

©June of 2023  
Anirudh Potturi

# **Programmability and Performance Analysis of Distributed and Agent-Based Frameworks**

**Anirudh Potturi**

Submitted In Partial Fulfillment of the Requirements

For the Degree of

Masters of Science in Computer Science and Software Engineering


Under Guidance of Dr. Munehiro Fukuda, Chair Committee

Committee Member Dr. Min Chen

Committee Member Dr. William Erdly

**University of Washington**

ORCID

 <https://orcid.org/0000-0002-9270-9628>



## Abstract

### Programmability and Performance Analysis of Distributed and Agent-Based Frameworks

*Anirudh Potturi*

Chair of the Supervisory Committee:

Dr. Munehiro Fukuda

Computer Science & Software Engineering

In big data, the importance shifts from raw text presentation of data to structure and space of the data. Computational geometry is an area of interest, particularly for the structure and distribution of data. Thus, we propose using Agent-Based Modelling (ABM) libraries for big data to leverage the benefits of parallelization and support the creation of complex data structures such as graphs and trees. ABMs offer a unique and intuitive approach to solving problems by simulating the structural elements over an environment and dispatching agents to break these problems down using swarming, propagation, collisions, and more. For this research, we introduce using Multi-agent Spatial Simulations (MASS) for big data. We compare the programmability and performance of MASS Java against Hadoop MapReduce and Apache Spark. We have chosen six different applications in computational geometry implemented using all three frameworks. We have conducted a formal analysis of the applications through a comprehensive set of tests. We have developed tools to perform code analysis to compute metrics such as identifying the number of Lines of Code (LoC) and computing McCabe's cyclomatic complexity to analyze the programmability. From a quantitative perspective, in most cases, we found that MASS

demanded less coding than MapReduce, while Spark required the least. While the cyclo-matic complexity of MASS applications was higher in some cases, components of Spark and MapReduce applications were highly cohesive. From a qualitative viewpoint, MASS applications required fine-tuning resulting in significant improvements, while MapReduce and Spark offered very limited performance enhancement options. The performance of MASS directly correlates with the data, in contrast to MapReduce and Spark, whose performance is not affected by the distribution of data.

## Acknowledgements

*Dedicated to Ma, Vinni, Babai, Pinni, Family and Friends.*

A special thanks to Josh Helzerman, who helped extend the collection of applications and perform their measurements. I would also like to extend my gratitude towards Matthew Sell for his guidance and invaluable suggestions. I had the pleasure of working with my fellow members of the Distributed Systems Laboratory.

## Contents

<b>List of Figures</b>	<b>9</b>
<b>1 Introduction</b>	<b>10</b>
<b>2 Background</b>	<b>12</b>
2.1 Distributed frameworks for data science . . . . .	13
2.1.1 Hadoop MapReduce . . . . .	13
2.1.2 Apache Spark . . . . .	14
2.1.3 Multi-Agent Spatial Simulations Library . . . . .	16
2.2 Previous Work . . . . .	17
<b>3 Related Work in Agent-based Approach</b>	<b>22</b>
3.1 Agent Based Frameworks . . . . .	22
3.1.1 Repast Symphony . . . . .	22
3.1.2 NetLogo . . . . .	22
3.1.3 DMASON . . . . .	23
3.1.4 Other multi-agent systems . . . . .	24
3.2 Summary . . . . .	25
<b>4 Implementation</b>	<b>26</b>
4.1 Tools Development . . . . .	26
4.1.1 Java Static Code Analysis Tool . . . . .	26
4.1.2 C++ Static Code Analysis Tool . . . . .	28
4.1.3 NodesXML File Generator Tool . . . . .	28
4.2 Application Development . . . . .	29
4.2.1 MASS Base Code for Repast Applications . . . . .	29
4.2.2 Breadth-First Search . . . . .	30
4.2.3 Range Search . . . . .	31
4.2.4 Largest Empty Circle . . . . .	32
4.3 MASS Fine-Tuning as Preparation for Performance Evaluation . . . . .	35
<b>5 Evaluation</b>	<b>37</b>
5.1 Verification of Evaluation Tools . . . . .	37
5.2 Programmability Analysis . . . . .	38
5.2.1 MASS versus Repast . . . . .	38

5.2.2	MASS versus MapReduce/Spark . . . . .	39
5.3	Performance Evaluation . . . . .	42
5.3.1	MASS vs Repast . . . . .	42
5.3.2	MASS versus MapReduce/Spark . . . . .	44
<b>6</b>	<b>Conclusions</b>	<b>53</b>
6.1	Summary . . . . .	53
6.2	Future Directions . . . . .	54
	<b>Acronyms</b>	<b>55</b>
	<b>References</b>	<b>56</b>
	<b>Appendices</b>	<b>60</b>





## LIST OF FIGURES

2.1	MASS Model . . . . .	18
4.1	Breadth First Search . . . . .	31
4.2	Range Search . . . . .	32
4.3	Largest Empty Circles Initial Migrations . . . . .	35
4.4	Largest Empty Circles: Finding Distance between City and Dump Site . . . . .	36
5.1	Lines of Code (Bar graph representation of MASS, MapReduce, and Spark) . . . . .	40
5.2	Breadth-First Search Repast Simulation . . . . .	43
5.3	CPP: 50,000 Points . . . . .	45
5.4	CPP: 100,000 Points . . . . .	45
5.5	Closest Pair of Points: CPU Scalability of MASS, MapReduce and Spark . . . . .	45
5.6	Convex Hull: CPU Scalability of MASS, MapReduce and Spark . . . . .	47
5.7	Euclidean Shortest Path: CPU Scalability of MASS, MapReduce and Spark . . . . .	48
5.8	Largest Empty Circles: CPU Scalability of MASS, MapReduce and Spark . . . . .	49
5.9	Point Location: CPU Scalability of MASS, MapReduce and Spark . . . . .	51
5.10	Range Search: CPU Scalability of MASS, MapReduce and Spark . . . . .	52

# Chapter 1

## Introduction

The Bureau of Labor Statistics projected a 36% increase in jobs for data scientist roles from 2021 to 2023 [1]. The exponential data volume and complexity growth have encouraged data scientists, researchers and practitioners alike to leverage big data technologies and methods to extract insights. Data scientists drive innovation and address real-world challenges in ways that were previously not feasible [2]. Traditional programming paradigms are unsuitable for processing large datasets, attributed to insufficient computing power and memory offered by a single computer (a node). Thus, a parallelizable framework is necessary to harness the computational power of a cluster of machines. A cluster of machines is a set of computers (nodes) connected to allow for communication and work as if they are a single and a much more powerful machine.

Big data frameworks such as Hadoop MapReduce (henceforth MapReduce) and Apache Spark (henceforth Spark) are popular tools used by data scientists today. These libraries run in distributed environments, allowing collective computational power and memory space. MapReduce processes data in batches, while Spark is suitable for processing data in streams. However, Spark can also process data in batches. Handling large-scale data processing involving big data is supported by both frameworks. This data is often processed as is, in its plain text form, since manipulating primitive data types to complicated structures in these frameworks is not supported.

Contrary to these libraries that stream flat data to their analyzing functions, Agent-Based Modelling (henceforth ABM) frameworks construct data structures over a cluster system and dispatch data-analyzing agents. The paradigm followed by ABMs is a mindset rather

than just a mere technology [3]. The low-level focus is on *agents* and simulating the most fundamental elements of a system [4]. In addition, the emergent collective group behaviour of agents reveals the nonlinear dynamics of the simulation space and data [5].

We work with geometric data, particularly as a prevalent example of distributed geo-data and data structures on top of a cluster system, thus over distributed memory. Our ABM solutions process graphs and multi-dimensional spaces, while their MapReduce and Spark iterations analyze plain text data. We focus on applications in computational geometry as they are critical tools for solving practical problems involving geometric and geo data. With a focus on the structure of data, ABMs offer intuitive approaches to these problems using swarming, propagation, collisions and migrations. Furthermore, these algorithms enable analysts to extract meaningful insights from spatial data and support decision-making in various fields, such as urban planning, environmental science, and transportation engineering.

Through this research, we seek to compare the programmability and execution performance of MASS against traditional big data libraries. A formal analysis of the programmability and performance has been pending with the current work available. We perform a comprehensive set of benchmark tests for each application at least three times. Later, we develop tools to analyze existing applications to draw insights, including code complexity and Lines of Code (henceforth LoC).

The rest of this paper is organized as follows. Chapter 2 discusses using current big data libraries in computational geometry. Chapter 3 briefly overviews known ABM libraries and their use in computational geometry. Chapter 4 describes the implementation of applications and tools required for this research. Experimental evaluations are discussed in Chapter 5, while Chapter 6 concludes the paper.

## Chapter 2

### Background

Data scientists seek to solve real-world problems using large sets of data. Their datasets cannot often fit into a single computing node's memory. While Graphical Processing Units (GPUs) handle time complexity, spatial complexity requires cluster computing to be handled by distributing workloads. In a cluster environment, the collective memory of machines can be used to distribute and process large sets of data, making it easier to handle. As a result, the need for frameworks capable of handling such data and the ability to solve them at faster speeds is necessary. Such frameworks leverage the computing capabilities of a cluster, a group of machines collaborating to derive valuable insights from data within a reasonable time frame. Of importance in cluster computing is hiding CPU boundaries by automating inter-process communication, thus giving a single-system image to data scientists. They seek to use user-friendly frameworks, meaning they expect to spend less time on programming, focus more on the data, and draw conclusions from the results.

Applying ABM frameworks for analyzing distributed data structures represents a groundbreaking approach. ABM libraries encompass complex adaptive systems that offer autonomous, responsive processes as agents operating within an environment. These agents not only exert distributed actions within a simulation environment but also exert some degree of distributed control over the environment [6]. Moreover, agents are equipped with social abilities to communicate with other agents using messages. By utilizing ABM frameworks, researchers can construct realistic scenarios and realistically simulate the behaviours of each problem component. Since ABMs are parallelizable and

are good at observing agents in an environment, the focus shifts from the textual representation of the data to the space and structure of the data.

## 2.1 Distributed frameworks for data science

This subsection first covers the literature on two big data libraries: MapReduce and Spark, including their applications in the area of computational geometry. Thereafter, we introduce the MASS library as a different approach to big-data computing.

### 2.1.1 Hadoop MapReduce

MapReduce is a framework designed to process large data sets in a distributed environment. Its MapReduce programming paradigm abstracts the concept of parallel programming into two functions: map and reduce. The map function transforms individual input records into intermediate key-value pairs, denoted as  $\langle k, v \rangle$ . On the other hand, the reduce function processes all values associated with the same key and produces the final output [7].

Early work with MapReduce to solve various computational geometry computational geometry was proposed by simulating the Bulk Synchronous Parallel (BSP) model. However, to the best of the authors' knowledge, there are no known actual implementations of the proposals suggested. It remains to be seen how to implement other algorithms that do not follow the BSP model [8].

The skyline and Voronoi diagram construction algorithms have been implemented using MapReduce by Zhang C et al. [9]. The Voronoi diagram construction was accomplished using the traditional divide-and-conquer algorithm to MapReduce's map and reduce functions. However, the merging operation caused a bottleneck to the scalability of this implementation. Furthermore, over the years, there has been a growing interest in utilizing the library to address computational geometry problems, particularly those in-

volving 2-dimensional data [10]. However, support for such features can only be emulated through extensions to the original library, such as the SpatialHadoop extension. This extension for MapReduce adds support for R-Trees and QuadTrees since the standalone library is not well suited for spatial operations and lacks support for these data structures.

Support for complex distributed data structures such as a generalized version of RTrees named SD-RTrees, K-Dimensional Tree (henceforth KDTree) based K-RP [11], and Quad-Tree based hQT have been proposed for performing spatial queries. However, these approaches could not scale better as they were not structurally suitable for MapReduce since specific tree traversals require message passing, which is unavailable in MapReduce.

Lastly, the `CG_Hadoop` suite is a collection of computational geometry applications developed using the SpatialHadoop extension. The algorithms implemented in these applications utilize a divide-and-conquer approach to leverage the parallel performance. The data used in the developed applications were strictly in the form of CSV (comma-separated values) files generated from shapefiles of geographical data.

### 2.1.2 Apache Spark

As the significance of spatial computing has continued to grow, the development of powerful libraries to support a wide class of applications in big data streaming frameworks such as `Spark` has been significant. Leveraging `Spark`'s flexible in-memory data caching capabilities allows user applications to manage and optimize their data processing workflows effectively [12], resulting in highly efficient and scalable spatial computing solutions. The interactive mode allows users to run `Spark` commands one at a time and engage in any fine-tuning. However, this mode is limited to a single computing node, usually on the master node.

One key advantage of `Spark` is its advanced job execution scheme, which utilizes a

Directed Acyclic Graph (DAG) of stages and a lazy evaluation approach to optimize the execution of iterative algorithms. This approach enables developers to understand the processing path better and optimize their applications accordingly. In recent years, numerous systems and algorithms have emerged that leverage Spark’s distributed processing capabilities to support spatial queries over large, distributed spatial datasets. These systems typically rely on built-in or on-top implementations, allowing developers to choose the best approach to meet their requirements. While some work focused on the library’s performance, using different parameters [13], other works focused on the efficiency of algorithms used.

Spark GraphX is built on top of Spark’s RDD to support parallel computation of graph-based RDDs such as GraphRDD and VertexRDDs [14]. It offers a platform to implement graph-processing systems with the underlying framework of Spark. To support message passing, which is essential in graph and tree algorithms, GraphX implements the Pregel API [15]. Messaging is required to pass information between vertices of a graph. The GraphX component also provides support for performing operations on directed multigraphs. Previous studies have been conducted using four worker nodes and show the framework’s potential in big data that involves preserving and modifying distributed graphs [16]. However, further work must demonstrate scalability in a cluster environment. Another disadvantage of GraphX is the inefficiency in the construction of dynamic graphs. It is the best fit for static graphs streamed in fixed-sized windows. As graphs change over time, GraphX reconstructs the entire graph repetitively, which adds computational overheads, causing an increase in execution times and memory usage [17].

GeoSpark is a three-layered architecture, an in-memory framework that supports spatial data in Spark. It extends the core of Sparks’ native Resilient Distributed Datasets (henceforth RDDs) with the Spatial Resilient Distributed Dataset Layer and the Spatial Query Processing Layer [18]. The Spatial RDD layer provides support for the QuadTree



and R-Tree data structures [19]. The Spatial RDD layer introduces three new RDDs, namely PointRDD, RectangleRDD, and PolygonRDD, that allow the representation of spatial data in Spark. These RDDs enable a wide range of geometrical operations, such as overlap detection, distance computation, and minimum bounding rectangle calculation.

SpatialSpark is a distributed computing framework that supports indexed spatial joins and range queries [20]. However, in specific scenarios involving data-intensive applications, it was observed that the framework’s scalability might be limited when running on multiple computing nodes as opposed to a single node. The communication overheads between computing nodes, which can become a bottleneck, are believed to be the stem of the issue, further influenced by the number of partitions. To optimize the performance of SpatialSpark for distributed spatial data processing, one could consider tuning the framework’s partitioning strategy, network configuration, and resource allocation based on the application’s specific data and computational requirements. Additionally, implementing efficient data serialization and compression techniques can reduce communication overheads and improve the overall scalability of the framework.

One essential type of algorithm used in computational geometry is the Range Query. A range query algorithm performs a search operation on a collection of values and identifies all the data points between a user-specified range. There are different approaches to performing search queries, for instance, the KDTree-based approach of Range Search. Another tree-based variant of range queries is the Spark-based Interpolation Search (SPIS), specifically developed for use with Spark, without which the filtering mechanisms would be inefficient [21].

### **2.1.3 Multi-Agent Spatial Simulations Library**

The `Multi-Agent Spatial Simulations Java` (henceforth MASS) Library is an ABM

library proposed to conduct spatial simulations in a parallel environment. MASS has successfully addressed challenges such as agent migration and good resource utilization. MASS has been under development for over a decade ever since. The use of MASS library in big data analysis has shown promising results over the years. The most abundantly tested features of MASS in these years have been in solving problems in multidimensional spaces.

In MASS, ABM components are represented using the two modelling objects: *Places* and *Agents*. Places represent the simulation's environment, while agents are the entities or actors, each called an agent within that environment. Places elements (each called a place) are distributed among computing nodes on a cluster and can be located using a globally known set of coordinates. For instance, Fig 2.1 shows a two-dimensional mapping of places using x and y coordinates. Once mapped, all places are assigned to threads and can communicate with each other and with agents that visit them during program execution. Agents are stored in bags on each process and can traverse the cluster to visit places. When agents move between nodes, their data is serialized and passed between the cluster nodes via TCP communication. Not all simulations require both places and agents, as some simulations may only need places for computation. The MASS library functions using a master-worker pattern to control the simulation. Computing node 0 acts as the master or host node, while the other nodes are the worker or remote nodes. User applications interact with the MASS host node, and MASS controls all supporting communication with the worker nodes internally.

## 2.2 Previous Work

Early work by S. Gokulramkumar focused on applying the foundational concepts of MASS in computational geometry [22]. While the work proved that users could develop applications in MASS without experience, users would have to develop complex data structures and algorithms. The work introduced four essential applications in the said domain: Clos-

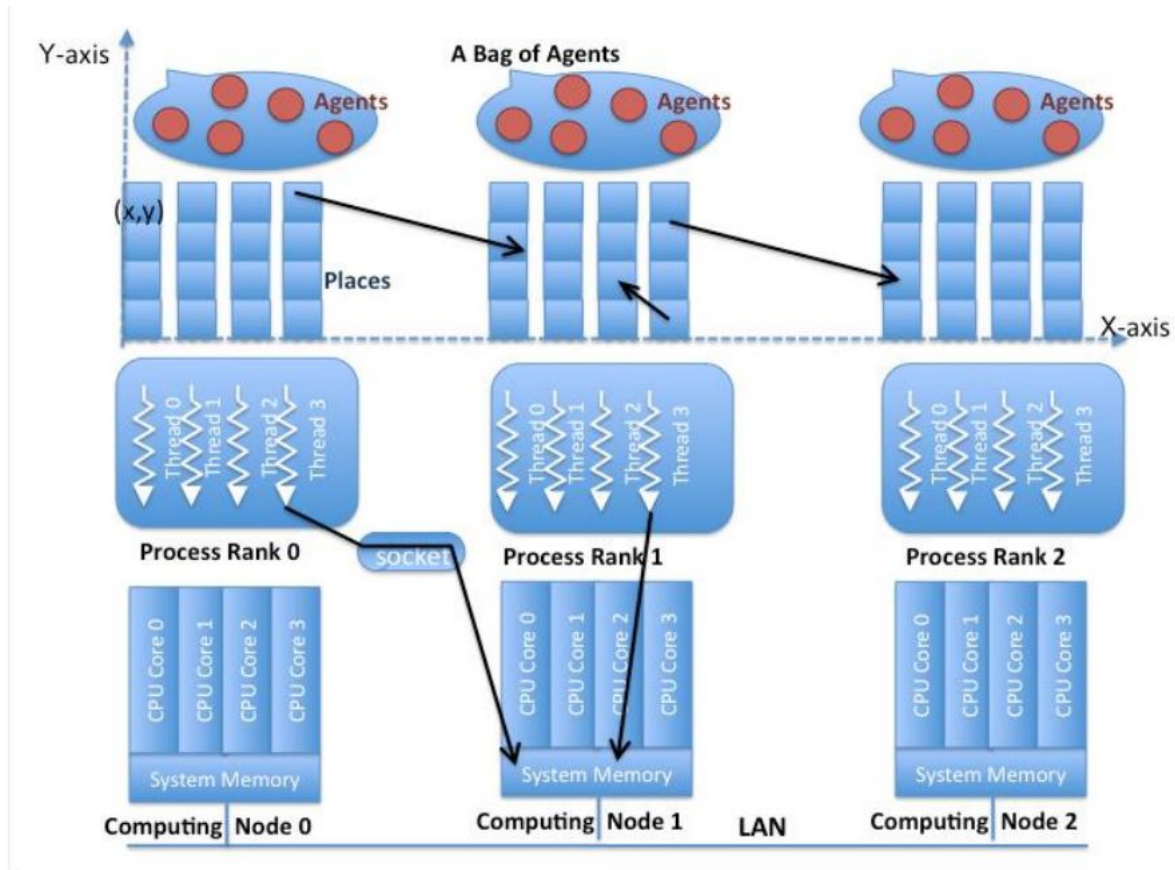


Figure 2.1: MASS Model

est Pair of Points, Convex Hull, Delaunay Triangulation, and Voronoi Diagram. At the time, the maximum number of nodes used to measure performance and scalability was eight.

For work to be extended and bolster the construction of graphs, the infrastructure in MASS was added by J. Gilroy [23]. The library's contributions include adding the GraphPlaces class, which contains the bulk of graph construction and maintenance functionality, and support importing graphs of HIPPIE and MATSim file types. Furthermore, the GraphPlaces component allows adding and removing individual vertices and edges in graphs. This work also addressed the challenges of creating dynamic data structures to allow continuous changes in the data structure.

Further work done by S. Paronyan focused on evaluating applications using MASS

against its counterparts [24]. The work added two applications to the existing applications library: Range Search, Largest Empty Circle, Point Location, and Euclidean Shortest Path. The work also compared MASS implementations to that of MapReduce and Spark counterparts. The measurements conducted at the time involved a maximum of 12 nodes.

To address issues such as wastage of memory and slow execution times of MASS in [22] and [24], Y. Guo [25] implemented and performed evaluations of three data structures in MASS: Binary Tree, Continuous Space, and QuadTree. Using the ContinuousSpace component optimized the creation of places by allowing each place to cover a specific range of coordinates. This way, there would be a reduction in the overall number of places created by MASS, resulting in reduced computational overheads. The SpaceAgent component's inclusion into the core library allows the use of agents over the SpacePlace data structure. The SpaceAgent component allowed users to define application-specific agent functionalities on top of the default propagate method. The propagate method could spawn agents in the von-Neumann and Moore Neighboring places of a place.

The development of BinaryTree and QuadTree data structures supported the creation of tree data structures in MASS since the only available option was GraphPlaces. Tree creation offered an opportunity to develop applications reliant on data storage, sorting and searching. The QuadTree data structure also helped solve the problem of handling uneven data distributions, impacting how sub-places to agents are assigned. The QuadTreeAgent component allows the use of agents over the tree data structures.

Lastly, V. Mohan identified and implemented eight automated agent navigation patterns to analyze structured data [26]. With this automated migration, an application developer using MASS would effectively perform less coding and focus more on their data. The research addressed shortcomings of MASS without the built-in functions that significantly impacted the performance of applications involving large datasets. Two of our benchmark applications were updated and measured for their accuracy, performance,

and programmability. The measurements conducted at the time involved a maximum of sixteen nodes.

The line-up of contributions from former students was pending a formal analysis of the performance and programmability. With respect to programmability analysis, metrics were collected manually previously. The metrics included LoC and, in some cases, cyclomatic complexity. The programmability analysis collectively for all applications must be redone using a code analysis tool to produce metrics that cannot be affected by human errors. In terms of performance analysis, the datasets used to test the applications during the time were insignificant for big data scenarios and lacked a formal evaluation methodology.

## **Summary**

While some work with MapReduce and Spark has extended big-data coverages from plain datasets to distributed data structures, their endeavours are not always successful due to the nature of data streaming. ABM provides an intuitive approach to solving geometry algorithms with swarming, propagation, and collision of agents. Since ABM solutions are parallelizable, we can leverage the benefits of handling complex data structures to solve applications in computational geometry. The data represented in computational geometry problems are normally sets of points in the Euclidean space. We selected fundamental problems for which algorithms have been described in [27]. Problems that initially originated as a means to accurately compute taxes for lands by the Pharaoh's now form the basis of a larger area of problems with a wide range of applications.

For this work, we use text files containing the input data for our applications. While the data files for the applications in this research do not require any special pre-processing for MASS to handle, data represented in MATSIM and HIPPIE file types must be pre-processed. The datapoints in MATSIM and HIPPIE files must be sorted by vertex IDs

where each of the file types follows a specific structure to organize the data [28]. Other data pre-processing tasks include the modification of shape files for geographic information systems to text format for MASS. Lastly, the fine-tuning of MASS offers the benefit of enhancing application performance by removing redundant code, in contrast to Spark and MapReduce.

## Chapter 3

### Related Work in Agent-based Approach

This section covers the literature on agent-based modelling toolkits and their attempts made for application in computational geometry. It is important to note that most of these toolkits are not parallel frameworks. Parallel-ABM libraries in Java are very limited in number.

#### 3.1 Agent Based Frameworks

##### 3.1.1 Repast Symphony

Repast Symphony is a single-node agent-based modelling toolkit based on Java [29]. This toolkit has been used to develop an application to solve the convex hull problem using the elastic band algorithm [30]. The space for the simulation is built-up in 2D and placing data points over it. Agents behave as an elastic band, starting from the outer perimeter of the space and moving inwards. As agents collide with the data points, they record the data points. Although this work focused on discovering the most natural way of solving the convex hull problem, the repast model underperformed compared to the primary Java version. Furthermore, using an additional library for performing a search algorithm adds to the execution time overhead. This search was necessary for the algorithm.

##### 3.1.2 NetLogo

NetLogo is a single node multi-agent based programming and modelling environment developed to simulate natural and social phenomena. It is well suited for modelling complex

systems developing over time. Modellers can instruct hundreds or thousands of agents, all operating independently, making it possible to explore the connection between the micro-level behaviour of individuals and the macro-level patterns that emerge from their interaction.

An extensive collection of built-in applications from art, biology, computer science, and mathematics to name a few, are packaged into the toolkit [31]. Agent migration patterns in ABM, such as the von-Neumann neighbourhood and Moore neighbourhood algorithms, are inspired by cellular automata. These patterns of agent migration are prevalent in computational geometry applications such as closest pair of points, where-in agents migrate outwards from source points in all directions. Lastly, the Voronoi diagram is also part of this collection of applications. Voronoi diagrams are the partitioning of a plane with  $n$  points into convex polygons such that each polygon contains exactly one generating point, and every point in a polygon is closer to its generating point than to any other.

### **3.1.3 DMASON**

MASON is a single-node open-source agent-based modelling simulation toolkit based on Java. It supports applying ABMs in geographical-information systems and social network simulations. MASON focuses on simulations of much more complex systems with a small, high-performance, self-contained simulation core so that many models can be run in parallel or involve up to millions of agents.

The DMASON framework is a parallel agent-based simulation framework written in Java based on the MASON toolkit that aims to hide the complexity of the distribution of agents between nodes to the developers. It was initially developed to harness unused computers in scientific centres that could take profit from distributing their MASON simulations and allows simply launching simulations on a distributed system using its



provided *Master* and *Worker* applications [32]. Currently, work still needs to be done using this framework for solving computational geometry applications. However, given the strengths of this library, it would be worth developing the applications.

### 3.1.4 Other multi-agent systems

The JaCaMo project is another framework used to describe multi-agent systems. JaCaMo provides a tuple-based communication infrastructure that allows agents to share data and coordinate their activities through a shared tuple space. The library has three key components: *agents* provided by Jason, *simulation environment* provided by Cartago and *organization* provided by Moise [33]. Jason is an interpreting platform that extends AgentSpeak, a language for agents based on their beliefs, desires and intentions to solve problems. Cartago is an infrastructure for modelling the environment of the simulation. Lastly, Moise is a framework used to define the structural elements of the rest of the simulation. In the area of computational geometry, no work exists using the JaCaMo library.

Jade is a distributed agent-based modelling library. It is meant for building agent systems but needs more support for simulation infrastructure [34]. Thus, it distinguishes itself from being a simulation framework because it has neither a scheduler nor the concept of having a ‘clock’ such as MASS or Repast. The scalability of this library is limited, thus, is not well suited for multi-agent-based simulations. A single agent is mapped to one thread, allowing them to migrate through environments. One helpful feature built-in is its GUI for monitoring remote agents, which can be helpful. However, given its limitations, very limited work has been done using this library. In the context of computational geometry, no work exists using this library.

The Ascape tool is written entirely and used with Java. It is a research-oriented software which takes an object-oriented approach. Similar to Repast, this toolkit is packaged into an IDE [35]. It provides a wide range of built-in functions and libraries for modelling

various aspects of complex systems, such as spatial environments, agent behaviours, and communication among agents. In the context of computational geometry, no work exists using this library either.

## 3.2 Summary

Studies have been conducted on utilizing simulation frameworks for agent-based modelling for data science using Repast Symphony and NetLogo [36]. However, the study pointed out the lack of support for systems to support such simulations. Similarly, the work done using ABM libraries in computational geometry is limited. One of the main reasons for this is that parallel ABM libraries are not widely researched and developed. The comparison between a parallel library (e.g., MASS Java) against single-node libraries in terms of the execution performance will be limited to single-node executions of MASS applications. Thus, we mainly focus on comparing the programmability. The big data libraries, on the other hand, focus on execution performance. It is an important factor when dealing with large sets of data. However, one of the critical aspects which still needs to be addressed is the target users of these libraries. Typically, it would be data scientists using such libraries. We move forward with the assumption that data scientists also give importance to the programmability of the library. They would want to write simple code in less number of lines. With frameworks such as Ascape and Repast Symphony which are good toolkits for agent modelling, the built-in features do not support a wide range of model-building approaches [37].

## Chapter 4

### Implementation

This section describes three achievements of this research: (1) tools development to compare MASS with Repast, MapReduce, and Spark in their programmability, (2) application development to compare their programmability and execution performance, and (3) preparation of performance evaluation.

#### 4.1 Tools Development

This section explains the development of two code analysis tools: each meant to analyze code in Java and C++ respectively. An additional tool was developed to ease the execution of MASS applications with automated node configuration file generation.

##### 4.1.1 Java Static Code Analysis Tool

A static code analysis tool is necessary to perform the programmability analysis of our applications. The Java static code analysis tool builds upon the work done by a former high school intern at the DSLab, Kent Fukuda. He developed a tool: LoC, short for Line of Code. The tool can read an input file, identify whether it is a Java or C++ file and read each line of the file. The tool identified the number of lines in the code, comments, and blank lines in each file without using a parser. However, a parser is required to analyze each file's contents further, identify variables and methods, and compute the cyclomatic complexity. The JavaParser library parses classes written in Java in this tool. JavaParser is an open-source library that provides many methods to parse, extract code fragments

and build abstract syntax trees.

The static code analyzer is a function-based tool which identifies and counts the number of conditionals, iteration, exception handling, logical operator, break, and return statements used in a class. Each of these statements introduces a new logical path into a program, causing a change in flow. Identifying these statements allows the computation of McCabe's cyclomatic complexity per method. This code complexity metric is a representative flow complexity of a program. It is an important metric to determine a program's stability and confidence level. LoC is a measure to identify how much programming is necessary to implement an application. Cyclomatic complexity is a metric used to determine the complexity and stability of a program based on the number of logical paths in it. The LoC and cyclomatic complexity are good metrics to explain better how much complex code one needs to develop for an application. The cyclomatic complexity is computed for each method of a program separately. Initially, the complexity of each method/function in the program is one ( $=1$ ), and for every aforementioned statement encountered, the complexity of the method is incremented by one. We also compute the average complexity of a method in a class.

This code analyzer consists of seven components (see Appendix A): (1) Runner initiates the code analysis process; (2) InitiateVisit initiates the MethodVisitor to visit sections of the code and increment the counter of each statement; (3) MethodVisitor component is the point of invocation of the Parsing process. A list maintains the count of each logical statement encountered in the code; (4) LinesOfCodeAnalyzer works on a directory. It combs the target directory for analysis and finds all source code files; (5) Pinter prints the results of the code analysis process onto the console and into a text file; (6) StatementCounter tracks the occurrences of statements; and (7) DataCollector collects all the metrics of an application directory.

Additionally, the tool can be run from any directory by the user, including the working

directory. For example, users can call the tool from a MASS Application's directory. The tool will identify the current working directory, explore all files and folders in the directory, and analyze relevant code files.

### **4.1.2 C++ Static Code Analysis Tool**

C++ code is hard to parse. We explored parsing libraries such as CPP-Parser and ANTLR (ANOther Tool for Language Recognition) before implementing the tool using ANTLR for its decently available documentation. ANTLR is a parser generator for C, CSharp, HTML and more. This tool uses ANTLR's C++ Grammar files. The grammar files contain rules that define how C++ code is parsed [38]. The rest of the tool's structure is similar to the Java code analysis tool (See Appendix B). During development, we realized that ANTLR surprisingly had no specific rules to parse an else-if statement as intended.

On the contrary, the parsing of 'if' and 'else' had no issues. The 'else-if' statement was parsed separately as an 'if' and an 'else' statement. The parsing introduced an inconsistency in our results because the counts of statements produced were incorrect. To fix this, we defined a rule in ANTLR's grammar file to correctly parse the 'else-if' statement.

### **4.1.3 NodesXML File Generator Tool**

An additional implementation to support the automation of benchmarking was required. MASS application users and developers require a node configuration file to specify the nodes/machines that will run the application. The configuration file is a file that lists the master and worker nodes of the cluster. The principle of this tool is simple. When called from within a MASS Application folder, the tool will instantly generate a nodes.xml file for the application. A user passes two arguments: (1) The number of nodes; and (2) The assigned port number.

A port number is necessary for MASS to perform messaging operations between nodes. A third and optional argument is the preference of the master node. By default, the master node is decided as 'cssmpilh.uwb.edu' by the tool. This tool utilizes the Javax XML Package to generate the XML document. The tool fetches the username required in the node configuration file for MASS, thus minimizing user inputs. Users can also call this tool from within a MASS application directory. The tool creates the configuration file in the currently active directory. This feature eases the use of the tool further.

## 4.2 Application Development

Using the code analyzing tools, this research conducted two comparative studies: (1) MASS Java versus Repast in automated agent migration and (2) MASS Java versus MapReduce/Spark in computational geometry. The automated agent migration in study 1 served as the basis of study 2's agent-based computational geometry. These studies needed to complete the following application development before the comparisons: 4.2.1: MASS base code for Repast applications, 4.2.2: Breadth-first search in Repast for study 1, and 4.2.3: Range search, and 4.2.4: Largest empty circle for study 2.

### 4.2.1 MASS Base Code for Repast Applications

The MASS base code extracts all the structural elements developed to mimic MASS in Repast for study 1: MASS versus Repast. This base code implements a set of routines that are necessary for conducting a simulation. The base code consists of seven components (see Appendix C): (1) The ApplicationBuilder component is a template component that provides a basic structure of context building in Repast. A ContextBuilder is necessary for all applications in Repast. A context consists of agents and the simulation environment, the places; (2) The FileInputReader is a utility component that reads input data files and populates place and vertex objects; (3) The AgentManager is essential since it manages operations such as agent creation, termination, and controlling agent migration;

(4) The agent component creates objects that will act as the agents in applications; (5) The Place component develops the space and acts as a point of invocation of method calls to the AgentManager; (6) The Graph component is used to construct the graph. It acts as a placeholder for agents, vertices, and places; (7) The Vertex component is used to create vertices of the graph. Each vertex object contains a list of its neighboring vertices and the distances.

Every agent is assigned a unique identifier, an agent ID. The identifier is the footprint of an agent. An agent leaves behind a footprint upon visiting a place. A place provides an environment for an agent and part of a graph or problem. A vertex object is stored in a place and is used to construct graphs. The footprint is beneficial in tracking and terminating agents when visiting a place already visited by another agent, a feature inspired by the PropagateRipple method implemented in MASS [26]. Agents in the closest pair of points application implement this logic, where agents terminate upon visiting a place visited by an agent originating from a different place.

### 4.2.2 Breadth-First Search

This application is used for MASS vs Repast comparison. The BFSBuilder is a component of this application that builds the simulation objects (see Appendix D). It adds objects of the graph into the context and renders it on the GUI. The GraphGen component generates a graph and neighbours of each node. Each place acts as the node of the graph, and each place has neighbouring places. The Graph object contains a list of place objects where each place holds one vertex of the graph. Each place is home to precisely one agent. At the beginning of the simulation, only agent 0 at place 0 is active. All other agents are dormant, meaning they cannot migrate or perform any activity.

A scheduled method in each place calls the AgentManager to allow all active agents to migrate to a neighbouring place. At the start of the simulation, agent 0 looks for its

neighbouring places and picks one to which it will migrate and return. On arrival back to the source place, the agent triggers a dormant agent at the visited place to be activated. The AgentManager performs the activation of an agent. From Fig 4.1, the agent signals the AgentManager to activate agent 1 residing in place 1. The AgentManager keeps track of all the places visited by agents. If more places have yet to be visited, the scheduled method in each place triggers the active agent residing in that place to migrate to a neighbouring place. Finally, with no more places to explore, the simulation ends.

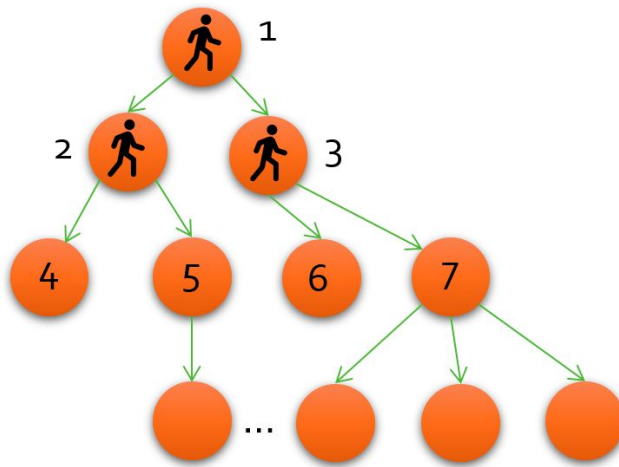


Figure 4.1: Breadth First Search

### 4.2.3 Range Search

The Repast Symphony application is developed by re-engineering the underlying implementation of the MASS application set in computational geometry that is used for comparison with MapReduce/Spark. The context builder for the application resides in the RangeSearch component (see Appendix E). The RangeSearch component reads the input file and constructs the KDTree before building the context. The AgentManager initiates the search for nodes in the KDTree within the query range.

From Fig 5.10, an agent first starts at the root node: node 1. If the coordinates' bounds are on the node's left child, the agent continues traversing, i.e., migrating to the



left child. If a point lies on the right child node, AgentManager triggers the activation of an agent residing in the child node to start traversing. To further understand agent behaviour, the number of agents used for the traversal, a method to count the total number of agents activated in the simulation is implemented. A critical difference between the Repast and MASS versions is that while MASS spawns a new agent, Repast Symphony activates an agent. Thus, activating an agent in the application increases the counter by one. Agents continue traversing and fetching results from the KDTree until all data points are collected.

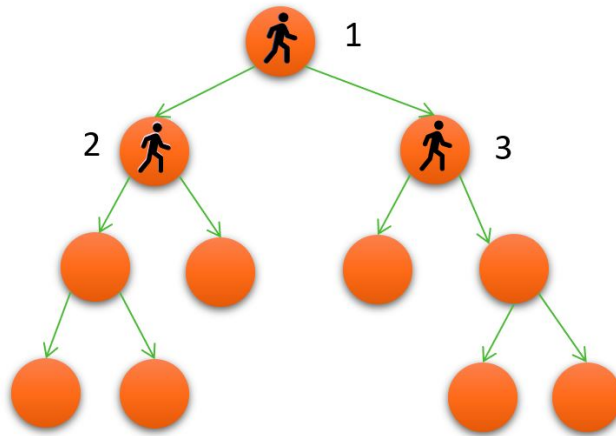


Figure 4.2: Range Search

#### 4.2.4 Largest Empty Circle

The previous implementation of the Largest Empty Circles application was incorrect. The implementation computed the farthest pair of points from the input data sets separately and applied a brute-force approach to computing the farthest pair of points among them. Additionally, the implementation is not executable because of flaws in the simulation setup. As a result, we re-implemented this application for MASS.

The Largest Empty Circles problem, also known as the Toxic waste dump problem, builds on two sets of data points: Cities and Dump Sites. The goal is to identify a dump site farthest from all the cities to dispose of toxic waste and minimize the impact

on human establishments. The space aspect of the application leverages the SpacePlace component of MASS to describe the problem over continuous space. The agent aspect of the application was developed using the SpaceAgent component of MASS and implemented automated agent migration patterns such as PropagateRipple. The underlying algorithm of this application follows a brute-force approach.

The application requires two sets of agents. The first set of agents resides in the points representing cities (henceforth city agents). These agents act as static observer agents. These agents do not propagate, and their job is to monitor if an agent originating from a different place enters the place where they reside. The second set of agents reside in the waste dump sites (henceforth site agents). These agents propagate to other places.

At the start of the simulation, a variable of type ‘double’ is initialized to the minimum value it can store. From Listing 4.1, the simulation continues for as long as site agents are active. The city agents first monitor their current place and identify if any site agents entered their place. If a site agent visits a city, we calculate the distance between the dump site from which the site agent originated from and the city. If this distance exceeds the current value of the maxDistance variable, we record the city and the dump site coordinates. After the end of the simulation, we print the maximum distance variable and the associated points.

Listing 4.1: Largest Empty Circles MASS implementation pseudo-code

```

1 double maxDistance = Double.MIN_VALUE;
2 while(vertexAgents.nAgents() > 0){
3     Object sitePairs = CityAgents.callAll(FarthestPairAgent.COLLECT_PAIRS, null);
4     for(Object siteob : (Object[]) sitePairs)
5     {
6         // Compute the distance between City and Dump Site from which Agent
7         originated. If this distance is greater than the current maxDistance
8         value, set maxDistance as the current distance and record Dump Site
9         and City Points.
10    }
11    vertexAgents.callAll(FarthestPairAgent.PROPAGATE_RIPPLE);
12    vertexAgents.manageAllSpace();
13 }

```

From Fig 4.3(a), initially, the data is scattered over the 2-dimensional space. MASS computes the space size depending on the maximum and minimum values of the data points from both sets. We map the data points to places. To observe agents' behaviour, Fig 4.3(b) shows a section of the space in a grid where each cell is one place. In the first migration, the agent residing in the site spawns child agents in the von-Neumann and Moore neighboring places.

From Fig 4.4(a), newly spawned child agents continue to spawn agents in their von-Neumann and Moore neighbouring places. Eventually, from Fig 4.4(b), an agent enters the city with the static agent. The static agent records the distance between a city and a site. We compare the distance between the points to the current value of the maxDistance variable. If the computed distance exceeds maxDistance, we set the value of maxDistance to the recently computed distance and save the site and city points. At this stage, we terminate the site agents originating from the site. This simulation continues until we compute all the distances between sites and cities.

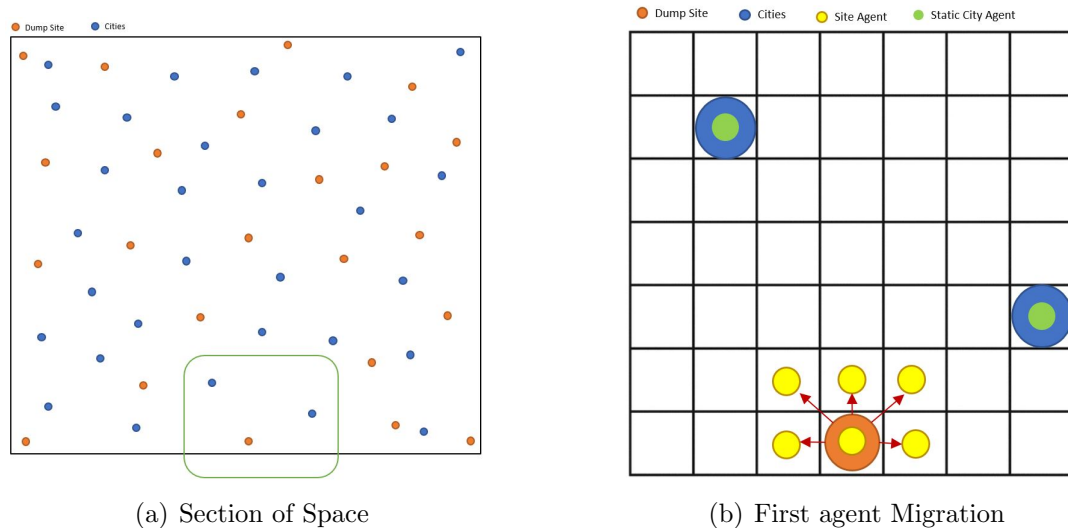
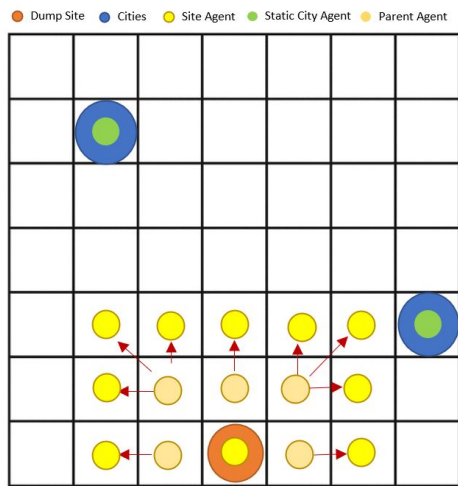


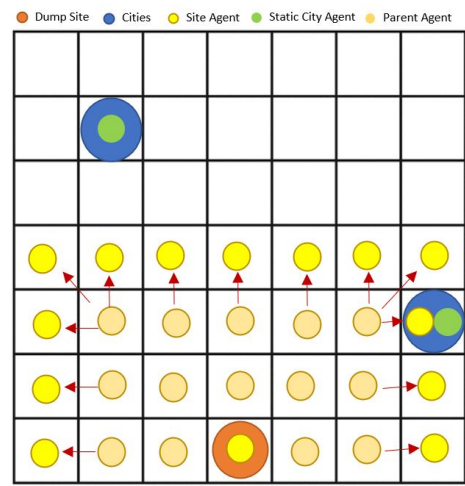
Figure 4.3: Largest Empty Circles Initial Migrations

### 4.3 MASS Fine-Tuning as Preparation for Performance Evaluation

MASS applications required fine-tuning to reduce wastage of resource utilization (Note that MapReduce and Spark benchmark programs have already been fine-tuned for computational geometry). All MASS applications perform logging operations included by former students for debugging. In a large-scale test, the logging operations become resource intensive. Thus, the logging operations in all the applications were turned off. Further and more importantly, optimizing the size of the simulation space is necessary. Especially the MASS applications that do not implement the SpacePlace component to define the space must be tweaked. The SpacePlace component efficiently handles place creation by only creating as many places as required depending on the minimum and maximum values of the dataset. The regular place component, on the other hand, does not perform such optimization by default. As a result, some applications, such as the convex hull, created an exponentially large number of places. To fix this, we implemented a helper method in the application to compute the minimum and maximum values in the dataset depending on which places are created.



(a) Agent Migration using Propagate Ripple



(b) Agent Migration Static City agent and Site agent Collision

Figure 4.4: Largest Empty Circles: Finding Distance between City and Dump Site

## Chapter 5

### Evaluation

We compared MASS and Repast in their agent migration and MASS versus MapReduce/Spark in computational geometry, both measured in programmability and execution performance. This section first verifies the correctness of the analyzing tools, then looks into programmability, and finally measure the execution performance.

#### 5.1 Verification of Evaluation Tools

The code analysis tool for Java was used to evaluate the programmability of all the applications in this research. Since no external tools are available that produce McCabe's cyclomatic complexity for applications in Java, manual verification is the only way to verify the correctness of the tool. As a result, we performed a manual collection of the metrics while comparing MASS and Repast applications. The metrics from the manual process and the code analysis tool were recorded in Excel spreadsheets and compared (see Appendix F).

The verification revealed inconsistencies in the results from the code analysis tool. The tool did not implement methods to identify *exception handling statements* and *logical operators*. To rectify the incorrect results, we updated the code analysis tool to support maintaining the occurrences of exception handling statements and logical operators in a *HashMap*.

## 5.2 Programmability Analysis

This section compares the programmability between MASS and Repast in agent migration as well as the programmability between MASS and MapReduce/Spark in computational geometry.

### 5.2.1 MASS versus Repast

This comparative work used the following four applications: Breadth-first search (henceforth BFS), Triangle counting, Range search, and Closest pair of points (henceforth CPP) to evaluate their agent migration features. MASS no longer requires applications to write fine-grain instructions to support automatic agent migration, tree traversal, and 2D propagation. The inclusion of automated agent migration over complex distributed data structures supports tree traversals and propagation.

To evaluate the programmability of the applications, we compute the Lines of Code (henceforth LoC) and cyclomatic complexity of the code. Agent LoC represents LoC used to define the agent behaviour in the simulation space. Space LoC is code that defines the space upon which the problem is distributed and built.

As shown in Table 5.1, the overall LoC in Repast is higher than MASS (i.e., 1545 compared to 1056 in total) because of the need for code defining the structural elements of the problems, including the creation of graphs and trees. Agent LoC is higher in Repast (i.e., 553 compared to 274) because of the mandatory use of AgentManager, much of which contributed to the agent LoC. Space LoC is higher, too, in Repast (i.e., 378 compared to 186) because it is used to develop the space and is a point of invocation of method calls to the AgentManager. This simple process seeks fine-grained instructions when initiating, performing, and ending the simulation.

CPP in MASS used an additional class, taking advantage of the strengths of object-oriented programming in storing the CPP results; in contrast, for Repast, a simple array of points stores the results. Repast’s CPP saw reduced code (i.e., 314 compared to 362) because of the simplified approach to recording the results. The cyclomatic complexity in MASS is increased through frequent iterations and conditionals, much of which is contributed by calls made to the base methods in MASS. It ranges from 2.25 to 3.944 compared to Repast’s cyclomatic complexity, which ranges from 1.785 to 2.6. Finally, in BFS with Repast, the modified AgentManager increased the agent LoC. Space LoC was majorly increased because, when compared to implementations of the other Repast applications, the generation of vertices and their neighbouring vertices was in-built into the application.

Table 5.1: Quantitative Programmability Comparison between MASS and Repast Symphony [39]

Measures	Libraries	BFS	Tri Count	Range Search	CPP
LoC	MASS	79	175	400	362
	Repast	432	260	539	314
Cyclomatic Complexity	MASS	2.25	3.875	6.83	3.1
	Repast	1.785	2.45	2.6	2.31
Agent LoC (A)	MASS	17	40	122	95
	Repast	229	76	139	109
Space LoC (S)	MASS	19	37	120	10
	Repast	111	94	130	43
Model Mgmt LoC - (A + S)	MASS	43	98	158	257
	Repast	92	90	270	162

## 5.2.2 MASS versus MapReduce/Spark

This comparative work used the following six applications: (1) Closest pair of points; (2) Largest empty circles; (3) Convex hull; (4) Euclidean shortest path; (5) Point location; (6) Range search in computational geometry. From Fig 5.1 and Table 5.2, MapReduce requires the largest LoC because of the configuration of jobs and helper components to construct complex data structures and finer-grained computations for computing results. On the other hand, Spark requires the least LoC because of the flexible and easy operations available to be performed on RDDs. LoC in MASS applications are in between



because of code describing the agent and space components. MapReduce and Spark for ESP required the support of custom components to construct multidimensional objects as obstacles.

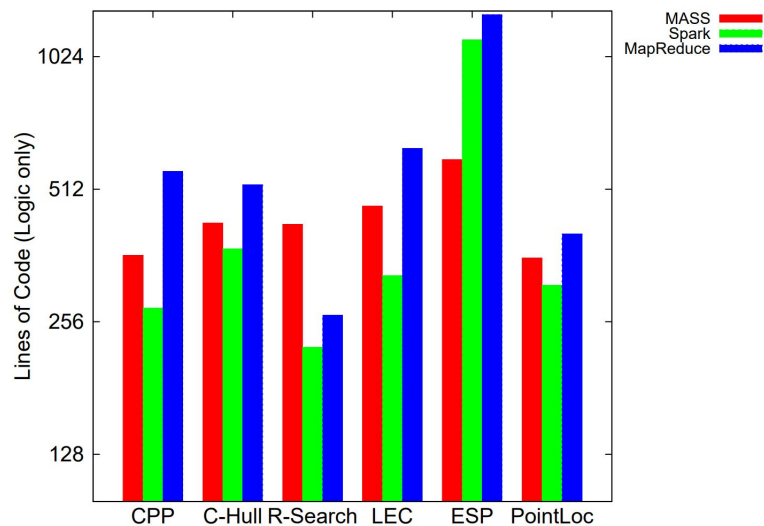


Figure 5.1: Lines of Code (Bar graph representation of MASS, MapReduce, and Spark)

The cyclomatic complexity in MASS is high because of calls to the core library for agent and space operations. With fewer functions performing the simulations, the average cyclomatic complexity per method increases naturally. As shown in Table 5.3, the cyclomatic complexity of MapReduce and Spark are relatively lower because of the increased use of modularized and highly-cohesive functions. In MapReduce, a significant impact on the cyclomatic complexity is caused by the *Map* (maximum of 9 in Range search) and *Reduce* (maximum of 45 in Point location) functions followed by methods developed to serve as helper functions or deduce results (maximum of 40 in Euclidean shortest path). In Spark, a significant impact to the cyclomatic complexity is caused by the Driver programs, i.e., the main methods (maximum of 38 in Point location). All other methods in the MapReduce and Spark implementations are minimal, reducing overall complexity.

In MASS, all methods implementing core library functions cause a significant impact

to the cyclomatic complexity (maximum of 36 in Range search). MASS implementation’s space and agent aspect involves multiple methods interacting with the core library. Frequent switch statements are used to call the appropriate methods from the library. Furthermore, methods describing initialization of the simulation space and agent behaviour frequently use loop statements resulting in a high flow complexity.

Table 5.2: Lines of Code of MASS, MapReduce, and Spark (Logic Only)

Application	Libraries		
	MASS	MapReduce	Spark
Closest Pair of Points	362	562	275
Convex Hull	429	524	375
Euclidean Shortest Path	598	1276	1118
Largest Empty Circle	469	634	326
Point Location	357	405	310
Range Search	426	265	224

Table 5.3: McCabe’s cyclomatic complexity of MASS, MapReduce, and Spark (per method)

Application	Libraries		
	MASS	MapReduce	Spark
Closest Pair of Points	4.05	2.43	3.00
Convex Hull	5.08	3.03	3.03
Euclidean Shortest Path	4.55	3.21	2.83
Largest Empty Circle	4.30	4.65	7.06
Point Location	2.54	4.45	4.16
Range Search	6.83	4.53	4.64

Applications that implement one of the eight built-in automated agent migration techniques result in reduced LoC for agent components. Applications that require problem-specific agent migration techniques to be developed often require detailed instructions. These detailed instructions are straightforward and frequently use conditional statements to call base methods from the core library. The space aspect of MASS applications is easy. Since most applications begin with the raw data (unmodified data) distributed over the space, the process is simple. For applications requiring special initialization logic (eg; Convex hull) to create places, there is an increase in the code complexity of the space components because of frequent loops and conditionals. Overall, MASS applications are easy to develop and require natural thinking approaches.

## 5.3 Performance Evaluation

### 5.3.1 MASS vs Repast

Repast applications depend heavily on the user interface. Each application and the structure of the problem space are rendered on the display. The application shown in Fig 5.2 is Breadth-first search with 50 vertices connected by edges. Buttons on this interface allow users to start and end simulations. From Table 5.4, we started facing issues while testing when we benchmarked the applications with the largest data sets. For example, the GUI struggled to render 10,000 vertices of the graph on display, which is unsurprising considering that the entire simulation was being performed on one system, resulting in insufficient memory. We continued testing the 10,000 vertices data set without rendering the graph on display, defeating the purpose of having an interface. Without the rendering, the application performed poorly, requiring over one hour to complete the execution. When compared to a single-node multi-threaded execution of MASS with 10, 100 and 1000 vertices, Repast was slower. For the 10,000 vertices, MASS was not able to complete the execution because of insufficient memory. However, on increasing the cluster size to 2 computing nodes with MASS, the execution of the application with 10,000 vertices was successful. MASS creates a higher number of agents compared to Repast. As a result, there is an increase in demand for memory.

Table 5.4: Breadth First Search Benchmarking Results Elapsed Time (in seconds)

Number of Vertices	MASS	Repast
10	0.059	1.6
100	0.187	1.5876297
1000	4.298	4.7715473
10000	-	3837.312504

Table 5.5 shows the results for Range search in Repast. The time to construct the KDTree is fast overall, regardless of the data size. The tree is constructed recursively by halving the data and creating left and right branches, making it an efficient approach.

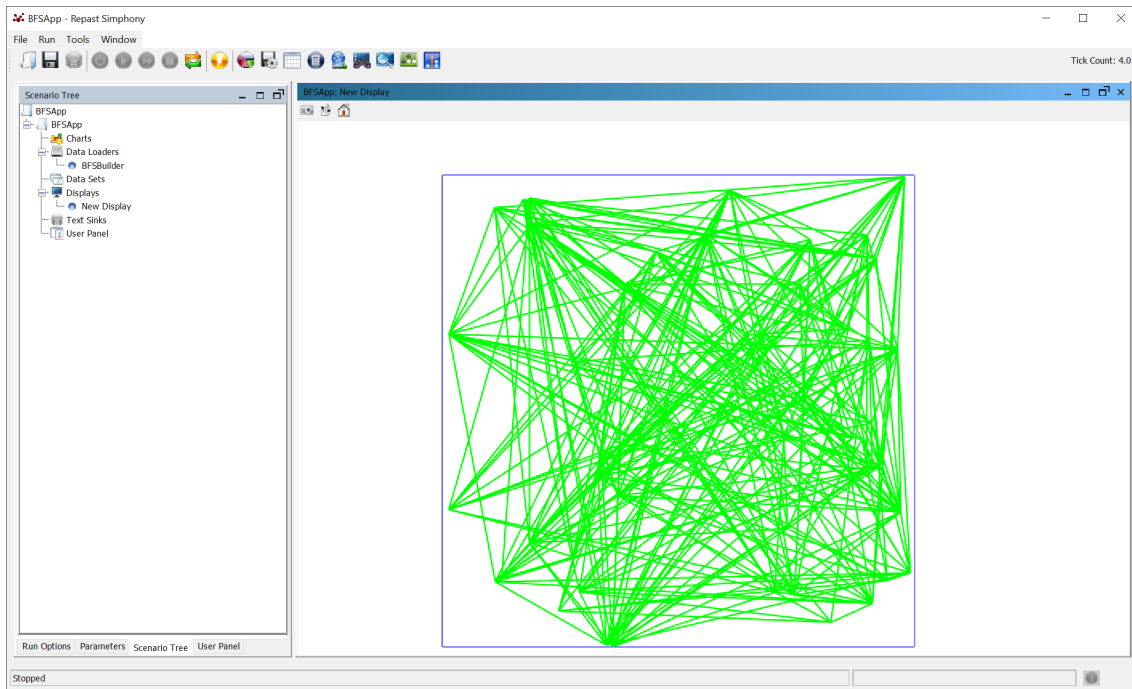


Figure 5.2: Breadth-First Search Repast Simulation

Overall, the Range search process was also very efficient because agents do not traverse the entire Tree. Instead, agents check the bounds/coordinates passed as input starting at the root node. If the points are present on the left branch, the agents traverse to the left. This process is vice versa if the bounds are on the right-side branch. At every stage, agents perform this check. As a result, there is a decrease in the total number of agents created for the simulation. An additional feature allowed us to calculate the total number of agents required in the application. When compared to a single-node multi-threaded execution of MASS with 100, 500 and 5000 vertices, Repast was slower. Observable differences in performance are seen in the largest dataset (10,000 vertices). The recursive KDTree construction is efficient, and the smaller datasets are not large that they cannot fit into the memory.

Table 5.5: Range Search Benchmarking Results Elapsed Time (in seconds)

Number of Vertices	MASS	Repast
100	0.131	1.5650227
500	0.159	1.5223931
5000	1.209	1.5709776
10000	4.095	3.2501259

### 5.3.2 MASS versus MapReduce/Spark

This section analyzes their performance to parallelize the six computational geometry applications. The illustration of each application results from an average of three tests performed for each node configuration. Detailed measurements are listed in tables in Appendix F-L. The benchmark measurements were performed when the operational load on the cluster was comparable.

#### Closest Pair of Point

All implementations of the Closest pair of points application use the same datasets. The MASS implementation maps data points to places in a 2-dimensional continuous space. Agents start at the data points. Agents perform the MigratePropagateRipple technique to find the closest neighbouring place. The MapReduce and Spark implementations of CPP follow a Divide and Conquer strategy by dividing the input points into stripes. Each stripe contains Three points. After merging adjacent stripes, the closest pair of points are computed for each resulting merged stripe. We follow this process until there is only One stripe remaining.

From Fig 5.5, the MASS implementation of CPP demonstrated significant improvements when the cluster size increased, followed by Spark. The MASS implementation benefits from the ABM approach, leading to a simple strategy. The MapReduce and Spark implementations rely on computations at each stage of merging stripes, increasing the execution time due to the computational overhead. MASS improved 12.50 times faster with 8 nodes over a single node for 50,000 data points and a 26.83 times improvement with 12 nodes over a single node for 100,000 data points. There was no significant change in the

execution speeds of MapReduce for both sets of data. The maximum size of the input files between both parameters was 1,150 Kilobytes. The flattened data file was small and could not leverage Hadoop's block partitioning. The partitioned data was of suitable size for just one node to process the entire block, leading to poor CPU scalability.

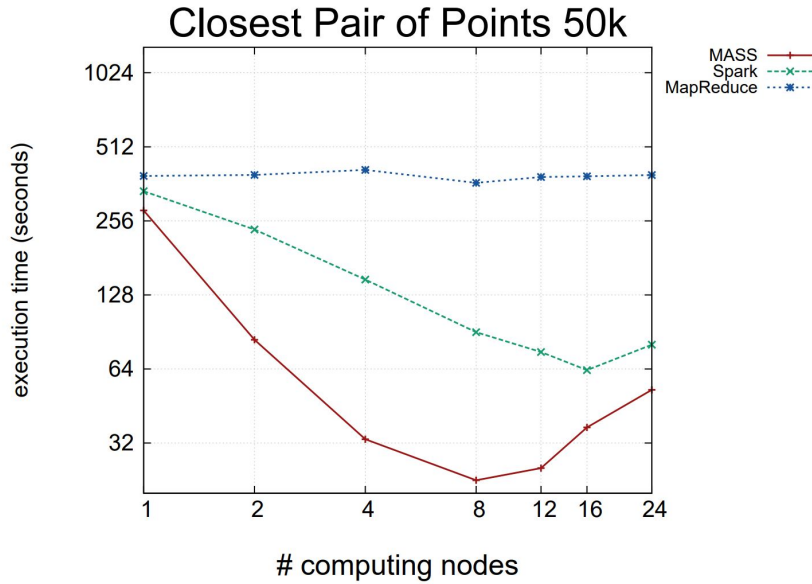


Figure 5.3: CPP: 50,000 Points

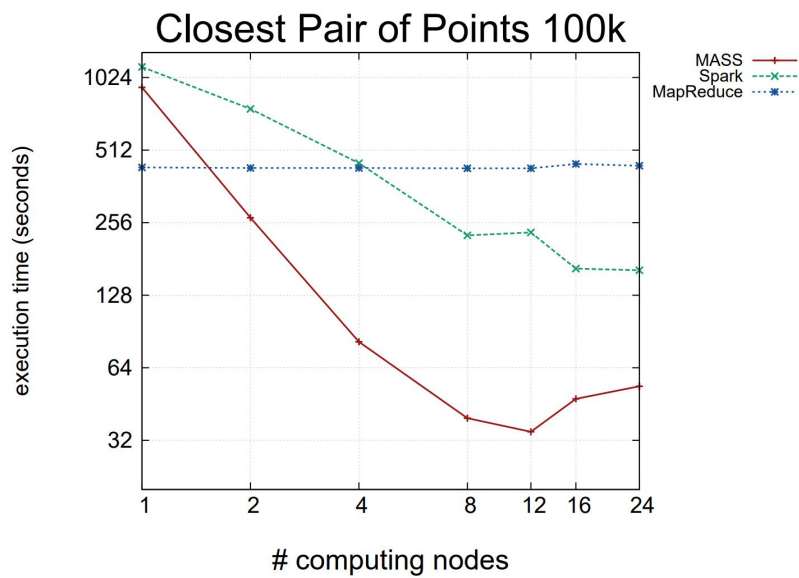


Figure 5.4: CPP: 100,000 Points

Figure 5.5: Closest Pair of Points: CPU Scalability of MASS, MapReduce and Spark

## Convex Hull

Figure 5.6 demonstrates the performance of all three implementations. The MASS implementation of convex hull implemented the elastic band algorithm approach [30]. The data points are mapped to places, and agents are initialized at the grid’s perimeter. Agents migrate inwards, resulting in collisions with data points. Upon collision, an agent is terminated. The data points agents collide with are collected till all agents are terminated. The MapReduce and Spark implementations, on the other hand, follow a Divide and Conquer strategy. The data is divided into groups containing one or two points. Convex hulls are created for each group individually. Finally, individual convex hulls are merged with neighbouring hulls iteratively until there is only one convex hull.

The Spark implementation performed better overall despite not significantly improving execution times with CPU scalability. Spark maintained an average of 11 seconds over all the node configurations for both datasets. We believe that the workloads cause the consistent results produced by Spark. The workloads for Spark are not computationally intensive to benefit from being distributed. On the other hand, MASS performed the fastest with 50,000 data points, achieving an execution time of 8.48 seconds with 8 nodes.

Lastly, MapReduce could not be benchmarked successfully with both the 12-node and 24-node configurations datasets. MapReduce tasks failed during the reduce operations while configured with 12 and 24 nodes, with an error indicating that the status was not reported to the master node within a timeout period. Increasing the timeout period from the default 600 seconds to 5000 seconds and above was unsuccessful. We are unsure of the likely cause of the timeout errors.

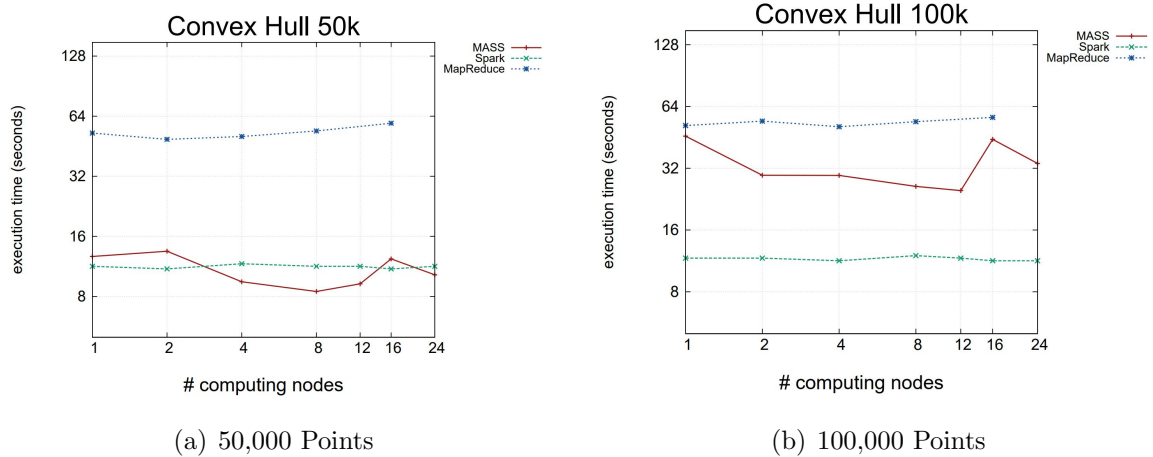


Figure 5.6: Convex Hull: CPU Scalability of MASS, MapReduce and Spark

## Euclidean Shortest Path

The Euclidean shortest path required the construction of geometric objects that would act as obstacles in all three implementations. Due to the lack of support for constructing complex objects in MapReduce and Spark, additional computations were necessary with the support of external libraries.

For the MASS implementation, the creation of obstacles was supported by places, adding a computational overhead. However, the obstacle initialization did not broadly impact the performance of MASS overall. Figure 5.7 demonstrates the performance of all three implementations. We saw promising results with MASS from 1 node through 4 nodes, after which the execution time increased rapidly. The MASS implementation currently sends the input data to all the nodes in the cluster and eventually to each place in the simulation. Given the wastage of memory caused by the current strategy, the master node is burdened by communicating the data file to all the nodes in the cluster. Thus, as the cluster size increases, the master node performs the additional work of sending the data file to each node. The measurements proved that our MASS implementation could have been improved for CPU scalability.



For Spark, the additional support for obstacle creation demanded significant computing power, resulting in the best performance overall. The performance of Spark on a single node was poor when compared to MASS'. With spatial scalability, the performance improved. Spark outperformed MASS in the smaller dataset but could not outperform MASS in the latter dataset (20k Obstacles). The high performance improvement of Spark in terms of percentage will be higher purely because of the poor performance on a single node. Lastly, MapReduce yielded the slowest execution times without significant performance improvements. Due to the lack of support for constructing complex objects in MapReduce, the Map functions were tasked with constructing the obstacle objects using custom components.

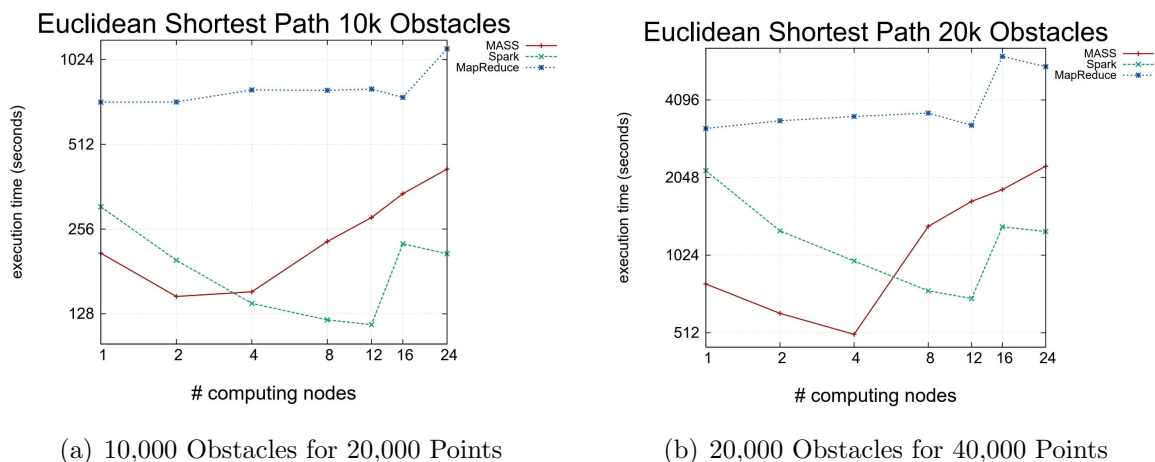


Figure 5.7: Euclidean Shortest Path: CPU Scalability of MASS, MapReduce and Spark

## Largest Empty Circles

Figure 5.8 demonstrates the execution times of our implementations of the Largest empty circles using MASS, MapReduce and Spark. In contrast to Spark and MapReduce, the MASS implementation showed significant improvements in execution time when increasing the size of the cluster. The MASS implementation took 252.95 seconds and 1017.46 seconds, with 50,000 points and 100,000 points in the *Sites* and *Vertex* (Cities) datasets. The MASS implementation in both cases performed the best with 12 nodes but was still outperformed by Spark. Although Spark did not demonstrate considerable improvements

in CPU scalability, the execution consistently outperformed the MapReduce and MASS implementations. The data shuffling process in Spark resulted in a stagnant performance. The execution time of MASS was increased by using the ManageAll operation for agents. The ManageAll operation is necessary to synchronize the status of all the agents during the simulation.

The MapReduce implementation was the slowest. The implementation required the use of seven different types of jobs. The first job was responsible for identifying the data points from the sites data that form a convex hull. The results of the first job are written into a temporary file. The second job reads the outputs from the first job to further compute a global hull, the results of which are written into a file. The third and fourth jobs perform the same tasks as the first and second jobs. The final three jobs are responsible for computing the largest empty circle and require intermediate results between each job. The increased execution times are attributed to the repetitive write and read operations between each job. The frequent read and write operations are disk intensive.

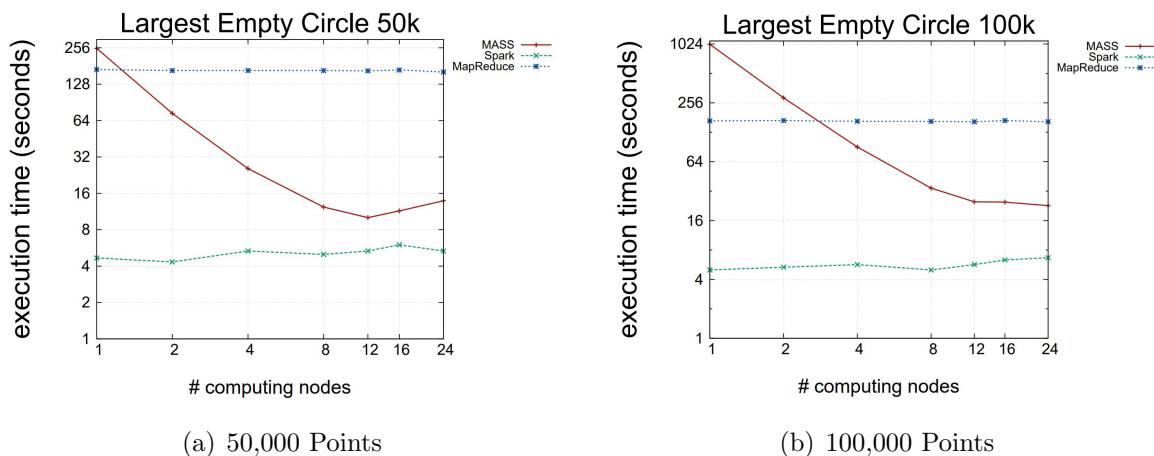


Figure 5.8: Largest Empty Circles: CPU Scalability of MASS, MapReduce and Spark

## Point Location

We benchmarked all three implementations of the Point location problem with 500,000 and 2,000,000 points. Fig 5.9 shows the performance of all three implementations with the three datasets. Overall, Spark demonstrated better performance with CPU scalability. The Spark implementation flattens the data before constructing trapezoid objects. Before searching for the query point, the Spark implementation takes advantage of the `mapPartitions()` method to transform Trapezoid RDD all at once instead of performing it on each row of data one by one. The Spark implementation balances out the heavyweight processing of the construction of the trapezoids with the efficient transformation of the trapezoids to improve overall performance. Spark performed the best with the 500,000 and 2,000,000 points datasets on 16 nodes (6.67 seconds and 7.67 seconds).

Vertical scaling did not improve the performance of our MASS implementation. Agents cause the slowdown in our MASS implementation. Our MASS implementation starts with one agent on each computing node. The number of agents increases as agents migrate through the trapezoids. Agent activity increases as we increase the number of nodes in our cluster. The `manageAll()` operation to synchronize the constantly migrating agents adds an overhead.

Finally, the MapReduce implementation did show significant improvements overall. The reducing phase of the implementation was responsible for creating trapezoid objects. The trapezoids are written to a temporary file for the next phase of Map and Reduce tasks to be executed for finding the query points. The intermediate writing and reading operations of the MapReduce implementation add an overhead.

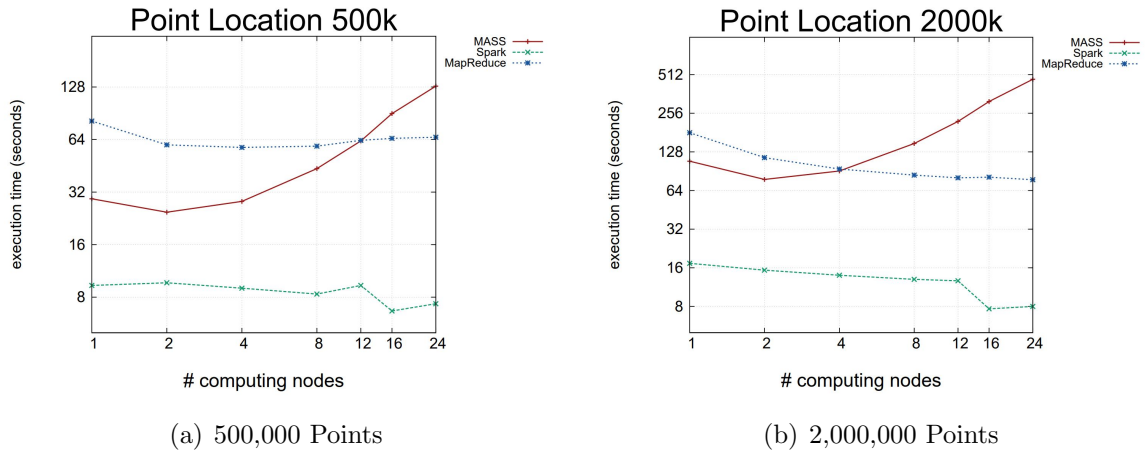


Figure 5.9: Point Location: CPU Scalability of MASS, MapReduce and Spark

## Range Search

Fig 5.10 demonstrates the execution times of our implementations of Range search using MASS, MapReduce and Spark. The MapReduce implementation significantly improved with the 1,000,000 coordinates dataset compared to the 500,000 coordinates dataset. However, MapReduce collectively did not demonstrate good CPU scalability. The fastest performance measurements were recorded with a single node up to 4 nodes, after which the performance declined. The KDTree for the MapReduce implementation is constructed in an array. As the input file size increases, the memory available on a single node must be increased to construct the KDTree. As a result, adding up to 4 nodes improved the performance. Furthermore, the search process for the query points is implemented during the Reduce phase, with the support of an ArrayList. All points within the user-specified query points are identified using a recursive process.

Our Spark implementation yielded strong results with CPU scalability to a maximum of 8 nodes. The Spark implementation reads the entire dataset into an RDD before being processed to create nodes of the KDTree. The overhead added by initialising points followed by the KDTree construction drastically increases the execution time. The high performance improvement of Spark in terms of percentage will be higher purely because of the poor performance on a single node.

Unfortunately, we could not successfully benchmark the MASS implementation in one specific case. The 1,000,000 points data set could not be tested with 2 nodes because the execution would get stuck after the construction of 65% of the KDTree. Threads enter the waiting state and never resume, causing a hangup. In all the other testing instances, the agent performance of MASS outperformed MapReduce and Spark.

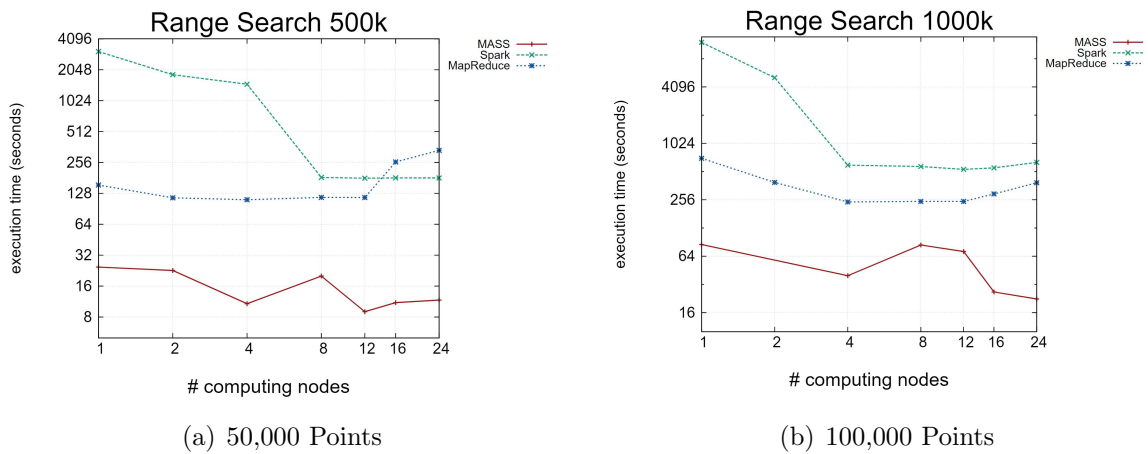


Figure 5.10: Range Search: CPU Scalability of MASS, MapReduce and Spark

## Chapter 6

### Conclusions

#### 6.1 Summary

We have conducted extensive studies in two directions. First, we analysed the automated agent migration of MASS against Repast Simphony. The study compared the applications' flow complexity and the code size necessary for describing the simulation, space and agents. We used four applications, each implemented using MASS and Repast. Our analyzing tools found that the flow complexity of MASS applications was higher than that of Repast but required less code describing finer-grained instructions. We also tested the capability of Repast for big data and observed that handling large sets of data was inefficient because of memory limitations, which showed the need for cluster computing.

For this evaluation, we constructed and stored data structures in the memory for MASS, in comparison to Spark, which flattens and streams the data into the memory. The approach for analyzing data structures with MASS is simple and is more effective for dispatching agents over it. We do not consider streaming data in MASS similar to Spark since it is not efficient. For instance, Spark's graph streaming allows dynamically changing structures but requires the construction of the entire graph each time the graph is modified.

Second, we performed a comparative analysis of MASS against MapReduce and Spark. To perform the quantitative analysis, we selected and benchmarked six applications in computational geometry; each implemented using MASS, MapReduce and Spark. Our

analysis shows that Spark required the least amount of code. The overall cyclomatic complexity of MASS applications was the highest. While MapReduce and Spark applications had lower flow complexities, the methods were highly cohesive. The driver applications in Spark were highly complex, whereas the Map and Reduce methods in MapReduce are significant contributors to increased flow complexity.

With respect to the ease of programming for big data, we cannot say that one framework is easier than the other. The paradigms of each of the three frameworks are entirely different from one another. Algorithm approaches for MapReduce and Spark follow Divide and Conquer strategy. On the other hand, algorithm approaches of MASS applications also include intuitive but Brute-force strategies. With respect to performance, MASS showed impressive spatial scalability when compared to MapReduce and Spark. While one application revealed a slowdown of agent migration in MASS, it outperformed Spark and MapReduce in most cases. MapReduce maintained poor performance in all the applications because of the repetitive expensive disk operations.

## 6.2 Future Directions

The static code analysis tools developed for Java and C++ in this research can be used to analyze future developments not limited to this work or libraries. The scripts developed to automate the execution of applications can be tailored to automate the execution of applications in the future. Additionally, the collection of benchmark applications in the field of computational geometry can be expanded by implementing the Voronoi diagram and Delaunay triangulation. Furthermore, the synthetic data used in our benchmarking can be replaced with natural data. For instance, the applications can be implemented into Geographical information systems (GIS). This research focused on the quantitative analysis of MASS, MapReduce, and Spark. Furthermore, a qualitative analysis of the three libraries must be performed. Lastly, data streaming for MASS can be considered only for adding and deleting data structures that have already been constructed.

# ACRONYMS

---

<b>Symbol</b>	<b>Meaning</b>
ABM	Agent-Based Modelling
ANTLR	ANother Tool for Language Recognition
LoC	Lines of Code
MASS	Multi-Agent Spatial Simulation
GUI	Graphical User Interface
BFS	Breadth-First Search
CPP	Closest Pair of Points
CH	Convex Hull
ESP	Euclidean Shortest Path
PL	Point Location
LEC	Largest Empty Circles
RS	Range Search
CPPParser	C++ Parser
RDD	Resilient Distributed Dataset
XML	Extensible Markup Language
GIS	Geographical Information Systems
TCP	Transmission Control Protocol

---



## REFERENCES

- [1] “Occupational outlook handbook.” Online accessed at <https://www.bls.gov/ooh/math/data-scientists.htm#tab-6>.
- [2] S.-H. Chen and R. Venkatachalam, “Agent-based modelling as a foundation for big data,” *Journal of Economic Methodology*, vol. 24, no. 4, pp. 362–383, 2017.
- [3] E. Bonabeau, “Agent-based modeling: Methods and techniques for simulating human systems,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 10, pp. 7280–7287, 2002.
- [4] L. Geaves, “Agent-based modeling of flood insurance futures,” 08 2020.
- [5] J. C. Jackson, D. Rand, K. Lewis, M. I. Norton, and K. Gray, “Agent-based modeling: A guide for social psychologists,” *Social Psychological and Personality Science*, vol. 8, no. 4, pp. 387–395, 2017.
- [6] “Agent-oriented programming: Intro.” Online accessed at <https://www2.econ.iastate.edu/tesfatsi/AOPRePast.pdf>.
- [7] “Mapreduce.” Online accessed at <https://hadoop.apache.org/docs/r1.2.1/mapreduce.html>.
- [8] M. T. Goodrich, N. Sitchinava, and Q. Zhang, “Sorting, searching, and simulation in the mapreduce framework,” 2011.
- [9] J. Zhang, X. Jiang, W.-S. Ku, and X. Qin, “Efficient parallel skyline evaluation using mapreduce,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1996–2009, 2016.
- [10] Y. Li, A. Eldawy, J. Xue, N. Knorozova, M. F. Mokbel, and R. Janardan, “Scalable computational geometry in mapreduce.” Online accessed at <https://www.cs.ucr.edu/~eldawy/publications/CGHadoop.pdf>.
- [11] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, “Voronoi-based geospatial query processing with mapreduce,” in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 9–16, 2010.
- [12] M. Guller, *Big Data Analytics with Spark: A Practitioner’s Guide to Using Spark for Large Scale Data Analysis*. Books for professionals by professionals, Apress, 2015.

- [13] Y. Zhang and A. Eldawy, “Evaluating computational geometry libraries for big spatial data exploration.” Online accessed at <https://www.cs.ucr.edu/~eldawy/publications/20-GeoRich-GeoLite.pdf>.
- [14] J. S. Andersen and O. Zukunft, “Evaluating the scaling of graph-algorithms for big data using graphx,” in *2016 2nd International Conference on Open and Big Data (OBD)*, pp. 1–8, 2016.
- [15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” Association for Computing Machinery, 2010.
- [16] J. S. Andersen and O. Zukunft, “Evaluating the scaling of graph-algorithms for big data using graphx,” in *2016 2nd International Conference on Open and Big Data (OBD)*, pp. 1–8, 2016.
- [17] Y. Hong and M. Fukuda, “Pipelining graph construction and agent-based computation over distributed memory,” in *2022 IEEE International Conference on Big Data (Big Data)*, pp. 4616–4624, 2022.
- [18] J. Yu, J. Wu, and M. Sarwat, “A demonstration of geospark: A cluster computing framework for processing big spatial data,” in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 1410–1413, 2016.
- [19] G. Mavrommatis, P. Moutafis, M. Vassilakopoulos, F. García-García, and A. Corral, “Slicenbound: Solving closest pairs and distance join queries in apache spark,” in *Advances in Databases and Information Systems*, (Cham), pp. 199–213, Springer International Publishing, 2017.
- [20] S. You, J. Zhang, and L. Gruenwald, “Large-scale spatial join query processing in cloud,” in *2015 31st IEEE International Conference on Data Engineering Workshops*, pp. 34–41, 2015.
- [21] Papadopoulos, A. N., S. Sioutas, C. Zaroliagis, and N. Zacharatos, “Efficient distributed range query processing in apache spark,” in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 569–575, 2019.
- [22] S. Gokulramkumar, “Agent-based computational geometry.” White Paper, 2020.
- [23] J. Gilroy, “Dynamic graph construction and maintenance.” White Paper, 2020.
- [24] S. Paronyan, “Agent-based computational geometry.” White Paper, 2021.

- [25] Y. Guo, “An implementation of agent-navigable contiguous 2d/3d spaces and trees.” White Paper, 2021.
- [26] V. Mohan, “Automated agent migration over structured data.” White Paper, 2022.
- [27] F. P. Preparata and M. I. Shamos, “Computational geometry an introduction,” 1985.
- [28] Y. Hong, “Graph streaming in mass java.” White Paper, 2022.
- [29] North, M. J, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, “Complex adaptive systems modeling with repast symphony,” *Complex adaptive systems modeling*, vol. 1, pp. 1–26, 2013.
- [30] S. Saadati and M. Razzazi, “Natural way of solving a convex hull problem.” Online accessed at <https://arxiv.org/ftp/arxiv/papers/2212/2212.11999.pdf>.
- [31] “Netlogo.” Online accessed at <https://ccl.northwestern.edu/netlogo/models/index.cgi>.
- [32] “Final report development of a distributed agent based simulation benchmark using d-mason.” Online accessed at [https://ddd.uab.cat/pub/tfg/2019/tfg182224/FinalReport\\_vFinal.pdf](https://ddd.uab.cat/pub/tfg/2019/tfg182224/FinalReport_vFinal.pdf).
- [33] “Jacamo.” Online accessed at <https://jacamo.sourceforge.net/doc/>.
- [34] “Jade.” Online accessed at <https://jade.tilab.com/>.
- [35] “Ascape.” Online accessed at <https://ascape.sourceforge.net/>.
- [36] F. Lorig and I. J. Timm, *Simulation-Based Data Acquisition*. Springer International Publishing, 2020.
- [37] Gilbert, Nigel, and S. Bankes, “Platforms and methods for agent-based modeling,” *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl\_3, pp. 7197–7198, 2002.
- [38] T. Parr, “Antlr website.” Online accessed at <https://www.antlr.org/>.
- [39] V. Mohan, A. Potturi, and M. Fukuda, “Automated agent migration over distributed data structures,” in *In Proceedings of the 15th International Conference on Agents and Artificial Intelligence - Volume 1*.
- [40] R. Ng, “Evaluation of euclidean shortest path, voronoi diagram and line segment intersection using mass, spark, and mapreduce.” White Paper, 2022.
- [41] J. Wang and J. L. Shaun Stangler, “Convex hull css 534.” UW Graduate Level Assignment Report, 2022.

- [42] M. BRAGEN, “Repast simphony system dynamics getting started.” Online accessed at <https://repast.sourceforge.net/docs/RepastSystemDynamicsGettingStarted.pdf>.
- [43] N. R. Jennings, “On agent-based software engineering,” *Artificial intelligence*, vol. 117, no. 2, pp. 277–296, 2000.
- [44] P. E. B. Pradnyana, K. M. Adhinugraha, and S. Alamri, “Highest order voronoi processing on apache spark,” in *Computational Science and Its Applications – ICCSA 2018*, (Cham), pp. 169–182, Springer International Publishing, 2018.
- [45] J. Yu, Z. Zhang, and M. Sarwat, “Spatial data management in apache spark: the geospark perspective and beyond.” Online accessed at [https://jiayuas.github.io/files/paper/GeoSpark\\_Geoinformatica2018.pdf](https://jiayuas.github.io/files/paper/GeoSpark_Geoinformatica2018.pdf).
- [46] A. Paudel, “Acceleration of computational geometry algorithms for high performance computing based geo-spatial big data analysis.” Online accessed at [https://epublications.marquette.edu/cgi/..](https://epublications.marquette.edu/cgi/)
- [47] S. Luke, R. Simon, A. Crooks, H. Wang, E. Wei, D. Freelan, C. Spagnuolo, V. Scarano, G. Cordasco, and C. Cioffi-Revilla, “The MASON simulation toolkit: Past, present, and future,” in *International Workshop on Multi-Agent-Based Simulation (MABS)*, 2018.

## APPENDICES

## A Class Structures of Static Code Analysis Tool for Java

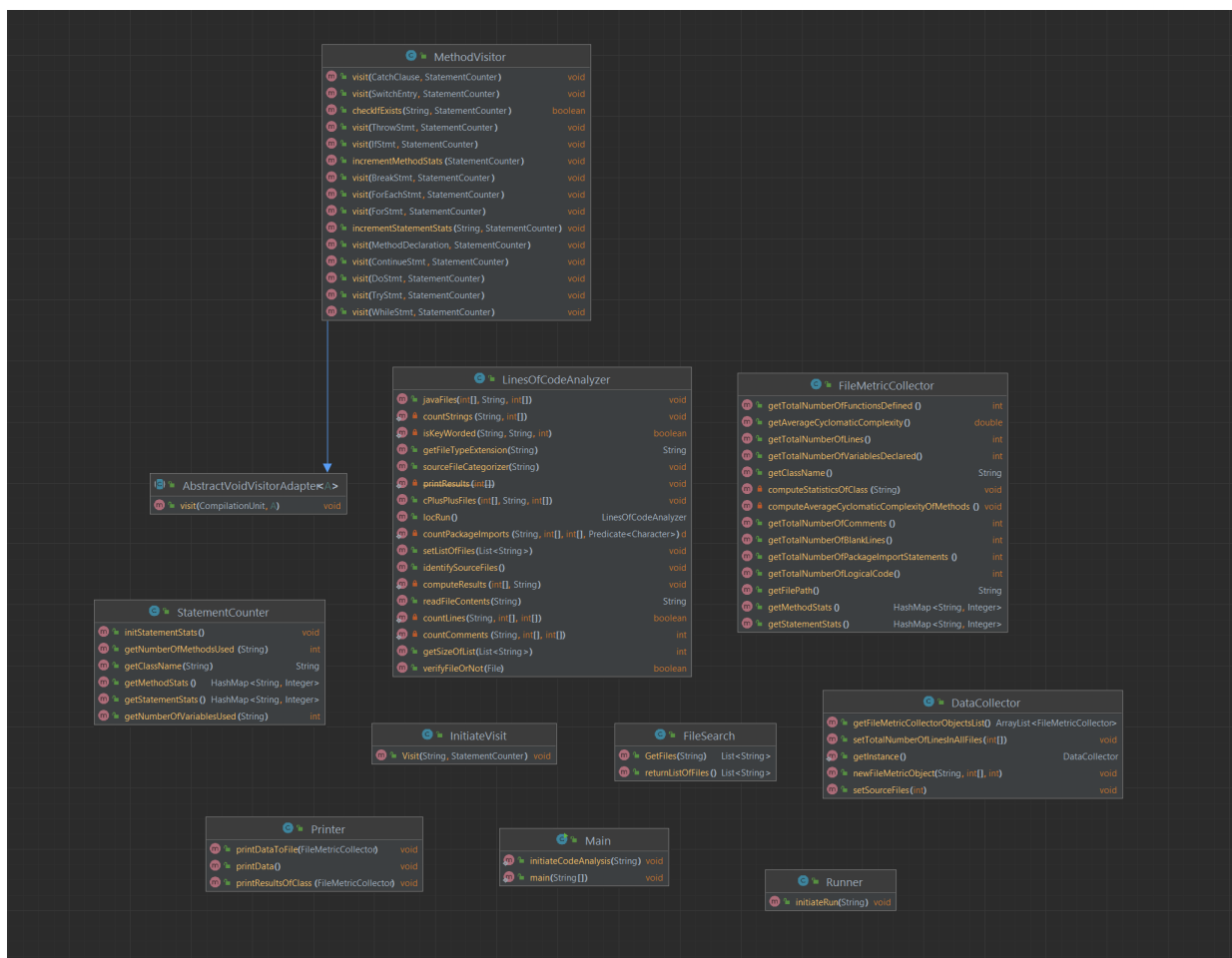


Figure A.1: Static Code Analysis Tool for Java

## B Class Structure of Static Code Analysis Tool for C++

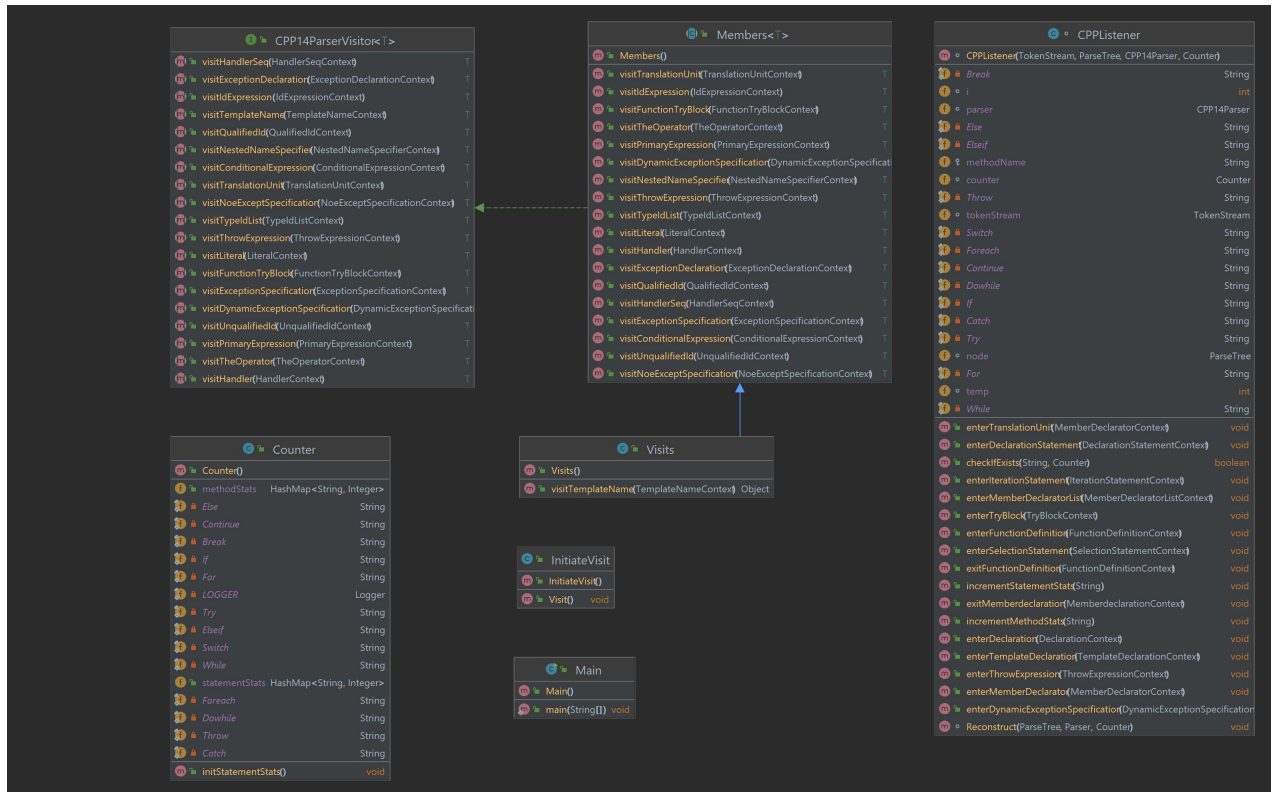


Figure B.1: Static Code Analysis Tool for C++

## C MASS Base Code for Repast

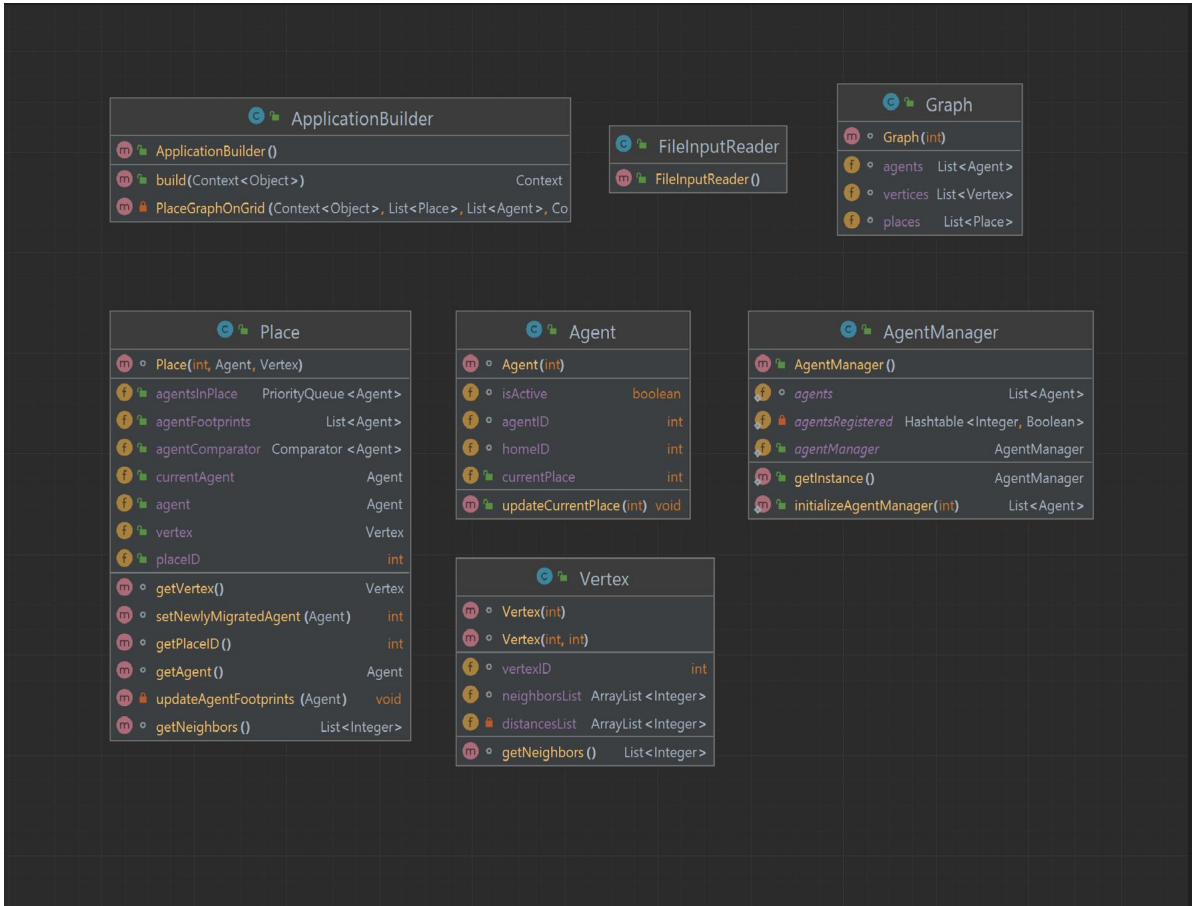


Figure C.1: MASS Base Code

## D Repast Breadth First Search

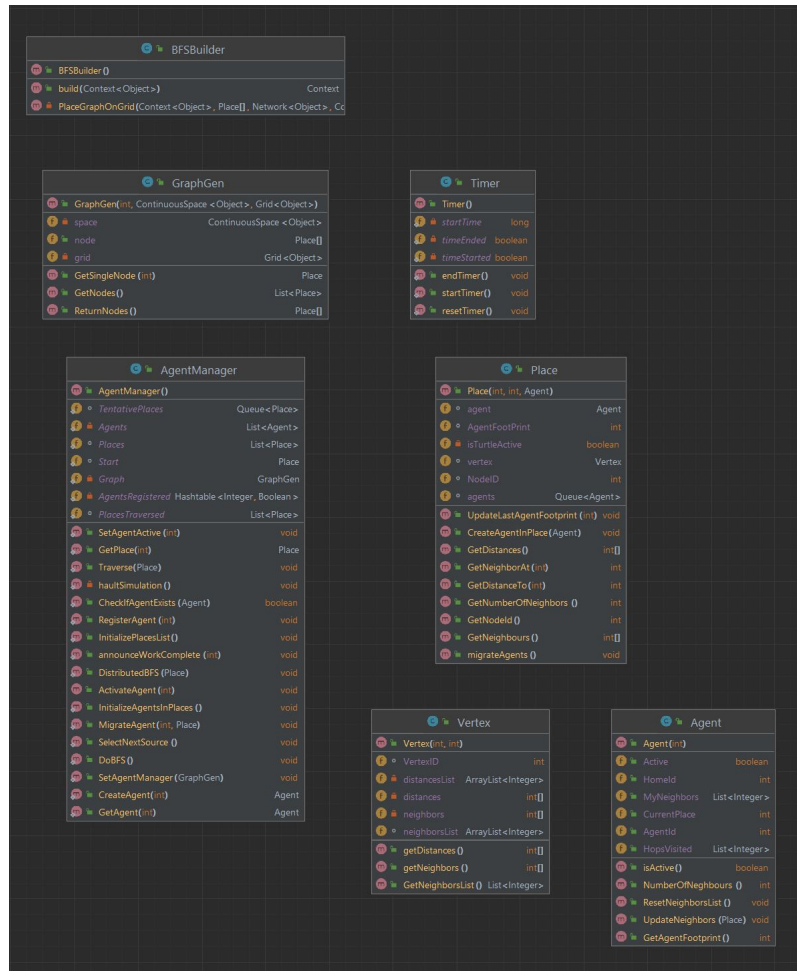


Figure D.1: Breadth First Search Components



## E Repast Range Search

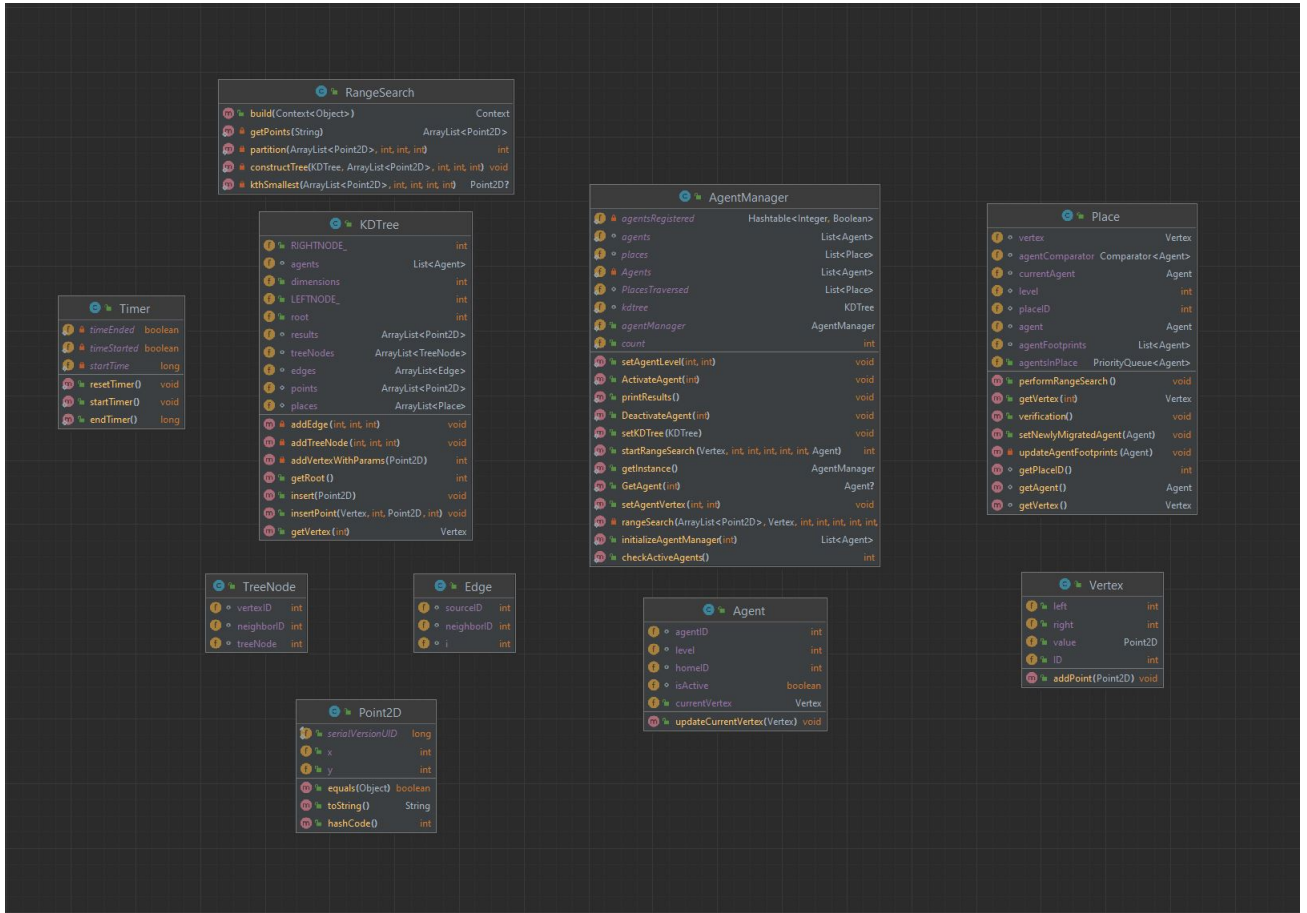


Figure E.1: Range Search Components

## F Verification of Evaluation Tools

Cyclomatic Complexity of MASS Java Code				MASS JAVA	
Application	Classes	Method names	Cyclomatic Complexity	Revised Cyclomatic Complexity	
BFS	BreadthFirst	Main method - public static void main	6	4	
		callMethod	2	2	
	Node	callMethod	2	2	
		init	1	1	
	Average Cyclomatic Complexity of methods of Application			2.75	2.25
Triangle Counting	CountTrianglesGraphMASS	run_triangle_counting	11	10	
		Main method - public static void main	9	6	
	NodeGraphMASS	callMethod	3	3	
		init	1	1	
	CrawlerGraphMASS	display	3	3	
		callMethod	5	5	
	CrawlerGraphMASS	init	1	1	
		getTriangle	2	2	
		Average Cyclomatic Complexity of methods of Application			4.375
	RangeSearch	KDTree	insert	3	2
insertPoint			17	15	
getRoot			1	1	
rangeSearch			17	9	
getVertex			1	1	
Point2D		toString	1	1	
		hashCode	1	1	
		equals	1	1	
		kthSmallest	7	4	
KD_RangeSearch		Main method - public static void main	9	7	
		getPoints	9	7	
		partition	8	5	
		constructTree	3	2	
		Vertex	hashCode	1	1
UpdateAgent		callMethod	3	3	
		DoSearch	17	9	
		update	1	1	
		OnArrivalEvent	1	1	
Average Cyclomatic Complexity of methods of Application			5.611111111	3.944444444	

Figure F.1: Manual verification of accuracy of code analysis tool metrics

## G Closest Pair of Points Results

Table G.1: Closest Pair of Points Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Points)	MASS Java	MapReduce	Apache Spark
1 Node	50,000	282.17	389.67	338.00
	100,000	931.98	433.67	1133.67
2 Nodes	50,000	84.10	393.67	236.00
	100,000	268.25	431.67	758.67
4 Nodes	50,000	33.14	412.67	147.67
	100,000	81.96	431.67	451.67
8 Nodes	50,000	22.57	365.33	90.33
	100,000	39.53	430.33	226.67
12 Nodes	50,000	25.30	386.00	75.00
	100,000	34.73	430.33	233.00
16 Nodes	50,000	37.01	388.33	63.33
	100,000	47.56	448.67	165.00
24 Nodes	50,000	52.64	393.00	80.33
	100,000	53.59	440.67	162.33

## H Convex Hull Results

Table H.1: Convex Hull Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Points)	MASS Java	MapReduce	Apache Spark
1 Node	50,000	12.70	52.67	11.33
	100,000	45.88	51.67	11.67
2 Nodes	50,000	13.49	49.00	11.00
	100,000	29.59	54.33	11.67
4 Nodes	50,000	9.48	50.67	11.67
	100,000	29.54	51.00	11.33
8 Nodes	50,000	8.48	54.00	11.33
	100,000	26.11	54.00	12.00
12 Nodes	50,000	9.26	-	11.33
	100,000	24.92	-	11.67
16 Nodes	50,000	12.34	59.00	11.00
	100,000	44.26	56.67	11.33
24 Nodes	50,000	10.28	-	11.33
	100,000	33.78	-	11.33

## I Euclidean Shortest Path Results

Table I.1: Euclidean Shortest Path Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Obstacles)	MASS Java	MapReduce	Apache Spark
1 Node	10,000	210.20	722.87	307.22
	20,000	791.03	3174.14	2176.43
2 Nodes	10,000	147.53	723.91	198.48
	20,000	608.49	3401.44	1271.59
4 Nodes	10,000	153.25	798.71	139.41
	20,000	504.50	3534.55	971.92
8 Nodes	10,000	231.34	795.84	121.84
	20,000	1323.96	3643.29	744.39
12 Nodes	10,000	281.38	804.30	117.18
	20,000	1654.91	3267.58	695.49
16 Nodes	10,000	341.84	751.52	227.10
	20,000	1838.38	6043.41	1318.82
24 Nodes	10,000	417.98	1117.837	209.24
	20,000	2267.14	5514.56	1264.31

## J Largest Empty Circle Results

Table J.1: Largest Empty Circle Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Sites and Vertices each)	MASS Java	MapReduce	Apache Spark
1 Node	50,000	252.95	169.00	4.67
	100,000	1017.46	167.33	5.00
2 Nodes	50,000	73.37	166.00	4.33
	100,000	288.29	168.67	5.33
4 Nodes	50,000	28.62	166.00	5.33
	100,000	90.46	166.00	5.67
8 Nodes	50,000	12.32	166.00	5.00
	100,000	34.33	165.33	5.00
12 Nodes	50,000	10.08	165.00	5.33
	100,000	24.88	164.00	5.67
16 Nodes	50,000	11.45	167.67	6.00
	100,000	24.71	168.67	6.33
24 Nodes	50,000	13.88	161.33	5.33
	100,000	22.74	164.00	6.67

## K Point Location Results

Table K.1: Point Location Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Points)	MASS Java	MapReduce	Apache Spark
1 Node	500,000	29.67	81.67	9.33
	2,000,000	108.23	180.33	17.33
2 Nodes	500,000	24.53	59.67	9.67
	2,000,000	78.07	115.33	15.33
4 Nodes	500,000	28.30	57.67	9.00
	2,000,000	90.73	94.00	14.00
8 Nodes	500,000	43.60	58.67	8.33
	2,000,000	148.70	84.33	13.00
12 Nodes	500,000	62.83	63.33	9.33
	2,000,000	221.00	80.33	12.67
16 Nodes	500,000	90.13	65.00	6.67
	2,000,000	315.73	81.33	7.67
24 Nodes	500,000	129.40	66.00	7.33
	2,000,000	470.57	77.67	8.00

## L Range Search Results

Table L.1: Range Search Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Obstacles)	MASS Java	MapReduce	Apache Spark
1 Node	500,000	24.52	154.43	3,080.33
	1,000,000	85.32	708.80	12,245.67
2 Nodes	500,000	22.72	115.96	1,831.33
	1,000,000	-	392.08	5,142.33
4 Nodes	500,000	10.78	110.95	1,479.67
	1,000,000	39.67	242.86	600.33
8 Nodes	500,000	20.04	116.96	183.00
	1,000,000	84.30	246.09	580.00
12 Nodes	500,000	9.02	116.62	180.00
	1,000,000	71.79	246.04	540.67
16 Nodes	500,000	11.05	258.93	181.33
	1,000,000	26.56	295.72	561.00
24 Nodes	500,000	11.69	336.02	181.00
	1,000,000	22.40	388.80	641.33

Table L.2: Range Search Benchmark Results - Distributed KDTree Construction (time in seconds)

Number of worker nodes	500,000 points	1,000,000 points
1	7.37	14.74
2	1603.42	-
4	2542.32	5054.24
8	3898.99	7998.26
12	3633.13	8201.09
16	3996.57	8507.11
24	4603.35	9682.51



## M Range Search Results

Table M.1: Range Search Execution Times (in seconds)

Number of Nodes	Dataset Size (# of Obstacles)	MASS Java	MapReduce	Apache Spark
1 Node	500,000	24.52	154.43	3,080.33
	1,000,000	85.32	708.80	12,245.67
2 Nodes	500,000	22.72	115.96	1,831.33
	1,000,000	-	392.08	5,142.33
4 Nodes	500,000	10.78	110.95	1,479.67
	1,000,000	39.67	242.86	600.33
8 Nodes	500,000	20.04	116.96	183.00
	1,000,000	84.30	246.09	580.00
12 Nodes	500,000	9.02	116.62	180.00
	1,000,000	71.79	246.04	540.67
16 Nodes	500,000	11.05	258.93	181.33
	1,000,000	26.56	295.72	561.00
24 Nodes	500,000	11.69	336.02	181.00
	1,000,000	22.40	388.80	641.33

Table M.2: Range Search Benchmark Results - Distributed KDTree Construction (time in seconds)

Number of worker nodes	500,000 points	1,000,000 points
1	7.37	14.74
2	1603.42	-
4	2542.32	5054.24
8	3898.99	7998.26
12	3633.13	8201.09
16	3996.57	8507.11
24	4603.35	9682.51

## N Running Benchmark Scripts

Find the benchmark applications under the following branch till updated in the develop branch: `anii_develop`

The scripts to run the benchmarks are placed in the corresponding folders of the applications. Each application consists of 3 sub-folders containing each of the implementations. The following applications folder names can be found in the MASS applications repository:

1. `ClosestPairOfPoints` under `Applications` folder
2. `LargestEmptyCircles` under `Applications` folder
3. `RangeSearch` under `Applications` folder
4. `ConvexHull` (developed by Shaun Stangler, Jason Lim, and Jaron Wang in [41]) under `ComputationalGeometry` folder
5. `EuclideanShortestPath` (developed by Richard Ng in [40]) under `Applications` folder
6. `PointLocation` under `Applications` folder

Each of the folders contains the MASS, MapReduce, and Spark implementations. Placed in each folder is the run script to automate the testing of the application on 7 different node configurations: 1, 2, 4, 8, 12, 16, and 24 nodes. The control access of the run scripts must be modified so that then scripts can be executable. This can be done using the following command: `chmod 700 runScriptFileName.sh`