©Copyright 2021

Satine Paronyan

AGENT-BASED COMPUTATIONAL GEOMETRY

Satine Paronyan

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2021

Committee:

Munehiro Fukuda, Chair

Michael Stiber

Min Chen

Program Authorized to Offer Degree:

Computing and Software Systems

University of Washington

Abstract

Agent-Based Computational Geometry

Satine Paronyan

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

The Multi-Agent Spatial Simulation (MASS) library is a parallel programming library that uses agent-based modeling (ABM) parallelization approach over a distributed cluster. The MASS library contains several applications solving computational geometry problems using ABM algorithms. This research aims to build additional four ABM algorithm-based applications: (1) range search, (2) point location, (3) largest empty circle, and (4) Euclidean shortest path. This research presents ABM solutions implemented with MASS library as well as divide and conquer (D&C) solutions to four problems implemented with big data parallelization platforms MapReduce and Spark. In this paper, we discuss design approaches used in solutions for the four problems. We present ABM and D&C algorithms with MASS, MapReduce, and Spark platforms. We provide a detailed analysis of programmability and execution performance metrics of ABM algorithm-based implementations with MASS against D&C algorithm-based versions with MapReduce and Spark. Results showed that the MASS library provides an intuitive approach to developing parallel solutions to computational geometry problems. We observed that ABM MASS solutions produce competitive performance results when performing computations in-memory over distributed structured datasets.

Table of Contents

Chapter 1 INTRODUCTION **8**

1.1 Problem Definition *8*

1.2 Research Objective *9*

1.3 Project Summary *10*

1.4 Report Structure *10*

Chapter 2 RELATED WORK **12**

2.1 Parallelization approaches to computational geometric algorithms *12*

2.2 Agent-based parallelization of in-memory data analysis *14*

Chapter 3 PARALLEL ALGORITHMS DEVELOPEMENT **17**

3.1 Range Search *17*

3.1.1 MASS range search implementation 18

3.1.2 MapReduce range search implementation 20

3.1.3 Spark range search implementation 20

3.2 Point Location *21*

3.2.1 MASS point location implementation 21

3.2.2 MapReduce point location implementation 23

3.2.3 Spark point location implementation 24

3.3 Largest Empty Circle *24*

3.3.1 MASS largest empty circle implementation 25

3.3.2 MapReduce largest empty circle implementation 26

3.3.3 Spark largest empty circle implementation 27

3.4 Euclidian Shortest Path *28*

3.4.1 MASS Euclidian shortest path implementation 28

3.4.2 MapReduce Euclidian shortest path implementation 30

3.4.3 Spark Euclidian shortest path implementation 30

Chapter 4 PARALLELIZATION **32**

4.1 MASS *32*

4.2 MapReduce and Spark *35*

Chapter 5 ALGORITHMS PERFORMANCE ANALYSIS **38**

5.1 Evaluation metrics *39*

5.2 Programmability *40*

5.2.1 Range Search 41

5.2.2 Point Location 42

5.2.3 Largest Empty Circle 43

5.2.4 Euclidian Shortest Path 44

5.3 Execution performance *45*

5.3.1 Range Search ... 45
5.3.2 Point Location ... 47
5.3.3 Largest Empty Circle ... 48
5.3.4 Euclidian Shortest Path ... 49

Chapter 6 DISCUSSION AND FUTURE WORK ... **51**

REFERENCES ... **53**

Appendix A OUTPUT SCREENSHOTS ... **56**

LIST OF FIGURES

Figure Page

Figure 3. 1 Range Search and KD tree construction 18
Figure 3. 2 Range Search computation with agent-based algorithm 19
Figure 3. 3 MapReduce and Spark range search algorithm 21
Figure 3. 4 Agent propagation and migration in MASS Point location 22
Figure 3. 5 The computation of the largest empty circle utilizing Voronoi diagram properties. . 25
Figure 3. 6 MASS agent-based largest empty circle computation 25
Figure 3. 7 Euclidean shortest path search with Places and Agents 28
Figure 3. 8 Spark Euclidean shortest path algorithm 31

Figure 4. 1 MASS execution model 32
Figure 4. 2 MapReduce execution model 36
Figure 4. 3 Spark execution model 37

Figure 5. 1 Execution performance for Range Search implementation 46
Figure 5. 2 Execution performance for Point Location implementation 47
Figure 5. 3 Execution performance for Largest Empty Circle implementation 48
Figure 5. 4 Execution performance for Euclidean Shortest Path implementation 49

LIST OF TABLES

Table Page

Table 3. 1 MASS range search algorithm 18
Table 3. 2 MapReduce and Spark range search algorithm 20
Table 3. 3 MASS point location algorithm 22
Table 3. 4 MapReduce point location algorithm 23
Table 3. 5 MASS largest empty circle algorithm 26
Table 3. 6 MapReduce largest empty circle algorithm 27
Table 3. 7 MASS Euclidean shortest path algorithm 29
Table 3. 8 MapReduce Euclidean shortest path algorithm 30

Table 5. 1 Programmability metrics of Range Search 41
Table 5. 2 Programmability metrics of Point Location 42
Table 5. 3 Programmability metrics of Largest Empty Circle 43
Table 5. 4 Programmability metrics of Euclidean Shortest Path 44

Chapter 1
INTRODUCTION

1.1 Problem Definition

The agent-based computational geometry research project explores the applicability of agent-based parallelization for designing and developing solutions to computational geometry problems. Computational geometry as a field of computer science involves design, analysis, and modeling of efficient algorithms to solve complex geometric problems. Computational geometry is applied in computer graphics, geometric modeling, computer vision, geolocation, statistics, motion planning, and parallel computing. Computational geometry applications are computationally complex and involve large datasets. A great number of O(n log n) efficient sequential algorithms have been designed to solve geometric problems [1][2]. However, the sequential algorithms are bound to one machine, which limits the amount of input data. Distributed memory addresses the scalability concerns and enables the analysis of large datasets more efficiently [3].

MapReduce and Spark are the common big-data parallelization tools for computational geometry applications. Yet, these tools have several programming and performance drawbacks. MapReduce is limited to map() and reduce() operations [3]. The other disadvantage is slower execution performance because of the disk-oriented computation method. Spark often produces better execution performance and programmability results in comparison with MapReduce. However, some problems remain with Spark computation as well. Spark computation technique involves many transformation and action function calls [4] that sometimes can slow down the performance if the function calls are repeatable because of immutable RDDs in Spark. Computational geometry needs different types of data structures such as 2D/ 3D space or a graph. Thus, the other approach is to maintain computational geometry structures over distributed memory and inject agents [5], which collaboratively find a solution. This approach enables us to design more intuitive algorithms that provide better execution performance and programmability.

Computational geometry problems often utilize the same structured dataset for data analysis operations. Big data parallelization tools, such as MapReduce and Spark spent considerable time on I/O and data shuffle operations instead of keeping the structure dataset in memory for the entire

duration of the computation. The MASS library provides Places, SpacePlaces, and GraphPlaces data structures for maintaining structure dataset over distributed memory. Agents migrate, spawn, and terminate to perform data discovery operations. To create application-specific places or agents for parallel solutions users derive their custom classes from the base classes and implement several required functions.

1.2 Research Objective

The key objective of this project is to model four agent-based parallel algorithms for solving four computational geometry problems using the MASS library. The project aims to prove that the MASS library provides an efficient approach for developing intuitive parallelizable algorithms to solve computationally complex computational geometry problems. The agent-based parallelization approach enables us to keep a structured dataset in memory and inject multiple agents, each solving a part of the geometric problem. Computational geometry problem solutions often use the same data set structure in different stages of the computation whereas the better technique is to utilize the ability of the MASS library of reusing the same distributed in-memory data set for the computations.

This research expands the work on agent-based computational geometry by parallelizing four additional geometric problems: (1) range search, (2) largest empty circle, (3) point location, and (4) Euclidean shortest path (also known as point robot motion or obstacle avoidance). This project uses three different parallelization programming frameworks such as MapReduce, Spark, and MASS to design and implement algorithms that solve the selected computational geometry problems. The specific goals associated with the project are:

1. design and implement agent-based algorithms to solve computational geometry problems,
2. implement solutions to the same computational geometry problems with MapReduce and Spark,
3. evaluate the execution performance and programmability of algorithms implemented with the MASS library in comparison with MapReduce and Spark implementations.

The developed agent-based algorithms, if demonstrate better efficiency than the one using other parallelization tools MapReduce and Spark, will prove that agent-based parallelization is a better choice for programming problems consisting of structural and geometric data analysis.

1.3 Project Summary

The four agent-based algorithms are designed with a similar approach by using a distributed data structure to organize the input data and set agents to discover data. In range search algorithms we use GraphPlaces to create a multi-dimensional (KD) tree that represents the input data. The range search operation is done by multiple agents that traverse the KD tree to find points within the given range. The use of GraphPlaces instead of the original Places class provides a more intuitive way to create, initializing, and traversing a binary tree. In the point location algorithm, we use original Places as a base class to map the input. However, we utilize event-driven programming to create custom search simulation. The event-driven programming requires the implementation of annotated functions within the application-specific agent class. In the largest empty circle, we use the latest addition to the MASS library that is a continuous space with SpacePlaces and SpaceAgent. The continuous space allows the creation of more compact Places for large datasets. In Euclidean shortest path, we use a similar approach as in point location by using Places and event-driven programming.

We benchmarked agent-based algorithms against divide and conquer algorithms implemented with MapReduce and Spark big data parallelization platforms. The performance evaluation includes programmability and execution performance metrics. The benchmarking results showed:

1. MASS superiority over MapReduce in programmability and performance: all applications
2. MASS superiority over Spark in programmability: point location and largest empty circle
3. MASS superiority over Spark in performance: range search and Euclidean shortest path

1.4 Report Structure

The project report is organized into the following chapters. Chapter 2 provides an overview of the previous related work. Chapter 3 outlines a definition of the considered computational geometry problems, designed and implemented parallel algorithms using three different parallel programming frameworks such as MASS, MapReduce, and Spark. Chapter 4 discusses the data

representation and key methods of parallel platforms in chapter 3, which are used to implement the designed parallel algorithms. Chapter 5 presents the benchmark results of the algorithms across all three platforms. Chapter 6 provides the conclusion by summarizing the project achievements and potential future improvements.

Chapter 2
RELATED WORK

Many applications of computational geometry are in computer graphics, robotics, pattern recognition, operations research, etc. The computational geometry problems solve analytical and statistical tasks over large data sets in real-time. Therefore, such complex computations require real speed and efficiency. The sequential solutions cannot be applied in actual real-world computations of large data sets. It is common to use parallel computation techniques to achieve performance goals. A variety of sequential algorithms are available for computational geometry problems. The popular Computational Geometry Algorithms Library (CGAL) is an open-source library of computational geometry problems that is written in C++, which also has bindings to support Java and Python [6]. CGAL uses sequential algorithms for solutions to common computational geometry problems. The sequential solutions rely on one computing node that limits the amount of data that can be processed. The limit of hardware power makes it impossible to achieve the needed real-world performance expectations. Thus, parallel computations tools are the best choice for computational geometry solutions. It is necessary to sturdy parallel computing approaches to algorithms solving computational geometry problems to identify new ways of gaining efficiency and speed. Another popular library that provides various algorithms for solving computational geometry problems is Boost.Geometry, a part of a collection of Boost C++ Libraries [7]. Boost.Geometry is a purely C++ library that has parallel computing capabilities. The library provides a C++ interface to multi-core CPU and GPU computing based on Open Computing Language (OpenCL) [8]. Yet, Boost library is bound to only C++ and provides a limited number of algorithms. This section discusses the research in the area of parallelized solutions to computational geometry problems and the parallelization of computational geometry algorithms with an agent-based approach.

2.1 Parallelization approaches to computational geometric algorithms

Computational geometry focuses on geometric problems that solve complex problems using large data sets. Multiple different parallel computation models have been developed to handle the

processing of large amounts of data and complex computations. The majority of the available parallel algorithms are based on shared memory or distributed memory models. The variety of tools available to implement parallel solutions to computational geometry problems starting with built-in multi-threading into programming languages and finishing with big data distributed computing platforms. In this sub-section, we will not focus on specific computational geometry problems rather on the approaches used in existing parallel solutions for computational geometry problems and the results.

In [9], Batista et al. discuss a few parallel geometric algorithms based on the CGAL library, using a shared-memory model [6]. The shared memory approach in this research includes multi-core CPU for multi-thread programming and OpenMP for thread control. OpenMP provides *#pragma omp task* to construct a task, *#pragma omp taskwait* to synchronize concurrent tasks execution, and *#pragma omp parallel* to process the tasks with user-specified threads [9][10]. The experiment results of this research showed the two times performance increase with parallel algorithms built using a shared-memory model over the sequential algorithms. In [11], Gonzalez-Escribano et al. present parallel algorithms for solving the same computational geometry problem based on shared and distributed memory models. The research uses three parallel algorithms: divide and conquer, sweep line, and Clarkson et al.'s to conduct performance measurements. The performance results showed that Clarkson et al.'s has a speedup of 14.75x for 32 processors with disc input over the sequential version. Divide and conquer and sweep line algorithms show a similar speedup of ~7x. In the same research, the performance of the same algorithms is also measured on the distributed memory environment to compare both environments against each other. The results show Clarkson et al.'s algorithm has a speedup of 15.16x for 64 processors with the disk input. The shared memory showed a better result in the performance of the divide and conquer and sweep line algorithms than distributed memory. The sorting algorithms perform poorly in distributed memory environment because of the need to physically send data to each computing node.

The first part of this subsection focuses on CPU scalability or speed-up of execution, the rest of the subsection focuses on memory/disk scalability or the capability of handling big data. These platforms provide an advantage of using distributed computing environment with clusters of any number of multi-core computing nodes. In [12], Puri et al. present the parallel algorithm

implemented with MapReduce and benchmarked against the sequential version. The experiment results of this research show that parallel algorithms implemented with MapReduce in the distributed environment have a speedup of 22x over its sequential version. This speedup is achieved by maximizing local processing and minimizing the communication overhead with shuffle and sort operations. In [13], Eldaway et al. present CG_Hadoop, which is a suite of scalable and efficient MapReduce algorithms for a vast variety of fundamental computational geometry problems. The experiment results of this research show that CG_Hadoop achieved up to 29x and 260x speedup than traditional algorithms when using Apache Hadoop and SpatialHadoop systems respectively [14]. Pradnyana et al. explored Apache Spark to parallelize implementation for the Voronoi diagram [15]. The results of the research showed 60% faster over sequential version and 17 % increase in input can be processed.

The distributed computing big data platforms are widely used to model and implement parallel algorithms for solving computational geometry problems. These platforms show considerably fast performance results compared to the sequential versions. Yet, bid data processing platforms such as MapReduce and Spark do not fit well for solving computational geometry problems. The input data for computational geometry problems mainly structured large data sets that do not need continuous modification throughout the whole computation time. Additionally, the MapReduce computation model involves reading and writing from disk which is time-consuming operations. Although Spark applications execute faster than MapReduce, the Spark computation model spends time on regular in-memory caching and data shuffling to redistribute or repartition data.

2.2 Agent-based parallelization of in-memory data analysis

Agent-Based Modelling (ABM) system is used for simulating behaviors of autonomous agents. The advantage of ABM over traditional techniques is the capability of analyzing behavioral patterns or interaction connections. There is a noticeable increase in the usage of ABMs for developing parallel computational solutions in a variety of domains such as computational science, economics, ecology, biology, and social simulations. Repast HPC and FLAME are common ABM platforms for repetitive analyses of structured datasets in memory [16].

Multi-Agent Simulation Suite (MASS) is an ABM parallel-computing library that focuses on multi-entity interactions [5]. The agents in MASS created in a cluster of multi-node computing nodes and over the distributed array. The MASS library has been applied as a primary tool for implementing agent-based parallel solutions in a variety of applications. In [17], Kipps et al. applied MASS to develop a parallel graph algorithm for biological network motif search, which is a uniquely recurring connected subgraph pattern. ABM fits well in a scientific area since it does not require deep parallel programming knowledge. The MASS library provides the ability to analyze a vast variety of scientific structured data. Woodring et al. applied ABM with MASS to develop an ABM parallel solution to analyze large climate data sets in a form of multi-dimensional structured data NetCDF [18]. The experiments of this research showed that MASS has several advantages over the traditional tools used for parallel scientific computations: (1) traditional programming and not a new parallel programming paradigm, (2) agent migration over an adjacency matrix structure that represents a graph, (3) agent migration mimics scientist familiar pattern of top to bottom dataset skimming, (4) support for runtime analysis, (5) interpretive and native executions in a single package.

Recursive Porous Agent Simulation Toolkit (Repast) is an open-source ABM and simulation platform that provides scalable agent simulations [19]. Repast suite is available in Java version as Repast Symphony and C++ version as Repast High Performance Computing (RepastHPC). RepastHPC developed similarly to Repast Symphony but was designed to support top high-performance computers (supercomputers). RepastHPC creates a network or continuous space projection and maps it over MPI rank while maintaining the simulation environment [16]. It is potentially possible to distribute the dataset as a projection letting agent navigation for data discovery with RepastHPC. Yet, the four difficulties within RepastHPC make it less usable for data sciences: (1) inability to stop computation under a certain condition instead of specific simulation task; (2) inability to perform parallel I/O operations instead of sequential I/O by the master node; (3) inability to modify projection during simulation; (4) absence of visualization or state-checking tools for tracking agent movements or projection state [16][20].

Flexible Agent Modeling Environment (FLAME) is an ABM platform developed to allow the agent and non-agent models to be in the same simulation environment [21]. FLAME users develop

an agent-based application in distributed computing environment consisting of a cluster of computing nodes. The platform guarantees automatic parallelization of user applications if users follow the programming guidelines within the platform. FLAME was applied by Eurance that is a massively parallel agent-based simulation of the European economy. This experiment showed that FLAME can create up to 500,000 agents [22][23]. Although FLAME allows high-performance ABM, Shih et al. observed two difficulties for non-computing specialists, while benchmarking FLAME: (1) tolerating limited flexibility in programming and understanding parallel-computing concepts [22]. The design choice used for implementing agent in FLAME is not well suited for computational geometry that is mainly involved big data analysis. Agents in the FLAME platform are not able to migrate between processes and unable to send direct messages to each other [22][23]. The other drawback is that each agent maintains the entire dataset by the applications can easily crash because of out-of-memory issues. On the contrary, agents in the MASS library migrate between processes, send direct messages, and are not required to carry the entire dataset.

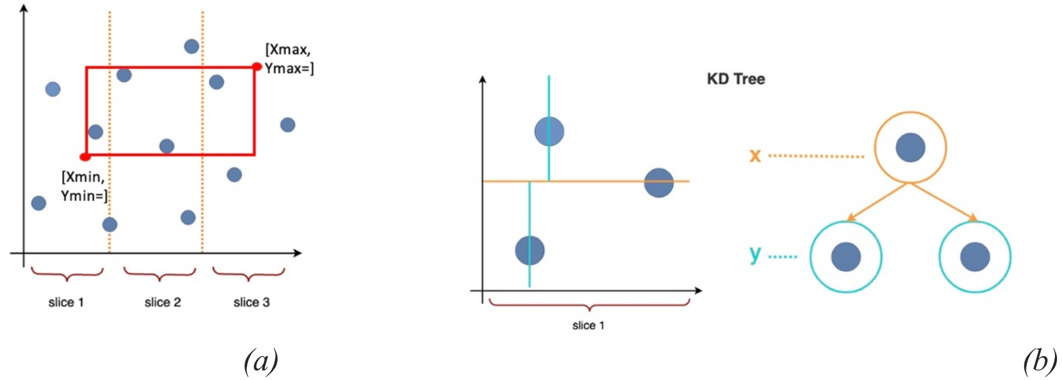
This research explores the applicability of ABM using MASS to modeling parallel algorithms to computational geometry problems. AMB may have a more intuitive approach to designing parallel algorithms for analyzing structured datasets. Additionally, ABM solutions may perform well in structure data analysis in terms of programmability and execution performance. The majority of computational geometry problems ask for dataset analysis that operated on the same dataset throughout multiple computation cycles. The MASS library allows to construct and maintain the original dataset throughout the entire computation time, which spares memory and execution performance. We will model four agent-based algorithms to solve four computational geometry problems to identify the strengths and fragilities of using ABM and MASS library.

Chapter 3
PARALLEL ALGORITHMS DEVELOPEMENT

We designed and implemented four agent-based parallel algorithms with the MASS library to solve four computational geometry problems and compared them against MapReduce and Spark implementations. We selected two query-searching and two statistic problems: Range Search, Point Location, Largest Empty Circle, and Euclidean Shortest Path. The chosen query and statistic problems are conventionally solved by divide and conquer parallel algorithms implemented by tools that fit best for data-streaming analysis tasks. Yet, the considered computational geometry problems use repeatedly the same structured input data set to analyze or compute the result. To solve these types of problems the traditional parallelization tools such as MapReduce or Spark may not be the best fit. The motivation for the chosen problems was to utilize a better suited parallelization approach to developing parallel algorithms for structured data analysis. We used the MASS library to develop four parallel algorithms that utilize agent-based data discovery mechanism over the same distributed data structure. The solution for the two query problems: Range Search and Point Location and two statistic problems: Largest Empty Circle and Euclidean Shortest Path, we designed four intuitive algorithms that use agent migration and propagation, distributed data structure, and distributed in-memory results to discover data attributes and compute the results. This section presents the overviews of the selected computational geometry problems, the designed parallel algorithms, and the additional implementation specifics.

3.1 Range Search

Definition. Given a set of N points in the plane, determine which points reside within a query rectangle (range). The query range includes four values: x- and y-minimum, x- and y-maximum coordinates in a 2D plane (see Figure 3.1a). The range searching problem is one of the fundamental problems in computational geometry [24]. Range searching has applications for range queries in databases, geolocation, data analysis, and statistics. Range searching is also often a subroutine step in the algorithms for solving more complex geometric problems.



(a) *(b)*

Figure 3. 1 Range Search and KD tree construction

The baseline of the range searching algorithms is the construction of a multidimensional binary tree (KD tree) [25]. KD tree for two-dimensional points is a modified binary search tree (BST), which alternates *x*- and *y*-axis as a key for inserting elements (see figure 3.1b). The alternating sequence starts with the *x*-axis. The construction of the KD tree consists of recursively partitioning the plane into two halfplanes, where the point positioned at the bisection line is the next point to be inserted into the tree with respect to *x*- and *y*-dimensions. Each bisection line is determined after sorting the points by *x*- or *y*-axis depending on the next dimension of the KD tree level. The bisection line is determined by dividing the number of points by two. Because of using sorting, bisection lines, as well as *x*- and *y*-dimension alternations, the KD tree is built as a balanced BST.

3.1.1 MASS range search implementation

The MASS implementation utilizes GraphPlaces and Agents classes. The GraphPlaces is used to construct a KD tree from the input data points. The graph structure remains unchanged in distributed memory throughout the computation. We spawn and migrate agents to discover the points that reside within the given range. Each agent carries its discovered points until no more points left to analyze. After traversing the KD tree each agent returns its list of discovered points.

Table 3. 1 MASS range search algorithm

Algorithm 1a: MASS Range Search

1. Initialize MASS
2. Read input points from a file
3. Sort points by x coordinate

4. Build KD Tree utilizing GraphPlaces, which is distributed among computing nodes in the cluster
5. Create an Agent at the root vertice
6. For each subtree where agents are created:

   While (root != null) do

   - Parent Agent checks if its current vertex's point (x, y) is in the query range. If yes, save point into agent's list of discovered points.
   - Parent Agent checks if its left and right children are not null. If both, children are present, Parent Agent migrates to the left node and spawns a child Agent for the right node. If only one child node exists Parent Agent migrate there without spawning any additional children
   - If nor more vertices to visit agent return a result and terminate

   end while

7. At the main program:

   While (Agents > 0) do

   - Collect all points sent by agents

   end while

Figure 3.2 provides an example of range search with agent-based parallel algorithm using GraphPlaces to construct KD tree and Agent to perform the search.

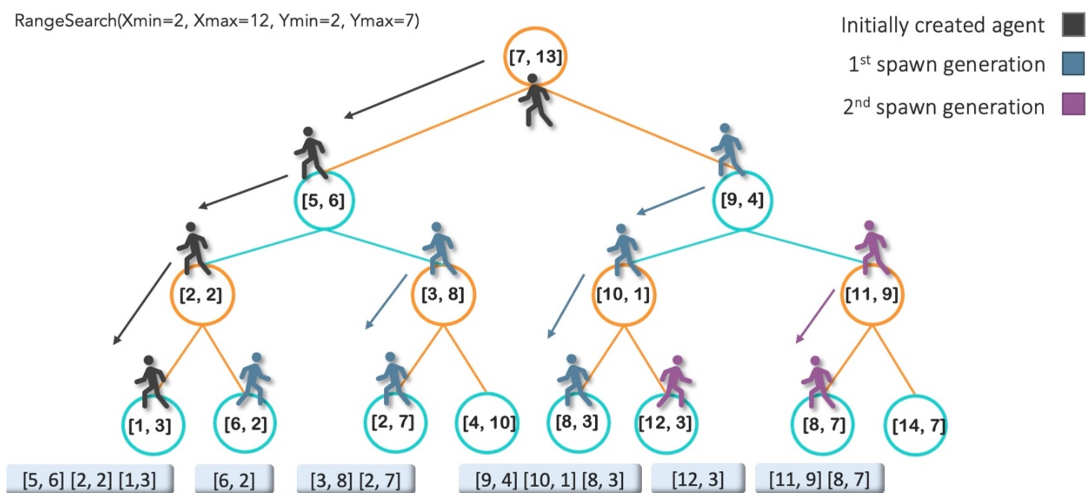


Figure 3. 2 Range Search computation with agent-based algorithm

3.1.2 MapReduce range search implementation

MapReduce implementation uses two sets of MapReduce jobs each having specific pair of mapper and reducer classes. After the input data points are read from the original data set, the points are sorted by *x* coordinate and partitions into small slices. The first pair of mapper and reducer map partitions to specific reducers, where each reducer builds a local KD tree from the points in the partition, then perform a search to determine the point within the given range. Finally, reducers output the search result to an intermediate file. The second mapper and reducer pair process the local results: the mapper reads the intermediate file, maps all the result points to the reducer, where the reducer collects all the result points to the final result output list.

Table 3. 2 MapReduce and Spark range search algorithm

Algorithm 1b: MapReduce and Spark Range Search

1. Read input points from a file
2. Partition point into small slices
3. Sort the points by x value
4. Partition points into N-number of slices
5. For each slice do
    - Sort points by x coordinate
    - Remove duplicate points if any
    - Build KD tree
    - Perform range search on KD tree by traversing the tree with respect to x and y dimensions and determining the points within the query range
    - Store the discovered points in a local temporary list
    - Return discovered points from the result list

   end for each
6. Collect all search results from each partition into the final output file

3.1.3 Spark range search implementation

Spark implementation utilizes the same parallelized algorithms as MapReduce implementation. However, Spark implementations differ by their core paradigm. Spark utilizes a set of transformations and actions on the input Resilient Distributed Datasets (RDDs) instead of map() and reduce(). Figure 3.1 shows the algorithm for Spark implementation.

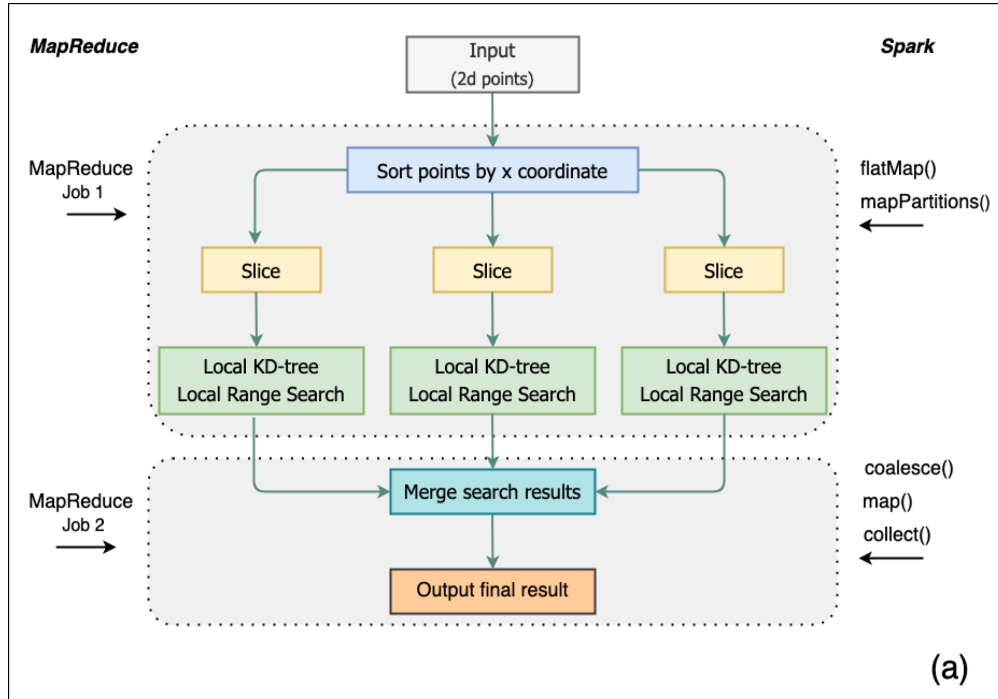


Figure 3. 3 MapReduce and Spark range search algorithm

3.2 Point Location

Definition. The most commonly used application of point location [26] is a location query. Given a map and a query point specified by its coordinates find the region of the map containing a query point. A map is nothing more than a subdivision of the plane into regions, a planar subdivision. A trapezoidal map is usually used to create a planar subdivision [2]. To create a trapezoidal map a vertical line is drawn from each point upward and downward.

3.2.1 MASS point location implementation

The MASS point location implementation uses Places and Agents classes. We also use event programming with the annotations feature of the MASS library to eliminate the need for manually synchronizing agents. The input for the MASS point location implementation is a trapezoidal map that comes in a form of coordinate points for each trapezoid in the trapezoidal map and neighbors list for each trapezoid. All trapezoids have a unique identifier (index). We read the input trapezoids and map them into the Places array after each Place assigned its trapezoid.

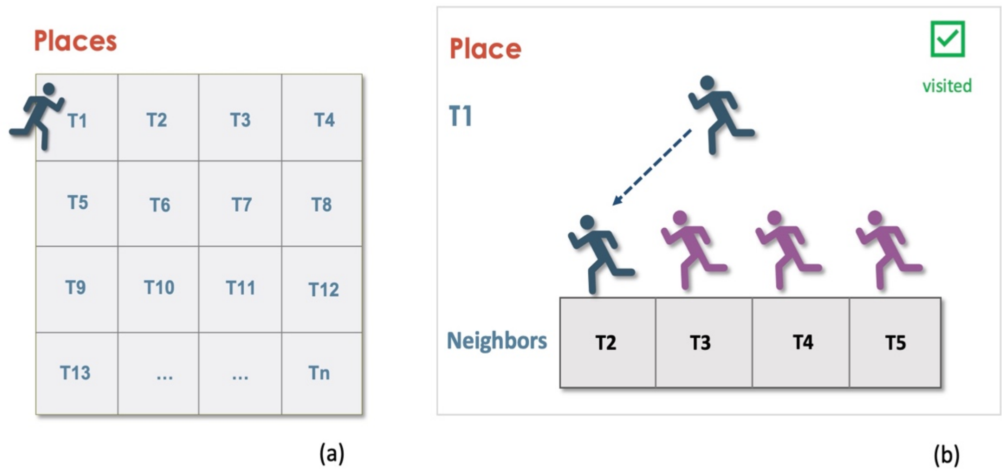


Figure 3. 4 Agent propagation and migration in MASS Point location

We start by creating one agent at the first Place in the Places array (Figure 3.4 a). As the agent arrives at the Place it checks if the trapezoid corresponding to the current Place locates the query point and if yes then the agent sends a message to all other agents to terminate and writes the result to the current Place (Figure 3.2 b). If the agent does not locate the query point at the current Place, then the agent will migrate to the unvisited trapezoid if one exists. An agent migrates to one of the unvisited trapezoids after it spawns additional agents for other unvisited trapezoids specified in the neighbors list of the trapezoid at the current Place. We continue spawning and migration of the agents until the trapezoid locating the query point is found or if such trapezoid does not exist then until completing the visit of each Places. Table 3.3 presents the agent-based parallel algorithm to solve the point location problem.

Table 3. 3 MASS point location algorithm

Algorithm 2a: MASS Point Location

1. Initialize MASS
2. Read input file with trapezoids
3. Create Places as 1D array with length equal to number of given trapezoids
4. Map each trapezoid to a specific Place
5. Create Agent at the first Place (first cell in the array)
6. While (number of Agents != 0) do

If (Place is not visited) {
- Agent sets place as visited
- Agent checks if the trapezoid at current place locates the query point.

If yes, the Agent broadcasts message to all other agents that it found the trapezoid locating the query point. After the agent writes the result at the place where it is located the result then the agent terminates. The agents that receive the message terminate as well.

If not, Agent checks if the current trapezoid has neighbors. If trapezoid has more than one neighbor, current Agent will migrate to one of these neighbors and for the rest it spawns child agents.

end if

end while

7. Call all Places for the result which received into Object[]. If the query point is not located within the given trapezoids then the results in Object[] are equal to null. Otherwise, one element in the Object[] contains the result, which is the trapezoid locating the query point.

3.2.2 MapReduce point location implementation

The MapReduce implementation uses two mapper and reducer classes to compute the point location. The first MapReduce job reads input trapezoids representing the trapezoidal map and splits them into smaller slices then maps slices to the corresponding reducers where each reducer performs the point location in its partition of trapezoids. The intermediate result from the first MapReduce job is collected to the temporary file. The second MapReduce job reads the output of the first MapReduce job and if the result for the point location query was found collects to the output file. The algorithm for the MapReduce implementation is a simple brute force divide and conquer approach.

Table 3. 4 MapReduce point location algorithm

Algorithm 2b: MapReduce Point Location

1. Read input trapezoids from input file
2. Execute first Mapper to partition all the trapezoids into small chunks.
3. Execute first Reducer to find the query point in the given partition.

For each partition do

- Search the query point within the trapezoids of this partition. If the query point is found return the result into temporary output file.

end for each

4. Execute second MapReduce job to map the result from intermediate files and collect the final result into the output file.

3.2.3 Spark point location implementation

Spark implementation follows a similar approach as MapReduce implementation with the difference of using transformations and actions on the input RDD instead of map() and reduce(). The Spark algorithm is identical to the algorithm in Table 3.4 except we do not use any intermediate file I/O operations since we use immutable RDDs. The advantage of Spark implementation over MapReduce implementation is that operations are performed in memory, which decreases the run time of the program. We create a different RDD and perform different transformation or action at each step in the algorithm presented in Table 3.4 to compute the result:

1. Read the input trapezoids into JavaRDD<Trapezoids>, where Trapezoid is the object to represent a single trapezoid in a trapezoidal map.
2. The *repartition()* transformation partitions the trapezoids into equal chunks among available computing cores.
3. The *mapPartitions()* transformation creates a new JavaRDD<Trapezoid> and applied to each partition to locate query point inside the trapezoids in a particular partition.
4. The *coalesce()* transformation is used to avoid data shuffling of the existing partitions and to reduce the number of partitions into one, a new JavaRDD<Trapezoid>.
5. The action collect returns the entire RDD containing the result to the driver program.

3.3 Largest Empty Circle

Definition. The largest empty circle problem states given a set of *S* site points, determine the largest empty circle whose interior does not overlap with any other obstacle [27]. It has been proven that the center of the largest empty circle must lay on the Voronoi vertex.

3.3.1 MASS largest empty circle implementation

The MASS implementation uses the Voronoi diagram properties as a baseline [28]. We use Voronoi diagram sites and vertices as input to the program. Figure 3.3 presents the important prove used in the computation. It has been proven that the center of the largest empty circle must lie on the Voronoi vertex [1][2].

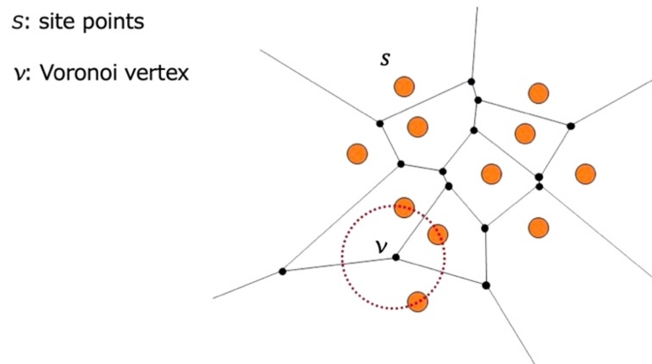


Figure 3. 5 The computation of the largest empty circle utilizing Voronoi diagram properties.

We read input sites and vertices presented and 2D points. The implementation uses SpacePlaces and SpaceAgents classes to compute the result. We create SpacePlaces and map the input points into corresponding SpacePlace. We use SpaceAgents to compute the farthest pair among sites and vertices. As the farthest pair of sites and vertices determined we compute the farthest (site, vertex) pair which makes the largest empty circle. Figure 3.4 illustrates the computation of the largest empty circle using SpaceAgents and the farthest pair of point computation.

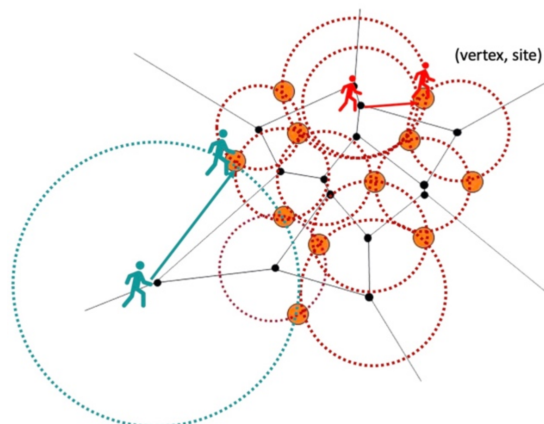


Figure 3. 6 MASS agent-based largest empty circle computation.

The farthest pair of points is computed using the modified implementation of the closet pair of points programs implemented by the previous graduate student at Distributed Systems laboratory. The farthest pair of points as the closest pair of points uses Von-Neumann patterns and Moore migration [29][30].

Table 3. 5 MASS largest empty circle algorithm

Algorithm 3a: MASS Largest Empty Circle

1. Initialize MASS
2. Read input files (sites and vertices)
3. Create SpacePlaces and initialize with points (sites and vertices)
4. Create Agents at each place

   While (farthest pair is not found) do
   - Agents.callAll(propagate agents)
   - Agents.callAll(migrate)
   - Agents.manageAll()
   - Terminate parent Agents from where child agents spawned
   - Loop through returned results and compute farthest pair with maximum distance

   end while
5. Determine the farthest pair among sites
6. Determine the farthest pair among vertices
7. Return the farthest pair that has maximum distance between vertices and sites

3.3.2 MapReduce largest empty circle implementation

The MapReduce implementation uses four sets of MapReduce jobs to compute the largest empty circle. The first two MapReduce jobs are used to read and map the input sites and vertices then compute the convex hull of sites and vertices. We use the points on the convex hull to compute the farthest pair of points. The last mapper and reducer classes are chained together to compute the farthest pair of points and sites and vertices. The last MapReduce job uses the farthest pair of (site, site) and (vertex, vertex) to compute the largest empty circle, which is the farthest pair of (vertex, site). Table 3.6 presents the divide and conquer algorithms used for MapReduce implementation.

Table 3. 6 MapReduce largest empty circle algorithm

Algorithm 3b: MapReduce Largest Empty Circle
1. Read input points from file 2. Execute two MapReduce jobs to compute convex hull of sites and vertices and collect output to intermediate file 3. Execute MapReduce job to compute farthest pair of point for sites and vertices and collect output to intermediate file 4. Execute MapReduce job to compute farthest pair (vertice, site) 5. Collect result to output file.

3.3.3 Spark largest empty circle implementation

Spark implementation follows the same approach as MapReduce with the difference of using transformations and actions on the input RDD. The Spark algorithm is identical to the MapReduce algorithm presented in Table 3.6 except we do not use intermediate files to store intermediate results. Each step in the algorithms in Table 3.6 creates a new RDD to which we apply certain transformation or action:

1. The input points representing sites and vertices are read into JavaRDD<String> for further input processing.
2. The *flatMap()* transformation creates a new JavaRDD<Point> presenting input point.
3. The *repartition()* transformation repartitions the points into equal partitions among available computing cores.
4. The *mapPartitions()* transformation creates a new JavaRDD<Point> and applied to each partition to compute the convex hull and farthest pair of points.
5. The *coalesce()* transformation is used to avoid data shuffling of the existing partitions and to reduce the number of partitions into one, a new JavaRDD<Point>.
6. The action *collect*() returns the entire RDD containing the result to the driver program.

3.4 Euclidian Shortest Path

Definition. Given a set of obstacles in Euclidean space and two points, find the shortest path between points that does not intersect any of the obstacles [31].

3.4.1 MASS Euclidian shortest path implementation

The MASS implementation uses Places and Agents. We also use event-driven programming with annotations to eliminate the need for manual synchronizations of Places and Agents [32]. The input trapezoidal map data is read and used to build the road map (a graph) that is used to compute the shortest path [2]. The nodes in a road map are mapped to the Places array. We assign a specific node to each Place as well as store the information about available edges from the node.

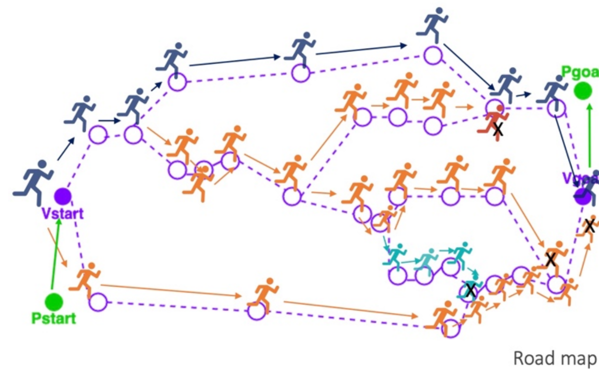


Figure 3. 7 Euclidean shortest path search with Places and Agents

The search starts by creating an agent at the starting node then spawn and migrate until the Euclidean shortest path is found (see Figure 3.7). Each agent carries its walked path from the starting node as a list of visited nodes. Each Place has the distance of the last visited agent that traveled from the starting node, as the new agent arrives at the Place it checks whether the Place is already visited. If the Place visited and the distance of the previous visiting agent was shorter than the distance of the current agent, the current visiting agent terminates. The intermediate distance check and agent termination eliminate the unnecessary pathfinding for agents that already known to have a longer path compared to the current known shortest path. If the current Place node has more than one available edge, the visiting agent spawns additional agents and before migrating to one of the edges provides the spawned agents with its traveled path information.

Finally, the result (the shortest path) is written to the Place that is assigned the goal node, so no additional Places traversal needed to get the final result. Table 3.7 present the agent-based parallel algorithm to compute Euclidean shortest path.

Table 3. 7 MASS Euclidean shortest path algorithm

Algorithm 4a: MASS Euclidean Shortest Path

```
1. Initialize MASS
2. Read input file with trapezoids
3. Create road map (presented by list of nodes)
4. Create Places as 1D array with length equal to number of given roam map nodes
5. Map each node to a specific Place
6. Create Agent at the first Place (first cell in the array)
7. While (number of Agents != 0) do
       If (Place was visited)
         - Agent compares its distance with the distance of the previous visiting agent.
           If this agent has longer distance, then agent terminates
       end if

       Agent sets current Place as visited
       Agent updates its distance then the writes it at the current Place

       If current place has the node that is the goal
           - Agent writes the path to the place and terminates.
       else
           If the node has only one edge
               - Agent migrates to the node on the other side of the edge
           else
               - Agent spawns an additional agent for the other edges
               - Agent migrates to the node following one of the edges
           end if
       end if

   end while
```

3.4.2 MapReduce Euclidian shortest path implementation

MapReduce implementation uses three sets of MapReduce jobs to implement a divide and conquer solution to the Largest Empty Circle problem. The first set of mapper and reducer is used to map nodes of the road map in small partitions to reducers, whereas reducers compute all local paths in their partition and collect the paths to intermedial file. The second set of mapper and reducer is used to map all local paths to reducers, whereas reducers construct the whole path from starting point to the goal point then output the shortest path in their partition to the intermediate file. Finally, the third set of mapper and reducer map all intermediate shortest paths determined in the previous MapReduce job then determine the resulting shortest path. Table 3.8 presents the parallel algorithm to solve the Euclidean shortest path problem with MapReduce.

Table 3. 8 MapReduce Euclidean shortest path algorithm

Algorithm 4b: MapReduce Euclidean Shortest Path

1. Execute MapReduce job #1:
   Mapper: Read input. Partition road map nodes into small slice and send to reducer
   Reducer: Compute local paths within its slice. Collect local paths into output.

2. Execute MapReduce job #2:
   Mapper: Read intermediate file produced by MR job 1. Partition local paths between reducers. Each local path that starts & ends by nodes within specific range sent to the according reducer.
   Reducer: Construct whole paths from original start to goal and find the shortest within its found paths. Collect search result into the output.

3. Execute MapReduce job #3:
   Mapper: Read input and sent it to the final reducer.
   Reducer: Determine the shortest path and collect result into the final output file.

3.4.3 Spark Euclidean shortest path implementation

The Spark implementation follows a similar approach as the MapReduce implementation. The Spark algorithm is identical to the MapReduce algorithm presented in Table 3.8 except we use RDDs, transformations, and actions instead of map() and reduce(). Figure 3.5 shows the flow of the Spark program.

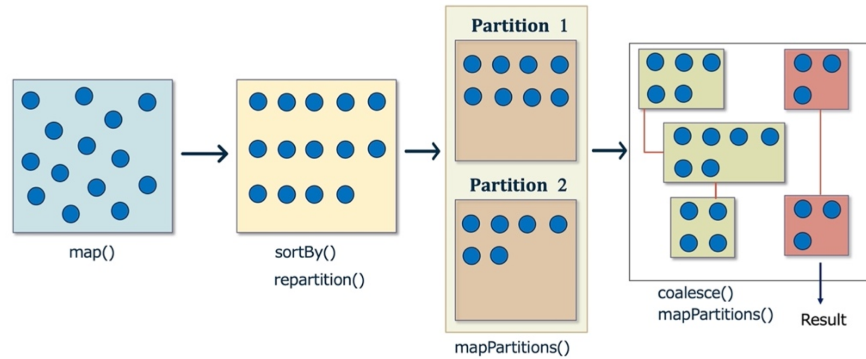


Figure 3. 8 Spark Euclidean shortest path algorithm

The difference between MapReduce and Spark implementations is that operations are performed in memory, which decreases the run time of the program. In the Spark implementation of the algorithm in Table 3.8 we create a different RDD of each step and apply curtain transformation of action:

1. The input file is read into JavaRDD<String> after which *map()* transformation is applied to map each line to JavaRDD<Node>, where Node is an object representing a node in the road map graph for the shortest path computation.
2. The nodes are sorted by transformation *sortBy()* and stored in a new JavaRDD<Node>.
3. The *repartition()* transformation repartitions the nodes into equal partitions among available computing cores.
4. The *mapPartitions()* transformation creates a new JavaRDD<Node> and applied to each partition to compute the sub-paths.
5. The *coalesce()* transformation is used to avoid data shuffling of the existing partitions and to reduce the number of partitions into one JavaRDD<Node>, which contains all the sub-paths computed at the previous transformation.
6. Another *mapPartitions()* transformation creates a new JavaRDD<Node> and applied to each partition containing a slice of sub-paths to construct the whole paths and compute the shortest path.
7. The *collect*() action returns the entire RDD containing the result to the driver program.

Chapter 4
PARALLELIZATION

4.1 MASS

Multi-Agent Spatial Simulation (MASS) library consists of two key components Places and Agents. Figure 4.1 illustrates the underlying architecture of MASS library execution [5]. Places represented as distributed matrix allocated over a cluster of multi-core computing nodes. Places instance is partitioned into small vertical stripes over the cluster. The stripes are allocated and executed by a different thread. Agents are organized into bags that are allocated to different processes. Agents are allocated to the process of the Place they are bounded. The base Agent class can be derived by a user to create application-specific agents. Agents can migrate to other places within the matrix, create their copies, control their activity. As MASS initialization starts, each computing node starts a process that spawns threads equal to the local CPU cores. The message passing technique is used to manage the multi-threaded processes.

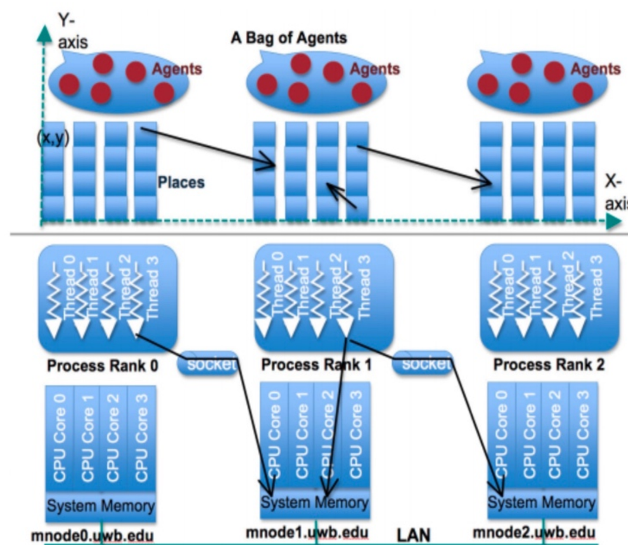


Figure 4. 1 MASS execution model

The simulation for typical MASS application performed by executing parallel function call to places and agents with *callAll(functionId)*; interchange data among places and agents with

exchangeAll(); and agent migration, propagation, and termination with *manageAll()*. The MASS library guarantees a barrier synchronization of agents with the execution of *callAll()*, *manageAll()*, and *doAll()*, however user manually implements this synchronization in the driver program. Event-driven programming with annotation feature eliminates the need to manually specify synchronization by implementing automatic invocation of synchronization functions. Three types of events annotated with *@OnCreate*, *@OnArrival*, and *@OnDeparture* are agent-associated functions. The functions *doWhile(lambda)* and *doUntil(lambda)* initiate the events and commit the annotated functions.

The implementations of parallel agent-based algorithms for four computational geometry problems (range search, point location, largest empty circle, and Euclidean shortest path) used different features of the MASS library.

1. *Range Search* implementation uses GraphPlaces to build a multidimensional (KD) tree and Agents to traverse the tree. GraphPlaces is recently added to the MASS library to provide a more intuitive way for graph creation. Another approach to create a graph is to use regular Places but it requires more lines of code and less intuitive implementation. VertexPlace is a base class that represents a single element in GraphPlaces. KDTreeNode class extends VertexPlace to implement application-specific vertex (node) for KD tree. RangeSearchAgent class extends the base class Agent to specify the behavior of the agent that crawls through the tree to discover a point in the requested range. The major methods for RangeSearchAgent:

    a). *callMethod(int functionID, Object agrs)* invoked by *callAll()* in driver program to invoke a specified function to all agents.

    b). *migrateTo(Object arg)* invoked to direct agent to the next place.

    c). *atVertex(Object args)* invoked to search of the query point at each KDTreeNode by each active agent.

    d). *spwanChildren(Object args)* invoked to create an additional agent if the current KDTreeNode has more than one child node.

    The overall algorithm implementation is consisted within the driver program. We read the input file to create a KD tree and implement tree traversal with agents. The agent migration, propagation, and termination are done by utilizing the manual synchronization approach.

2. *Point Location* implementation uses Places, Agents, as well as event-driven programming feature to eliminate manual synchronization overhead. Point Location algorithm uses a trapezoidal map to search for the query point. In the driver program, we read the input file with trapezoids to initialize each place with a trapezoid. Cell class extends Place base class and Crawler class extends Agent base class to implement application-specific place and agent. Crawler class mainly consists of methods with annotations that will be invoked by *doWhile(agents::hasAgents)*.

    a). *initCrawler()* with *@OnCreation annotation* invoked upon agent any agent is created.

    b). *onArrival()* with *@OnArrival annotation* invoked as agent arrives at a new place. This method implements agent behavior for point location search.

    c). *receiveMessage()* with *@OnMessage annotation* invoked to accept any sent messages from any of the agents.

3. *Largest Empty Circle (LEC)* implementation uses continuous space - the newest addition to the MASS library. We use SpacePlaces and SpaceAgent, which are two major classes of continuous space. SpacePlaces is similar to Places but it represents data space in continuous space. SpacePlace is an individual element of SpacePlaces. PlaceSpaceAgent that is derived from Agent resides within SpacePlaces elements.

    a). *computeLEC(String inFile, Boolean flag)* invoked by driver program to compute the largest empty circle based on user-provided input files. The input file is passed to SpacePlaces constructor to create places. Each place initialized with a two-dimensional point.

    The baseline for the LEC algorithm is the use of Voronoi diagram vertices and sites. The driver program reads two input files, one containing vertices and the other sites. The majority of the algorithm is implemented in computeLEC function of LECComputation class.

4. Euclidean Shortest Path implementation similar to point location uses Places, Agents, and event-driven programming. The baseline for an algorithm to Euclidean shortest path solution is the creation of Places object and shortest path search by agents. Places object is used to create a road map, which is a directed graph used by agents to determine the shortest path from the start point to the goal point. RobotMotionCell class derived from Place base

class to implement algorithm-specific node in a graph. RobotCrawler class extends Agent base class to implement graph search algorithm.

a). initCrawler() with *@OnCreation annotation* invoked upon agent any agent is created.

b). *onArrival()* with *@OnArrival annotation* invoked as agent arrives at a new place. This method implements agent behavior for point location search.

The driver program reads the input file with trapezoid then creates nodes for the graph based on the trapezoidal map. The simulation starts by creating Places with a size equal to the number of nodes in a graph. We initialize each Place with Node object using callAll() function. The shortest path search is started by creating Agents then invoking *agents.doWhile(agents::hasAgents)*. The agent that finds the shortest path stores the result in the Place that holds the goal node.

4.2 MapReduce and Spark

Hadoop MapReduce framework allows to perform distributed and parallel processing of large amounts of data on a Hadoop cluster consisting of any number of servers. The data is stored on a high fault-tolerant distributed file system called Hadoop Distributed File System (HDSF). The distributed file system makes data available to all the computing nodes in the cluster. HDFS maintains reliability and fault tolerance by storing and replication of the inputs and outputs of each job. Hadoop MapReduce architecture based on Job Tracker-Task Tracker model corresponding to master-worker model. Job Tracker initiates the communication to Task Trackers by distributing the work or results. As the user submits a job Task Tracker breaks it into map and reduce tasks, assigns tasks to Task Trackers, monitors the progress, and informs the user about task completion. The cluster nodes can be Name Node, Secondary Name Node, or Data Node. The worker node is always Data Node and Task Tracker. Data Node holds and manages its local file system data. The master node can be both Name Node and Job Tracker, however, Name Node manages file system metadata and access control, and Job Tracker distributed tasks to the worker nodes. Secondary Name Node downloads periodic checkpoints from the Name Node to ensure fault tolerance.

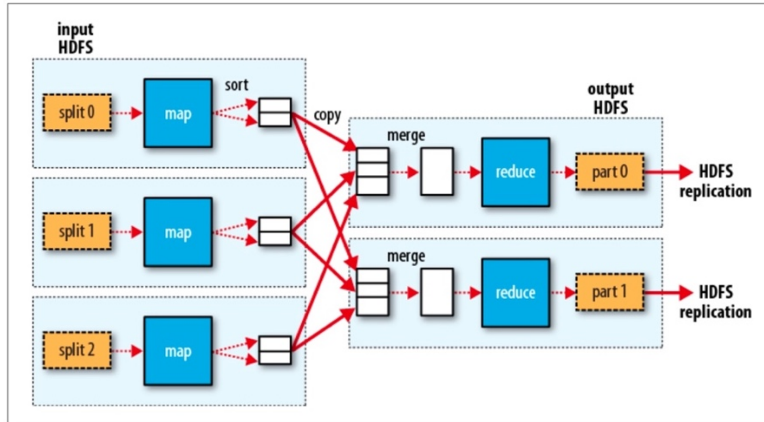


Figure 4. 2 MapReduce execution model

Figure 4.2 presents the MapReduce execution model [3]. Hadoop MapReduce distributes the task across multiple servers in a cluster. Hadoop MapReduce job consists of two phases: map phase and reduce phase. The input and output for each phase represented by <key, value> pairs. Map tasks executed in parallel on the partitioned input data blocks. The output pairs from map shuffled among different reducers based on keys. The reducers accept only the pairs that match their assigned key. Users specify the logic for map and reduce phases by creating application-specific mapper and reducer classes and implements map() and reduce() methods correspondingly.

1. *Range search* implementation consists of two MapReduce jobs. First MapReduce maps input points based on x coordinate to corresponding reducers while reducers build local multidimensional (KD) trees and perform local range search. The second MapReduce job reads the discovered range search points from the first MapReduce job output and collects to resulting output.
2. *Point location* implementation utilizes two MapReduce jobs. First MapReduce job maps input trapezoids to corresponding reducers while reducers perform a search for the trapezoid locating the query point. The second MapReduce job gets the output from the first MapReduce job and collects the search result to the final output.
3. *Largest empty circle* implementation uses two MapReduce jobs to compute the convex hull of sites and vertices. We then use the points of the convex hull to execute another two sets of MapReduce jobs that compute the farthest pair of points within convex hull site points

and convex hull vertex points. The final MapReduce job computes the farthest pair of points as (vertex, site) which forms the largest empty circle.

4. *Euclidean shortest path* implementation executes three MapReduce jobs. First MapReduce job maps the input point based on x-coordinate to corresponding reducers while reducers compute local paths. Second MapReduce job maps local paths, constructs the whole paths from starting node to goal node, and determines the shortest path within a partition. Third MapReduce job maps all discovered shortest paths from previous MapReduce job and performs the final filtering to determine the resulting shortest path.

Similar to Hadoop MapReduce, Apache Spark is a cluster computing platform utilized to process large amounts of data in parallel. However, data access and storage in MapReduce is disk-based while in Spark computations it is in-memory based, which speeds up Spark applications. Spark works with data by using a *resilient distributed dataset* (RDD) data model. RDD is an immutable distributed set of elements (objects). RDDs are split into partitions that are computed on different cluster computing nodes. Two types of operations can be performed on RDDs: *transformations* and *actions*. Transformations construct and return a new RDD from the previous one, for example when we use map() and filter(). Actions perform computation on RDD and then return the result to the driver program or storage, for example, count() and first(). Spark performs lazy evaluation, which means the computation occurs when transformed RDD is used in an action. Spark can run on a Hadoop cluster and can access any Hadoop data source, often used HDFS. Figure 4.2 illustrates the Spark execution model, as soon as the driver program creates RDDs from a dataset in parallel by accessing data storage then applies transformations and actions to get the result [33].

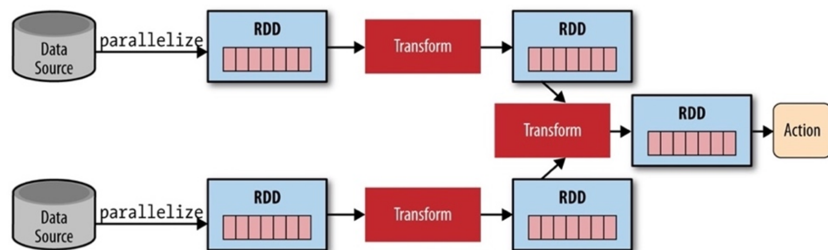


Figure 4. 3 Spark execution model

Hadoop MapReduce and Spark platforms implement a divide and conquer algorithms (D&C) for all four applications: range search, point location, largest empty circle, and Euclidean shortest path. However, the parallel implementation approaches are different in each platform. The D&C parallel algorithms for solving the four computational geometry problems implemented in MapReduce by creating application-specific Mapper and Reducer classes and running multiple MapReduce jobs. Each Mapper class implements map() function that specifies in which manner the job performs mapping. Similarly, each Reducer class implements reduce() function, which indicates the computation logic in the MapReduce job. The same parallel algorithms are implemented in Spark within the driver program with multiple RDDs, transformations, and actions.

1. *Range search* implementation uses several resilient distributed datasets (RDDs) and transformations to implement the same algorithm as the MapReduce version. We apply mapPartitions() to each partitioned data slice to build a local KD tree and perform a local range search. The transformation coalesce() reduces the search results from each partition into one. To get the result from the RDD into the driver program we use collect() action.
2. *Point location* implementation initially uses flatMap() to represent input trapezoids and repartition() to partition the trapezoids among available computing cores. To determine the trapezoid locating the query point, we apply mapPartitions() transformation. To get the result, we apply coalesce() transformation and collect() action.
3. *Largest empty circle* implementation as initial steps uses flatMap() to represent the original input dataset and repartition() to partition the point among available computing cores. We use mapPartition() to each input site point and vertex point partitions, which compute the convex hull of site points and vertex points then computing the farthest pair of points (site, site) and (vertex, vertex). To return the result to the driver program, we use collect() action. To determine the largest empty circle, we compute the farthest pair of vertex to site.
4. Euclidean shortest path implementation uses map() transformation to represent the input points then applies sortBy() transformation to sort points. The repartition() transformation partitions points among available computing nodes. To determine the shortest path we apply mapPartitions() transformation to each partition. The mapPartitions() transformation follows the same computation logic used in the second and third MapReduce jobs of the MapReduce version. To get the search result we use collect() action.

Chapter 5
ALGORITHMS PERFORMANCE ANALYSIS

The computational geometry algorithms implemented using three parallelization tools: MASS, MapReduce, and Spark. The implementations are evaluated by two metrics: execution performance and programmability. Programmability in its turn includes four evaluation metrics: boilerplate code, lines of code, number of classes, and cyclomatic complexity. This section discusses the evaluation metrics, development environment and tools, gathered results, and analysis of these results.

5.1 Evaluation metrics

We compared agent-based parallel algorithms implemented with MASS against the conventional divide and conquer parallel algorithms implemented with MapReduce and Spark. The evaluation of the implementations is measured by execution performance and programmability.

1. *Programmability*

   Programmability is measured by four different metrics: boilerplate code, lines of code, number of classes, and cyclomatic complexity. To measure the total number of lines of code (LOC) we used Statistic plugin installed on IntelliJ IDEA editor [34][35]. To measure LOC, we eliminated the commented lines, empty lines, include/import lines, and console/debugging output lines.

   - Boilerplate code – number of lines of code required to set up the environment
   - Lines of code (LOC) – total number of lines of code in the implementation
   - Number of classes – total number of classes in the implementation
   - Cyclomatic complexity - number of linearly independent paths through an algorithm.

   Cyclomatic complexity metric is used to estimate the overall complexity of the application, specific functionality within it, or specific algorithm. We use cyclomatic complexity metric to estimate the complexity of the implementations of algorithms within the developed

applications. To calculate the cyclomatic complexity of algorithms we construct the control flow graph based on algorithm implementation and apply the following formula:

$$\text{Cyclomatic complexity} = E - N + 2*P$$

The variables in the formula present the values calculated based on control flow graph nodes and edges: *E = number of edges, N = number of nodes,* and *P = number of nodes that have exit point*. Each node on the graph represents indivisible or commands within the algorithm. If an algorithm consists of zero decision point such as *If, For* then the complexity value is one because it contains single path in the algorithm. In addition, cyclomatic complexity metric can be used to determine the probability of errors within the program and path testing where number of test cases is equivalent to the cyclomatic complexity of the program.

2. *Execution performance*

   The metric measures the execution time of three different implementations: MASS, MapReduce, and Spark. Execution Performance of the applications will be measured by their run time on a cluster of computing nodes. We use different configurations of a cluster consisting up to 12 Hermes computing machines at University of Washington Bothell. All cluster Hermes machine have 15GB of RAM and either 4 or 8 processor cores. To be more concrete 4 computing machines have 8 CPU cores, and the other 8 machines have 4 CPU cores. The computing machines connected to 1 gigabit per second (Gbps) Ethernet. To determine the execution time, we run each application multiple times and take the average time.

5.2 Programmability

The programmability metric for the three parallel programming tools (see section 5.1) provides an insight into the ability of the tool for an intuitive and simple programming experience to the users. This subsection presents the programmability metric for the four computational geometry problems: Range Search, Point Location, Largest Empty Circle, and Euclidian Shortest Path.

5.2.1 Range Search

Table 5.1 presents the programmability metrics for the range search implementation utilizing MASS, MapReduce, and Spark.

Table 5. 1 Programmability metrics of Range Search

Parallel Framework	Boilerplate lines of code	LOC	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	25	367	6	3
Spark	4	270	3	3
MASS	3	298	4	4

The comparison of three different implementations of Range Search showed that MASS requires the fewest number of boilerplate code to set up the environment in contrast to MapReduce and Spark. This boilerplate code consists of initializing MASS, specifying nodes.xml, and shutting down MASS when the computation is finished. The total number of lines of MASS and Spark implementations are relatively compatible. Yet, due to the need for creating subclasses of Agent and Place classes to accommodate implementation needs, the total number of lines is slightly higher than Spark implementation. The fewest number of classes is required by Spark implementation, whereas MapReduce highest number of classes. The MASS metric for a total number of classes is comparable to Spark implementation.

Cyclomatic complexity is used to measure the complexity of the algorithms based on three parallelization tools - MASS, MapReduce, and Spark. Spark and MapReduce use the same divide and conquer algorithm to perform range search, and the cyclomatic complexity is equal to three for both. Agent-based algorithm with MASS uses places and agents, which increases the value of the cyclomatic complexity of the algorithm since agents should traverse the binary tree while the unvisited nodes still exist. The cyclomatic complexity metric showed that algorithms designed for MASS provide similar complexity in comparison to divide and conquer algorithms designed for MapReduce or Spark. We observe from programmability metrics that MASS is a better programming tool in terms of the required steps needed to set up the environment and Spark is a better programming tool in terms of the fewer LOC required for the implementation.

5.2.2 Point Location

Table 5.2 presents the programmability metrics for the point location implementation utilizing MASS, MapReduce, and Spark.

Table 5. 2 Programmability metrics of Point Location

Parallel Framework	Boilerplate lines of code	LOC	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	23	283	8	2
Spark	4	236	3	2
MASS	3	265	7	4

Spark implementation of the Point Location has the fewest number of lines of code and number of classes in comparison with MASS and MapReduce. Yet, MASS implementation has the best results in the number of boilerplate code lines metric. It also has a better result than MapReduce in terms of the number of classes needed to implement the algorithms. The cyclomatic complexity of MASS algorithms is higher in comparison to Spark and MapReduce algorithms due to using doWhile() loop which is needed to do agent simulations.

MASS implementation has the second-best result in terms of the total number of lines of code (LOC). This is due to the fact that the algorithm implementation utilizes more code. To initialize each Place with trapezoid, we extended the Places class and customized to the needs of the algorithms. We also have a class Trapezoid, which describes the Trapezoid object for each Place. For the agent to perform point location search we extended the Agent class to describe the behavior of each agent. Also, we have one more additional class that is used to pass arguments to each created (spawned) agent.

We significantly decreased the number of lines of code for this benchmark in comparison to using the traditional programming approach of the MASS library. This improvement is due to utilizing event-oriented programming using the annotations [32] feature of the MASS library. MASS library traditionally uses barrier synchronization of agent upon executing callAll(), manageAll(), or doAll(), which users repeatedly invoke from the main() function [32]. The event-programming

with annotations automatically invokes such functions when the corresponding event occurs. Thus, we eliminated the need of writing additional lines of code to manually synchronize the agent.

5.2.3 Largest Empty Circle

Table 5.3 presents the programmability metrics for the largest empty circle implementation utilizing MASS, MapReduce, and Spark.

Table 5. 3 Programmability metrics of Largest Empty Circle

Parallel Framework	Boilerplate lines of code	LOC	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	76	549	10	2
Spark	2	320	5	2
MASS	3	211	6	2

The programmability analysis for LEC programs in all three platforms MASS, Spark, and MapReduce showed that MapReduce implementation has the worst programmability results in comparison with the other two programs. As the metric indicates MapReduce implementation requires the highest number of boilerplate code, lines of code, and classes (see table 5.3). These high numbers are due to the need of setting up the environment and multiple MapReduce jobs. On the contrary, LEC Spark implementation requires the fewest number of lines of boilerplate code to set up the environment. Also, Spark implementation has the advantage of having the fewest number of classes. Spark benefits from its programming model that has resilient distributed datasets (RDDs), which overall shortens the required number of lines and number of classes. The decrease in the number of boilerplate code for LEC Spark implementation in comparison with PointLocation Spark implementation (subsection 5.2.2) is because we do not broadcast any variables to Spark context as it is the case for PointLocation Spark implementation where we have to broadcast two additional variables x and y coordinates of the query point.

Largest empty circle (LEC) MASS implementation outperforms Spark and MapReduce implementations in terms of a total number of lines of code (LOC). It also outperforms MapReduce by having the fewest number of boilerplate code lines and the number of classes. The good

programmability performance of LEC MASS is due to the embedded SpacePlace and SpaceAgent classes in the MASS library. The algorithm for LEC MASS does not override the SpacePlace and SpaceAgent classes, thus, simplifies the overall implementation and increases programmability performance. However, small changes are made in the SpaceAgent, which locates directly in the MASS library. These changes are needed for computing the farthest pair of points instead of computing the closest pair of points that is coded into the original implementation.

5.2.4 Euclidian Shortest Path

Table 5.4 presents the programmability metrics for the Euclidean shortest path implementation utilizing MASS, MapReduce, and Spark.

Table 5. 4 Programmability metrics of Euclidean Shortest Path

Parallel Framework	Boilerplate lines of code	LOC	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	46	914	13	3
Spark	3	633	7	3
MASS	3	673	10	4

The programmability analysis of the Euclidean shortest path implementations in three platforms MASS, Spark, and MapReduce showed that MASS algorithm implementation is the second-best in terms of the number of boilerplate lines of code, total lines of code, and the number of classes. The boilerplate code for MASS includes initializing MASS, specifying nodes.xml file, and shutting down MASS. The boilerplate code for Spark includes the creation of JavaSparkContext, setting up the parameters, and stopping spark context. The boilerplate code for MapReduce includes the set up of three MapReduce jobs where for each of the job we create JobConf, specify the input and output files paths, setup of mapper and reducer classes, setup of input and output formats, and set up of parameters. MASS and Spark have compatible boilerplate code metrics, which simplifies the setup of the platform and shortens the implementation time.

MR implementation has the highest LOC due to the multiple implemented mapper and reducer classes and a high number of boilerplate lines of code. Spark has the shortest implementation code (LOC) due to its programming paradigm of RDDs, transformations, and actions that simplify and

shorten the implementation. MASS implementation is lagging behind Spark implementation by LOC metric due to the need for custom agent and place classes. Yet, we utilized the event-driven programming feature of the MASS library that eliminated a significant amount of synchronization code that usually should be manually coded by the user [32].

The total number of classes of MapReduce implementation is the highest in comparison with Spark and MASS implementations. This increased value for MapReduce is due to the need for three sets of mapper and reducer classes to implement the divide and conquer solution. Whereas Spark implementation takes advantage of its programming model and allows to implement the same algorithm utilizing RDDS, transformations, and action in the same driver class. The MASS implementation utilizes agents and places to implement the agent-based algorithm. We extended the original Agent and Place classes of the MASS library and customized them to accommodate the needs of the algorithm. MASS implementation for Euclidean shortest path utilizes the PointLocation MASS implementation as a subroutine during shortest path computation. However, we embedded the point location computation into the Euclidean shortest path program which also increases the total number of classes.

Spark and MapReduce implementations use the same divide and conquer algorithm. Thus, the cyclomatic complexity metrics for Spark and MapReduce have the same value. The MASS implementation uses an agent-based parallel algorithm (section 3). To implement an agent-based algorithm we utilized event-driven programming with annotations that allows us to simplify the agents and places synchronization. However, this required us to use a while loop that increased the value for the cyclomatic complexity metric.

5.3 Execution performance

5.3.1 Range Search

We conducted execution performance tests for range search using 500,000 input points with multiple clusters having a different number of computing nodes. Figure 5.1 presents the execution performance results.

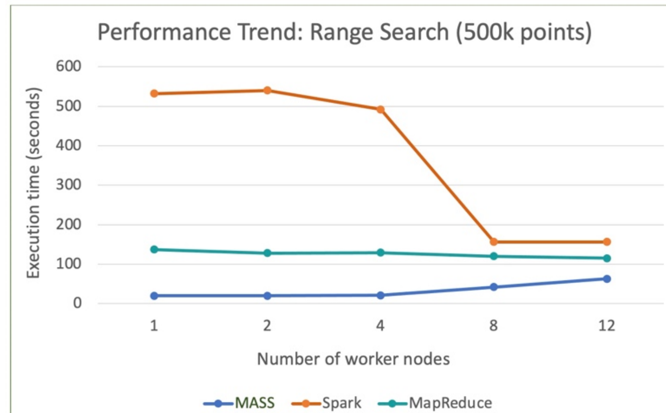


Figure 5. 1 Execution performance for Range Search implementation

Figure 5.1 shows that MASS implementation produces better execution performance results in comparison with MapReduce and Spark applications. According to the performance results gathered during multiple tests, the execution time of MASS implementation increases by increasing the number of cluster nodes greater than four. This inefficiency in MASS computing node scalability is suspected to be due to the fact that to create and initialize the utilized data structure in this MASS benchmark program we read the entire input file on each computing node while each computing node needs only its corresponding portion of the data in the file. We used a relatively large input file (500k point) and the read operation is time-consuming especially when each computing node reads the entire file. The possible solution to overcome the file read operation bottleneck could be to create a better approach for parallel input/output operations, for example opening the file at the master node once and letting each computing node read only its portion of the file in parallel. MapReduce program showed a decrease in execution time as the number of computing nodes in the cluster increases. Yet, in contrast to Spark implementation, the execution performance of MapReduce implementation does not significantly decrease with an increase in the number of computing nodes.

The Range Search implementation with the MASS library benefits from the fact that MASS allows maintaining the original dataset structure for the duration of computation. The data is not copied continuously or moved as in MapReduce and Spark. The same binary tree containing points on its nodes is used throughout the entire computation. The main reason for the MASS performance result is the fact that we utilize the binary tree implementation. The binary three is distributed

binary tree among all computing nodes, where each computing node has a portion of the overall binary tree. Each computing node contains its local tree, which provides us the ability to create an agent for each root node of each local tree at the beginning of the computation. This shortens the tree traversal time of the overall binary tree since each local tree is traversed at the time locally on each computing node.

5.3.2 Point Location

To conduct execution performance tests for all three versions of Point Location applications we used the data set of 500,000 trapezoids and Figure 5.2 presents the gathered results.

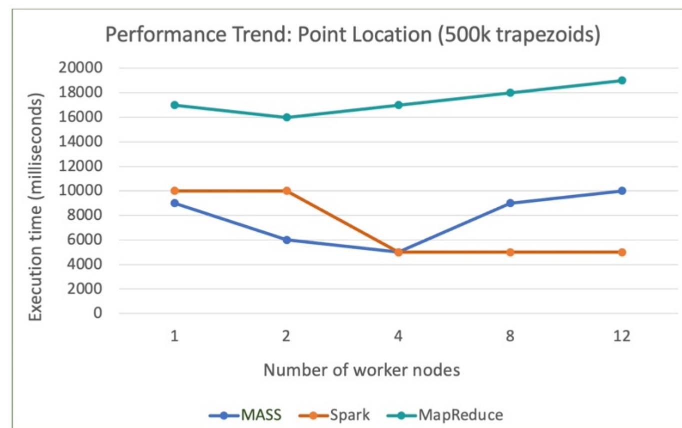


Figure 5. 2 Execution performance for Point Location implementation

Overall, the execution performance of the MASS implementation of Point Location is better than the MapReduce implementation. MapReduce takes a longer execution time due to the overhead of intermediate input/output operations between MapReduce jobs. We can see in Figure 1 that when a number of computing nodes in the cluster are less or equal to four, Point Location MASS performed better than both Spark and MapReduce implementations. After adding more computing nodes to the cluster Point Location MASS execution time starts increasing. This increase in execution time is due to having more agent activities over the network. After the agent discovers the trapezoid that locates the query point, the agent sends messages to all the other agents over the network. The message broadcast and receive operations are time-consuming. In addition, because we have more computing nodes in the cluster when creating Places, the places are distributed over all these computing nodes and during the search, agents have to migrate between more computing

nodes which also increases execution time. The MASS library uses Hazelcast as an underlying in-memory data grid in the MASS library for distributing data evenly among the cluster computing nodes [36]. Hazelcast has a disadvantage of using transmission control protocol (TCP), which is known to be more reliable but slower than user data protocol (UDP) for over-the-network communication. The potential solution to mitigate this overhead in the future is to switch from Hazelcast to our own implementation.

5.3.3 Largest Empty Circle

To conduct the execution performance tests for the largest empty circle applications we used data set of 500,000 points and Figure 5.3 presents the gathered results.

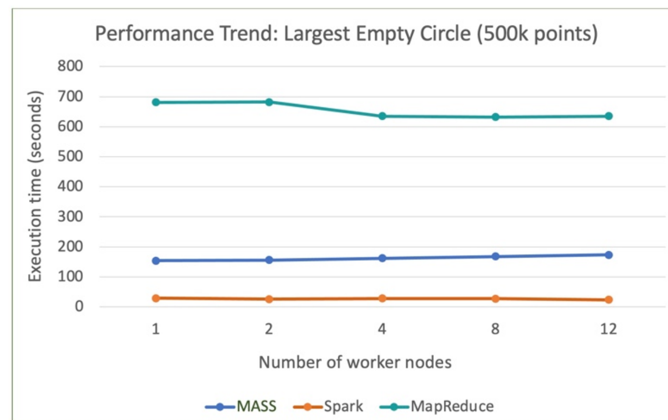


Figure 5. 3 Execution performance for Largest Empty Circle implementation

The LEC MapReduce and Spark implementations use the divide and conquer approach. However, LEC MapReduce implementation has the highest execution time metric in comparison to due to the overhead of running multiple MapReduce jobs that also read and write intermediate files to conduct the computation. Spark implantation produced the faster execution time results among all three programs. Yet, as we can observe from figure 5.3 the increase in the number of computing nodes in the cluster does not make a significant change in decreasing the execution time. This relatively same execution time of Spark program among all used clusters due to the data shuffle that occurs multiple times during computation. The MASS implementation shows competitive execution time against Spark implementation. We used the new addition to the MASS library – MASS spaces, specifically we utilized SpaceAgent and SpacePlace classes that use agent-

navigating Space data structure. MASS spaces also do not use Hazelcast as an underlining data structure, which by our assumption is the reason of slowing down the MASS application that uses the original Agent and Space classes.

5.3.4 Euclidian Shortest Path

To conduct execution performance tests for all three versions of Euclidean shortest path applications we used a considerably large data set of 500,000 graph nodes (consist of 2d points). Figure 5.4 shows the execution performance results of Euclidean shortest path application with MASS, MapReduce, and Spark.

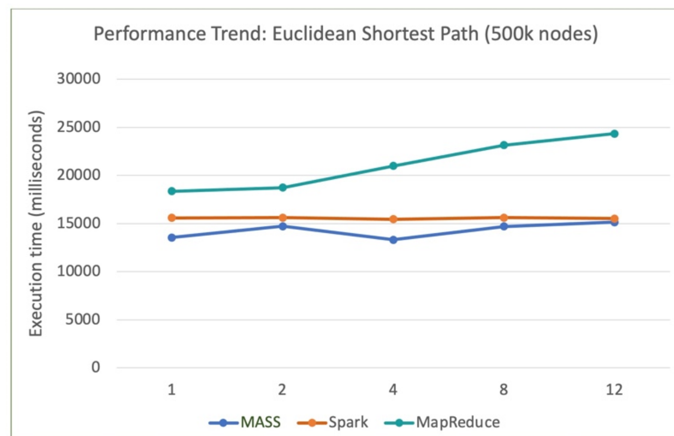


Figure 5. 4 Execution performance for Euclidean Shortest Path implementation

We can see from execution performance metrics for all three implementations of Euclidean shortest path that MASS implementation demonstrates the best run time result against Spark and MapReduce implementations. The MASS program also shows a decrease in execution time with some of the cluster configurations. While the execution performance of the MASS program starts increasing as the number of computing nodes in the cluster is greater than eight, yet the result is better than the Spark program. The reason for inefficiency as the cluster size grows is suspected to be the same reason as discussed earlier for Point Location benchmarking results (sub-section 5.3.2), which is the use of Hazelcast [8] as an underlying data grid used for data distribution. MapReduce implementation did not show any increase in the execution time when the number of computing nodes is increasing. The execution time of the MapReduce program constantly

increases, which is due to the overhead of running multiple MapReduce jobs that on its turn utilize multiple file reads and write to accommodate the computation needs. The Spark implementation produced competitive to MASS implementation results. Yet, Spark implementation did not show a significant execution time decrease during performance tests. The execution time of Spark implementation relatively the same, which is due to the overhead of multiple data shuffles during the computation. In comparison with MapReduce and Spark implementations, MASS implementation utilizes the same constructed data structure to perform data analysis, which eliminates the overhead of need for repetitive data shuffles, reads, or writes.

Chapter 6
DISCUSSION AND FUTURE WORK

This project explored the applicability of agent-based parallelization of algorithms to computational geometry. We used the MASS library to design and implement the designed algorithms. We aimed to prove that agent-based parallelization provides a more intuitive approach to developing more efficient algorithms for solutions for computational geometry problems. Some research was done in this area using MapReduce and Spark parallelization platforms that use the divide and conquer approach. Yet, the research of this area is not sufficient enough and needs improvements. The major contribution of this research is:

1. We provided agent-based parallel solutions for four computational geometry problems within the MASS library.
2. We measured and analyzed the performance of implementations with MASS against implementation with big data parallelization tools (MapReduce and Spark).
3. We determined that the MASS library provides more intuitive programming for users of non-computing backgrounds.
4. The MASS library provides a common programming paradigm utilizing high-level programming languages. It does not require the adjustment to a new programming paradigm as MapReduce and Spark.
5. MASS library fits better into applications that use the same data structure for the duration of computation. We used Places to create an in-memory distributed data structure to map input data, which is not required to be replicated or shuffled for the duration of the entire computation.

Following are the areas where future work can be done based on this work.

1. Switch from Hazelcast with TCP: the MASS library currently uses an in-memory data grid Hazelcast for distributing data evenly among the cluster computing nodes. The switch to other or our own UDP-based implementation may increase computing node scalability and

execution performance for developed applications using GraphPlaces and Places of MASS library as discussed in section 5.3.

2. Additional public interfaces for GraphPlaces: while the current GraphPlaces implementation provides an intuitive way to graph creation, several graph essential public interfaces could be added to increase the usability of GraphPlaces to users with a non-developer background. For example, to migrate an Agent to a specific graph's vertex, it is required to retrieve two types of vertex information, such as vertex's unique identifier (ID) and GlobalIndexForKey. The following example demonstrates how the user should get the vertex place where an agent to migrate, MASSBase.getGlobalIndexForKey (unique ID). This operation is a low-level MASS library function call for GraphPlaces, which is apparent to users.
3. Automated agent propagation to graph edges: users of the current GraphPlaces propagate new agents over the graph by manually specifying the agent creation and migration that is time-consuming and not intuitive. A better approach would be to implement automatic agent propagation that will allow propagating agent to the available edges of any vertex.
4. Use of multiple search queries: MASS based implemented applications can be adjusted to perform multiple search queries and benchmarked to examine query performance separate from the precomputation setup such as KD tree and road map (graph) construction. For example, the current range search application spends most of the execution time on KD tree construction and only after that perform search based on one range search query.
5. Benchmarking hardware utilization: the current performance metrics of the implemented four applications examine programmability and execution performance. Additional inquiry of hardware utilization by the executing application may provide details on efficiency of data distribution and utilization.

REFERENCES

[1] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction. 1993

[2] M. Berg, O. Cheong, M. Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications. 2008

[3] T. Whire, Hadoop: The Definitive Guide. 2012

[4] H.Karau, A. Konwinski, P.Wendell, M. Zaharia, Learning Spark. 2015

[5] M.Fukuda, Parallel-Computing Library for Multi-Agent Spatial Simulation in Java, 2010

[6] P. Alliez, A. Fabri. Cgal: the computational geometry algorithms library. 2016

[7] Boost.org. Boost.Geomtry library.

[8] OpenCL.org. OpenCL.

[9] V. Batista, D. Millman, P. Sylvain, J. Singler. Parallel Geometric Algorithms for Multi-Core Computers. 2010

[10] OpenMP.org. OpenMP.

[11] A. Gonzalez-Escribano, D. R. Llanos, D. Orden, B. Palop. Parallelization alternatives and their performance for the convex hull problem. 2005

[12] S. Puri, D. Agarwal, X. He, S. K. Prasad. MapReduce Algorithms for GIS Polygonal Overlay Processing. 2013

[13] A. Eldawy, Y. Li, M. Mokbel, R. Janardan. CG_Hadoop: computational geometry in MapReduce. 2013

[14] A. Eldawy, M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. 2015

[15] P. E. B. Pradnyana, K. M. Adhinugraha, S. Alamri. Highest Order Voronoi Processing on Apache Spark. 2018

[16] J. Gilroy, S. Paronyan, J. Acoltzi, M. Fukuda. Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory. 2020

[17] M. Kipps, W. Kim, M. Fukuda. Agent and Spatial Based Parallelization of Biological Network Motif Search. 2015

[18] J. Woodring, M. Sell, M. Fukuda, H. Asuncion, E. Salathe. A Multi-Agent Parallel Approach to Analyze Large Climate Data Sets. 2017

[19] RepastHPC. Repast.github.io/repast_hpc.

[20] J. Gilroy, Dynamic Graph Construction and Maintenance. 2020

[21] Flame.ac.uk. FLAME.

[22] C. Shih, C. Yang, M. Fukuda. Benchmarking the Agent Descriptivity of Parallel Multi-Agent Simulators. 2018

[23] C. Deissenberg, S. Hoog, H. Dawid. Eurace: A massively parallel agent-based model of the European economy. 2008

[24] Wikipedia.org. Range searching – Wikipedia.

[25] H. M. Kakde, "Range Searching using Kd Tree." 2005

[26] Wikipedia.org. Point location – Wikipedia.

[27] Wikipedia.org. Largest empty circle – Wikipedia.

[28] Wikipedia.org. Voronoi diagram – Wikipedia.

[29] Wikipedia.org. Von Neumann neighborhood – Wikipedia.

[30] Wikipedia.org. Moore neighborhood – Wikipedia.

[31] Wikipedia.org. Euclidian shortest path – Wikipedia.

[32] M. Sell, M. Fukuda, Agent Programmability Enhancement for Rambling over Scientific Dataset, 2020

[33] B. Bengfort, J. Kim. Data analytics with Hadoop: An Introduction for Data Scientists. 2016

[34] IntelliJ idea. Available: https://www.jetbrains.com/idea/

[35] Statistic plugin. Available: https://plugins.jetbrains.com/plugin/4509-statistic

[36] Hazelcast.org. Hazelcast open-source projects.

Appendix A
OUTPUT SCREENSHOTS

I: MASS Range Search output

```
20:39:16.518 [main] [DEBUG] VertexPlace constructed.
20:39:16.518 [main] [DEBUG] VertexPlace constructed.
20:39:16.519 [main] [DEBUG] VertexPlace constructed.
20:39:16.519 [main] [DEBUG] VertexPlace constructed.
20:39:16.519 [main] [DEBUG] VertexPlace constructed.
20:39:16.519 [main] [DEBUG] VertexPlace constructed.
20:39:16.520 [main] [DEBUG] VertexPlace constructed.
20:39:16.520 [main] [DEBUG] ***** Built KDTree from buildKDGraph method.
20:39:16.520 [main] [DEBUG] ***** Built Graph. Added and Initialized vertices with points.
20:39:16.520 [main] [DEBUG] ***** Build graph root id is 100
20:39:16.520 [main] [DEBUG] ***** Initial Agents instantiation done.
20:39:16.525 [main] [DEBUG] ***** Starting range search
20:39:19.527 [main] [DEBUG] ***** Elapsed time: 3 seconds
20:39:19.529 [main] [DEBUG] ====== TOTAL NUMBER OF POINTS: 10000
20:39:19.530 [main] [DEBUG] ======= Points found in requested range:
20:39:19.531 [main] [DEBUG] ======= Found points in range:
20:39:19.533 [main] [DEBUG] ***** 856,2807 857,2398 852,2880 894,2856 991,1562 948,2958 994,
2643 908,2632 910,2863 930,2967 911,2349 958,2425 970,2913 870,2956
```

II: MASS Point Location output

```
MASS.init: done
####### Point Location: Places of dimension: 10 creat
Oct 26, 2020 5:09:18 PM com.hazelcast.internal.partit
INFO: [10.158.82.18]:5701 [dev] [3.12.9] Initializing
localAgents:1
####### Exited doWhile !
####### Graph Generation Time: 58 milliseconds
####### Elapsed Time: 270 milliseconds
####### RESULT found in trapezoid [3]
finsh
Oct 26, 2020 5:09:18 PM com.hazelcast.core.LifecycleS
INFO: [10.158.82.18]:5701 [dev] [3.12.9] [10.158.82.1
Oct 26, 2020 5:09:18 PM com.hazelcast.internal.partit
```

III: Largest Empty Circle output

```
[sparon@hermes05 MASS_Apps]$ ./run.sh
ERROR StatusLogger No log4j2 configuration file found. Using
Creating Places...
Number of Points 72001
Places of dimension: 270 x: 270 created!
FPP: Creating Agents...
     72001 agents created.
*** propagate method ***
     648009 Agents created

####### Farthest pair of points:
sites:
        point1: (41810.0, 23075.0)
        point2: (42187.0, 23455.0) distance = 535.284

File is created!
Creating Places...
Number of Points 430065
Places of dimension: 657 x: 657 created!
FPP: Creating Agents...
     430065 agents created.
*** propagate method ***
     3870585 Agents created

####### Farthest pair of points:
vertices:
        point1: (72834.0, 34879.0)
        point2: (73000.0, 34724.0) distance = 227.1145

File already exists.
====================================
LEC RESULT:
vertice: [73000.0, 34724.0]
site: [41810.0, 23075.0]
distance: 33294.37341353641
Time taken: 153947 milli seconds
Time taken: 153seconds
====================================
[sparon@hermes05 MASS_Apps]$
[sparon@hermes05 MASS_Apps]$
[sparon@hermes05 MASS_Apps]$
```

IV: Euclidian Shortest Path output

```
####### Agent [3] has arrived at [30]
####### Agent [3] at [30], neighbors size [1
####### Agent [3] entered onArrival
####### Agent [3] has arrived at [31]
####### Agent [3] at [31], neighbors size [1
####### Agent [3] entered onArrival
####### Agent [3] has arrived at [1]
####### Agent [3] at [1], neighbors size [0
####### Exited doWhile !

####### Elapsed Time: 328 milliseconds
####### SHORTEST PATH:
Node[0] [25.0,85.0] ->
Node[5] [50.0,152.0] ->
Node[6] [81.0,157.0] ->
Node[7] [100.0,160.0] ->
Node[16] [105.0,155.0] ->
Node[18] [120.0,125.0] ->
Node[19] [130.0,125.0] ->
Node[24] [140.0,135.0] ->
Node[25] [154.0,149.0] ->
Node[26] [160.0,155.0] ->
Node[32] [172.0,152.0] ->
Node[33] [190.0,150.0] ->
Node[1] [210.0,85.0] ->

finsh
```