Figure 4.6. Execution time for the Range Search application.

4.2 Summary of evaluation

Overall, compared to the current MASS library, the Continuous Space, Quad Tree, and Binary Tree improve programmability and performance of applications, including execution time and memory consumption in following aspects:

- Programmability - all of the three implementations reduce LOC and number of classes for applications.
- Execution time – all of the three implementations reduce execution time for applications.
- Memory consumption – all of the three implementations reduce memory consumption for applications.

Strengths and challenges

Overall, these three data structures improve programmability and performance in different applications. At the same time, we also found there are limitations that could be improved. For programmability evaluation, we use the four metrics - boilerplate code, LOC, number of classes and number of methods in this study. However, these metrics might not be enough. To measure the complexity directly, we will use the cyclomatic complexity which measures the number of “linearly independent paths” through a piece of code in the future. The strengths and challenges of the three data structures are discussed as follows:

(1) Continuous Space

Strengths: The Continuous Space creates more compact Places by using a continuous range of coordinates for each SpacePlace object. With the same functionalities, a smaller number of Places and Agents reduce execution time and memory usage. In addition, the SpacePlace object is sub-divided into sub-places with user defined granularity. The Agent’s information in sub-places is traced using a Hashtable of <Integer, Set<Agent>> in which the key is the linear index of sub-place and the value is a set of Agents in the corresponding sub-place. Furthermore, Agent migrates to the destination Place using coordinates and the user do not need to specify the index of the destination Place. This feature facilitates the user to migrate Agent without knowing the internal structure of the Places.

Challenges: Because all Places and sub-places are evenly partitioned, if the data points are not evenly distributed, in some Places there might be multiple Agents allocating in a single Place and other Places might be vacant. This leads to low efficiency of memory usage. Another challenge is that the granularity needs to be optimized by the user. On one hand, the value of granularity determines the fineness of sub-places, which would affect program output in some cases. For example, in Voronoi Diagram application, if the granularity is too low, some Voronoi edges will be missing. On the other hand, the value of granularity will affect application’s performance (e.g. a high granularity will slow down the execution and increase memory consumption) because of the increased computation. Therefore, the user has to optimize the granularity for the best performance of their application.

(2) Quad Tree

Strengths: In order to create Places in an efficient way, the Quad Tree is used to create Places if a new data point resides in a Place which is already occupied. This approach avoids multiple Agents allocating in the same Place in Agent's initialization. In addition, Places are further divided into sub-places whose sizes equal to the deepest QuadTreePlace object. A similar Hashtable is used to trace the Agent's information in the place as described in the Continuous Space. Furthermore, the user-defined granularityEnhancer is used to further divide sub-places into a finer grid if needed.

Challenges: As the data points are distributed over cluster in a spatial-driven manner, the data points are always partitioned into power of two (e.g. 1, 2, 4, 8, 16…). Therefore, the number of computing nodes are restricted to the power of two. For example, if there are 11 machines in the cluster system, only 8 will be used for computing, which would be a waste of resource.

(3) Binary Tree

Strengths: The Binary Tree allows users to apply a log N search and divide-and-conquer algorithm in MASS library for their application. Although the Graph in MASS could also mimic a tree structure, users have to implement the tree in their application. In Binary Tree, because each computing node has its local tree, agents are initialized to traverse the tree in each computing node parallelly. This minimizes communications between computing nodes and reduces overhead of the program. Meanwhile, as the BinaryTreePlace has a global index, Agent is able to migrate to Place at different computing nodes.

Challenges: In the current implementation, for parallelization, the master node reads an entire file and calculates the boundary of each node. Then each computing node reads the entire input file and build its local tree with data points within the boundary. However, if the input file is huge, reading the entire file can be time consuming. In order to avoid this problem, the input file could be partitioned and the computing node reads its corresponding part directly. Another drawback of the current implementation is that, to build a balanced binary tree, the data point in the tree node is the median value of its sub-trees. However, for real-time data streaming, data items are added sequentially, and the tree won't be balanced anymore. To overcome this limitation, we will develop new algorithm to balance the tree when adding tree nodes dynamically.

Chapter 5. Conclusion

In conclusion, we achieved our goals of this research project through the implementation of three data structures, including Continuous Space, Quad Tree, and Binary Tree. The major contribution of this research includes:

(1) We have designed and implemented Continuous Space, Quad Tree, and Binary Tree that extend data structures supported by MASS library.

(2) We have verified the implementation of the three data structures and compared their performance to the data structures in the current MASS using different applications.

(3) The Continuous Space and Quad Tree outperform the original MASS in geometric applications and the Binary Tree outperforms the MASS Graph in Range Search application, which will facilitate users to apply MASS to their applications.

Below we discuss the potential areas where future work can be done based on the outcome of this work.

1. Input file format: The Continuous Space, Quad Tree, and Binary Tree are able to instantiate Places from an input file directly. The text file is the only format supported by current MASS. As the MASS library will be applied in scientific fields that uses a various type of files, we will expend MASS to support other input formats that are commonly used, such as NetCDF, CSV, and GeoJSON.
2. Parallel I/O: In the current implementation of Continuous Space, Quad Tree, and Binary Tree, each computing node reads an entire file to instantiates Places. In order to read a file efficiently, especially for a large input file, the file should be partitioned and each

computing node reads its corresponding section. Furthermore, in big-data analysis, an input dataset might be too huge to fit distributed memory even after being partitioned. To solve this problem, we will design and implement a streaming algorithm that keep streaming data to a cluster system.

References

1. Hadoop, “Accessed on: July 6, 2020. [Online]. Available: http://hadoop.apache.org/.”

2. Spark, “Accessed on: August 10, 2020. [Online]. Available: http://spark.apache.org/.”

3. Storm, “Accessed on: June 30, 2020. [Online]. Available: http://storm.apache.org/.”

4. Unidata – NetCDF, “Accessed on: September 12, 2019. [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/.”

5. HIPPIE, “Accessed on: February, 2019. [Online]. Available: http://cbdm-01.zdv.uni-mainz.de/mschaefer/hippe/index.php.”

6. E. Bonabeau. Agent-based modeling: Methods and techniques for simulating human system. *PNAS*, pages 7280-7287, 2002

7. RepastHPC, “Accessed on: February. 1, 2020. [Online]. Available: https://repast.github.io/repast_hpc.html.”

8. FLAME, “Accessed on: February 1, 2020. [Online]. Available: http://www.flame.ac.uk.”

9. M. Fukuda, C. Gordon, U. Mert, M. Sell. An agent-based computational framework for distributed data analysis. *Computer*, 53(3):16 - 25, 2020.

10. Munehiro Fukuda and Distributed Systems Lab. MASS Java Manual.

11. J. Woodring, M. Sell, M. Fukuda, H. Asuncion, E. Salathe, M. Kipps, W. Kim, M. Fukuda Agent and Spatial Based Parallelization of Biological Network Motif Search, *17th IEEE International Conference on High Performance Computing and Communications - HPCC 2015*, New York, August 24-26, 2015

12. Y. Shih, C. Gordon, M. Fukuda, J. van de Ven, C. Freksa. Translation of String-and-Pin-Based Shortest Path Construction into Data-Scalable Agent-Based Computational Models. *Proc. of the 2018 Winter Simulation Conference*. pages 881-892, Gothenburg, Sweden, December 2018

13. J. Gilroy, S. Paronyan, J. Acoltzi, M. Fukuda, “Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory”, 2020 BigGraphs Workshop at IEEE BigData'20, pp 2957-2966, December 2020

14. S. Gokulramkumar, “Agent Based Parallelization of Computation Geometry Algorithms”, Master Thesis, University of Washington, March 2020

15. R. Sedgewick, K. Wayne, Altorithms (4th edition). April 2011

16. T. White, “Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale (4th edition)”, April 21, 2015

17. H. Karau, A. Konwinski, P. Wendell, M. Zaharia, “Learning Spark: Lightning-Fast Big Data Analysis (1st edition)”, February 27, 2015

18. Repast Simphony Reference Manual, “Accessed on: October 2020. [Online]. Available: https://repast.github.io/docs/RepastReference/RepastReference.html .”

19. M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, P. Sydelko, Complex adaptive systems modeling with Repast Simphony, Complex Adaptive Systems Modeling, March 2013

20. C. Shih, C. Yang, M. Fukuda, Benchmarking the Agent Descriptivity of Parallel Multi-Agent Simulators, International Workshops of PAAMS 2018, Highlights of Practical Application of Agents, Multi-Agent Systems, and Complexity, pages 480-492, Toledo, Spain, June 2018

21. J. Van, M. Fukuda, H. Schultheis, C. Freksa, T. Barkowsky, Analyzing Strong Spatial Cognition: A Modeling Approach, German Conference on Spatial Cognition, Spatial Cognition 2018, Tubingen, Germany, pages 197-208, September 2018

22. J. Emau, T. Chuang, M. Fukuda, A Multi-Process Library for Multi-Agent and Spatial Simulation. In Proc. of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Prcessing - PACRIM'11, pages 369-376, Victoria, BC, Canada, August 24-26, 2011

23. J. Kleinberg, E. Tardos. Algorithm Design (1st edition). March 16, 2005

24. Franco P. Preparata and Michael Ian Shamos. Computational Geometry: An Introduction. Springer Science Business Media., 1993

25. F. Aurenhammer, Voronoi diagrams survey of a fundamental geometric data structure. In: ACM Computing Surveys, vol. 23, pp. 345–405. 1991

26. W. Alsalih, K. Islam, Y. Nunez-Rodriguez, H. Xiao, Distributed Voronoi diagram computation in wireless sensor networks. SPAA 2008: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, 2008.

27. S. Garrido, L. Moreno, M. Abderrahim, F. Martin, Path Planning for Mobile Robot Navigation using Voronoi Diagram and Fast Marching. IEEE International Workshop on Intelligent Robots and Systems, Beijing, China, October 9-15, 2006.

28. E. Langetepe, G. Zachmann. Geometric Data Structures for Computer Graphics. 2006.

29. A. Okabe, B. Boots, K. Sugihara, S. N. Chiu. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams (2nd edition) Chichester, West Sussex, England: John Wiley & Sons, 1999.