Agent Based Parallelization of Computational Geometry Algorithms

Saranya Gokulramkumar

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

Master of Science

University of Washington

2020

Committee:

Munehiro Fukuda

Min Chen

Clark Olson

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

Abstract

Agent Based Parallelization of Computational Geometry Algorithms

Saranya Gokulramkumar

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 utilizes agent-based modeling (ABM) to parallelize big data analysis. In this research, we aim to build on the previous research using MASS and extend the applicability of the library to a computationally complex problem area – computational geometry. We have developed agent based algorithms for four problems in this area – Closest pair of points, Voronoi diagram, Convex hull, and Delaunay triangulation, which is a maiden effort using ABM for such problems. This research presents parallel solutions to these four problems using two other big data analysis platforms – Hadoop MapReduce and Apache Spark. We provide a comprehensive analysis of how MASS based implementations compare to the implementations using the other two frameworks. Programmability and execution time are key criteria used to evaluate the parallel solutions. This paper discusses design approaches and algorithm specifications for four problems in all three parallel platforms and then proceeds to discuss the results. Results showed that MASS library fares well in terms of providing a capability to build intuitive parallel solutions and to perform multiple analyses in-memory on the input data. Furthermore, we discovered potential areas of enhancement for the library, which can situate the MASS library as a better contender for parallelizing data analysis in the future.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
INTRODUCTION

The agent-based parallelization of computational geometry algorithms project aims to leverage the key advantages of agent-based modeling to parallelize solutions to specific problems in computational geometry. Computational geometry has applications in the fields of biology, computational physics, spatial cognition, image processing, and GIS (Geographic Information Systems), and often involves analyzing big data. Efficient sequential algorithms have been developed to solve these problems, which have achieved a theoretical limit on execution time possible on current processors. Thus, users currently seek better data scalability instead of execution time, due to the increasing growth in data size and demand for real-time responses from applications using computational geometry. Naturally, it is logical for users to pursue the ability to parallelize these solutions. With the recent trend in parallel and distributed computing, and readily procurable commercial cloud infrastructure services, users can avail compute clusters quickly. There is also the availability of parallelization frameworks that allow end-users to seamlessly develop parallel data analysis solutions to run on a cluster of multi-core computing nodes. However, not all problems can intuitively fit into the programming models of the existing frameworks. Additionally, there is scope for frameworks that provide the capability to run multiple analyses on data in the distributed memory. In this project, we use three different parallel programming frameworks – Spark, MapReduce, and MASS – to parallelize solutions for four computational geometry algorithms. We aim to prove that the MASS library provides an intuitive way for parallelization and also provides the capability to perform multiple data analyses on the distributed in-memory data.

1.1 Parallel data streaming

Parallel data streaming allows applications to exploit a form of parallel processing. In frameworks that implement parallel data streaming like Hadoop MapReduce (MR) [40], and Spark [41], given a stream of data, a sequence of operations is applied to each element in the data stream. The input data is streamed from data sources and is processed in parallel on a multithreaded multi-core computing cluster, after which the results are output to downstream systems. Stream processing systems are designed to have a set of worker nodes, each of which will run one or more operations on a certain partition of input data, one record at a time. After the individual worker nodes have completed their tasks, intermediate results are reduced to form the overall result at the main program. In the following subsections, we briefly discuss the parallel programming execution model of data streaming based frameworks – MapReduce and Spark.

1.1.1 MapReduce

Hadoop MapReduce is a programming framework that can be used to define distributed computations on massive amounts of data. MapReduce works by breaking down the processing into two phases: the map phase and the reduce phase. Key-value pairs are the basic data structure in MapReduce. Keys and values can be primitives such as integers, floating-point values, strings, etc. or they may be programmer-defined complex structures. The programmer defines a mapper and a reducer with signatures that look like the following:

map: (k1, v1) $\rightarrow$ [(k2, v2)]

reduce: (k2, [v2]) $\rightarrow$ [(k3, v3)], where [] denotes a list.

Figure 1.1 shows the MapReduce execution model [22]. Mappers and reducers are objects that implement the MAP and REDUCE methods, respectively. A mapper object is initialized for each map task, and the MAP method called for each key-value pair in the input data. Similarly, a reducer object is initialized for each reduce task, and the REDUCE method is called once per intermediate key. In addition to this, there is a shuffle and sort operation performed by the execution framework to aggregate values by keys after the map phase.

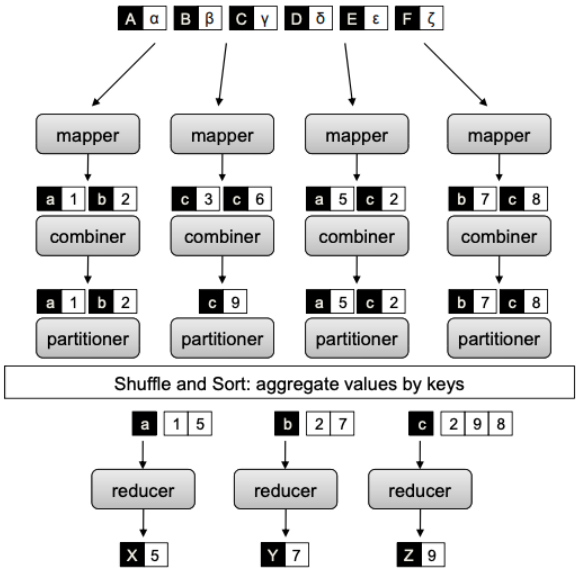


Figure 1.1: MapReduce execution model

Partitioners and combiners are the two other important elements of MapReduce. Partitioners divide the intermediate keyspace and assign key-value pairs to reducers. Combiners are an optimization in MapReduce that allows for local aggregation of values corresponding to unique keys before the shuffle and sort phase in order to reduce the high communication cost.

1.1.2 Spark

Apache Spark is a general-purpose cluster computing platform, which extends the MapReduce model to efficiently support multiple types of computations, including interactive queries and stream processing. The key programming element in Spark is the Resilient Distributed Dataset (RDD), an abstraction representing an immutable collection of items distributed across many nodes that can be operated in parallel. Input data will be represented as RDDs in a Spark application, and each RDD is split into multiple partitions, which may be assigned to different nodes on the cluster. A *driver program* is responsible for launching various parallel operations on the cluster. The nodes running parallel operations on the cluster are

called the *executors*. Figure 1.2 shows the Spark execution model [6].

Two types of operations can be performed on an RDD. *Transformations* on RDD results in the construction of a new RDD. *Actions* on an RDD compute a result based on the RDD and return the result to the driver program or write to disk. Passing functions as parameters form the crux of the programming model in Spark. Spark automatically takes the function defined in the single driver program and ships it to the executor nodes, essentially running parts of the program on multiple nodes.

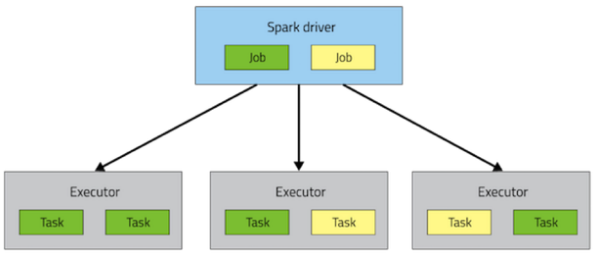


Figure 1.2: Spark execution model

Although there are multiple advantages in using a data streaming framework to parallelize data analysis, not all problems fit into this model of parallelization, including certain computational geometry problems. Problems that involve the processing of structured datasets become significantly time-consuming when implemented in MapReduce or Spark. For example, in graph analysis, each unit of execution in both MapReduce and Spark serves as a different graph vertex, and direct element-to-element communication is not supported by both these frameworks. This drawback results in the implementation of iterative MapReduce or repetitive RDD transformations to emulate inter-vertex communication in a graph, thereby incurring additional communication overhead due to the need to shuffle excessive amounts of data over the network of computing nodes. Furthermore, multiple analyses cannot be performed on in-memory distributed data due to the execution model, which requires data to be streamed from the disk for every new operation. Also, implementing computational geometry algorithms in Spark and MapReduce entails the design of complex data structures [Appendix A] and often requires the use of complex divide and conquer algorithms

to fit into the parallel programming model.

1.2 Parallel data discovery

Agent Based Models (ABM) are used to simulate the behavior of individual or collective agents on a system. MASS, a parallel ABM library, populates reactive agents over structured data, which is organized as distributed memory array elements over a cluster system, to discover important attributes of the dataset.

1.2.1 MASS

MASS (Multi-Agent Spatial Simulation) [9] is a parallel computing library utilizing agent-based modeling (ABM) of applications in physical, biological, social, and strategic domains in a given virtual space. MASS library constitutes two key classes ***Places*** and ***Agents*** which emulate distributed array of input data and the reactive entities that migrate within the array respectively. Places are mapped to threads, and agents are mapped to individual processes. Places are organized into small vertical stripes in the X-axis direction, which are executed each by a different thread. Agents, on the other hand, are grouped into bags which are allocated to different processes. Threads that are part of the same process that has a bag of agents allocated to it can then operate on these agents by executing them. The control of a MASS application is within the main program which executes one of the following three functions to carry out a simulation: *callAll()* to invoke functionality on all agents or places in parallel, *exchangeAll()* to communicate data among array elements or the agents, and *manageAll()* to spawn, move, or terminate agents. In addition to these functions, MASS provides an important feature, which is asynchronous agent migration through the *doAll()* method. This feature allows users to design iterative simulations of *callAll()* and *manageAll()* functions by eliminating the need for synchronization at the main program. Using this feature, agents' synchronization overhead can be reduced. Figure 1.3 shows how the places and agents are structured and their operations over the cluster [12].

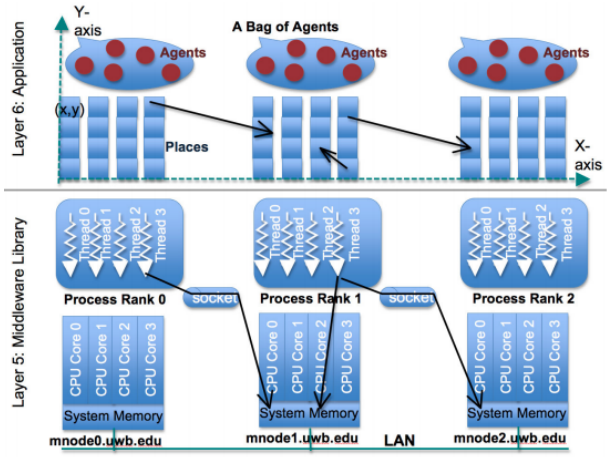


Figure 1.3: MASS execution model

The strength from using a data discovery approach lies in the fact that multiple different algorithms can be simulated using MASS, while data is still present on the distributed nodes. This provides the advantage of not having to stream data into memory for each algorithm, as required by a library using the data streaming approach. In addition to this, MASS provides an intuitive parallelization capability which will aid end users from non-computing background to get acquainted with the library easily.

1.3 Target domain

Computational geometry problems fall under the category of problems that need structured data analysis. The MASS library provides an elegant approach to organizing data using spatially distributed *Places* objects. This approach provides the ability for users to design algorithms such as the closest pair of points, by mapping input points onto the spatial data structure (Places) and having agents spawn and migrate to discover the attributes of the input data points. Additionally, certain problems require the input datasets to be in memory to support multiple analyses on the data. One such problem in the area of computational geometry is the computation of the Voronoi diagram since the diagram possesses unique properties. After the Voronoi diagram is computed for a set of generator points, the diagram

can be used to construct the Delaunay triangulation of the same points more efficiently, due to these unique properties. Furthermore, the convex hull problem can also be solved in memory using the result of the Voronoi diagram, without having to stream data from the disk for each operation. Our work aims to benchmark the MASS based implementations of such computational geometry algorithms against implementations using two big data analysis tools, MapReduce and Spark.

1.4 Research objective

The main objective of this thesis is to identify problems in the area of computational geometry, for which solutions can be parallelized with more ease using the MASS library. The main focus lies in proving that MASS provides an intuitive design capability and supports multiple data analyses of distributed in-memory data. Being able to design intuitive parallel solutions to problems is necessary because end-users from non-computing background need to be able to quickly adapt to using a parallelization framework without the need for technical proficiency in using the framework. Focusing on utilizing a parallel data discovery approach, specific goals for this research are:

1. applying different algorithms to the same structured dataset,
2. developing intuitive algorithms for computational geometry problems using agents, and,
3. evaluating computation overheads incurred by agents and proposing potential solutions as future work.

This thesis is organized as follows: Section 2 provides a review of relevant previous work done in the area of parallelization of computational geometry applications. Section 3 outlines a brief definition of the problems considered. Specific implementation details and design approaches used for each algorithm in the different parallel programming frameworks – Spark, MapReduce, and MASS are also discussed in Section 3. Section 4 presents specifications of how input data is represented and key methods used for parallelizing the four computational geometry algorithms discussed in section 3 across the three platforms. Section 5 presents

the benchmark results of parallelizing algorithms within the three platforms in terms of programmability, execution time, and operability. Finally, Section 5 presents the conclusion of this research and lays out potential future improvements that could be worked on.

Chapter 2

RELATED WORK

A multitude of sequential algorithms exist currently for computational geometry problems. CGAL (Computational Geometry Algorithms Library) is a C++ library [2] of sequential geometric algorithms and data structures developed as part of the open-source CGAL project. Such sequential algorithms are generally accepted to have achieved the theoretical limit on computational complexity possible on existing processors. These solutions also often rely on the computational capacity of the target computing node and thus are limited by the current hardware capacities. Hence, users no longer pursue speed up in execution time. With the increase in the availability of large datasets, users solicit the ability to build scalable data analysis algorithms. Thus, naturally, we would like to study parallel solutions to these problems. In the sub-sections below, we survey the research done in parallelizing computational geometry algorithms using shared memory and distributed memory paradigms. We also discuss parallelization efforts in this problem area that use big-data analysis tools. Finally, we present some related work done in the field of parallelization using the MASS library.

2.1 General parallelization of computational geometry algorithms

Work on developing design paradigms for parallelizing computational geometry algorithms has been in progress since the early 1980s. New parallel design paradigms are required because the sequential algorithm design paradigm cannot be adapted easily to the parallel processing environment. Different theoretical parallel computation models can be used to develop parallel solutions. Two of the models that have been researched extensively are *shared memory* and *distributed memory* models. Many theoretical parallel algorithms have been proposed based on the above models. In this survey, we only consider research that

includes implementations of algorithms and reports performance results.

2.1.1 Shared memory-based parallelization

In [4], Batista et al. present several parallel geometric algorithms built on top of the CGAL [2] library, utilizing the shared memory parallelization model. The target architecture in this research is shared memory parallel computers with multi-core CPUs that provide the capability to execute multiple instructions on different cores simultaneously. Paudel et al. utilize OpenMP (a programming interface that provides a set of compiler directives to allow application programmers to parallelize existing C, C++, or Fortran code) to parallelize a solution for Voronoi diagram [27]. The OpenMP-based solution is built on top of the CGAL library and achieved a speedup of 2x compared to the sequential version of the algorithm (Fortune's plane sweep). Ray et al. present a GPU based parallel algorithm for the Voronoi diagram in [34], which achieves an order of magnitude faster execution time compared to parallel implementations available in CGAL. Wynters shows the use of a C++ template library called *Thrust* [51] that supports parallel implementations of algorithms to interface with CUDA and develop a parallel solution to the Voronoi diagram.

Although the shared memory-based parallelization achieves better performance compared to sequential versions, there is a drawback in the fact that we cannot leverage distributed computing. Additionally, for GPU (Graphics Processing Unit) based implementations, users cannot use the common data structures to solve computational geometry problems. Instead, users should modify their code to make use of the GPU shared memory concept and take into account multiple factors with respect to optimal memory access by threads.

2.1.2 Distributed memory-based parallelization

Ghodsi et al. present a distributed memory-based library of basic primitives needed for the implementation of computational geometry applications, called *ParLeda* in [13]. The library is built on top of the existing computational geometry library called LEDA [25] to leverage its data structures and computations. It uses MPI (Message Passing Interface) for communication in the parallel environment (a heterogeneous network of UNIX machines). Additionally,

the library provides data partitioning methods for common cases encountered in computational geometry problems. Peterka et al. offer parallel algorithms to the Voronoi diagram and Delaunay triangulation using a distributed memory-based programming library called *DIY* [30], [29]. This library, built on top of MPI, provides configurable data partitioning and scalable data exchange in a distributed memory HPC environment [28].

The drawback of using MPI based parallelization is that MPI collective communication, in many cases, has the main program as a focal point. Computation results are often collected to the main program for input/ output (I/O) or further processing unless MPI I/O is used. Other key constraints include:

1. compiled native execution is not applicable to heterogeneous computing,
2. not fault-tolerant,
3. requires more programming changes to go from serial to parallel version, and,
4. performance is limited by the communication network between the nodes.

2.2 Parallelization with big-data computing tools

Big data computing tools such as Hadoop MapReduce, Apache Spark, Storm, etc. are widely adopted to build parallel data analysis applications to analyze big data. The target architecture for these tools is a distributed computing environment with multiple multi-core computing nodes. In [21], Eldawy et al. present a suite of scalable and efficient MapReduce algorithms for some of the key computational geometry problems such as convex hull, closest pair of points, etc., called *CG_Hadoop*. Their experiments on a 25-machine cluster show that CG_Hadoop achieves up to 14x and 115x better performance than traditional algorithms when using Apache Hadoop and *SpatialHadoop* (a Hadoop based system more suitable for spatial operations) systems respectively [8], [7]. Xia et al. provide MapReduce based implementations of the Voronoi diagram and Delaunay triangulation for SpatialHadoop system. Their method uses an incremental insertion algorithm to construct Delaunay triangulation and then transform it to the Voronoi diagram with the Quad-Edge structure (computer representation of the topology of a 2D or 3D map).

Although MapReduce has its advantages in big data analysis, applications largely involve disk-oriented computation, which is a major bottleneck for execution time. Furthermore, developers are required to transform sequential D&C based solutions into *map* and *reduce* phases, sometimes requiring multiple methods chained to achieve the program objective. This can result in difficult-to-understand programs and increase the total number of lines of code and boilerplate code percentage.

2.3 Agent based parallelization and previous work using MASS

Agent-Based Models (ABM) are used to simulate the behavior of individual or collective agents on a system. MASS, a parallel ABM library, populates reactive agents over structured data, which is organized as distributed memory array elements over a cluster system, to discover important attributes of the dataset. Successful implementations of agent-based simulations have been carried out in the areas of transport, ecology, and biology.

Woodring et al. [50] have worked on developing a parallel implementation for a global-warming analysis based on NetCDF climate data [43]. Another application in the area of biology is the biological network motif search by Kipps et al. [19]. Six benchmark programs have been implemented using MASS to compare execution time and programmability with Hadoop MapReduce and Apache Spark programs by Gordon et al. [10], [15], and [38]. The programs implemented are: Breadth-first search (BFS) and triangle counting (TC) in graph problems, ant colony optimization (ACO) and particle swarm optimization (PCO) in computational optimization, and k-means clustering and k-nearest neighbors in data sciences. Per Fukuda et al. in [11], results have shown a few challenges in certain algorithms with respect to programmability but revealed significant strengths in using the MASS library for parallelization of these algorithms. A shortest path search algorithm using MASS was simulated, which showed good results in intuitive programmability [38] and spatial scalability beyond a single machine [11]. Furthermore, Fukuda et al. outline the strengths identified as part of the above benchmarks in [11]. Their work provides a motivation to explore application areas that MASS could be used in, to give a more intuitive way of developing parallel solutions to problems in these areas.

In this research, we aim to extend the investigation on applications built using the MASS library to a new problem area: computational geometry. Agent-based modeling may have strength in navigating structured datasets such as graphs and 2D/ 3D spatial input. Hence, we feel that computational geometry is one of the good fit application areas for MASS-based parallelization. Keeping in mind the strengths discovered as part of the previous work using the MASS library, our goal is to identify strengths and weaknesses in using the library to parallelize solutions for computational geometry problems. This study will discuss possible improvements and performance enhancements to the MASS library based on the research outcome.

Chapter 3
IMPLEMENTATION

We have considered the following four computational geometry algorithms to be parallelized using the MASS library and benchmarked against MapReduce and Spark implementations: (1) Closest pair of points, (2) Voronoi Diagram, (3) Convex Hull, and (4) Delaunay Triangulation. The motivation to choose the closest pair of points problem was to implement an agent migration mechanism and 2D space management, which would then form the basis for parallelizing solutions to other problems considered. Since the closest pair of points problem is based on input data that is spatial in nature, discovering various attributes (here, the Euclidean distance) of the data is an intuitive task using agents. The Voronoi diagram problem, on the other hand, was chosen because it possesses unique properties which can lend itself to the computation of solutions to convex hull and Delaunay triangulation. This problem is a best fit for MASS-based parallelization because we can perform multiple analyses on distributed in-memory Voronoi diagram results to compute the convex hull and Delaunay triangulation for the input points. In the sections that follow, we present the problem definition, parallel design approach adopted, and other specifics of implementation for all of the four problems chosen for the project.

3.1 Closest pair of points

Definition. Given N points in a plane, find two points whose mutual distance is the smallest [49]. The closest pair of points problem is a fundamental problem in computational geometry. It forms the basis of many complex computational geometry algorithms, such as the *All Nearest Neighbors* problem. A real-time application of the closest pair of points problem is in the air-traffic control system, to detect two closest aircraft that are at a greater danger of collision.

3.1.1 MASS closest pair of points implementation

The MASS implementation uses both *Places* and *Agents* classes. The input data points are mapped to the Places matrix based on their coordinates since the point coordinates map elegantly to *Places* indices. We then spawn and migrate agents using the Von Neumann and Moore neighborhood patterns. Von Neumann neighborhood defines the neighborhood of a cell as its four neighbors at a Manhattan distance of 1 [47]. Moore neighborhood is defined on a two-dimensional square lattice and is composed of a central cell and the eight cells that surround it [46].

Table 3.1: MASS closest pair of points algorithm

Algorithm 1: MASS Closest Pair of Points

1. Initialize MASS
2. Read input from file
3. rows, cols = maximum value of x and y coordinates in input
4. Create Places = dimension rows * cols
5. Create Agents = equal to number of input points
6. Closest pair1 = ComputePair(VonNeumann, 4)
7. Closest pair2 = ComputePair(Moore, 8)
8. Return one pair among closest pair1 and closest pair2 that has a minimum distance between points.

Algorithm 2: ComputePair(neighborhood, neighbors)

1. REPEAT until closest pair found:
   (a) Agents.callAll(SPAWN, neighbors)
   (b) Agents.callAll(MIGRATE, neighborhood_pattern)
   (c) Closest pair results = Places.callAll(COLLECT_AGENTS)
   (d) Loop through all return objects and IF there are more than one non-null objects, do an O(n) comparison to compute the closest pair. ELSE continue.

Table 3.1 shows the closest pair of points algorithm which executes function calls on *Agents* and *Places* instances iteratively until the closest pair is found. Within each iteration, either four or eight agents are spawned at each place that has an agent assigned to it. Each place also maintains a set of footprints of other agents that visited that place. Maintaining an additional data structure within places provides the capability to check for collision between agents at places where other agents have already visited and left. This reduces the number of active agents in memory at any given point of time, thereby reducing the memory footprint. Since agents traveling diagonally travel at a higher speed ($\sqrt{2}$ times faster) than that of agents traveling in horizontal or vertical directions, we use two separate simulations using the two migration patterns. This is due to the fact that if we use the Von-Neumann pattern alone, points that are diagonally closer might take longer to get discovered compared to points that are horizontally or vertically closer and vice versa with Moore migration. This will produce incorrect results for certain inputs.

As shown in steps 5 and 6 of the algorithm in Table 3.1, we compute the closest pairs by running one simulation using the Von-Neumann pattern and one using Moore pattern. The *ComputePair()* routine is used to compute the closest pair using each of these migration patterns. As shown in Table 3.1, we iteratively spawn and migrate agents to appropriate neighborhood locations until an agent collision is discovered on any place. Upon discovering a collision, the routine returns with the closest pair value to the driver program. At the driver program, once the two simulations using different migration patterns are complete, we return the pair that has the minimum distance between points among the two simulations as the final closest pair. Figure 3.1 shows the *Von Neumann* and *Moore* migration patterns followed by agents to compute the closest pair result.

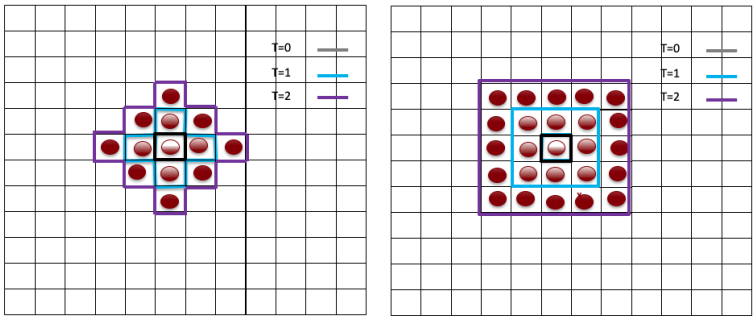


Figure 3.1: Von Neumann and Moore agent migration at time intervals 0, 1, and 2

3.1.2 MapReduce closest pair of points implementation

In the MR implementation, we have used three mappers and reducers chained together to compute the closest pair. The input data is sorted in the x dimension, and points are split into stripes of three points each. The first MR job takes care of the sorting, the second job computes the local closest pair within each stripe using a brute force approach, and the third job merges adjacent stripes together. The third MR job is run iteratively, merging stripes two at a time until we are left with only one stripe. At this point, the closest pair of the last stripe is the result of the entire input dataset.

Table 3.2: MapReduce closest pair of points algorithm

Algorithm 3: MapReduce Closest Pair of Points

1. Read input from file
2. Execute first MapReduce job to sort input points in x dimension and write output to disk.
3. Divide points into stripes of three points each and write output to disk.
4. Execute second MapReduce job to compute local closest pair for each stripe (among three points) using brute-force approach.
5. REPEAT until left with only one stripe:
   (a) Execute third MapReduce job to merge adjacent stripes
      i. map() method: modify stripe number of adjacent stripes to have the same value.
      ii. reduce() method: combine stripes with same key (stripe number) into one stripe. Compute closest pair of the combined stripe using the divide and conquer approach for closest pair problem.
6. Return closest pair from the last stripe.

Table 3.2 shows the MapReduce implementation for the closest pair of points using the *Divide and Conquer* (D&C) strategy proposed by Shamos and Hoey in [32]. In every MR job, the output from the *map()* method is written to the local file system (disk) and the *reduce()* method reads its input from this file. After each MR job is finished, intermediate output is written to the *Hadoop Distributed File System* (HDFS) – underlying distributed file system of the Hadoop ecosystem. The major performance overhead in this implementation is due to these disk-based operations. This significantly increases the execution time of the implementation. Figure 3.2 shows the closest pair computed using the D&C approach. Figure 3.3 shows the MR execution flow for the closest pair implementation. Left hand side of the diagram indicates the different MR jobs running at each step of the process. The steps

included in the dotted rectangular box are the steps performed in the third MR job. This job is run iteratively until there are no more stripes to merge, and we are left with only one stripe, which will contain the final closest pair result.

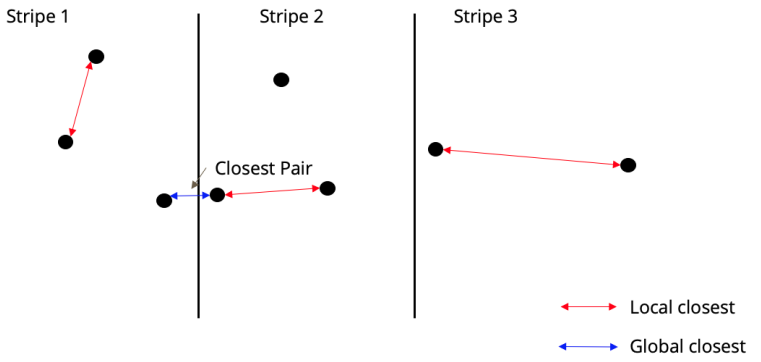


Figure 3.2: MapReduce and Spark closest pair of points implementation using D&C approach

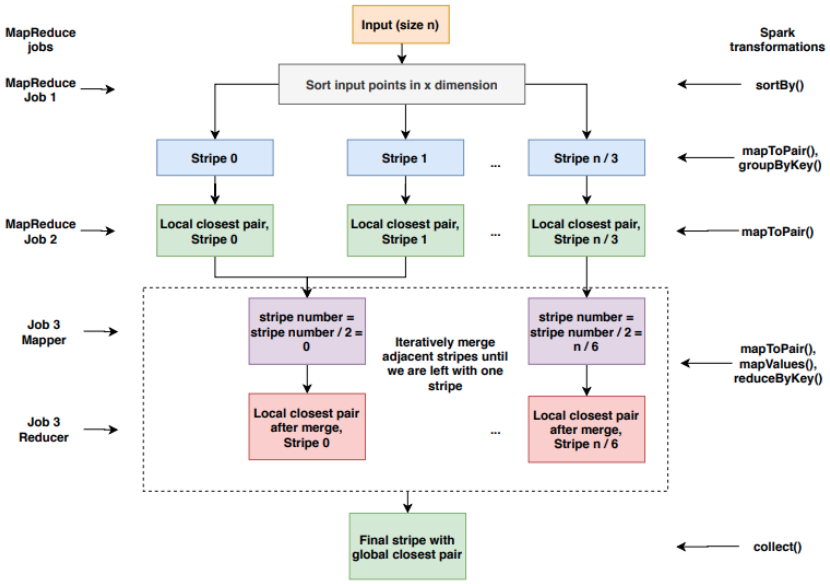


Figure 3.3: MapReduce and Spark closest pair of points algorithm

3.1.3 Spark closest pair of points implementation

The Spark implementation follows a similar strategy as MapReduce implementation of the closest pair of points. The only differentiating factor is that instead of using mappers and reducers, we use a series of Spark *transformations* and *actions* on the input RDD. The Spark algorithm is identical to the algorithm in Table 3.2. Figure 3.2 shows the Spark execution flow for the closest pair of points implementation. Right hand side of the figure presents different transformations and actions used in each step of the algorithm. For each of the steps in the algorithm, we present the different RDDs created below:

1. Input points are stored in a JavaRDD⟨Point⟩, where *Point* is the data structure to represent each point in terms of x and y coordinates.
2. The points are sorted using the *sortBy()* transformation and the results are stored in a new JavaRDD⟨Point⟩.
3. The *mapToPair()* transformation is used to create a new JavaPairRDD⟨Integer, Point⟩, where the key represents the stripe number and the value is the point mapped to it. Also, in step 3 of the algorithm, we group the points by their respective stripe numbers. We use the *groupByKey()* transformation to create a new JavaPairRDD⟨Integer, Iterable⟨Point⟩⟩.
4. We use a brute force approach to compute the local closest pair within each stripe using the *mapToPair()* transformation. The results will be stored in a new RDD JavaPairRDD⟨Integer, ClosestPair⟩, where the ClosestPair data structure contains information about the closest pair within the stripe, list of points in the stripe, etc.
5. In step 5, we use *mapValues()* transformation to modify the stripe numbers so that every two adjacent stripes have the same number to facilitate reduction based on the key (merge step). We create a new JavaPairRDD⟨Integer, ClosestPair⟩. We then use the reduceByKey() transformation to merge two stripes with the same key and store the results in a new JavaPairRDD⟨Integer, ClosestPair⟩.

After merging all the stripes, the final stripe will contain the closest pair result. The

major advantage of Spark implementation compared to MapReduce implementation is that all operations on RDDs are performed in-memory, and hence the execution is faster.

3.1.4 Design decisions

In this section, we present some of the key design decisions made for the closest pair implementations in the three platforms. The MASS closest pair of points implementation has the following design decisions:

1. X and y coordinates of input points are integers.
2. Rows and columns for the places matrix dimension will be calculated based on the largest value in x and y dimensions.
3. Both Von Neumann and Moore migration patterns will be used for agent migration because agents travel at different speeds diagonally vs. horizontally and vertically.

The MapReduce and Spark closest pair of points implementations have the following design decisions:

1. Choice of divide and conquer approach, because it fits both the programming models.
2. X and y coordinates of input points are integers.

3.2 Voronoi diagram

Definition. Given a set S of N points in the plane, for each point pi in S, the locus of points closer to pi than to any other point, denoted by V(i), is a convex polygonal region having no more than 8 N-1 sides. V(i) is called the Voronoi Polygon associated with pi. These N regions partition the plane into a convex net, which is referred to as the Voronoi diagram [48].

Voronoi diagrams are widely used in applications belonging to the areas of natural sciences, health, engineering, informatics, etc. For example, in networking, Voronoi diagrams can be used to derive the capacity of a wireless network [48]. Voronoi diagram possesses multiple unique properties which can help solve other computational geometry problems

efficiently. Some of the problems include convex hull, closest pair of points, Delaunay triangulation, etc. [32].

3.2.1 MASS Voronoi diagram implementation

The MASS implementation of the Voronoi diagram follows a similar strategy as that of MASS closest pair of points. We read the input data points and map them to the *Places* matrix utilizing the correspondence between point coordinates and the *Places* indices. We spawn and migrate Agents from each of the points representing a Voronoi site following an alternating *Von Neumann* [47] and *Moore* [46] migration patterns. Figure 3.4 shows the agent migration pattern and agent ripple collisions. With the use of this kind of migration, we tried to emulate agents migrating in the form of a ripple originating from each Voronoi site. When agents meet at the mid-point of any two points, a perpendicular bisector is constructed, and the edge is inserted into the Voronoi region of both the sites. The Voronoi edges are tracked as a *Java HashMap* with the key being a parametric form of the line, containing slope and intercept information and the value as the edge itself. This data structure helps us store a single entry for an edge with the same slope and intercept for cases where multiple collisions cause the same edge to be computed. We also check for collisions of three agents to compute Voronoi vertices. In addition to this, each place will contain a set of points that belongs to agents that visited it, called the *Agent Footprints*. We use this list of points to check for collisions among agents that might have traveled to a place in any of the previous iterations and left.

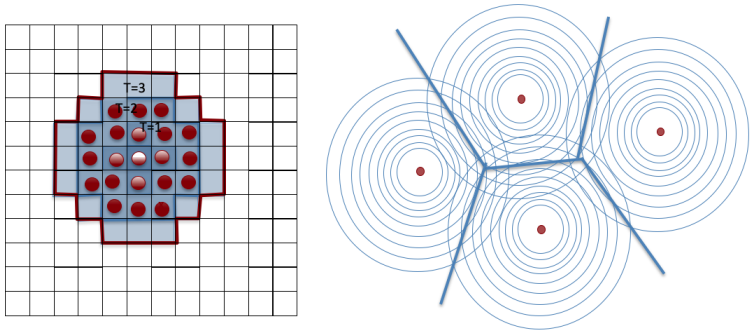


Figure 3.4: Alternating agent migration in Von Neumann and Moore pattern, and agent ripple collisions

Table 3.3: MASS Voronoi diagram algorithm

Algorithm 4: MASS Voronoi diagram algorithm

1. Initialize MASS
2. Read input from file
3. rows, cols = maximum value of x and y coordinates in input
4. Create Places = dimension rows * cols
5. Create Agents = equal to number of input points
6. REPEAT for user specified number of iterations:
   (a) Agents.callAll(SPAWN, neighbors)
   (b) Agents.callAll(MIGRATE)
   (c) Agents.callAll(CHECK_COLLISION)
   (d) Agents.callAll(MIGRATE_TO_SOURCE)
   (e) Agents.callAll(UPDATE_VORONOI_EDGES)
   (f) Agents.callAll(UPDATE_VORONOI_VERTICES)
7. Voronoi Results = Places.callAll(COLLECT_AGENTS)
8. Loop through the returned objects from Places and construct the final Voronoi diagram.

Table 3.3 presents the MASS Voronoi diagram algorithm. Step 5 in the Table 3.3 spawns ***neighbors*** number of agents in every iteration. This number is ***four*** for Von Neumann

migration and ***eight*** for Moore migration. Once the agents are spawned (step 6.i)), we migrate the agents to appropriate neighbor locations based on the current iteration's migration pattern (step 6.ii)). We then check for collisions at all places to identify agents colliding at midpoints of any two points and also check for agents colliding at centers of circles passing through three different Voronoi sites (step 6.iii)). Agents that are part of a collision migrate to their source locations (the point mapped to the current agent), carrying the information they discovered at the collision site (perpendicular bisector or Voronoi vertex) (step 6.iv)). This information is then updated at the source Voronoi sites in terms of either adding a Voronoi edge (perpendicular bisector) or Voronoi vertex (steps 6.v and 6.vi)). After the user-specified number of iterations are completed, we collect the information from each of the Voronoi site (step 7) and construct the final Voronoi diagram (step 8).

The number of times step 6 in the algorithm runs is determined by user input. The other alternative to this is to run the simulation until all active agents get killed due to the inability to migrate beyond the *Places* boundary. This alternative proved to be a performance bottleneck. Also, the maximum iterations required by agents to collide is given by the maximum distance between any two points in the input. Given this fact, we do not have to run the simulation until all active agents die; we only need to run as many iterations as required to construct an accurate Voronoi diagram. An accurate Voronoi diagram should satisfy the following conditions:

1. Two agents mapped to points that must have a perpendicular bisector in the final Voronoi diagram, should collide.
2. Three agents mapped to points between which there is a Voronoi vertex in the final Voronoi diagram, should collide.

Thus, we expect input from the user for the number of iterations of agent migration required. Based on a trial-and-error method (by visualizing the computed Voronoi diagram and checking for the presence of either of these: infinite edges where the intersection is not detected or undetected Voronoi vertices), users can determine the appropriate number of iterations for the current input. This could be improved in the future in terms of reducing

user intervention by computing the following:

1. Two neighboring points that have the farthest distance among other pairs in the input – distance1.
2. Circle with the largest radius with three points on its circumference – the distance between any one of the three points and the center of the circle is the distance needed for the three points to collide – distance2.

The maximum of distance1 and distance2 will give us the appropriate number of iterations required.

3.2.2 MapReduce Voronoi diagram implementation

The MapReduce implementation for the Voronoi diagram is based on the D&C algorithm proposed by Shamos et al. in [32]. Table 3.4 presents the overview of steps followed by the MapReduce implementation. We execute a sequence of three MR jobs to compute the Voronoi diagram. In step 1, input points are sorted in the x dimension, and in step 2, we split the sorted points into stripes of two points each, similar to the strategy used in the closest pair of points implementation. This Voronoi diagram algorithm requires that the convex hull of the input points be computed as a prerequisite. So, in step 3, we construct local convex hulls for the points in each stripe. Following this, we also construct local Voronoi diagrams for points within each stripe (which includes adding a perpendicular bisector for the points within the stripe). We use the *Doubly Connected Edge List* (DCEL) data structure to represent the boundary of each of the Voronoi regions in the final output. This data structure will consist of the Voronoi edges and vertices that demarcate the current Voronoi site from other sites. Figure 3.5 shows MR execution flow for the convex hull, Voronoi diagram, and Delaunay triangulation implementations.

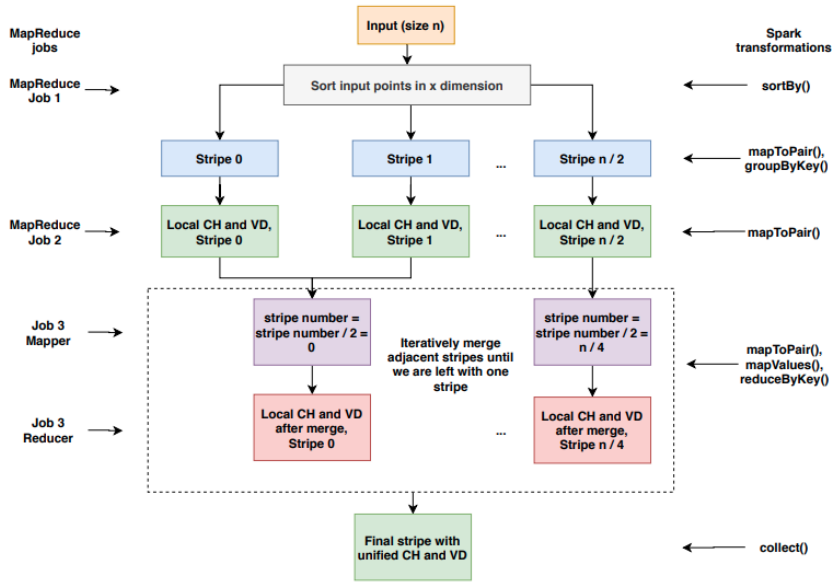


Figure 3.5: MapReduce & Spark convex hull, Voronoi diagram, and Delaunay triangulation algorithm

Table 3.4: MapReduce Voronoi diagram algorithm

Algorithm 5: MapReduce Voronoi Diagram

1. Read input from file
2. Execute first MapReduce job to sort input points in x dimension and write output to disk.
3. Divide points into stripes of three points each and write output to disk.
4. Execute second MapReduce job to compute local Voronoi diagram for each stripe (among two points) using brute-force approach.
5. REPEAT until left with only one stripe:
   (a) Execute third MapReduce job to merge adjacent stripes
      i. map() method: modify stripe number of adjacent stripes to have the same value.
      ii. reduce() method: combine stripes with same key (stripe number) into one stripe. Compute Voronoi diagram of the combined stripe using the divide
      and conquer approach for the closest pair of points problem.
6. Return final Voronoi diagram from the last stripe.

Figure 3.6 shows the construction of the dividing chain while merging two stripes to compute the unified Voronoi diagram using the D&C algorithm [18]. In step 4 of the algorithm in Table 3.4, the third job is run iteratively, combining adjacent stripes two at a time until we are finally left with one stripe. During the merge phase, we first merge the convex hulls into a unified hull and use the two points that form the upper tangent (Figure 3.8) of the unified hull as the starting point to construct the Voronoi diagram. The left point becomes the left Voronoi site, and the right point becomes the right Voronoi site. We start by constructing a perpendicular bisector (dividing chain, shown in the blue line in Figure 3.6) between these two points. We then traverse the left Voronoi site clockwise and the right

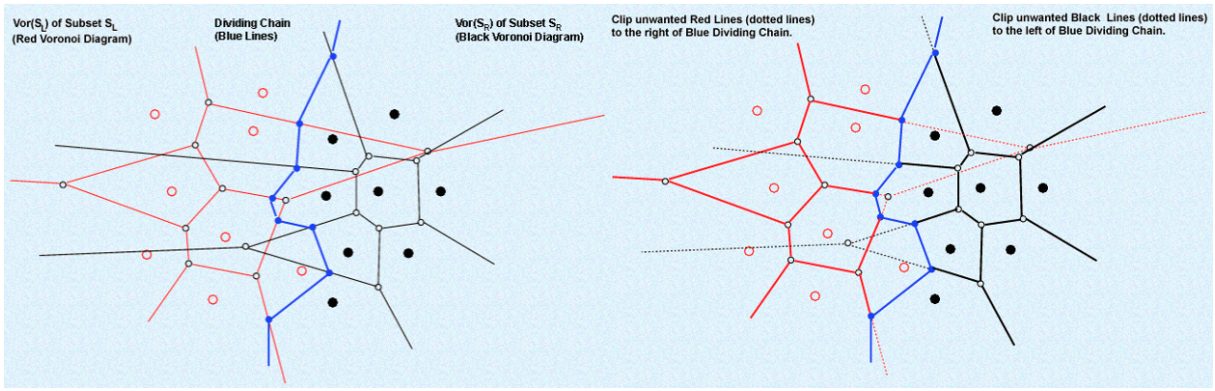


Figure 3.6: Construction of dividing chain and the unified Voronoi diagram during the merge step of MapReduce and Spark implementations

Voronoi site counterclockwise to find the closest edge that intersects the dividing chain. If an edge from the left Voronoi site is chosen, we change the left site to be the other endpoint of the chosen edge that is not the current site. If an edge from the right Voronoi site is chosen, we do the same for the right site. In step 5, after the end of iterations, we will be left with one final stripe with the final Voronoi diagram result. Although this implementation is able to handle large input size, there are multiple disadvantages:

1. Due to the disk-oriented operations in each map-reduce phase, there is a tangible effect on execution time. In addition to this, since the convex hull of points is a prerequisite for the Voronoi diagram D&C algorithm, we had to design complex data structures to implement this algorithm.
2. The input and output formats available in the library allow users to specify what format the mappers and reducers should use for reading and writing data on the. Finding an appropriate input and output format for the complex data structures was a challenge. Due to the inability to perform in-memory operations, and the need for the Voronoi diagram algorithm to run iteratively and modify the constructed diagram from previous iterations, we had to serialize the Voronoi diagram object to write it to disk. One solution was to use the MapReduce *Writable* interface and manually

designate which fields should be serialized within each of the application classes. This process seemed tedious for the complex data structures we had for the Voronoi diagram implementation. Instead, we opted to serialize the intermediate Voronoi diagram object to file using Google's Gson serialization library [16]. The Voronoi diagram java object will be serialized to a text format, which we then write to file using the standard text output format in MapReduce. We then read it from file for subsequent MR jobs and deserialize it to a java object using the Gson library.

3. The data structures used in this implementation contain references to objects of the same class within each object (cyclic references), which Gson was not able to serialize and deserialize. Due to this drawback, we had to convert the references and store it in an additional data structure (list) to enable serialization. During deserialization, appropriate references will be built manually, which increased the development effort and time.

Drawbacks like these and the enormous learning curve for the MapReduce framework were major setbacks in application development time for the Voronoi diagram using MapReduce.

3.2.3 Spark Voronoi diagram implementation

The Spark implementation follows a similar approach to the MapReduce implementation. The algorithm used is the same D&C algorithm outlined in Table 3.4. We sort the input points in the x dimension, split them into stripes of two points each. We then iteratively merge adjacent stripes by applying a sequence of transformations and actions on the RDDs until we are left with only one stripe.

The Spark implementation also required complex data structures to represent and construct the Voronoi diagram. This led to significant effort and time for design and implementation. Additionally, due to a large number of application-level instances in memory during execution, this implementation fails to run for points above size 128. We encounter an *OutOfMemory* exception for java heap space. This is due to the fact that all of the output is collected to the driver program to construct the final Voronoi diagram. This bottleneck

can be alleviated if we write the output to disk from all of the distributed computing nodes. However, the advantage of the Spark implementation over MapReduce implementation is that we did not have to handle serialization and deserialization at the application level. This is due to the fact that all operations in Spark are performed in-memory.

3.2.4 Design decisions

In this section, we present some of the key design decisions made for the Voronoi diagram implementations in the three platforms. The MASS Voronoi diagram implementation has the following design decisions:

1. The x and y coordinates of input points are integers.
2. Number of rows and columns for the *Places* matrix dimension will be calculated based on the largest value in x and y dimensions.
3. Every two integers will have ten intervals between them to accommodate the mid-points of two points on the *Places* matrix. Hence, we created a finer mesh for the *Places* matrix.
4. Both Von Neumann and Moore migration patterns will be used for agent migration.
5. We used *asynchronous* agent migration to improve application performance. With this feature, the control doesn't come back to the main program during the construction of the Voronoi diagram. We invoke the *doAll()* method on the agents and pass it a list of methods to invoke on each iteration.

The MapReduce and Spark closest pair of points implementations have the following design decisions:

1. Choice of divide and conquer approach, because it fits both the programming models.
2. X and y coordinates of input points are integers.

3.3 Convex hull

Definition. Given an arbitrary subset L of points in the Euclidean space Ed, the Convex Hull Conv(L) of L, is the smallest convex set containing L. Where convex set is a subset C of Ed, in which for all x and y in C, the line segment connecting x and y is included in C [49].

Convex hulls have applications in pattern recognition, image processing, statistics, geographic information system, etc. [49].

3.3.1 MASS convex hull implementation

The MASS convex hull implementation uses agent-migration techniques to construct the convex hull. We supplement this with Graham's scan algorithm [32] to detect concavities in the convex hull and eliminate points that create them. A prerequisite for this implementation is that the Voronoi diagram for the input points is computed and be present in the distributed memory. This enables us to do multiple analyses on the same input data in memory without having to read the intermediate data from disk. We start by instantiating one agent at the lowest point of the input data. We migrate this agent along the unbounded regions of the Voronoi diagram and add points along the route to the convex hull points. Every time we add a point, we check for concavities formed by the last three points added to the list of points in the convex hull. Say the last three points are p1, p2, and p3. If the angle(p1p2p3) is concave, we remove point p2 from the list (an example point shown in red in Figure 3.7). When we find a concavity, we keep running this check until we find three points that form a convex angle in the list. This algorithm saves a lot of time compared to just using Graham's scan algorithm because we limit the sites that the agent visits only those that have an unbounded Voronoi region in the Voronoi diagram. We stop when the agent comes back to the place of origin and output the final convex hull. Figure 3.7 shows the construction of the convex hull with a Voronoi diagram as the base.

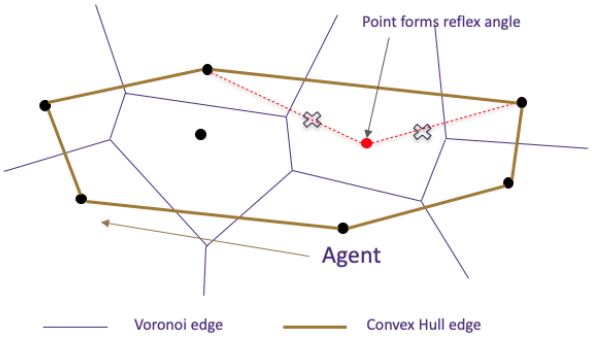


Figure 3.7: Convex hull construction using the Voronoi diagram output and traversing unbounded regions using a single agent

Table 3.5: MASS convex hull algorithm

Algorithm 6: Convex Hull

1. Create Agents: one agent that will traverse the unbounded regions of the Voronoi diagram
2. INITIAL_POINT = compute the point with lowest y coordinate in the input points
3. REPEAT while the convex hull agent is alive:
   (a) Agents.callAll(COMPUTE_NEXT_POINT)
   (b) Agents.callAll(MIGRATE)
4. Loop through the returned objects from Places and construct the final convex hull.

Table 3.5 shows the MASS algorithm for the convex hull. We iteratively call functions on the convex hull agent in step 3 to compute the next point and migrate the agent to that location. The method to compute next point loops through all the edges that are part of the current site's Voronoi region from the Voronoi diagram and checks if there are any infinite edges. Voronoi regions that contain at least one infinite edge is treated as an unbounded region and will be considered as the next target location for the convex hull agent to migrate to. Additionally, when an agent migrates to a Voronoi site if the site is found to

be unbounded, we add it to the list of convex hull points. In step 4, we collect results from *Places* and construct the final convex hull.

3.3.2 MapReduce convex hull implementation

The MapReduce convex hull implementation follows the algorithm presented in Table 3.4 for MapReduce Voronoi diagram implementation. Since computing the convex hull is a prerequisite for the implementation of the Voronoi diagram, we have designed the Voronoi diagram data structure to include the data structure for the convex hull. Using the 'composition' java language feature, we have implemented the Voronoi diagram class to include an instance of the convex hull object. The implementation starts by sorting the input points in x dimension and splitting them into stripes of two points each. Each stripe is represented by a key – the stripe number and value – the Voronoi diagram instance. The Voronoi diagram instance contains the list of points that are part of the current stripe and also contains a reference to the convex hull for the stripe. When we iteratively merge stripes using the D&C strategy, in step 5 of Table 3.4, we first merge the convex hulls for two adjacent stripes. Following this, we use the unified convex hull as the base to merge the Voronoi diagrams belonging to the adjacent stripes into one. Figure 3.8 shows the construction of the convex hull while merging two stripes using the D&C approach.

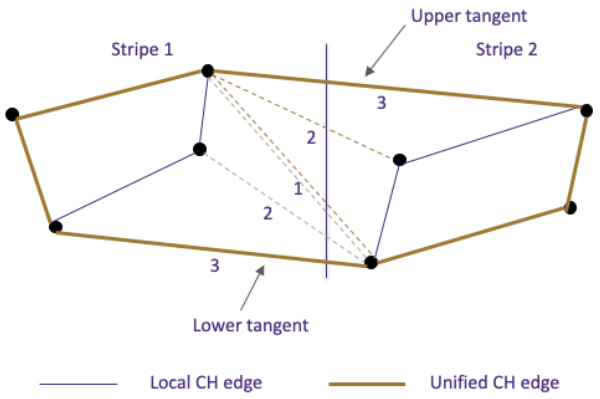


Figure 3.8: Merging two stripes to construct a unified convex hull

During the merge phase, we start by finding the rightmost point in the left stripe and the leftmost point in the right stripe. We compute the upper and lower tangents that connects the local convex hulls from the two stripes. The upper tangent is computed by determining two points, one on the left hull and one on the right hull. To compute the right point, we keep the left point constant and keep changing the right point to be its clockwise neighbor until the edge formed by the left and right points don't cut the right convex hull. We then keep the right point constant and keep changing the left point to be its counterclockwise neighbor until the edge formed by the left and right points don't cut the left convex hull. Similarly, we compute the two points that form the lower tangent. The iterative process of computing upper and lower tangents is shown by the dotted lines in Figure 3.8. At the end of the iterative merge phase, the final stripe will contain the unified hull for all of the input points. A similar argument as that of the Voronoi diagram implementation can be made for the convex hull implementation with respect to the complexity of data structures. The execution time is poor as well when compared to the Spark implementation due to the disk-oriented operations and requirement to serialize and deserialize intermediate convex hull information.

3.3.3 Spark convex hull implementation

The Spark convex hull implementation follows the algorithm in Table 3.4. As explained in section 3.3.2, the Voronoi diagram instance will contain a reference for the convex hull corresponding to the points in every stripe. We merge the adjacent stripes by applying multiple transformations and actions on the RDDs to compute the final convex hull.

3.3.4 Design decisions

In this section, we present some of the key design decisions made for the convex hull implementation in MASS. The MASS convex hull implementation has the following design decisions:

1. X and y coordinates of input points are integers.

2. The decision to use Graham's scan algorithm to detect concavities – there might be few regions in the Voronoi diagram that, even though should be bounded, will be unbounded because of the restriction that agents cannot travel beyond place boundaries. This results in the need to perform additional checks to see if a point should be in the convex hull. We tried to check if edges in the region have intersections and then exclude such regions from the convex hull, but some unbounded regions that shouldn't be part of the convex hull did not have any intersecting edges, and we did not have a way to differentiate between these. So, we opted to use Graham's scan algorithm to check for concavities and eliminate non-convex hull points.

3.4 Delaunay Triangulation

Definition. Given a set P of points in Ed, a *Delaunay triangulation* is a triangulation Delaunay triangulation(P) such that no point in P is inside the circumcircle of any triangle in Delaunay triangulation(P) [45]. The Delaunay triangulation of a set of points P corresponds to the dual graph of the Voronoi diagram of P. Hence, the Delaunay triangulation can be constructed using the Voronoi diagram. Applications of this problem include terrain modeling using a set of sample points, path planning in automated driving, etc. [45].

For the Delaunay triangulation implementation, a similar strategy was used in all three of the platforms, MASS, MapReduce, and Spark. During the design for Voronoi diagram implementation, we factored in additional data structures and methods to compute the Delaunay triangulation while the Voronoi diagram is being constructed. Voronoi vertices are discovered either by means of three agents colliding in MASS or by the detection of edge collision during the merge phase in MapReduce and Spark. When a new Voronoi vertex is detected, we identify the three sites of which the Voronoi vertex is a part of and record these three sites within the Voronoi diagram. Once the Voronoi diagram computation is complete, we get the Voronoi vertices information stored in the Voronoi diagram instance to construct the Delaunay triangulation. For each Voronoi vertex, we read the points that it is a part of and construct edges between each pair of points in the combination. Figure 3.9 shows the Delaunay triangulation constructed using the Voronoi diagram as the base.

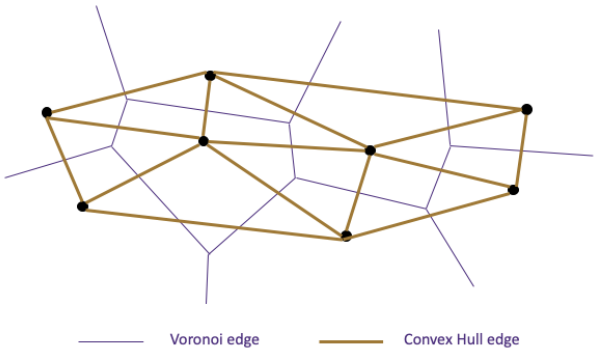


Figure 3.9: Delaunay triangulation constructed using Voronoi diagram as the base

Chapter 4

PARALLELIZATION

In this section, we present specifications of how input data is represented for parallelizing the four computational geometry algorithms discussed in section 3 across the three platforms. We also discuss the key methods used to implement the four algorithms in MASS, MR, and Spark.

4.1 MASS

The MASS closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation implementations, are centered around the *Place* and *Agent* classes. Methods are invoked on each of these instances iteratively to discover attributes of the input dataset. The **Place** parent class defines the attributes and functionality required to create and maintain a distributed array of elements across the nodes in the computing cluster. Each element is pointed to by a set of network-independent matrix indices. The **Agent** parent class similarly defines the attributes and functionality required to create, migrate, and execute methods on agents. Each agent is an execution instance that can reside on a place, migrate to any other place with matrix indices, and interact with other agents as well as places. Table 4.1 indicates key data structures and classes used to represent the input within MASS implementations for all four applications.

Table 4.1: Input dataset construction using MASS

	Closest pair of points	**Voronoi diagram**	**Convex hull**	**Delaunay triangulation**
1.	A 2D space represented with 2D Places			
2.	Point class			
3.	ClosestPair class – closest pair of points and their distance	Voronoi diagram represented by a Hashtable, maintained at Place (= point)	convex hull represented by an ArrayList, maintained at Place (= point)	Delaunay triangulation represented by an HashSet of Voronoi vertices, at Place (= point)
4.	-	Edge class, ParametricLine class	-	VoronoiVertex class

From Table 4.1, we can observe that for the four MASS implementations, we represent the input data as a point (*Point* class) attribute that each place contains. For the closest pair of points, we need an additional class to represent the closest pair and the distance between them. For Voronoi diagram implementation, we represent the Voronoi diagram using a *Java Hashtable* with keys as the parametric form (*ParametricLine* class) of Voronoi edge and the value as a set of edges in the region (*Edge* class). We represent the convex hull as a *Java ArrayList* of points within places that have points mapped to them. We represent the Delaunay triangulation as a *Java HashSet* of Voronoi vertices (*VoronoiVertex* class). This class contains information about the three sites that are on the circumference of the circle with the center as the Voronoi vertex point. Table 4.2 indicates the algorithm parallelization approaches used for each of the applications.

Table 4.2: Agent algorithm for parallelization

Closest pair of points	**Voronoi diagram**	**Convex hull**	**Delaunay triangulation**
Agent propagation using Von Neumann pattern first, followed by Moore pattern	Agent propagation using Von Neumann pattern in odd iterations and Moore pattern in even iterations	Agent walk on the unbounded regions of Voronoi diagram	Collect Voronoi vertices from Voronoi diagram, no agents involved

In the subsections below, we briefly discuss how the Places and Agents data structures are constructed and details about the key methods used in each of the algorithms.

4.1.1 MASS Place data structure and key functionality

In our closest pair of points application, we have a class *Plane2D* that inherits from the **Place** base class. This class represents the 2D plane that will be partitioned and distributed across nodes in the cluster. Listing 4.1 shows the code snippet used to instantiate the *Plane2D* object. This function will create the Places matrix, with each element being an instance of the class *Plane2D*.

```
Places plane2D = new Places(1, Plane2D.class.getName(), null, rowsAndCols[0]
, rowsAndCols[1]);
```

Listing 4.1: Code to create Place instances

The *Places* constructor takes five arguments including:

1. int handle – A unique identifier that designates a group of places. Must be unique over all machines.

2. String className – Name of the user-implemented class Places are constructed from.
3. Object argument – Arguments for user-defined Place constructor.
4. int... size – Size of the matrix the user wants to create (the three dots indicate that field accepts a variable number of arguments depending on the dimensions of the *Places* matrix to be created). Since the target dimensionality is two, we use two arguments to specify the number of rows and columns of the matrix.

The matrix dimensions are calculated based on the input point coordinates. We loop through the input points and identify the maximum value in both x and y dimensions and use these values as the dimensions. For example, if the input points have the maximum x coordinate as 20 and the maximum y coordinate as 30, the dimensions of the *Places* matrix will be 20x30. We then call the *initializePlane()* method to map the input points to the matrix. Each point will be mapped to an index based on its x and y coordinates. For example, the point (10,10) will be mapped to the *Places* matrix index (10,10).

In the list below we present key *Place* functionality. Methods 1 through 5 are used in the closest pair of points implementation, and methods 1 through 10, except method 5 are used in the Voronoi diagram implementation.

1. callMethod() – facilitates invoking appropriate methods on each place. This method is called when the main program invokes the *Places.callAll()* method. The method to be invoked is passed as an argument from the calling program.
2. initializePlane() – initializes the initial point locations and invokes methods 3 and 4 below.
3. initializeVonNeumanNeighbors() – initializes indices for the Von Neumann neighbors for each place in a vector.
4. initializeMooreNeigbors() – initializes indices for the Moore neighbors for each place in a vector.
5. collectAgents() – used in closest pair of points implementation. Returns the closest pair information from all places upon collision of two or more agents, null otherwise.

6. collectVoronoiRegions() – returns a java hashtable containing a list of edges with a parametric form of the edge as the key and the edge itself as the value.
7. removeNonVoronoiEdges() – removes edges that do not have either one of the endpoints as a Voronoi vertex contained in the current Voronoi site. This method is called during the post-processing step of the Voronoi diagram implementation.
8. removeNonVoronoiVertices() – removes Voronoi vertices that are not contained in all of the three sites that form the Voronoi vertex. During the Voronoi diagram construction, some Voronoi vertices will be removed, and this information is updated only in the corresponding Voronoi site. The other two sites that are part of this vertex are not updated due to the inability of agents to access the place instance of a location other than its own. This method is called during the post-processing step of the Voronoi diagram implementation.
9. collectConvexHullPoints() – returns the final convex hull points.
10. collectVoronoiVertices() – returns the final Voronoi vertices list to construct the Delaunay triangulation.

For the Voronoi diagram implementation, the creation of the *Places* matrix is slightly different than that of the closest pair of points. In order to capture the midpoint between any two points in the input, we need to have corresponding indices within the *Places* matrix as the point coordinates. Since we have considered the point coordinates to be integers only for this implementation, we can expect midpoints to be decimals and have at most one digit followed by the decimal point. To capture such coordinates, we have divided the *Places* matrix into a fine mesh, by allowing ten intervals between every single integer interval. For example, in order to have ten intervals between the digits 1 and 2, we multiply these digits by a factor of 10. By doing so, the *Places* matrix will have indices ranging from 10, 11, 12, up to 20. We then map each point to the appropriate place index by multiplying its coordinates by 10 as well.

4.1.2 Agent data structure and key functionality

In the closest pair of points, Voronoi diagram, and convex hull applications, we have agent classes that inherit from the **Agent** base class. This class represents the execution instances that we will be using to compute the closest pair of points, Voronoi diagram, and convex hull. Listing 4.2 shows the code snippet used to instantiate the *ClosestPairAgent* object for example. Similar syntax is used to create initial Voronoi diagram agents as well, but with a different class for the Voronoi diagram agent. This function will create a specific number of agents, each of which is an instance of the class *ClosestPairAgent*.

```
Agents closestPairAgents = new Agents(2, ClosestPairAgent.class.getName(),
null, plane2D, initPopulation);
```

Listing 4.2: Code to create Agent instances

The *Agents* constructor takes five arguments including:

1. int handle – A unique identifier that designates a group of Agent objects. Must be unique over all machines
2. String className – The name of the user-defined class to instantiate
3. Object argument – The argument to pass to each Agent as it is being instantiated
4. Places places – The Places instance that will contain the Agents
5. int initPopulation – The number of Agent Objects to create.

The number of agents to create corresponds to the number of points we have in the input dataset. All of the agents created in this step will reside in the *Places* matrix at index (0,0). We migrate these agents to indices in the matrix where input points are mapped. After the agents are at their respective initial locations, we start the respective computations for each application by iteratively spawning, migrating, and executing methods on agents.

In the list below we present key *Agent* functionality. Methods 1 through 4 are used in the closest pair of points implementation. Methods 1 through 8 are used in Voronoi diagram implementation, and methods 9 through 12 are used in the convex hull implementation.

1. callMethod() – facilitates invoking appropriate methods on each active agent. This method is called when the main program invokes the *Agents.callAll()* method. The method to be invoked is passed as an argument from the calling program.
2. migrate() – migrates agent to the next target neighborhood location based on the arguments passed.
3. spawnAgents() – spawns a specific number of agents indicated by the arguments, for each active agent. The new agents reside initially at the same place as their parent agents. We then use a *migrate()* call to migrate them to different neighborhood locations.
4. killDuplicateAgents() – there is a possibility that after spawning agents one or more agents mapped to the same input point might migrate to the places that have already been visited by another agent with the same point. This method looks at the AgentFootPrints list at the destination place on agent arrival and kills the agent if the list contains the point mapped to it.
5. checkCollision() – checks for collisions at each place with an agent. Two types of information can be discovered on collision – Voronoi edge or Voronoi vertex. This method checks for the collision of two agents first to compute Voronoi edges and then checks for the collision of three agents to compute Voronoi vertices.
6. updateVoronoiEdges() – inserts a new perpendicular bisector at both the source Voronoi sites.
7. updateVoronoiVertices() – inserts a new Voronoi vertex at three Voronoi sites that are part of this.
8. evaluateEdges() – upon collision of three agents, this method is called to compute the center of the circle formed by the three points mapped to these agents. This is returned as the Voronoi vertex.
9. initializeConvexHull() – computes the initial point location to migrate to before starting the convex hull computation, which is the point with lowest y coordinate. This method initializes the next target location for the agent to migrate to once the starting point is found.

10. computeNextPointIntConvexHull – computes the next point to migrate to based on the infinite edges in the current Voronoi region. This method picks the next infinite edge, which should be an edge between the current Voronoi site and an adjacent site, and sets the agent to migrate to the adjacent site in the next iteration.
11. evaluateAngle() – calculates the angle between three points. This method is called every time a new point is added to the convex hull points to check if the last three points in the list form a concavity.
12. killAgentAndStopExecution() – stops execution of the convex hull simulation by terminating the convex hull agent. This method is invoked when the agent reaches back to the starting point.

4.2 MapReduce and Spark

Table 4.3 indicates key data structures and classes used to represent the input within MR and Spark implementations for all four applications.

Table 4.3: Input dataset construction using MapReduce and Spark

	Closest pair of points	**Voronoi diagram**	**Convex hull**	**Delaunay triangulation**
1.	Point class			
2.	ClosestPair class – closest pair of points and their distance	ConvexHull class	VoronoiDiagram, DCEL, DirectedEdge, and Edge classes	VoronoiVertex class

From Table 4.3, we can notice that all four algorithms implemented using MR and Spark represent the input point using the Point class. The closest pair of points implementation uses the *ClosestPair* class similar to that of MASS implementation to represent the closest pair. The Voronoi diagram is represented using the *VoronoiDiagram* class, which contains a *Java*

HashMap with key as the Voronoi site and value as the DCEL data structure, representing the Voronoi regions. Additionally, the Voronoi diagram object contains a list of Voronoi vertices of this region, a reference to the incident edge, which is the edge last inserted into the region, and the direction of edges in this region. The incident edge of a region can be used to find either the next edge or the previous edge within the region. The direction of a region is set to clockwise or counterclockwise (edges will be traversed in this direction during the Voronoi diagram merge phase). The *DirectedEdge* class represents an edge in the Voronoi diagram with direction information. Adjacent Voronoi sites will share the same directed edges but with opposite directions. This instance will contain a reference to the directed edge twin, which will point to the edge in the adjacent Voronoi region to facilitate jumping from one region to another during the merge phase. The *ConvexHull* class contains the list of points that lie on the convex hull for a particular stripe and contains references to the upper and lower segments that connect two convex hulls from adjacent stripes after merging. The *VoronoiVertex* class represents Voronoi vertices in the Voronoi diagram and also contains references to the three points that the Voronoi vertex is between. The Delaunay triangulation is constructed based on the set of Voronoi vertices that the Voronoi diagram contains.

Table 4.4 shows the algorithm parallelization approaches used for each of the applications in MR and Spark.

Table 4.4: Divide and Conquer algorithms for parallelization

Closest pair of points	**Voronoi diagram, Convex hull, and Delaunay triangulation**
Divide input into stripes of three points each and merge adjacent stripes together and find closest pair of merged stripe, until left with one stripe	Divide input into stripes of two points each and merge adjacent stripes together and create unified convex hull, Voronoi diagram, and Delaunay triangulation, until left with one stripe

Now, we present the key methods used in each of the implementations using MR and Spark. The MR and Spark closest pair of points implementations use two classes to represent the points and the closest pair result. The *ClosestPair* class represents a pair of points that are the closest to each other in a given stripe. The class defines the two Point objects, the distance between them and other key attributes used in the D&C algorithm. In the iterative merging of stripes in both the platforms, we use important methods to compute the closest pair of the new combined stripe. The methods are:

1. constructPointSetWithinDelta() – constructs a list of points that are at a distance delta (argument) from the line that divides the two stripes, that belong to both stripes.
2. computeDistanceAlongStrip() – checks for points within the list of points from the above method to check if the distance is smaller than delta.

In the iterative merging of stripes step in both MapReduce and Spark of Voronoi diagram implementation, we use important methods to compute the convex hull, Voronoi diagram, and Delaunay triangulation of the new combined stripe. Key Voronoi diagram methods are:

1. mergeDiagrams() – called during the reduce phase of both MapReduce and Spark implementations. Invokes method to merge convex hulls of the two stripes first and then invokes the method to merge Voronoi diagrams of the two stripes.
2. mergeVoronoiDiagrams() – merges the Voronoi diagrams of the two stripes using the D&C approach.

Key convex hull methods are:

1. mergeConvexHulls() – merges the convex hulls of the two stripes using a D&C approach.
2. computeUpperTangent() – computes the upper tangent that connects the hulls from both stripes, as shown in Figure 3.8.
3. computeLowerTangent() – computes the lower tangent that connects the hulls from both stripes, as shown in Figure 3.8.

Chapter 5

RESULTS

We evaluated implementations for the four computational geometry algorithms in MASS, Spark, and MapReduce using three metrics: programmability, and execution time. In the sections that follow, we first present the environment and supporting software used to test the implementations. We then present data for the three metrics and analysis of the outcome.

5.1 Evaluation environment and procedures

The subsections below highlight some of the crucial elements, important considerations and assumptions that were used in the research.

5.1.1 Population

Since the problem area that we are focusing on is computational geometry, all input data will be in the form of 2D points representing multiple points on a cartesian coordinate plane. We have implemented a generator program that randomly produces the required number of data points based on the input number. Some of the constraints used for generating a random set of points include:

1. Point coordinates will be positive integer values,
2. No two points will have the same x **and** y coordinates (duplicates),
3. No three points will lie on the same line (co-linear), and
4. No four points will lie on the circumference of the same circle (co-circular).

The reason for having these constraints is that it tremendously reduces the time required to implement solutions for the algorithms considered. This allows us the choice of not having to handle edge cases created by the input data in case these constraints were not in place. For

example, having point coordinates as decimals instead of integers will require additional logic to handle mapping the points onto the Places matrix in MASS implementations. Also, we would have to create a finer mesh to incorporate decimals, which would increase the number of iterations that agents need to collide in the Voronoi diagram implementation. Thus, having the first constraint helps us focus on building an initial solution. More importantly, the second and third constraints help avoid generating input data that constitute a degenerate case for certain algorithms such as convex hull and Voronoi diagram. A degenerate case for the convex hull is when the input points contain three points that are collinear. A degenerate case for the Voronoi diagram is when the input points contain four points that lie on the circumference of the same circle (co-circular). The current implementations for these problems will not handle the degenerate cases as part of this research since it requires more involved and complex logic. Having these constraints directs focus towards the important question that this research is aiming to answer, which is that of comparison of execution time and programmability of implementations that use the three different frameworks. Handling edge cases like the ones mentioned above will not necessarily make any impact on the results.

5.1.2 Operational definition

The performance will be measured using the below-mentioned variables for MASS implementations versus MapReduce and Spark implementations. Variables to be used are i. Execution time, and ii. Programmability.

1. Execution time: The metric used to measure the execution time of different implementations is the total run time of each of the programs, which includes the time taken to perform disk reads and writes.
2. Programmability: Programmability of the solutions using the three different parallel programming models is measured by computing the following metrics:

   (a) Boilerplate code – number of lines of code required to set up the parallel programming environment
   (b) LOC – total number of lines of code

(c) Number of classes – number of classes or number of RDDs (Resilient Distributed Dataset) for Spark, that are created for the implementations.

(d) Number of methods - number of unique system defined methods invoked by implementations in each platform.

The development lifecycle phases include: i. Design implementations using MapReduce, Spark, and MASS ii. Implementation, iii. Test, and iv. Benchmark implementations.

5.1.3 Materials used

To test the implementations, a cluster of multi-core Linux based machines made available by the University of Washington (UW), was used. The cluster is physically located at the UW, Seattle campus. The size of the cluster is currently eight nodes that can be used for parallelization efforts.

5.1.4 Supporting software used

A java process profiling tool developed by Sedlacek et al. [35], was used to identify performance bottlenecks of the java implementations using MapReduce, Spark, and MASS. Metrics such as method invocation counts, CPU runtime of each method in the implementation were identified using the profiler, which aided in fine-tuning the implementations to achieve better performance. To compute programmability metrics such as the number of lines of code for each implementation, we used a plugin developed by Topinka et al. for IntelliJ IDEA IDE (Integrated Development Environment) in [42]. Hadoop MapReduce distribution version 0.20.0 [40], Apache Spark distribution version 2.4.3 [41], and MASS Java release version 1.2.1 was used to test the implementations using the respective frameworks. In addition to this, we developed two supplemental programs below:

1. A program to generate input points: this java program was used to generate input points for the computational geometry applications. This implementation includes the constraints mentioned in section 5.1.1 to eliminate duplicates and degenerate cases in

the generated points. Users are expected to specify the number of points required and the maximum value to use in x and y dimensions as program arguments.

2. A program to visualize convex hull, Voronoi diagram and Delaunay triangulation output: this java program uses the JFrame and JComponent classes from the Java AWT (Abstract Window Toolkit) package to visualize the computed convex hull, Voronoi diagram and Delaunay triangulation using components such as points and lines.

5.2 Programmability

Programmability for the three parallel programming frameworks – MASS, MapReduce, and Spark – refers to the ease of using each framework to implement applications. We measure programmability using the three metrics stated in 5.1.2.ii. In the subsections below, we present measured values for each of the programmability metrics and analyze the results.

5.2.1 Boilerplate code

Table 5.1 shows the boilerplate code percentage for the MASS, Spark, and MapReduce implementations of the closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation. We have grouped the Voronoi diagram, convex hull, and Delaunay triangulation metric into one column because all three implementations are part of a single java package, and it would be reasonable to report metrics this way instead of individually.

Table 5.1: Boilerplate code percentage for all implementations

Parallel Framework	**Closest pair of points(%)**	**Voronoi diagram, Convex hull, and Delaunay triangulation(%)**	**Average(%)**	**Total LoC**
MASS	1.02	0.33	0.68	2,382
Spark	1.04	0.23	0.64	1,539
MapReduce	5.08	2.08	3.58	2,369

The boilerplate code percentage is calculated by dividing the number of lines of code required within each implementation to set up the parallel execution framework, by the total number of lines of code. For MASS, the boilerplate code could include reading the *nodes.xml* file, setting up the logging level, initializing the MASS execution framework, etc. For Spark, the boilerplate code could constitute creating the *JavaSparkContext* instance using which the workers interact with the driver, setting parameters for the jobs such as application name, etc., and stopping the spark context at the end of the program. For MapReduce, it could include code to create *JobConf* instances for each of the MR job that we define in the application, define the file system path (HDFS), setup mapper and reducer classes for each job, setup input and output formats for each job etc. From the table, we can infer that MASS and Spark have comparable results for average boilerplate code percentage among all the implementations. Whereas the MR implementation has a higher percentage of code required to set up its parallel execution environment. Even though MASS and MapReduce have approximately close values for the total number of lines of code across all four apps, MR requires additional lines of boilerplate code compared to MASS. This is due to the fact that for MR implementation, we have to specify input/ output formats and paths, input/output key and value classes, mapper and reducer classes, etc. But MASS does not have the need for most of these parameters. The input/output paths are accepted as program command-line

arguments, and all of the setup information related to master and worker nodes are obtained from the nodes.xml file. This eliminates the amount of work the application developer has to do in terms of setting up the execution environment and allows them to focus on the application logic instead.

5.2.2 Lines of Code

Table 5.2 shows the total number of lines of code for the MASS, Spark, and MapReduce implementations of the closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation.

Table 5.2: Lines of code for all implementations

Parallel Framework	**Closest pair of points**	**Voronoi diagram, Convex hull, and Delaunay triangulation**	**Total LoC**
MASS	585	1,797	2,382
Spark	286	1,253	1,539
MapReduce	688	1,681	2,369

The total number of lines of code is determined using a plugin called *Statistic* within the IntelliJ IDE [42]. We have considered only the code lines for the above metrics and eliminated comments and empty lines. We can observe from the table that MASS implementations required approximately as many lines of code as that of the MR implementations. Even though the closest pair of points MASS implementation has fewer LoC compared to MR, the Voronoi diagram, convex hull, and Delaunay triangulation LoC is higher than that of MR implementation. This is because of the MASS Voronoi diagram implementation:

1. defines 2D space management as part of the application. This includes logic to create a fine mesh for the Places matrix-based, mapping input points to each place, initiating

and migrating agents equal to number of input points and migrating them to the places that have a point mapped, and

2. has defined the logic for agent migration to Von Neumann and Moore neighborhoods.

Since these operations are common to any application using 2D points, in future, if the library provides these operations as part of its API, we could reduce the total LoC required for MASS implementations.

5.2.3 Number of classes

Table 5.3 shows the total number of classes defined for the MASS, Spark, and MapReduce implementations of the closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation.

Table 5.3: Number of classes for all implementations

Parallel Framework	**Closest pair of points**	**Voronoi diagram, Convex hull, and Delaunay triangulation**
MASS	10	14
Spark	11	10
MapReduce	12	18

We have considered the number of java classes that were required for each of the implementations in the three frameworks for this metric. We can observe from Table 5.3 that MASS implementations required fewer classes compared to that MapReduce. This is due to the fact that MR implementation for the Voronoi diagram required the creation of complex data structures to represent it in memory and define the functionality required by the D&C algorithm. However, the MASS Voronoi diagram implementation maintains all of the required data structures in static entities called *Places* in terms of java collections, thereby

eliminating the need to define such complex data structures. Furthermore, MR implementation defines one mapper, and one reducer class for each of the MR job run in the application. The closest pair of points and Voronoi diagram, convex hull, and Delaunay triangulation implementations each run a chain of three MR jobs, thereby requiring a total of 6 classes each just to define the mappers and reducers. Clearly, MASS implementations have an advantage in this aspect compared to MR implementations.

5.2.4 Number of methods

Table 5.4 shows the number of unique methods invoked by the closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation implementations in MASS, Spark, and MapReduce. We have included all library methods invoked that is not part of any boilerplate code. Only methods that launch parallel operations on each of the data partitions are considered.

Table 5.4: Number of methods for all implementations

Parallel Framework	**Closest pair of points**	**Voronoi diagram, Convex hull, and Delaunay triangulation**
MASS	2	3
Spark	9	10
MapReduce	2	2

We can note from Table 5.4 that MASS and MapReduce have comparable values for the number of methods invoked during the application execution. This is mainly because for MASS parallel operations are executed either on the distributed static entities (*Places*) or on the dynamic entities (*Agents*). Various operations can be performed on both these entities using the *callAll()* method and barrier synchronization is performed using the *manageAll()* method. In addition to these two methods, in the Voronoi diagram implementation, we use

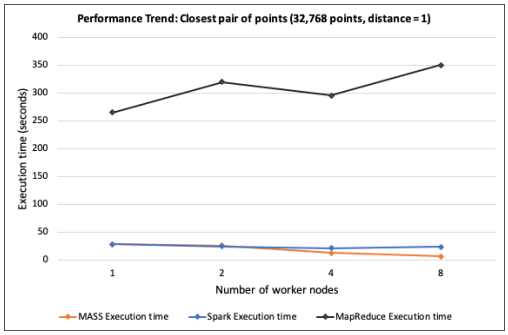
the *doAll()* method for asynchronous agent migration. For MapReduce, we use the *map()* and *reduce()* methods to perform parallel operations across data partitions. On the other hand, Spark has a larger number of method invocations. This is because of the parallel programming model of the framework, which requires data to be transformed iteratively until the final result is obtained. Thus with respect to the number of methods, MASS has better programmability compared to Spark.

5.3 Execution time

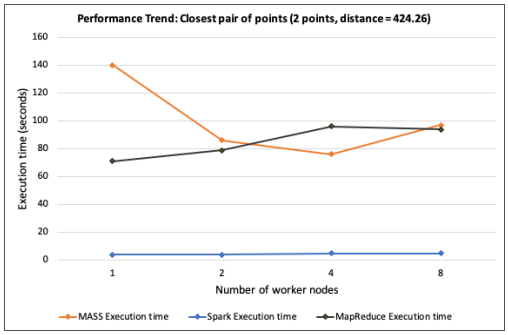
In this section, we present the measurements taken for execution time by running each of the implementations for the three platforms. We also present an analysis of these different execution times. We have used different node configurations (1, 2, 4, and 8 nodes and 1 thread per node) for the parallel computing cluster to measure the performance of each of the implementations across the three parallel platforms. We were not able to test the MASS implementations with more than 1 thread per node due to an exception in multi-threaded configuration with MASS. Thus to provide a level ground, we have configured the MapReduce and Spark platforms to use 1 thread per node while measuring the execution time required for the implementations.

5.3.1 Closest pair of points

Figure 5.1 shows the closest pair of points execution performance of the three implementations with two different inputs.



(a) Input points = 32,768 and distance = 1



(b) Input points = 2 and distance = 424.26

Figure 5.1: Performance trends for closest pair of points implementations

Figure 5.1(a) shows the trend for an input size of 32,768 points on different node configurations. The closest pair in this input had a Euclidean distance value of 1. We can observe from the graph that MASS and Spark implementations have comparable performances with such a large input size. But MapReduce implementation performs poorly due to a large number of input points. Figure 5.1(b) shows the trend for an input size of 2 points on different node configurations. However, this time the Euclidean distance between the two points is 424.26, which is much higher (threshold case for MASS, at which the execution time is no longer better than MapReduce). In both scenarios, Spark implementation outperforms MASS and MR implementations. Additionally, in the second scenario, as expected, with the reduction in the input size (number of points), the MR implementation takes less time compared to that of the first run with 32,768 points. But the MASS implementation has a tangible increase in time compared to the previous run. This is due to the fact that we use agent migration to discover the closest pair of points. As the distance between the closest pair increase in the input data, the agents have to travel further to be able to collide with other agents. This causes the program to run more iterations of agent spawn and migrate compared to when the input points have a closest pair with a shorter Euclidean distance. This is a drawback for the current MASS implementation. This performance bottleneck can be alleviated in the future with a more compact and efficient approach to 2D space

management in MASS closest pair of points implementation.

5.3.2 Voronoi diagram

Figure 5.2 shows the Voronoi diagram execution performance of the three implementations.

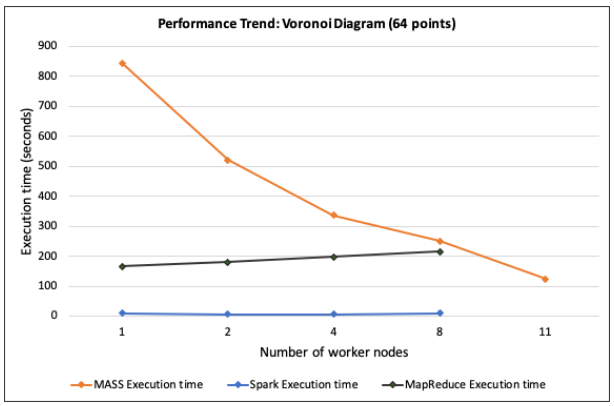


Figure 5.2: Performance trends for Voronoi diagram implementations

Since we were unable to test Spark and MapReduce implementations on 11 node configuration, the graph does not include data points for these implementations on 11 nodes. We can observe from the above graph that the MASS Voronoi diagram implementation takes longer to execute compared to Spark and MapReduce implementations. Multiple performance enhancements were made to the MASS Voronoi diagram implementation, including:

1. monitoring time spent by the application within each method in the code and adding in performance tweaks,
2. switching synchronous agent migration with asynchronous agent migration with the use of *doAll()*, thereby giving complete control to agents to spawn, migrate and execute functionality on their own. This eliminates the synchronization that happens when control returns to the main program while using *callAll()* and *manageAll()* functions instead, and,
3. eliminating multiple collision checks for the same set of points by maintaining a common

data structure among places to store information about three-point combinations for which collision has already been checked and a Voronoi vertex is identified etc.

We observed reduction in execution time after the addition of the above performance tweaks, but not so much that it would surpass the execution time of implementations in the other two platforms. Some of the areas for improvement for the current MASS Voronoi diagram implementation are presented below:

1. Allowing multi-threaded configuration for the MASS implementation could improve the execution performance of the MASS Voronoi diagram implementation, by leveraging the increase in parallelism.
2. The current implementation spends a significant amount of time checking for collisions of two or three agents in each iteration. This is one of the reasons for the longer execution time. Since agents migrate in the form of a ripple for the Voronoi diagram implementation, there could be multiple places where collisions between the same set of two or three points will be checked. But once a vertex is computed and recorded in its respective Voronoi region, further checks for collision for the same set of points need not be performed. In our current implementation, only places that have a point mapped to them will have information about the Voronoi, which cannot be accessed by other agents or places. Thus, in order to eliminate such duplicate checks, we created a static variable that contains all Voronoi vertices discovered so far. But since static variables are tied to the Java JVM (Java Virtual Machine) context, there will be independent copies of these variables on each of the nodes in the cluster. So, if a Voronoi vertex is discovered in say, for example, node 3, node 1 will not have this information in its copy and hence will still perform collision checks for the same set of points.
3. 2D space management is an issue because, as the distance between points in the input grows larger, the agents need longer to collide with other agents to gather information about potential Voronoi edges and vertices.
4. Input points are mapped to *Places* matrix based on the correspondence between their coordinates and the place indices. The *Places* matrix is partitioned by calculating the

total number of place objects (rows * columns) and dividing it by the number of nodes available on the cluster. So, for example, if we have a 4x4 matrix, there are a total of 16 places, and if we have 4 nodes on the cluster, each node will be assigned 16/4 = 4 places. If the first four places contain all of the input points, then the other three nodes will not be participating in the computation. Thus, load balancing is another issue with the current implementation.

Although we have used input data of small size (64 points) for the Voronoi diagram implementations, we expect to test these applications in the future with larger input size. When MASS implementation is no longer behind MapReduce and Spark (as shown in Figure 5.2) with smaller input data, we could increase the data size and test for a higher number of points.

5.3.3 Convex hull

Figure 5.3 shows the convex hull execution performance of the three implementations.

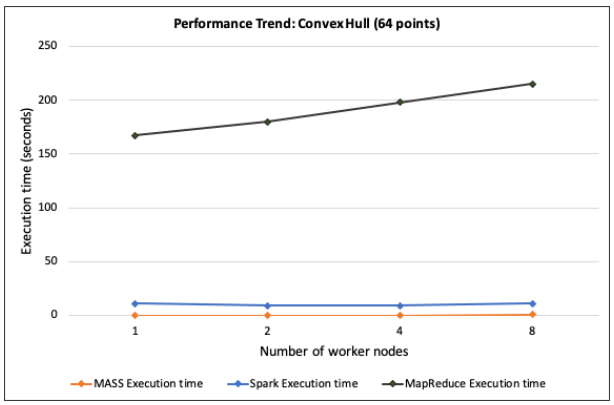


Figure 5.3: Performance trends for convex hull implementations

We can observe from the above graph that the MASS convex hull implementation outperforms Spark and MR implementations in terms of execution time. This is due to the fact that the convex hull implementation uses a single agent to traverse the unbounded regions of the Voronoi diagram to compute the convex hull. Leveraging the unique features that the

Voronoi diagram of an input dataset possesses, our implementation uses an efficient algorithm to compute the convex hull. The Spark & MR implementations use a D&C approach to construct the convex hull of the input points. Since we merge two stripes at a time within these parallel implementations, we end up processing all of the input points before finding the final convex hull. However, in MASS implementation, the convex hull agent starts from the point with the lowest y coordinate and traverses the Voronoi region of this point to find the next unbounded region to migrate to. It continues migration until it reaches its starting point. With this implementation, the agent hits only the points that lie on the convex hull and few others that have an unbounded Voronoi region even though they should be bounded. We discussed the reason for such cases in previous sections and will skip the discussion here. Due to this fact, the MASS convex hull implementation runs faster than the Spark and MR implementations.

5.3.4 Delaunay triangulation

Figure 5.4 shows the Delaunay triangulation execution performance of the three implementations. Since we were unable to test Spark and MapReduce implementations on 11 node configuration, the graph does not include data points for these implementations on 11 nodes. The Delaunay triangulation implementations in all of the three platforms follow the same strategy. We maintain information about the Voronoi sites that are part of each Voronoi vertex discovered during the program execution in additional data structures. We then use these data structures to construct the dual of the Voronoi diagram, which is the Delaunay triangulation. Due to the fact that we should complete the execution of the Voronoi diagram program to obtain the Delaunay triangulation results, we have a similar performance trend for the Delaunay triangulation implementations as that of Voronoi diagram. Similar arguments as that of MASS Voronoi diagram implementation can be made to account for the fact that the current MASS Delaunay triangulation implementation performs poorly compared to that of Spark and MR. There is scope for improvement in the future in the areas of 2D space management and agent migration support.

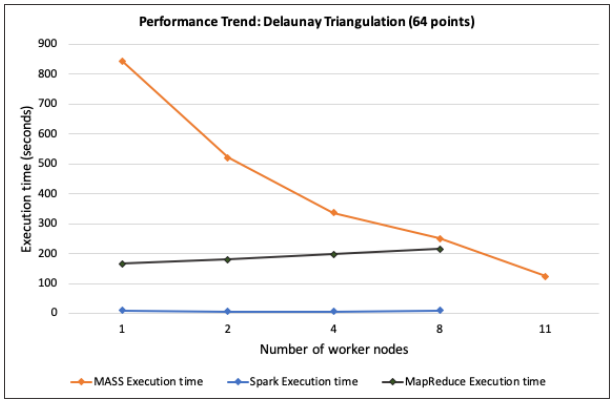


Figure 5.4: Performance trends for Delaunay triangulation implementations

5.4 Operability

In this section, we discuss how we were able to apply different algorithms to the same data structure in the implementations for the Voronoi diagram, convex hull, and Delaunay triangulation. Since the Voronoi diagram has multiple important properties, we were able to leverage them to perform multiple analyses on the same input data.

5.4.1 MASS

The MASS Voronoi diagram implementation uses agents to spawn, migrate, and collide with other agents to construct the Voronoi diagram of the input dataset. The *Places* contains all data structures required to represent and record all of the information for the Voronoi diagram. Since Places are static entities within the distributed cluster, we maintain the bulk of the data structures within these entities instead of within the dynamic *Agents* entities. With the MASS library, we have the advantage of creating the 2D space with the input points mapped to it in terms of *Places* and then reuse this to apply multiple analyses. For all of the three implementations – Voronoi diagram, convex hull, and Delaunay triangulation – we only read the input data once and map it to places. After the Voronoi diagram is computed, we terminate all of the Voronoi diagram agents, which marks the completion of the first analysis on the input data. Now the Places that have points mapped to them will contain the final Voronoi diagram as a *Java HashMap* with the keys as the Voronoi sites and values

as a *Java HashSet* of edges within this Voronoi region. Additionally, each place will also contain information about the Voronoi vertices in their respective regions as a *Java HashSet*. We use the Voronoi diagram HashMap as the base to compute the convex hull, without having to read any intermediate data from disk. Following this, we use the Voronoi vertices HashSet to construct the Delaunay triangulation of the input dataset. Finally, when all three analyses are complete, we output all of the results to disk and terminate the program. Thus, the advantage of using the MASS library for these types of analyses is that it eliminates the need for disk input/output for intermediate data, thereby reducing execution time to a certain level.

5.4.2 MapReduce & Spark

The MapReduce and Spark implementations, on the other hand, use a D&C approach to compute the Voronoi diagram, convex hull, and Delaunay triangulation for the input dataset. In these implementations, we construct the convex hull, Voronoi diagram, and Delaunay triangulation in tandem. But within each iteration of the merging adjacent stripes, we first merge the convex hulls of the stripes and then use this to construct the Voronoi diagram for the combined stripes. We have designed data structures to support the construction of the Voronoi diagram based on the unified convex hull in each iteration. Spark provides the capability to perform multiple analyses on in-memory data similar to MASS. Although we were able to implement appropriate data structures to leverage the properties of the Voronoi diagram and convex hull to perform multiple analyses on the input data, the MR implementation heavily relies on disk input/ output for each MR job. This introduces the need to read/ write intermediate data from and to disk, respectively. As a result of which, we cannot perform multiple analyses on in-memory data using MapReduce.

5.5 MASS library strengths and challenges

One of the key goals of this research was to assess the strengths of using the MASS library for parallelizing applications using 2D spatial data. MASS implementations exhibited strengths in certain aspects but presented challenges in certain others that needed focus. As mentioned

by Fukuda et al. in [11], the addition of features such as *asynchronous* agent migration to the MASS Java library, has relieved the performance bottlenecks for applications built using the library. Our experiments demonstrated the following strengths of the MASS library and also presented a few challenges that are summarized below.

Programmability: The Lines of Code (LoC) required to set up the parallel execution environment in MASS implementations was generally comparable to that of MapReduce. The MASS program logic in all of the four implementations are compact and intuitive compared to the other two platforms and does not require complex data structures and D&C logic. Moreover, MASS implementations require a smaller percentage of boilerplate code on an average and fewer classes in each application, compared to that of MR implementations. However, there were certain challenges with respect to programmability. Collective/ reductive operations in order to construct the final results are currently controlled by the *main* program, which is an overhead. In addition to that, MASS needs more LoC than Spark when constructing distributed data structures with MASS *Places* [11]. Furthermore, since we have implemented 2D space management and added support for agent migration to different neighborhoods within the Places matrix, the LoC for MASS based implementations are on the higher side.

Execution time: A significant speedup was achieved with the use of the automatic agent migration feature in MASS Voronoi diagram implementation. Additionally, the convex hull implementation outperformed the Spark and MR implementations. However, With respect to execution time, there were some challenges that we faced with MASS implementations. When the distance between the closest pair increases, we saw the MASS closest pair of points program take longer to compute the results due to agents taking more iterations of spawn and migrate to discover the closest pair. The MASS Voronoi diagram and Delaunay triangulation implementations have a similar drawback with respect to space management. When the distance between points increases in the input dataset, it takes longer for the agents to identify Voronoi edges and vertices, which directly affects the execution time. In addition to this, there is another bottleneck on execution time stemming from the fact that we create a fine mesh (every adjacent integer has ten intervals between them to capture

mid-points) for the *Places* matrix.

5.6 MASS performance with increase in cluster nodes

All of the implementations were tested on a cluster of eight nodes (hermes cluster). However, towards the end of this research, we were able to test the MASS implementations on a different cluster with eleven nodes (cssmpi cluster). As expected, with the addition of nodes MASS implementations performed better compared to the previous executions on eight nodes. Figure 5.5 shows the performance trend for three applications - Voronoi diagram, convex hull, and Delaunay triangulation for the two clusters with eight and eleven nodes.

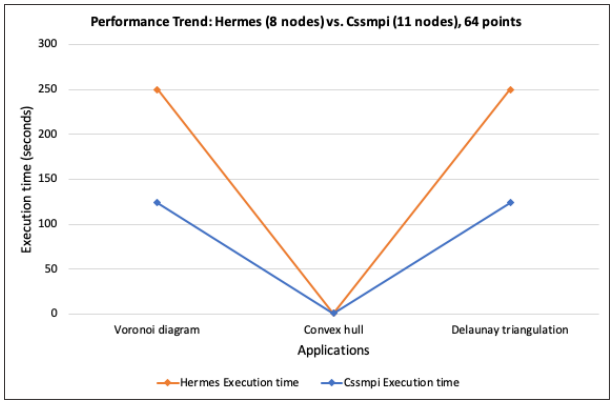


Figure 5.5: Performance trends for MASS implementations on hermes and cssmpi clusters

From the above graph, we can observe that with addition of nodes, the Voronoi diagram execution time reduced by 50%. We can see a similar drop in execution time for Delaunay triangulation, since the time it takes to compute it is the same as the Voronoi diagram. We were not able to compare the execution time of MASS applications against MapReduce and Spark due to setup issues on the new cluster. But the fact that MASS implementations' execution time decreases with increase in number of nodes is promising. Also, currently the input points generated have integer x and y co-ordinates. The constraints we have added to eliminate some input points that create degenerate cases for convex hull and Voronoi diagram,

restrict the number of points that can be created for a given range of x and y dimensions. In future, after 2D space management is added to the MASS library, the capability to handle decimals will be available. Using this feature, we could generate larger number of points to test the implementations on.

Chapter 6
CONCLUSION

In order to identify the application areas where the MASS library can be a better parallelization framework, our work explored the problem area of computational geometry. Although there is some research done in this area using the MapReduce framework, there is no work done using Spark and MASS. Furthermore, benchmarks for the parallelized solutions in these three platforms are not available. This research is a steppingstone to parallelizing computational geometry solutions using ABM. The major contribution of this research is:

1. We have provided ABM solutions for four computational geometry problems using the MASS library, which has not been worked on previously,
2. We have compared and analyzed the performance of MASS implementations against big data analysis tools such as Spark and MapReduce,
3. This research established that using MASS library users can develop applications intuitively which is an advantage for people from non-computing backgrounds, and
4. We proved that multiple analyses could be performed using MASS, without having to construct the *Places* matrix again for each analysis. Also, we do not have to depend on disk-based operations, and we need not develop complex data structures and D&C algorithms for MASS based parallelization.

In the sections below, we discuss the potential areas where future work can be done based on the outcome of this research.

6.0.1 Future Work

Multiple research opportunities have presented themselves as an outcome of this research. Following areas are some of the most significant ones, where further research could address

the current challenges facing the MASS based computational geometry implementations:

1. Support for 2D and 3D space management: the MASS library currently does not offer support for space management. Adding this support could tremendously reduce the application development time and enhance programmability by reducing the application LoC.
2. Support for agent migration: support could be added to the MASS core library for agent migration to different neighborhood locations using patterns for migration such as Von Neumann, Moore, etc.
3. Support for shared data structures among *Places* and *Agents*: Adding support for storing a shared variable among places will eliminate overhead caused by duplicate collision checks as discussed in section 5.3.2 and improve the execution performance.
4. Support for handling decimals: Support could be added to MASS closest pair of points, Voronoi diagram, convex hull, and Delaunay triangulation applications to handle decimal inputs, which will enhance the applicability to a wider type of input data.
5. Use of parallel input/ output: In future, the four MASS based implementations could be tested with parallel input/ output to check if performance improves.
6. Utilizing multiple threads per node: In future, the MASS based implementations could be tested with more than one thread per node to utilize the computing capability provided by all cores on a node. This could increase the performance of MASS closest pair of points, Voronoi diagram, and Delaunay triangulation implementations.

We believe that with the availability of benchmarks provided by this research, scientists from non-computing backgrounds will have the benefit of choice with respect to parallel programming libraries. While there are some challenges in using the MASS library, the advantages in intuitive programming capability and ability to perform multiple analyses on in-memory distributed data, prove to outweigh them. Furthermore, with the addition of 2D space management and support for agent migration within the MASS library in the future, MASS can become a better contender for big data analysis.

BIBLIOGRAPHY

[1] Algorithm Tutor. An efficient way of merging two convex hulls — Algorithm Tutor.

[2] Pierre Alliez and Andreas Fabri. Cgal: the computational geometry algorithms library. *ACM SIGGRAPH 2016 Courses*, pages 1–8, 07 2016.

[3] Mikhail J Atallah. Parallel Techniques for Computational Geometry. *Proceedings of the IEEE*, 80.9:1435–1448, 1992.

[4] Vicente H F Batista, David L Millman, Sylvain Pion, and Johannes Singler. Computational Geometry : Theory and Applications Parallel geometric algorithms for multi-core computers . *Computational Geometry: Theory and Applications*, 43(8):663–677, 2010.

[5] Nathan Bell and Jared Hoberock. Thrust: Productivity-Oriented Library for CUDA. *GPU computing gems Jade edition*, 7:359–371, 2012.

[6] Cloudera.com. Spark execution model.

[7] Ahmed Eldawy. SpatialHadoop : A MapReduce Framework for Spatial Data . *2015 IEEE 31st international conference on Data Engineering*, 1:1352–1363.

[8] Ahmed Eldawy. CG _ Hadoop : Computational Geometry in MapReduce. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 294–303, 2013.

[9] John Emau, Timothy Chuang, and Munehiro Fukuda. A multi-process library for multi-agent and spatial simulation. *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, pages 369 – 375, 09 2011.

[10] Munehiro Fukuda, Collin Gordon, Utku Mert, and Matthew Sell. An agent-based computational framework for distributed data analysis. *Computer*, 53(3):16–25, 2020.

[11] Munehiro Fukuda and Wooyoung Kim. OAC Core: Small: RUI: Agent-Based Scientific Data Discovery. (3):1–21, 2019.

[12] Munehiro Fukuda and Distributed Systems Lab. MASS Java Manual.

[13] Mohammad Ghodsi and Mehdi Sharifzadeh. ParLeda : A Library for Parallel Processing in Computational Geometry Applications 1 Introduction. *International Journal of Engineering*, (September 2001), 2014.

[14] Michael T Goodrich. Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry. *arXiv preprint*, pages 1004–4708, 2010.

[15] Collin Gordon, Utku Mert, Matthew Sell, and Munehiro Fukuda. Implementation Techniques to Parallelize Agent-Based Graph Analysis. *International Conference on Practical Applications of Agents and Multi-Agent Systems*, pages 3–14, 2019.

[16] Google Inc. Gson, 2008.

[17] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly Media, Inc., 2015.

[18] Kent State University. Voronoi Diagram using Divide-and-Conquer Paradigm.

[19] Matthew Kipps, Wooyoung Kim, and Munehiro Fukuda. Agent and spatial based parallelization of biological network motif search. *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, HPCC-CSS-ICESS 2015*, pages 786–791, 2015.

[20] Sourav Kumar, Sumit; Gulati. *Apache Spark 2.x for Java Developers*. Packt Publishing Ltd, 2017.

[21] Yuan Li, Ahmed Eldawy, Jie Xue, Nadezda Knorozova, Mohamed F Mokbel, and Ravi Janardan. Scalable computational geometry in MapReduce. *The VLDB Journal*, 28.4:523–548, 2019.

[22] Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. 2010.

[23] Berna L. Massingill, Beverly A. Sanders, and Timothy G. Mattson. *Patterns for Parallel Programming*. Pearson Education, 2004.

[24] David M Mount. CMSC 754 Computational Geometry 1. Technical report, 2012.

[25] Stefan Näher and Kurt Mehlhorn. LEDA: A Platform for Combinatorial and Geometric Computing. 1999.

[26] Mahmoud Parsian. *Data Algorithms*. O'Reilly Media, Inc., 2015.

[27] Anmol Paudel. Parallelization of Plane Sweep based Voronoi Construction with Compiler Directives. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 1:997–1007, 2019.

[28] Tom Peterka and Carolyn Phillips. High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation. *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 997–1007, 2014.

[29] Tom Peterka and Robert Ross. Versatile Communication Algorithms for Data Analysis. *European MPI Users' Group Meeting*, pages 275–284, 2012.

[30] Tom Peterka, Robert Ross, Wesley Kendall, Han-wei Shen, and Teng-yok Lee. Scalable Parallel Building Blocks for Custom Data Analysis. *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 105–112, 2011.

[31] F P Preparata and S J Hong. Convex Hulls of Finite Sets of Points in Two and Three Dimensions. *Communications of the ACM*, 20.2(2):87–92, 1977.

[32] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer Science Business Media., 1993.

[33] Meng Qi, Ke Yan, and Yuanjie Zheng. GPredicates : GPU Implementation of Robust and Adaptive Floating-Point Predicates for Computational Geometry. *IEEE Access*, 7:60868–60876, 2019.

[34] Nicolas Ray, Dmitry Sokolov, Université De Lorraine, Sylvain Lefebvre, Bruno Lévy, and Université De Lorraine. Meshless Voronoi on the GPU. volume 37.6, pages 1–12, 2018.

[35] Jiri Sedlacek and Tomas Hurka. VisualVM, 2017.

[36] Rahil Sharma. Shared and distributed memory parallel algorithms to solve big data problems in biological , social network and spatial domain applications. 2016.

[37] Alex Shavlovsky. VoronoiDiagramJavaRecursive/src/voronoi at master · alexshavlovsky/VoronoiDiagramJavaRecursive · GitHub.

[38] Yun-Ming Shih, Collin Gordon, Munehiro Fukuda, Jasper van de Ven, and Christian Freska. Translation of string-and-pin-based shortest path construction into data-scalable agent-based computational models. *2018 Winter Simulation Conference (WSC)*, pages 881–892, 2018.

[39] Future Studio. Gson — Getting Started with Java-JSON Serialization & Deserialization.

[40] The Apache Software Foundation. Apache Hadoop.

[41] The Apache Software Foundation. Apache Spark™ - Unified Analytics Engine for Big Data.

[42] Tomas Topinka. Statistic Plugin for IntelliJ, 2000.

[43] UCAR. Unidata — NetCDF.

[44] Tom White. *Hadoop, The Definitive Guide*. O'Reilly Media, Inc.

[45] Wikipedia.org. Delaunay triangulation - Wikipedia.

[46] Wikipedia.org. Moore neighborhood - Wikipedia.

[47] Wikipedia.org. Von Neumann neighborhood - Wikipedia.

[48] Wikipedia.org. Voronoi diagram - Wikipedia.

[49] Wikipedia.org. Wikipedia.

[50] Jason Woodring, Matthew Sell, Munehiro Fukuda, Hazeline Asuncion, and Eric Salathe. A Multi-agent Parallel Approach to Analyzing Large Climate Data Sets. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1639–1648, 2017.

[51] Erik Wynters. Examples from computational geometry that demonstrate the potential of using the thrust library to implement parallel processing on GPUs. *Journal of Computing Sciences in Colleges*, 28.6:148–155.

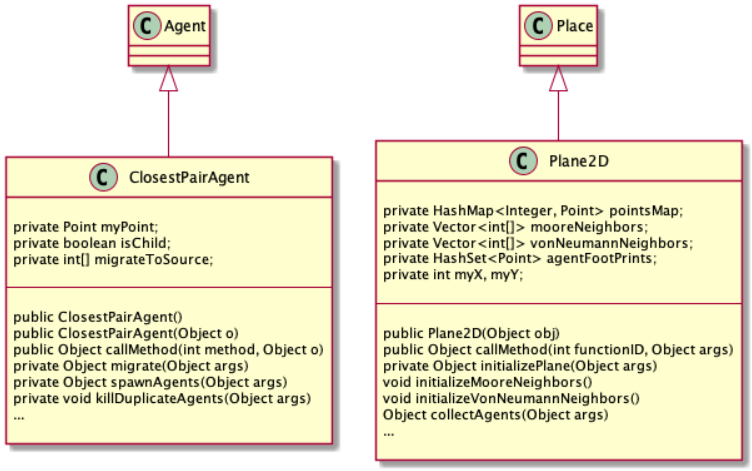
[52] Ying Xia, Xiaobing Wu, Xu Zhang, and Hae Young Bae. Parallel Voronoi Diagram Construction Method with MapReduce. *Parallel Computing*, 3(3):655–661, 2016.

[53] Justin Zobel. *Writing for Computer Science*. Springer Publishing Company, Incorporated., 2015.

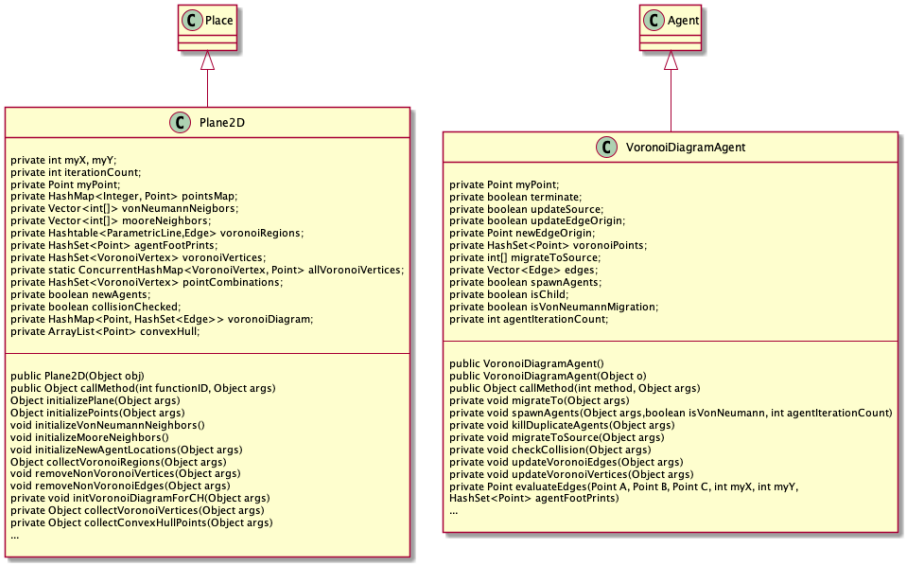
Appendix A

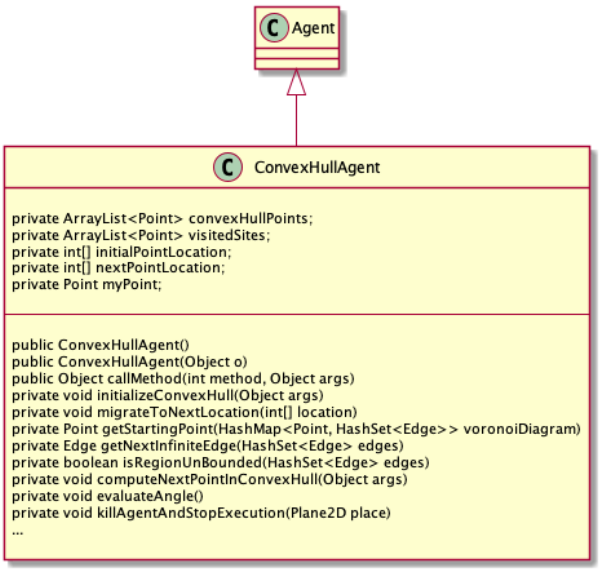
UML DIAGRAMS

I: MASS closest pair of points key classes

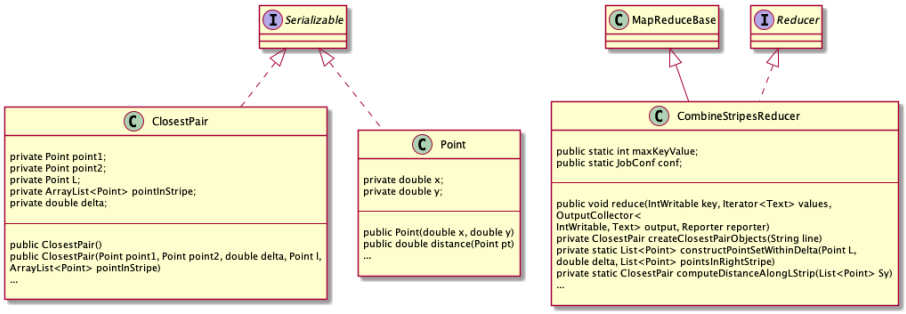


II: MASS Voronoi diagram, convex hull, and Delaunay triangulation key classes

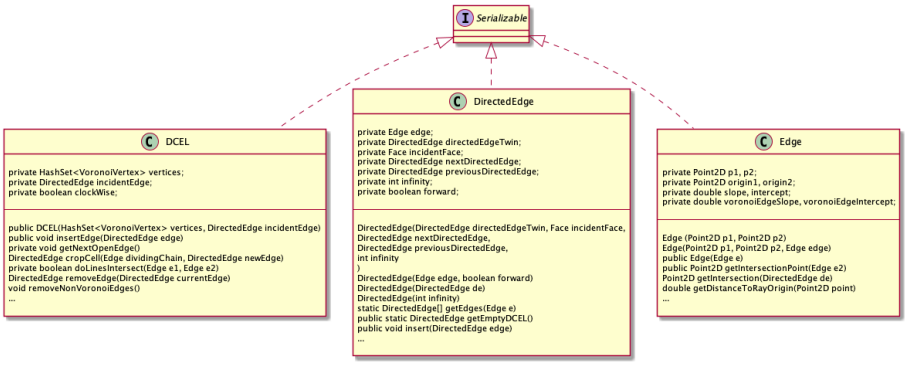


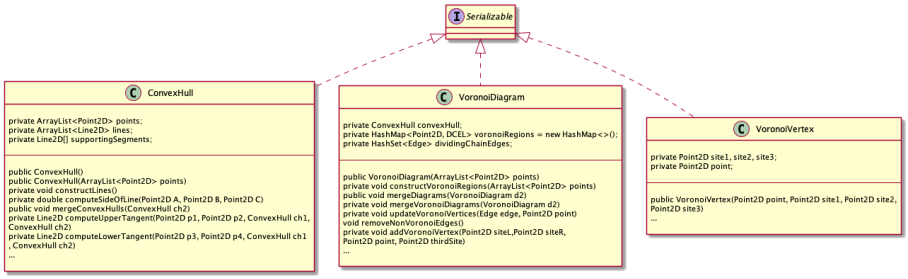


III: MapReduce & Spark closest pair of points key classes



IV: MapReduce & Spark Voronoi diagram, convex hull, and Delaunay triangulation key classes

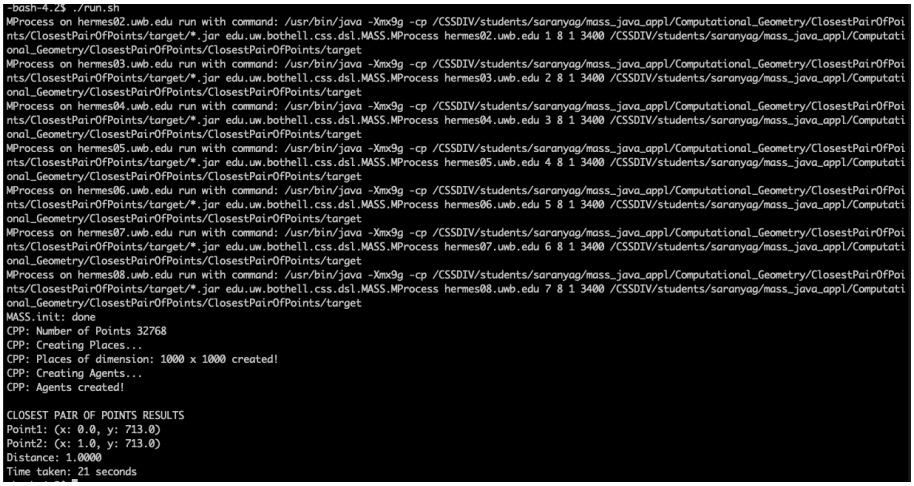



Serializable
ConvexHull
private ArrayList<Point2D> points;
private ArrayList<Line2D> lines;
private Line2D[] supportingSegments;
public ConvexHull()
public ConvexHull(ArrayList<Point2D> points)
private void constructLines()
private double computeSideOfLine(Point2D A, Point2D B, Point2D C)
public void mergeConvexHulls(ConvexHull ch2)
private Line2D computeUpperTangent(Point2D p1, Point2D p2, ConvexHull ch1, ConvexHull ch2)
private Line2D computeLowerTangent(Point2D p3, Point2D p4, ConvexHull ch1, ConvexHull ch2)
...
VoronoiDiagram
private ConvexHull convexHull;
private HashMap<Point2D, DCEL> voronoiRegions = new HashMap<>();
private HashSet<Edge> dividingChainEdges;
public VoronoiDiagram(ArrayList<Point2D> points)
private void constructVoronoiRegions(ArrayList<Point2D> points)
public void mergeDiagrams(VoronoiDiagram d2)
private void mergeVoronoiDiagrams(VoronoiDiagram d2)
private void updateVoronoiVertices(Edge edge, Point2D point)
void removeNonVoronoiEdges()
private void addVoronoiVertex(Point2D siteL,Point2D siteR, Point2D point, Point2D thirdSite)
...
VoronoiVertex
private Point2D site1, site2, site3;
private Point2D point;
public VoronoiVertex(Point2D point, Point2D site1, Point2D site2, Point2D site3)
...

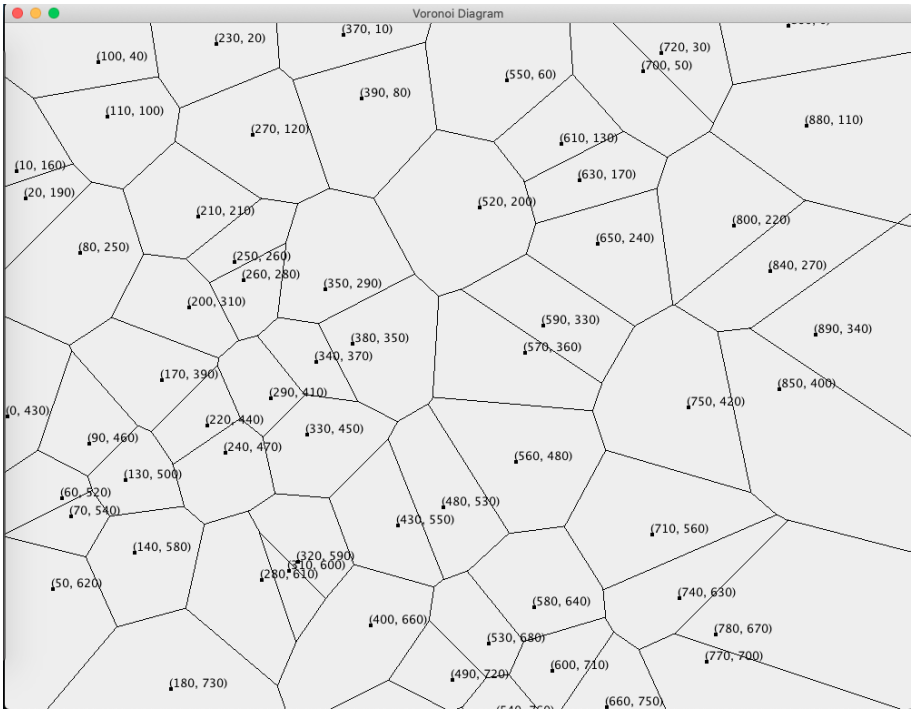
Appendix B

OUTPUT SCREENSHOTS

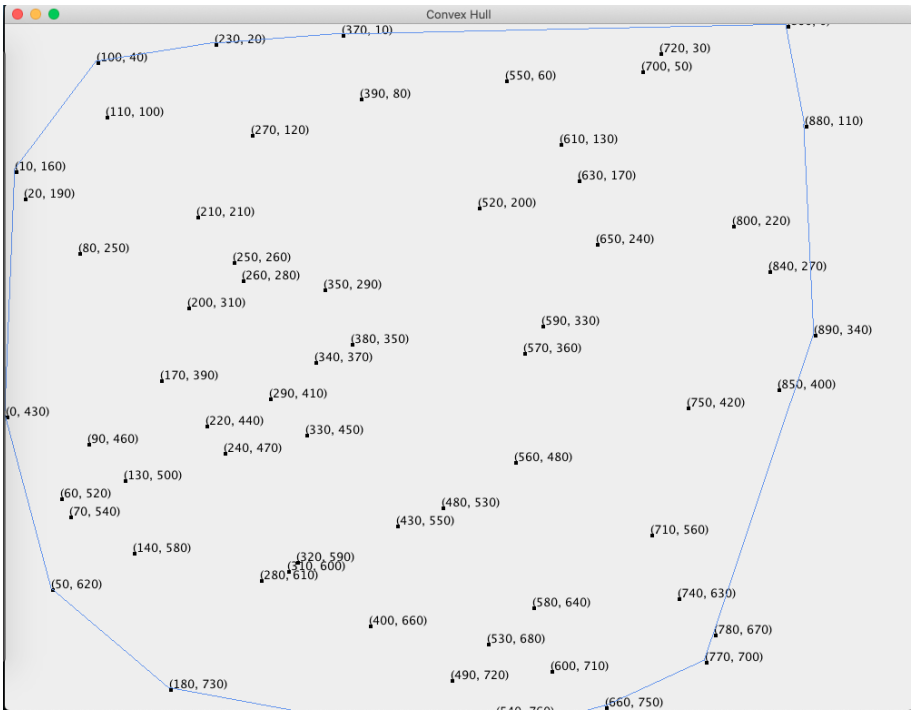
I: Closest pair of points output



II: Voronoi diagram output



III: Convex hull output



IV: Delaunay triangulation output