AGENT-BASED COMPUTATIONAL GEOMETRY

Leung-Tsan Ng

A white paper
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2023

Committee:
Prof. Munehiro Fukuda, Chair
Prof. Kelvin Sung
Prof. Thamilarasu

University of Washington

Abstract

Parallelization of computational Geometry Algorithms

Leung Tsan Ng

Chair of the Supervisory Committee:

Dr. Munehiro Fukuda

Computing and Software Systems

Computational geometry is a study of efficient algorithms for solving geometric problems. The aim of it is to optimize the algorithms as it always involves a very large dataset in the real world. For such a dataset, a little improvement in the time complexity of algorithms can save a lot of time. Besides this, parallelization is another way to shorten the computational time. Instead of optimizing the time complexity, we can assign the sub-problems to different computing nodes and let them compute the problem individually. As a result, the division of workload can reduce the computational time largely. However, many computational geometry algorithms, such as Convex Hull, Voronoi Diagram, and Closest Pair, have been researched for many years, and their performance had been tuned up to be very good in sequential versions. It is very difficult to compare the parallel version and sequential version in this case. One major advantage of parallel programming is the amount of space. As we scale up the number of computing resources, the space such as memory and disk space increases. Therefore, parallel programming can handle much larger datasets. In this project, we focus on extending the previous projects including Euclidean Shortest Path and Voronoi Diagram that were done by previous students. We improve

these two problems by using different approaches to handle larger amounts of data. The implementation is done with MapReduce, Spark, and Multi-Agent Spatial Simulation (MASS). These three frameworks have their own features. This project shows their performance on different geometric problems and compares their advantages and disadvantages.

TABLE OF CONTENTS

Contents

LIST OF FIGURES

Chapter 1: Introduction

1.1 Overview

Computational geometry is part of the study of computer science that consists of solving geometry algorithms. In multi-dimensional problems, it becomes much more challenging, and this is where the researchers are interested. In many studies, some common computational problems including the Closest Pair of Points, Convex Hull, Voronoi Diagrams, Euclidean Shortest Path, and so on, are being studied and their computational complexity is improved over years. These researches are very important as every minor complexity improvement can result in large differences in time cost especially when the data set is huge. Dealing large data sets with sequential computing has one critical limitation as it bounds to only one computing node. To obtain further improvement for larger datasets, parallelization is introduced. Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously[1]. When a big problem can be divided into multiple smaller ones, parallel computing has the potential to speed up the performance up to 20 times faster than the normal sequential version according to Amdahl's Law[2]. In computational geometry, some of the problems can be entirely or partially split into serval sub-problems. These problems are beneficial from parallel programming. Even though many problems can be benefited from parallel programming, some of the problems are not because their current performance is already optimized so far, especially the computational geometry problems which are researched for many years. Parallel programming can be useful in other aspects of these problems in terms of spatial scalability. Spatial scalability refers to the ability to scale the system by adding more computing nodes. Spatial scalability is particularly important for problems that involve large datasets or complex algorithms. By distributing the computation across multiple nodes, the overall computation time can be reduced significantly, making it possible to handle larger datasets or more complex algorithms. In this project, we focus on Euclidean Short Path (ESP), Trapezoidal decomposition, and Voronoi Diagrams. These problems can be partially parallelized. The sequential approach to these problems sometimes is hard to transfer to the parallel approach. In this paper, we implemented the algorithm based on the idea that is

similar to “Divide and Conquer”. Splitting the big problems into small parts and solving the heavy computational process parallel in different machines. Thus, reducing the overall computational time.

1.2 Objective

The objective of this project is to solve Euclidean Shortest Path (ESP) and Voronoi Diagrams using MASS. And the sub-problem of ESP, trapezoidal decomposition using MapReduce and Spark. ESP is done by previous students with MapReduce and Spark by using a pre-defined data source. The data source providing a neighboring relationship has not fully fulfilled the definition of ESP which is given a set of polyhedral obstacles in a Euclidean space, and two points, find the shortest path between the points that do not intersect any of the obstacles [3]. In this project, we use obstacles data as the data input for the trapezoidal decomposing implementation using MapReduce and Spark. Trapezoidal decomposing transforms the obstacles into data points with neighboring information. Then, using these data points, we can use the project from previous students to perform full ESP. The full ESP processing picture is shown in figure 1. In the MASS version, we utilize the agent property to find the shortest path between the source and the destination. We use the agent propagation approach for that. Agents keep spawning child agents until it hits the destination. A detailed explanation will be shown in chapter 4.

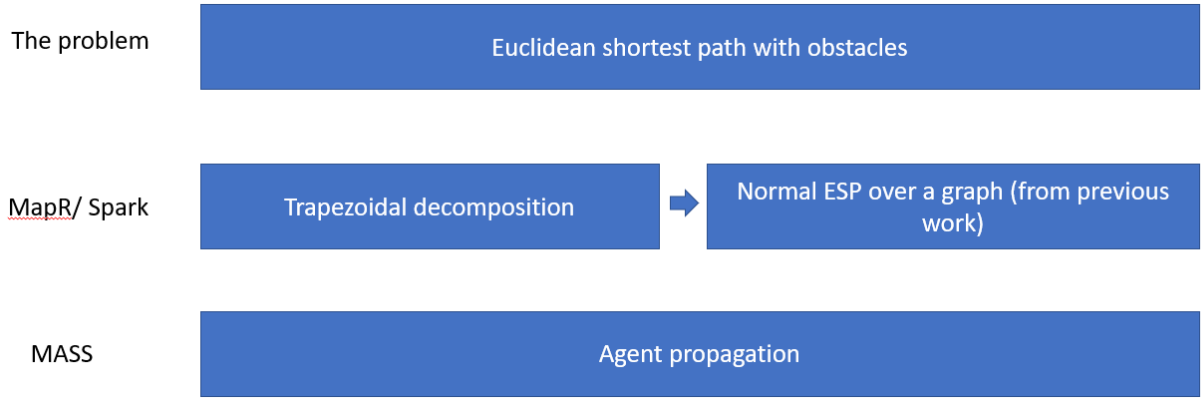

Figure 1: Euclidean Shortest Path full picture

For Voronoi Diagrams, the previous project limits the data point to less than 100. When the data point is too little, it is difficult to get a meaningful analysis in parallel programming because sometimes the communication overhead is heavier than the actual algorithm. In this project, we aim to release the data point issue and complete an analysis with 500k data points using MASS.

1.3 Project Summary

1. Developed the MASS version of Euclidean Shortest Path with propagation method that can handle around 250k data points depending on the size of places. The size of places does matter because we share the dataset to each place. Therefore, when the size of places increases, more datasets must be cloned. Memory thus becomes an issue.

2. Developed sequential version and parallel version of Trapezoidal decomposition using Spark and MapReduce. The parallel version can handle around 250k data points. The parallel version has the same issue which is the memory issue. Some portions of the implementation require shared variables. When the dataset is large, the number of shared variables increases as well. Thus, larger dataset might cause the heap issue.

3. Implemented Voronoi Diagrams using MASS library. The implementation can handle around 200k data points which is a lot better than the previous project. The performance depends on proximity of the data points. When there are lots of data points sitting in a crowded space, the program stops early since it is faster to fully occupy the entire map. Oh the other hands, when there are less data point in a larger space, it will takes longer time to fully take over the entire map.

Chapter 2: Tools and frameworks for parallelization

Parallel programming is a lot different from sequential programming. The process of serial programming is step by step. A task starts after the pervious tasks are completed. In parallel programming, a large task is divided into serval sub-tasks. These tasks are running simultaneously, which makes it a lot harder to get a good view of the program. And it creates additional concerns such as data race. A data race is said to exist when two or more concurrent processes of a parallel computation access the same resource in an unsynchronized fashion, and at least one of these accesses is a modify operation[4]. Fortunately, people built some high-level parallelization frameworks such as MapReduce and Spark, so we do not have to take care of the low-level problems. MASS, Spark, and MapReduce are all parallel computing frameworks that are designed for processing large datasets. MASS is a parallelizing library for multi-agent spatial simulation, which is specifically designed for simulating large-scale systems that involve multiple agents moving and interacting in a spatial environment. Spark is a general-purpose data processing framework that supports various types of workloads, including batch processing, stream processing, machine learning, and graph processing. MapReduce is a batch processing framework that is specifically designed for processing large datasets in a distributed computing environment. These frameworks have their own advantages. In this project, we are going to implement two algorithms and compare their performance.

2.1 MapReduce

MapReduce is a programming pattern in Hadoop framework. It is used to process big data stored in the Hadoop File System. The data access and storage are disk based. Data is usually stored as files. Files can be broken into small blocks, so MapReduce can handle different blocks individually in different machines concurrently. MapReduce only has two interfaces – Map and Reduce. For the Map function, data split into smaller blocks. Each of them is sent to a mapper to process the data. For the Reduce function, it waits for

the completion of the Map function and group the data by key, then typically perform an aggregation or summarization operation. The MapReduce framework provides fault tolerance, scalability, and load balancing, which enables it to handle large amounts of data efficiently. Figure 2 shows the full procedure of MapReduce. In this project, we can use MapReduce pattern to process our trapezoidal decomposition.

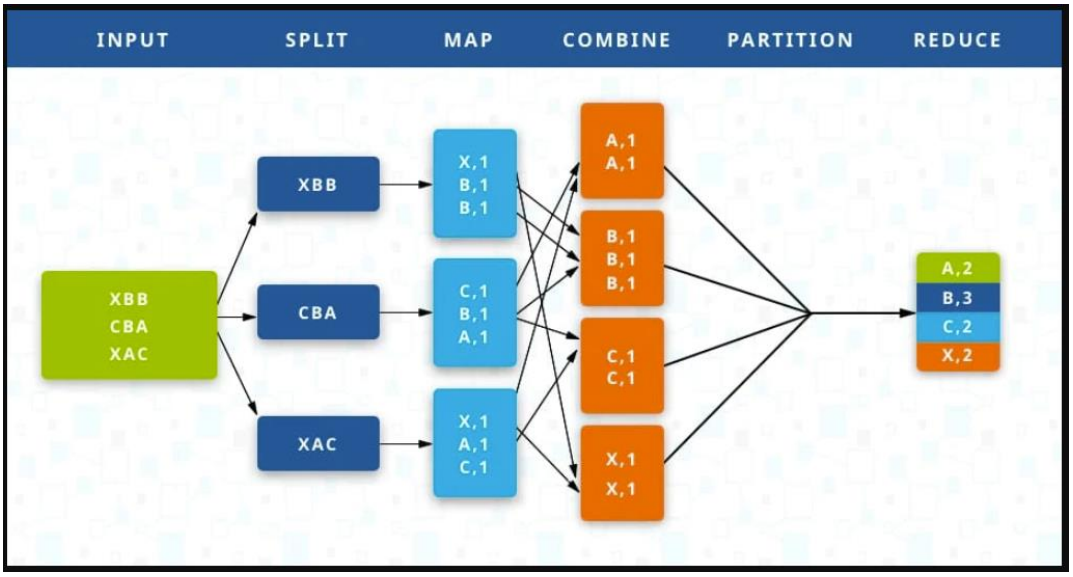

Figure 2: MapReduce procedure [5]

2.2 Spark

Spark is a data processing framework that can process large amounts of data across multiple computers in parallel. Due to the in-memory data engine, processing time in Spark is said to be up to one hundred times faster than MapReduce. The fundamental data structure of Spark is Resilient Distributed Dataset (RDD). It is an immutable distributed collection that can be any type defined by the users. The data set in RDD is then divided into multiple partitions across the clusters to be processed in parallel. Spark has two types of operations – Transformation and Action. Transformation creates a new data type from the old data.

Action returns one or a list of values based on the custom-defined computation. All transformations are inactive due to the lazy evaluation feature until an action is called. This feature makes Spark more efficient. The reason is that we can avoid the overhead of the sending intermediate mapped data set to the drive node. When shared variables are needed, Spark provides “broadcast variables”. These variables can be cached in memory on all salve nodes and let the computer nodes use the variables from the main driver. In our project, we used Spark for the trapezoidal decomposition to transform the obstacles data into trapezoids.

2.3 MASS

MASS (Multi-Agent Spatial Simulation) [6] is a parallel computing library utilizing agent-based modeling (ABM). ABM represents the simulation of agent behavior in a system. Two key classes in MASS are “Places” and “Agents”. In geometry, data usually resides in multi-dimensional space. “Places” is emulating the space where the data are at. “Places” is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes [6]. Figure 3 demonstrates how “Places” are distributed in a two-dimensional area. Assuming there are three computing nodes, if the “Places” size is 12 x 4, each computing node will be assigned to have a 4 x 4 “Places”. “Places” will be evenly distributed into multiple vertical stripes along the X-axis direction by default. Agents are a set of execution instances that can reside in a place, migrate to any other places with array indices (thus duplicating themselves), and interact with other agents and places [6]. In this project, we implemented the Euclidean Shortest Path and Voronoi Diagrams with MASS in a propagation approach manner.

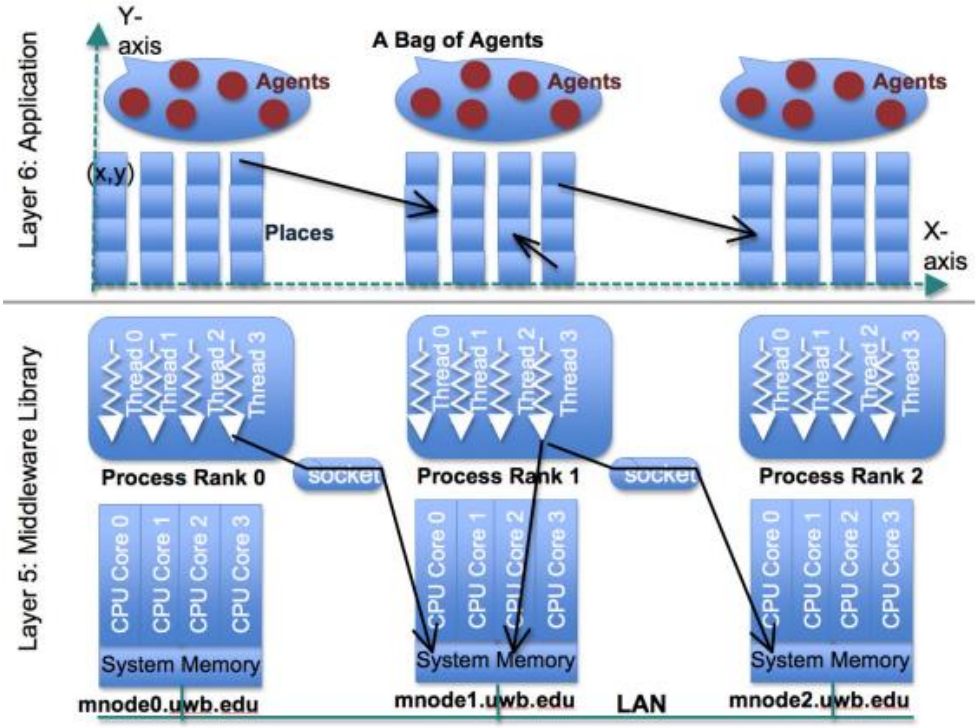

Figure 3: Parallel execution with MASS library [6]

Chapter 3: Related works

The time complexity and space complexity of computational geometry algorithms are important factors. In the real world, a dataset with hundreds of millions of data points is used to process user cases. Efficient algorithms are needed for faster outcomes. Scientists keep improving algorithms to achieve this goal. For example, the shortest path algorithm with non-negative weights is developed in 1956 with O(V^2EL) time complexity [7]. (V, E, and L represent the vertices, edges, and maximum weight among all edges). It is being improved to O(E+VloglogV) in 2004. In this section, we introduce a few algorithms and applications that are related to our project. On the other hand, efficiency in time complexity often takes advantage of using more space. In distributed computing, when we have more computing resources, we have more memory to use. As a result, we do not have to use the most time-efficient algorithms to solve our problems. Instead, understanding the trade-off and taking the balance becomes more important.

3.1 Slab decomposition

Slab decomposition is usually applied to the Point Location problem. It is based on subdividing the obstacles using vertical lines that pass through each vertex from the obstacles. The region between two consecutive vertical lines is called a slab. Notice that each slab is divided by non-intersecting line segments that completely cross the slab from left to right [8]. The idea can be applied to trapezoidal decomposition. The result is shown in figure 4. The diagram is split into multiple trapezoids. The only concern is, there are some invalid trapezoids that reside inside the obstacles. This can be fixed by checking if the centroid of the trapezoid is sitting on the obstacle. One of the limitations is storage consumption. The algorithm takes up to O(n^2) space complexity. It will be a problem when the data set is huge.

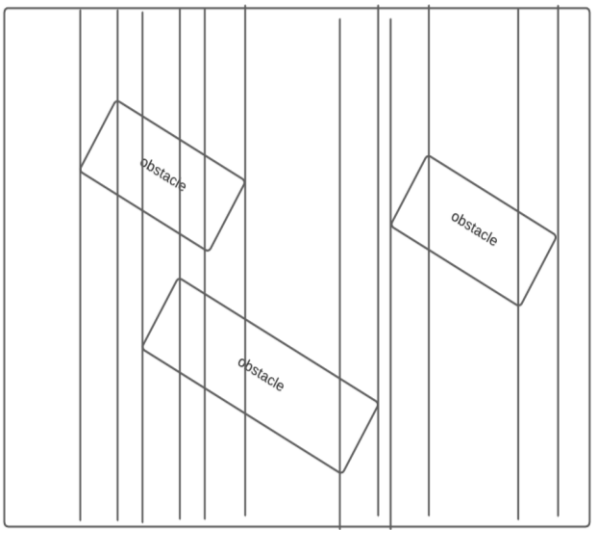

Figure 4: Slab decomposition for trapezoidal decomposition

3.2 Seidel's trapezoidal partitioning algorithm

Seidel's partitioning is a randomized incremental algorithm. It assumes that there are no two endpoints directly above and below one another. This avoids a special edge case (a stacked ladder shape). Incremental algorithms are usually simple. But they do not do great in terms of performance. Randomization helps to solve this problem. In this algorithm, the time complexity is improved to O (n log n).

The algorithm consists of a special data structure – the search tree. In figure 5, the left figure represents an example of trapezoidation, and the right figure shows the related search tree. The search tree has three components.

1. The edge (rectangle): a line on the trapezoid map
2. The vertex (diamond): vertices of a line
3. The trapezoid (circle): a built trapezoid.

These components are stored in a binary tree-like structure. Each operation costs O (log n) time due to the nature of the binary tree. As a result, the overall performance is O (n log n). Since the algorithm is an incremental algorithm, each step relies on the completion of the previous steps. Therefore, it only works in sequential programming.

Seidel's partitioning and slab decomposition are both geometric algorithms for solving computational geometry including Euclidean Shortest Path. In sequential programming, they are fine to compute the trapezoidal decomposition. Due to the drawbacks that are mentioned above. Therefore, we cannot use these two algorithms to compute trapezoidal decomposition in parallel.

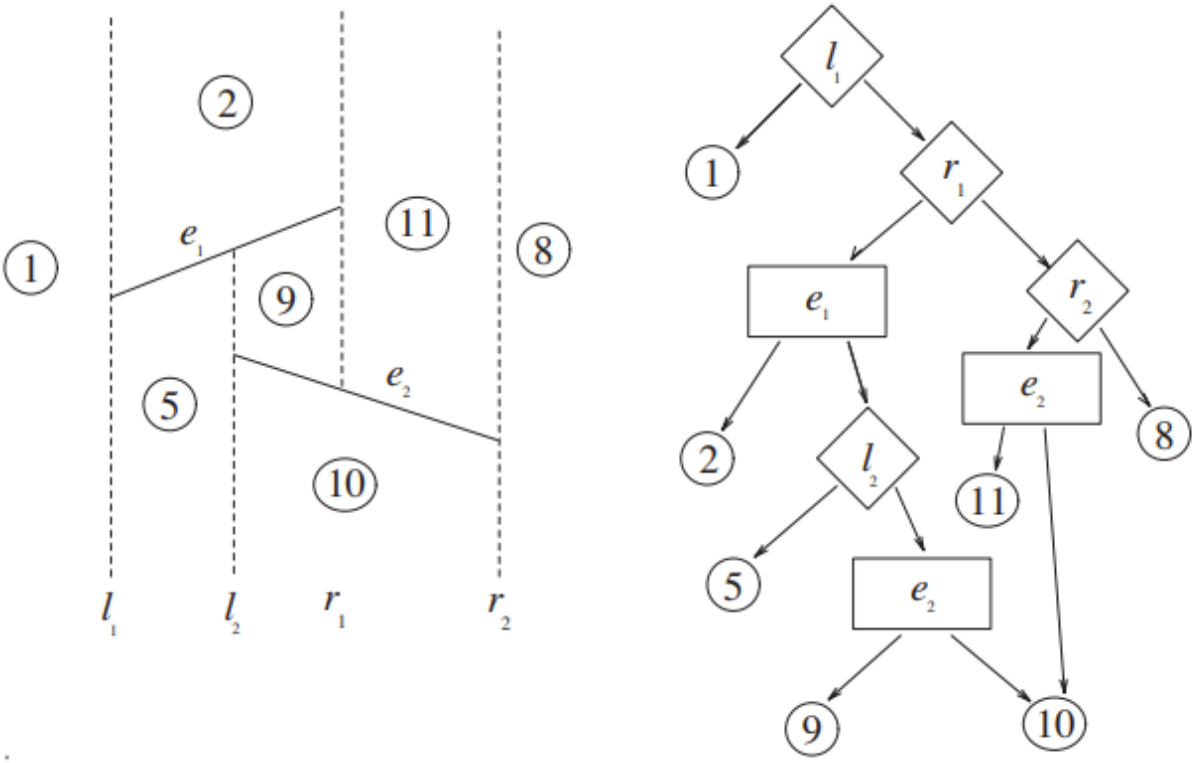

Figure 5 Left: post-trapezoidation. Right: the search tree [9]

3.3 NetLogo

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It was authored by Uri Wilensky in 1999 and has been in continuous development ever since at the Center for Connected Learning and Computer-Based Modeling [10]. The fundamental idea is based on agent-based modeling and simulation. There are four types of agents in NetLogo.

1. Turtle: the mobile agent that can be placed and move around the world.
2. Observer: instruct other agents.
3. Patches: a stationary agent that makes up the world.
4. Link: a connection between two distinct turtles.

NetLogo is an open-source multi-agent programmable modeling environment that is specifically designed for agent-based modeling and simulation. It has a user-friendly interface that allows users to build models using a visual programming language, which makes it easy for non-programmers to use.

NetLogo and MASS are two different tools used for agent-based modeling and simulation, but they have different features and capabilities. NetLogo is more suitable for small to medium-scale simulations and is particularly well-suited to social science research. In contrast, MASS is more suitable for large-scale simulations that require high-performance computing resources and is particularly well-suited to scientific research and engineering applications.

Chapter 4: Implementation

The project consists of three geometry algorithms including (1) Euclidean Shortest Path (ESP), (2) Trapezoidal decomposition, and (3) Voronoi diagram. This section includes the general idea of how the Trapezoidal decomposition works and how to solve ESP and Voronoi diagrams based on agent propagation.

Trapezoidal decomposition is a subproblem of Euclidean Shortest Path when the data given only contains obstacle data points in a planar graph. In general, ESP problems require input data to have data points with its neighbor relationship information, in short, a graph data structure. With the graph data, we are able to find a valid path and the distance between two locations and eventually a valid path from the source location to the destination. In the case of input data only consisting of obstacles information, we must build the graph data by ourselves. One of the common algorithms to build graph data is called trapezoidal decomposition. The basic idea is to break down the planar graph into multiple pieces of trapezoids. The trapezoids then can be easily transformed into data points with neighbor information. Hence, we can use a generic solution to solve ESP with the new data. A more detailed explanation will be presented in section 4.1.

For ESP, as a previous student had parallelized ESP with MapReduce and Spark. We will use her program to evaluate the performance with data built by the trapezoidal decomposition. Due to the natural features of MapReduce and Spark, the implementations are based on Divide-and-Conquer. In this project, we are using a different approach to achieve the same goal. We mainly focus on using multi-agent spatial simulation (MASS) with the agent propagation method. A sourcing agent expands to the nearby area until it finds the destination.

4.1 Trapezoidal decomposition

Definition: A trapezoidal decomposition is obtained by shooting vertical bullets going both up and down from each vertex in the original subdivision. The bullets stop when they hit an

edge, and form a new edge in the subdivision [8]. An example figure is shown in figure 6. Obstacles/blocks are placed on a planar graph. These obstacles can be in any polygon shape. One limitation is the placement of the obstacles. We assume that all the obstacles do not overlap with each other. Overlapping obstacles can cause incorrect output for this algorithm. This limitation can be solved by applying the Convex Hull algorithm but is not included in this project. We call the corners of the obstacle vertices. Each vertex extends a vertical line upward and downward until it hits another obstacle/boundary. After this operation is completed, the graph will be divided into many trapezoids just like the shape in figure 6. Hence, the trapezoidal map is formed.

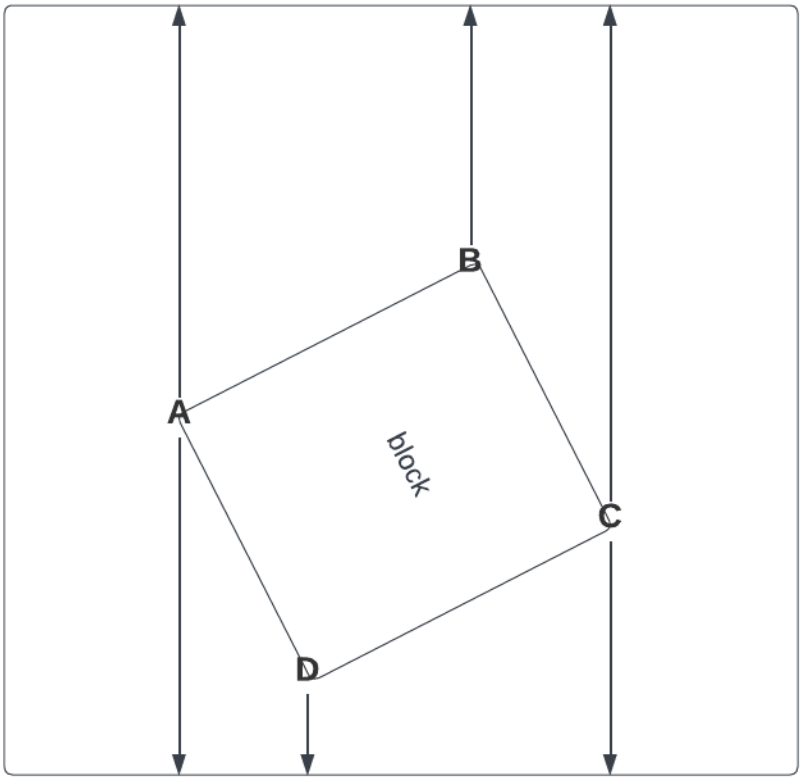

Figure 6: An example of a planar graph with an obstacle

4.1.1 Data pre-process

There are two types of data that we must find out before moving to the computation part. The first one is the extensions of a vertex. As mentioned above, we are shooting an upward and downward extension from the vertex until it hits another obstacle or boundary. In other

words, we are looking for the closest intersection pointing upward and downward. The information is needed for constructing the edge. An edge uses the vertex and its closest intersection to construct. The second helpful piece of information is the type of vertex. The types of vertices can be classified as “IN”, “MID”, and “OUT”. In figure 6, vertex A belongs to type “IN”, vertex B belongs to type “MID”, and vertex C belongs to type “OUT”. Each type has its own features. Type “IN”, always has two types of extension, upward and downward extension. Type “OUT” only has either an upward or downward extension. Type “OUT” is like type “IN” which also has two extensions. The difference is the direction of the vertex angle. Type “IN” points to the left-hand side and type “OUT” points to the right-hand side. This data will be used later in the search for the matching pair of the vertex. More details will be mentioned later in this chapter.

4.1.1.1 Calculation of line intersection

Two lines are said to be intersected if they share a common point. There are three scenarios when calculating line intersections.

1. There is an intersection at some point
2. The two lines are parallel
3. Line coincident

Case 2 and 3 result in no line intersection.
To calculate the intersection point, we can use mathematical formulas depending on what information. Generally, there are two common formulas to find the line intersection shown in equation 1 and equation 2 respectively.

$$Px = \frac{(x1y2 - y1x2)(x3 - x4) - (x1 - x2)(x3y4 - y3x4)}{(x1 - x2)(y3 - y4) - (y1 - y2)(x3 - x4)}$$
$$Py = \frac{(x1y2 - y1x2)(y3 - y4) - (y1 - y2)(x3y4 - y3x4)}{(x1 - x2)(y3 - y4) - (y1 - y2)(x3 - x4)}$$

Equation 1: line intersection using two points on each line [11]

$$ax + c = bx + d$$

$$P = (\frac{d-c}{a-b}, a\frac{d-c}{a-b} + c)$$

Equation 2: line intersection using line equations [11]

In our case, if we are looking for the upward intersection, we create a line from the vertex toward the upper boundary. The bottom point data will be the vertex data. The upper point data will be the same X-axis and 0 Y-axis because the upper boundary always sit on 0 Y-axis. Therefore, we prefer to use the formula equation 1 as we have all the data needed for this equation. However, the formula assumes the two lines extend infinitely. There is always a intersection point somewhere as long as they are not parallel. But the edge in our obstacles have a limited length. Using the formula directly may get to the intersection point that is out of the length of the edge which means the intersection found does not reside on the two lines (the two edges we are checking for the intersection). We have to add one more step to determine if the intersection is the actual intersection that we are looking for. A simple way to do that is to validate if the intersection point is in the min boundary of the two lines. In figure 7, there are two lines intersecting at (x,y). The max boundary here of X is min(x1, x3) to max (x2, x4). The Y boundary is min(y1, y3) to max (y2, y4). If the intersection is located within the boundary, it is guaranteed that the intersection is accurate.

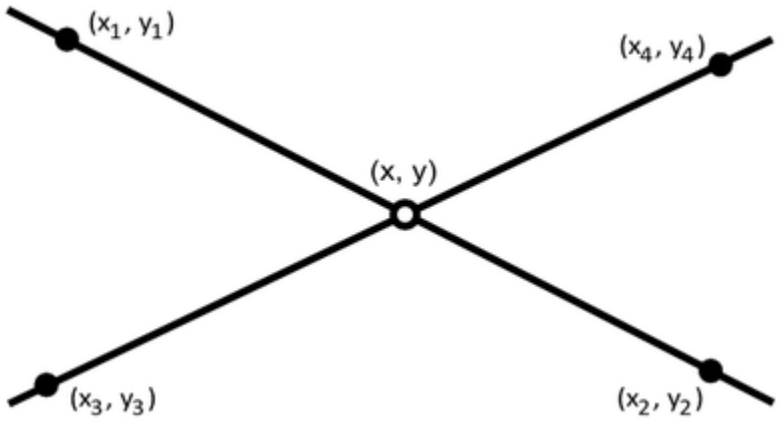

Figure 7:max and min bound of two lines [11]

4.1.1.2 Calculation of upward / downward extension

Not all the vertices have a valid upward/downward extension. When shooting a line upward / downward, it is almost like there are intersections somewhere. Having intersections does not necessarily mean it has a valid extension. To determine if the extension is valid, the number of intersections is the key.

Figure 8 shows how we can determine if a point is inside a polygon. To do that, it creates an infinite line from the point to the right. If the intersection count is odd, the point is located inside the polygon and vice versa. This idea is very useful for us to determine if a vertex has a valid extension. To find the valid upward extension, we created a line from the vertex to the upward boundary. If the total number of intersections is odd, it means it has an upward extension. For example, there is no other obstacle above the current vertex, extending a straight line toward upper boundary will only hit the upper boundary edge. So, there is one intersection. The downward extension can be found using the same logic. In figure 6, vertex B is an example of having an upward extension and no downward extension. Vertex D is the opposite example. The idea behind this is that all the obstacles are assumed to be 2-D and there is an outer boundary that wraps all the obstacles. If a line passes through an obstacle completely, the number of intersections is always equal to 2. If a line only passes half of the obstacle, the number of intersections will be 1. In the end, a line will always hit the upper boundary, or the bottom boundary, and this intersection is counted as 1. As a result, an odd number of intersections means the line passes through some obstacles and finally hits the boundary, in this case, there will be a valid extension. After we determine whether a vertex has a valid extension, we only need the closest one. Using the closest intersection and the vertex, we can create a trapezoid edge later in the algorithm.

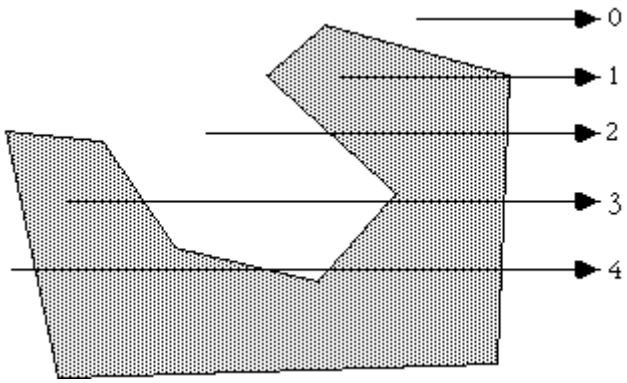

Figure 8: check if a point is inside a polygon [12]

4.1.1.3 find the valid pair to build a trapezoid

Definition of a valid pair: two successive vertices that are directly accessible to each other with no obstacles in between.

After the data pre-processing is done. We have sorted vertices in ascending order and found their closest extension on both sides. To start looking for a valid pair. We check the successive vertex in order. For example, in figure 9, we want to find the valid pair for vertex 3. We check its next greater vertex which is 4. But there is an obstacle between them. So, we check the next available one which is vertex 5. Since vertex 3 can directly access vertex 5, they are a good match. In this case, we use the upward extension of vertex 3 and the downward extension of vertex 5 to build a trapezoid.

Sometimes, there is a more complicated case. Vertex 6 in figure 9, it has both upward and downward extensions which means there are two trapezoids that need to be built. Therefore, we must find all the valid successive vertices but not just the first found one. In this case, it has 5 potential good matches which are vertex 7, 8, 9, 10, 11, 12, and 13. Check each vertex and stop when both upward and downward extensions find a good pair. For the lower part, vertex 7 is a good match because there is no obstacle in between. So, we add it to the potential valid pair list. The next step is to find the match of the upper part of vertex 6. The

next few available vertices are 8,9,10 because there is no block in between. However, these are only valid to match the downward extension of vertex 6. But the lower part of vertex 6 is already matched to vertex 7, we no longer have to match its lower part. Therefore, these vertices are ignored. Vertex 11 will be the next good pair for the upper part of vertex 6. And the computation is stopped here once both upward and downward valid pairs are found.

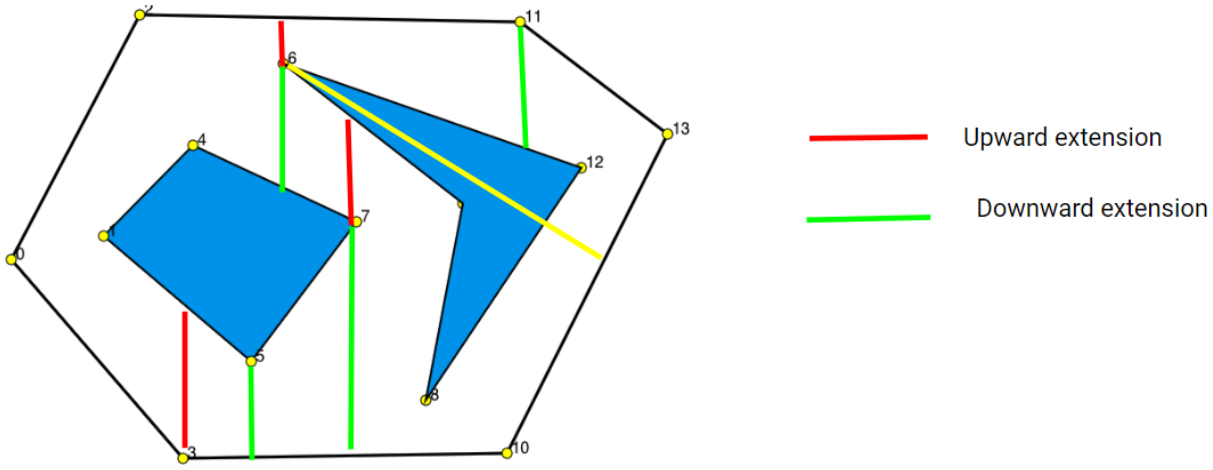

Figure 9: Valid pair example

There is a potential issue determining whether we are using the correct upper or lower part of the vertex to match the next available vertex. In the above case, we use “above” or “below” to determine the matches. For example, vertex 7 is “below” vertex 6. So, we use the lower part to match vertex 7. However, imagine if there is no vertex 11. Vertex 12 will be a good match for the upper part of vertex 6. But vertex 12 is “below” vertex 6 visually, it then matches the lower part instead of the upper part of vertex 6 which is inaccurate. To clarify, a vertex is above vertex 6 when it is located above the middle line of the vertex. The middle line is a line in the middle of two near edges that are connected to vertex 6. In figure 9, the

yellow line is the middle line of vertex 6. Vertex 12 is above the middle line; therefore, we should match it to the upper extension of vertex 6.

4.1.1.4 Trapezoid neighboring and final data point decision

Neighboring the trapezoids is intuitive. When two trapezoids are sharing the same edge, they are next to each other. In figure 10, the right edge of trapezoid 7 is sharing the left edge of trapezoids 8 and 9. They are counted as neighbors to each other.

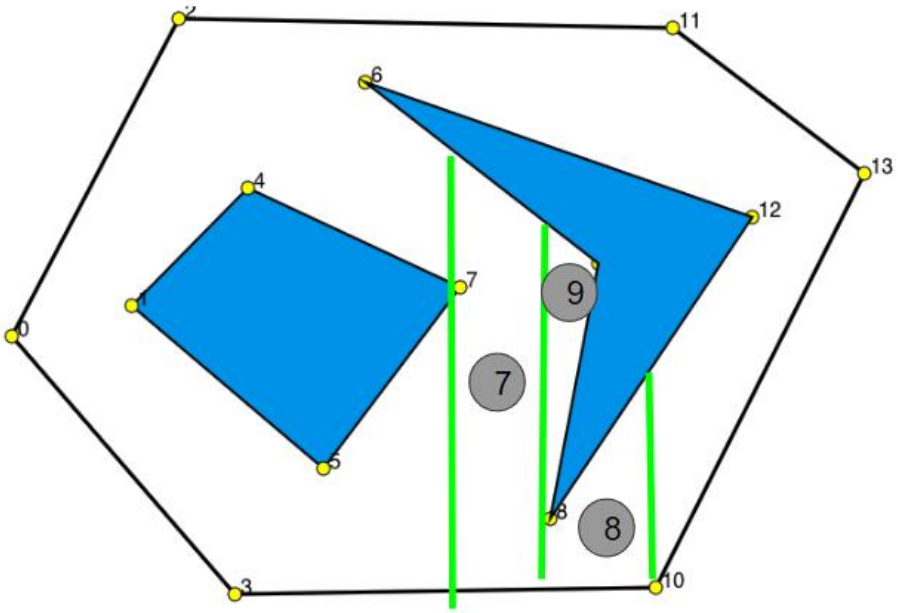

Figure 10:Trapezoid centroid bug

Besides, the neighboring relation, the distance between the neighbors is also the data we needed. Usually, we calculate the centroid of the trapezoid. Mark it as the data point location. And calculate the distance between those centroids. But with some obstacles on the map, this may not work. In figure 10, if we mark the centroid of trapezoids 7 and 8 as the data point and try to connect them using Euclidean distance, the Euclidean distance between the centroid of trapezoids 7 and 8 will cross the obstacle on the right-hand side. In this case, using the centroid of the trapezoid as the final data point does not seem a good idea. As we are using the shared edge to determine if they are neighbors, we could simply use the same

idea to mark the data point. We pick the middle point of the left edge of each trapezoid as the data point. This ensures all the neighbor data points can access their neighbor data points.

```
// Trapezoidal decomposition code snippet

// 1. Read source file and transform them to obstacle data
while (scanner.hasNextLine()) {
    buildObstacleUsingDataPoint();
    addObstacleToList();
}

// 2. Set up the vertex information from all obstacles
for (Obstacle obstacle : obstacleList) {
    determineExtensionsAndIntersections(obstacle);
    determineVertexType(obstacle);
}

// 3. Find the next non-blocking vertex
for (Vertex vertex : obstacle.getVerticesFromLeftToRight()) {
    Line line = createLineBetweenTwoVertices(vertex,
nextVertex);
    if (!hasIntersection(line)) {
        if (vertex.onlyNeedOneNextNonBlockingVertex()) {
            return;
        } else {
            if (vertexType == in) return vertex;
            qualifiedVertices.add(nextVertex);
        }
    }
}
for (Vertex qualifiedVertex : qualifiedVertices) {
    qualifiesForUpperOrLowerExtension(qualifiedVertex)

    if (allExtensionFound()) {
        break;
    }
}

// 4. Build the trapezoids (the vertices now are paired with the
next non-blocking vertex)
for (Vertex vertex : allVertices) {
    Line upperExtensionLine =
getClosestUpperExtensionLine(vertex);
    Line lowerExtensionLine =
getClosestLowerExtensionLine(nextVertex);
    Trapezoid trapezoid = buildTrapezoid(vertex, nextVertex,
upperExtensionLine, lowerExtensionLine);
}

// 5. Connect the trapezoids
for (Trapezoid trapezoid : trapezoidList) {
    connectAdjacentTrapezoids(trapezoid);

}
```

4.1.2 Parallelization and shared variables

For MapReduce, parallelization of MapReduce refers to the process of dividing the input data and distributing it across multiple machines for processing in parallel. This is a fundamental aspect of the MapReduce programming model, which is designed to support large-scale data processing on distributed systems.

For Spark, Parallelization of Spark refers to the process of dividing a large data processing task into smaller, more manageable subtasks and distributing them across multiple machines in a cluster for parallel execution.

In our MapReduce and Spark implementation, we mainly focus on the division of the heavy workload. There are two heavy calculations in the implementation. We will assign them to different computing nodes and utilize as many computing resources as possible to reduce the overall computational time. In Figure 11, it shows two computations that need shared data to complete their calculations. These two calculations must iterate the entire shared data. We try to assign the tasks into different computing nodes using MapReduce and Spark to reduce the computing workload.

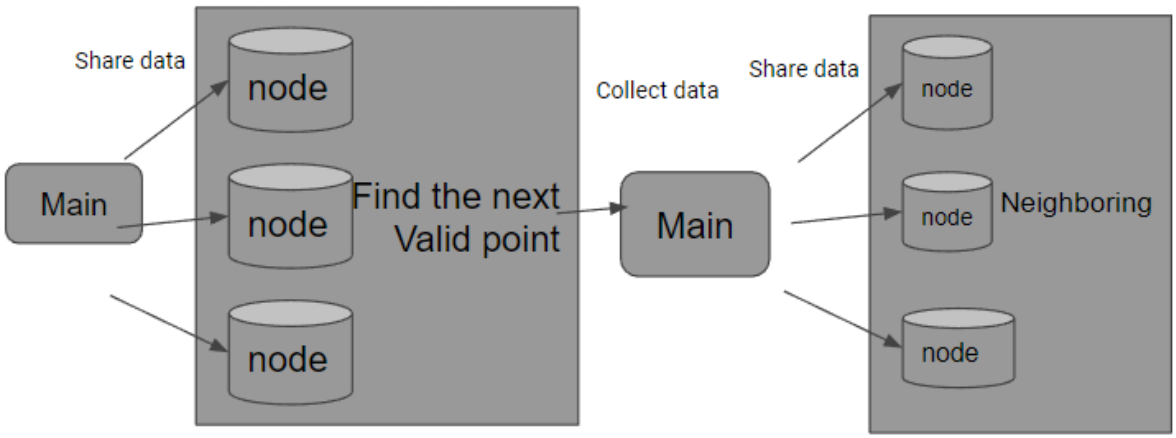

Figure 11:Division of workload

In the above diagram, we mentioned that those two tasks need to have shared data to support their calculations. Making use of their default system helps to better off the performance.

In MapReduce, a distributed cache is a feature that allows data to be cached and distributed across the nodes in a cluster. The distributed cache is a way to share files or data among all the nodes in the cluster, making it possible for all nodes to access the same data efficiently. The distributed cache can help to improve the performance of a MapReduce job by reducing the need to repeatedly read the same data from disk. By caching data in memory across the nodes in the cluster, the distributed cache can significantly reduce the amount of data that needs to be read from disk, improving the overall performance of the job.

In Spark, an accumulator is a shared variable that is used to accumulate values across different tasks in a distributed computation. Accumulators are write-only variables that can be updated by the tasks in a Spark job but can only be read by the driver program. Broadcast variables, on the other hand, are read-only variables that are cached on each node in a Spark cluster. Broadcast variables are used to share large read-only data structures, such as lookup tables or configuration settings, across a Spark job. By caching the data on each node in the cluster, broadcast variables can improve the performance of a Spark job by reducing the need to transfer large amounts of data over the network.

They project implemented the above feature to achieve the goal of shared data for the heavy computations.

4.2 Voronoi Diagrams

Definition: Voronoi Diagrams are the partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other [13].

The pattern created by Voronoi Diagrams is very common in the real world. For example, patterns on corn, giraffe skin, insect wing, and so on. This pattern is also useful in different areas. In mining, it can help to estimate the overall mineral resources based on exploratory drill holes. In epidemiology, they can help in identifying the source of infections[14].

To construct Voronoi Diagrams, there are a few algorithms we can use. For example, Fortune's algorithm, Lloyd's algorithm, and Bowyer-Watson's algorithm [15]. These algorithms are sequential based. There are improvements that can be made when utilizing parallel programming. One of the examples is using the MASS library. The MASS library can simulate the growth of each source cell in different threads and different computing nodes as well as improve efficiency on a large scale.

4.2.1 Implementation based on MASS library

In figure 3, MASS is utilizing threads and processes to achieve multi-task programming. All the parallelization is implemented internally because of that. The major components of it are Agents and Places. Applying the Voronoi Diagrams, we can use agents to represent the cells and places to represent the area that is occupied by the cells. Cells are expanding each iteration until they hit each other or the boundaries. When all the places are occupied, we can collect the collision data points where the expansion stopped. Gathering all the collision points with the source point, we can build the Voronoi Diagrams. Figure 12 shows two examples of Voronoi Diagrams based on different distance measurement methods.

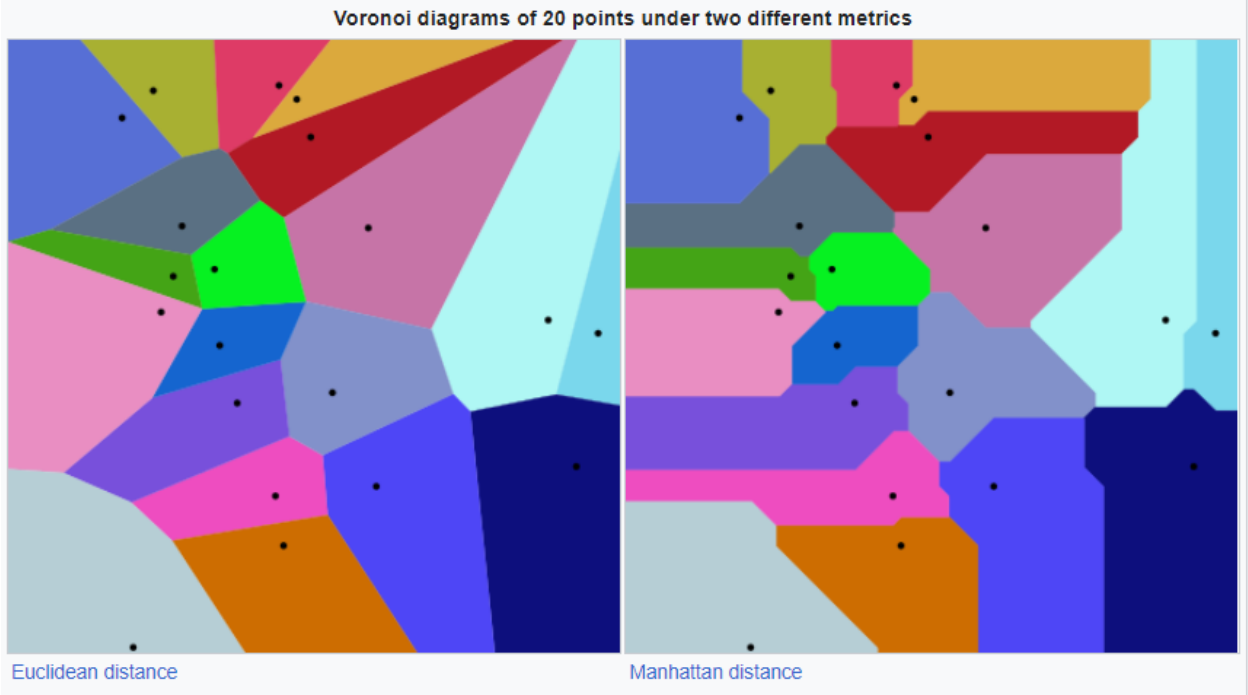

Figure 12: example of Voronoi diagrams [15]

4.2.2 Two different distance measurements

Figure 12 shows there are two different outcomes when we use different distance calculation methods. Euclidean distance and Manhattan distance are both measures of the distance between two points in a multi-dimensional space. However, they are calculated in different ways and are therefore suited to different applications. Euclidean distance is the straight-line distance between two points in space. It is calculated as the square root of the sum of the squared differences between each corresponding dimension of the two points. For example, in two-dimensional space, the Euclidean distance between points (x1, y1) and (x2, y2) is sqrt ((x2 - x1) ^2 + (y2 - y1) ^2). Manhattan distance, also known as taxicab distance, is the distance between two points measured along the axes of a grid-like structure, such as a city block. It is calculated as the sum of the absolute differences between each corresponding dimension of the two points. For example, in two-dimensional space, the Manhattan distance between points (x1, y1) and (x2, y2) is abs (x2 - x1) + abs (y2 - y1)

The Euclidean distance method is relatively better suited for applications where the spatial relationship between two points. It is because the calculation is more accurate as it includes the decimal places. In MASS, the area structure is like a 2-D array structure. We use one array cell as one unit of Voronoi place. When the expansion occurs, we try to simulate it to the real-world expansion which rings shape expansion. However, the array structure limits how it can be achieved. As a result, the Manhattan distance outcome is the only outcome that we can do using the MASS library.

4.2.3 Expansion sequence

The implementation of MASS Voronoi Diagrams is based on propagation. As the propagation place is like an array data structure, each propagation progress is based on one unit of the array cell. There are two common wave propagation patterns shown in figure 13 which are the Moore neighborhood and the Neumann neighborhood.

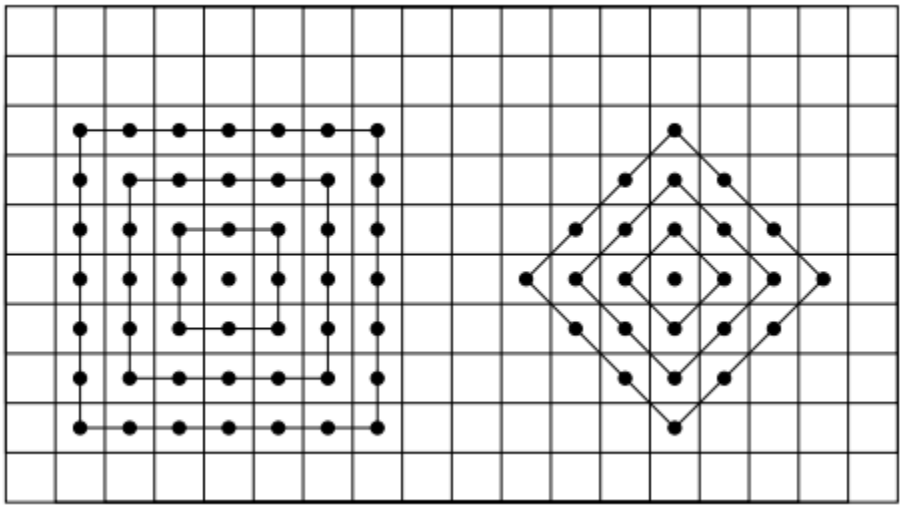

Figure 13: Propagation with Moore neighborhood pattern (left) and Von Neumann neighborhood (right) [16]

When either one of the patterns is applied to the Voronoi expansion, the outcome will not be close to what it should be as the pattern is not growing as a circle-like pattern. To make the propagation closer to real-world propagation, we must use a combination of the above two patterns.

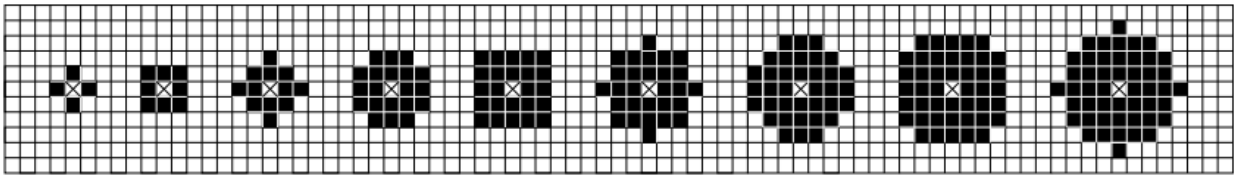

Figure 14: Circle like expansion [16]

Figure 14 is what we are expecting for the propagation pattern. The idea of the combination of the above two patterns is based on the agent's generation. When using agents to simulate the propagation pattern, we spawn their child agents and assign them to the new place according to the patterns we used. We can record the generation each child spawned. Therefore, based on their generation, we can decide which propagation pattern we use for the propagation. For example, if it is even number generation, we propagate the agents using the Moore pattern and, for the odd number generation, we use the Von Neumann pattern. This can result in more circle-like propagation which is closer to real-world propagation.

4.2.3 Agent's initialization trade-off

When initializing agents on their starting location, we have two methods to do it. The first one is to create agents automatically by the MASS's internal logic. As the places are indexed, agents will be placed accordingly starting from the index (0,0). Most of the time, all the agents will be created on the host machine. Then, we move the agents to their actual source location by the data point file. Since all the agents are sitting at the host machine, the process of moving agents to their source location will only be processed in the host machine. As a result, it takes a long time for that as there is only one computer node that is working.

The other method is to create agents in every place. Then terminate the invalid agents. The invalid agents are the agents that are being placed at a place that is not a source point. To achieve this, we must pass the source point data to every place and check if the index of the places matches any source point data. If not, we terminate the agent at that place. This method utilizes all the computing nodes. The drawback is, when the data file and the size of places are large, sharing this file with every place might cause a memory issue.

The two methods both have their pros and cons. They can be used interchangeably according to the needs. In this project, we are using the latter one as we are trying to tune up the performance.

```
// Voronoi Diagrams (MASS) code snippet


// 1. Read source file and transform them to point data
while (scanner.hasNextLine()) {
    buildDataPoint();
}

// 2. init MASS, places and agents
MASS.init();
Places places = new Places();
Agents agents = new Agents(); // assign agents to each place
killInvalidAgent(); // remove agent from non-source point

// 3. iterata the loop until all the places are occupied

while (n.Agents() > 0) {
    places.markVisitor();
    collectBoundarySourcePointPair();
    agents.propagate();
    agents.migrate();
    agents.killInvalid();
}
```

4.3 Euclidean Shortest Path

In computational geometry, Euclidean shortest path refers to the shortest path between two points in a two-dimensional or three-dimensional space that considers any obstacles that are present on the map.

In figure 15, there are examples of how ESP works. The grey polygons represent the obstacles, and the line represents the Euclidean shortest path between points A and B. The path follows a straight line wherever possible but deviates from it to avoid obstacles.

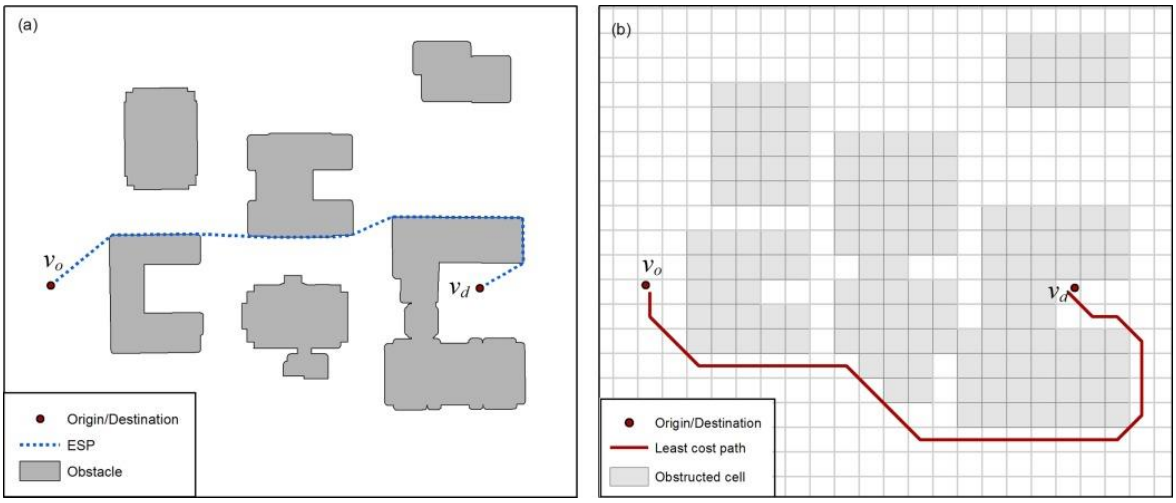

Figure 15: Example of ESP in Geometry [17]

Euclidean shortest path can be applied to various fields such as transportation, robotics, and computer graphics. Here are some examples:

1. Navigation: Euclidean shortest path is used in GPS and map applications to find the shortest route between two locations. This can be used for driving directions, walking directions, and even public transportation.
2. Logistics and transportation: Euclidean shortest path is used in logistics and transportation to optimize the routes of delivery trucks and other vehicles, minimizing travel time and fuel consumption.
3. Robotics: Euclidean shortest path is used in robotics for path planning, where robots are programmed to find the shortest path to move from one location to another. This is used in applications such as autonomous vehicles, drones, and warehouse automation.

4.3.1 Implementation in MASS

There are multiple ways to practice ESP in MASS. In this project, we use the method of agent propagation that is similar to the Voronoi Diagram implantation above.

First of all, we define the agent property.

1. The previous starting point: this is used to calculate the distance between the current location and the previous location.
2. The next location: this is used to indicate where the agents should migrate for the next propagation.
3. Is parent: this is used to indicate the agents have already propagated and are no longer needed.

Secondly, define the places.

1. Read the obstacles data and get its outer boundary. In figure 16, the wrapped area with black lines is the obstacle. The outer boundary is the area with the read lines.

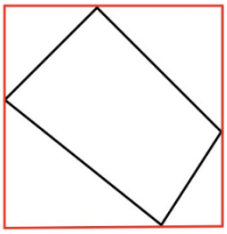

Figure 16: Example of outer boundary of an obstacle

2. Loop the entire outer boundary and mark the black area as an obstacle when creating places.

Thirdly, we start the agent propagation until any of the agents reach the destination. The agent's propagation with a pattern that is mentioned above in Voronoi Diagrams implementation. When an agent hits obstacles or boundaries, it gets terminated as this area is an invalid area. When an agent hits a vertex of an obstacle, it travels through the edge to the neighbor vertex. In figure 17, whenever an agent hits the green vertex, it will be spawning its

next generation to the neighbor vertices which are blue in color. This saves some resources from redundant agent propagation. Whenever a agent reach the destination, the iteration stop and return the shortest path.

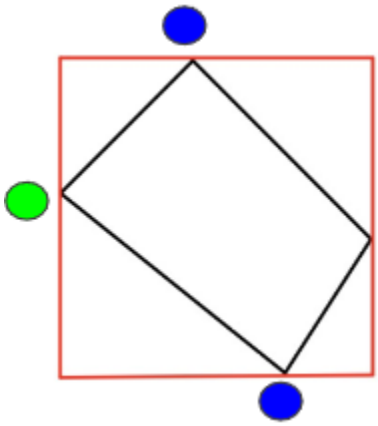

Figure 17: Example of vertex neighbor

4.3.2 Simulation to the real-world movement

In our practice, we try to simulate agents as real movements from a human. Therefore, we used the circle-like propagation pattern to mimic the movement. But an agent hits a vertex, it immediately moves from the vertex to its neighbor vertex. This causes a conflict with the real-life movement. Thus, we added an idle timer to those agents who walk faster than the iteration count. In each iteration in MASS, we say the agents can only move one unit of distance. When the agents are faster than the iteration count, they will stay idle and wait until iteration count exceeds their current distance. This is the method that we used to ensure the propagation movement is synchronized.

Chapter 5: Evaluation

In this Chapter, we benchmark the Trapezoidal composition of Spark and MapReduce and discuss the its performance when increasing the computing nodes. Also, we benchmark the MASS program including the ESP and the Voronoi Diagrams.

5.1 MapReduce performance

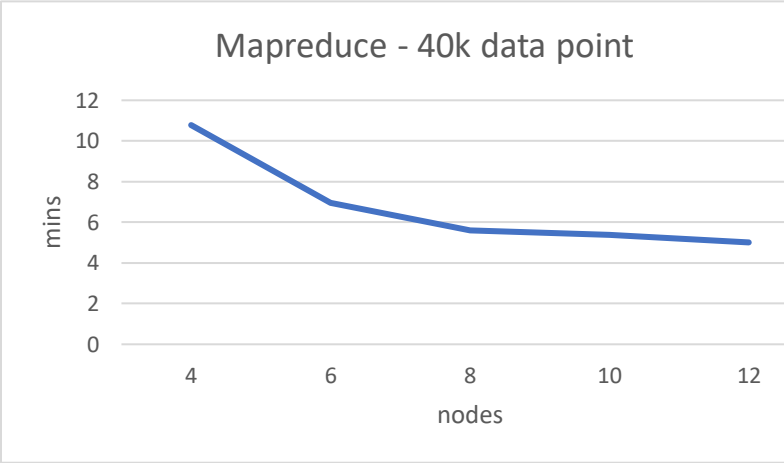

Figure 18: Line graph of MapReduce 40k data point performance

In figures 18, it shows the performance of the trapezoidal decomposition using MapReduce running on 4-12 computing nodes for 40k data point. The fastest running time is when we are using 12 computing nodes whereas the slowest running time is using 4 computing nodes. The line graph forms a downhill pattern as the number of nodes used increases. This information tells us with the increasing number of computing nodes, the algorithm can perform better.

5.2 Spark performance

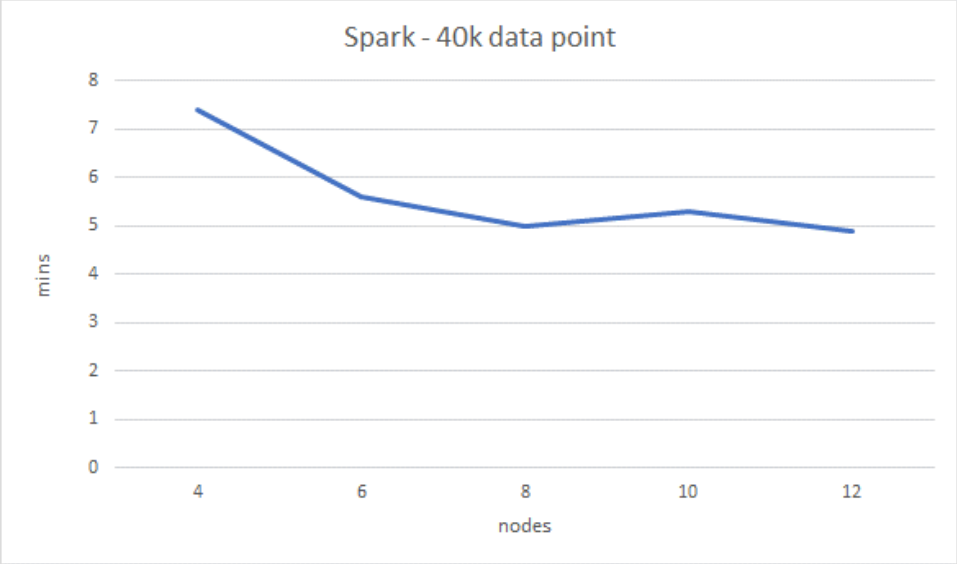

Figure 19 Line graph of Spark 40k data point performance

In figures 19, it shows the performance of the trapezoidal decomposition using Spark running on 4-12 computing nodes for 40k data point. The running time has a significant decreasing trend from the number of computing nodes 4 to 8. However, with the increasing of computing node from 8 – 12. There is no obvious message sign that showing the performance is improved when the number of computing nodes increases. At 12 computing nodes in figure 19, the performance is worsened compared to the 10 computing nodes. This may indicate that there are some overhead when the number of computing nodes exceed certain amount.

5.3 MapReduce and Spark performance comparison

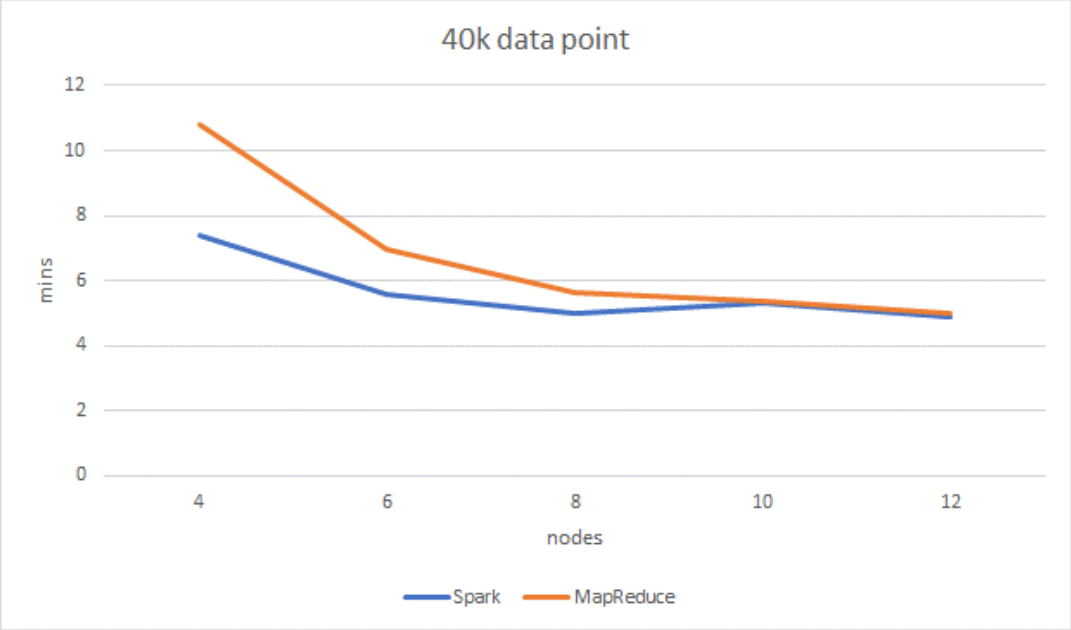

Figure 20: Line graph of Spark and MapReduce 40k data point performance

In figures 20, we combine the performance of using Spark and MapReduce. With 40k data points, Spark outperforms MapReduce when using fewer computing nodes. When the number of computing nodes exceed 8, they have a similar performance.

The result is not as expected. Spark is supposedly faster than MapReduce because of in-memory processing. Spark can process data in memory, whereas MapReduce needs to write intermediate results to disk. This means that Spark can be faster for iterative algorithms or tasks that require multiple passes over the data.

The minor performance difference indicates that there are some overhead in Spark that is worser than in MapReduce. In the implementation of Spark, it involves indexing and sorting. Spark is not optimized for sorting tasks. Whereas MapReduce provides a built-in sorting functionality that is optimized for sorting large datasets in a distributed computing

environment. Therefore, this is the reason why the performance of Spark is not as fast as expected.

5.4 Euclidean Shortest Path (MASS)

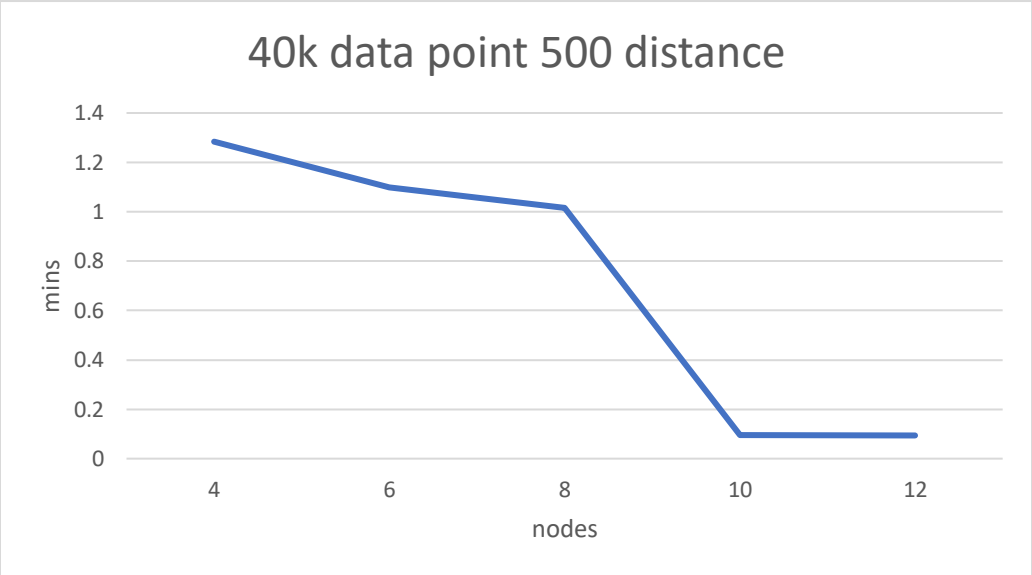

Figure 21: Line graph of ESP using MASS Library

In figures 21, it shows the performance of the ESP using MASS library running on 4-12 computing nodes for 40k data point. It appears to have a downward trend when the number of computing nodes increases. The best performance is at 12 computing nodes. It is expected as more computing resources mean agents and places can be distributed more evenly among all the computing resources.

5.5 Voronoi Diagrams (MASS)

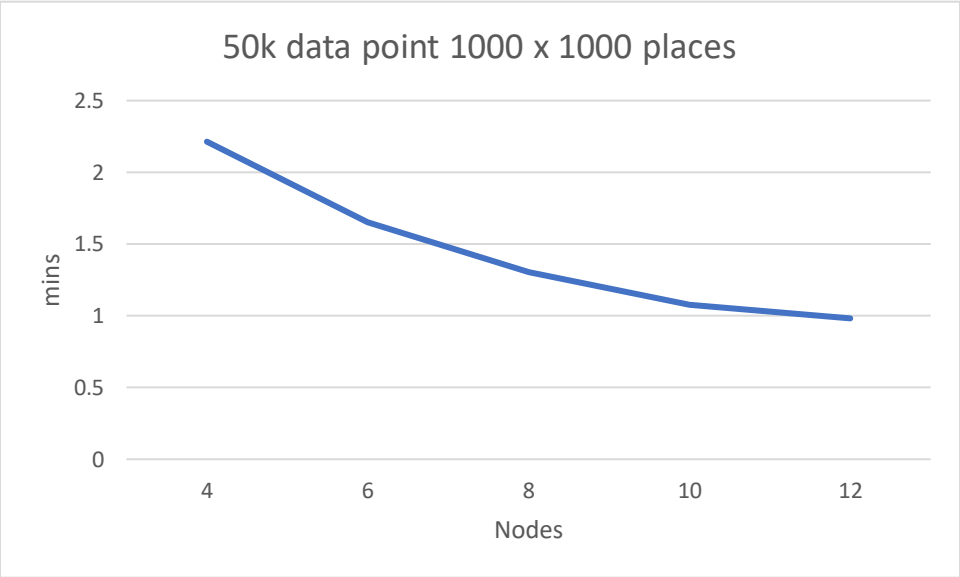

Figure 22: Line graph of Voronoi Diagrams using MASS Library

In figures 21, it shows the performance of the Voronoi Diagrams using MASS library running on 4-12 computing nodes for 50k data point in a 1000 by 1000 places. There is a smooth downward trend when the number of computing nodes increases. The data points are randomly generated and quite evenly distributed. This means agents and places are sharing the workload quite evenly. As a result, more computing nodes will have a better share of workloads.

The limitation of the program includes the proximity of the data points. When there are few data points, it takes more time to complete the program. The reason is that it takes more iterations to fully occupy the entire place. Therefore, proximity of data points plays an important role in the evaluation.

Chapter 6: Conclusion

The project discusses an implementation of Trapezoidal decomposition using MapReduce and Spark, Euclidean Shortest Path and Voronoi Diagrams using MASS library. Based on the result of evaluation in general, we can conclude that for parallel programming, when there are more computing resources, the overall performance of the paralleling applications will have a better performance. As well as the increased memory, some of the applications including the Voronoi diagram in this project, it can run more data points when the number of computing nodes increase.

For the limitation, the implementation of Trapezoidal decomposition has too much overhead for the shared data. Once there is a larger dataset, the amount of shared data increases as well. Thus, the performance has not gotten much better. For the MASS applications, it limits memory usage as well. Since they need to send the data source file to each place, the amount of places and the size of the source data affect the performance of the algorithm.

In the future, all of the applications in this project have potential to be improved. For Trapezoidal decomposition, the limitation is all about the shared variable. I think there is a way to break down the data beforehand so I can decrease the amount of shared data used. Therefore, decrease the overhead. For the MASS programming, currently I am sending the data file to each place. This wastes too much memory, and it can be avoided. Sharing data based on the number of computing nodes should be an feasible solution. And each computing node can share the data to their corresponding places. This will largely decrease memory usage.

BIBLIOGRAPHY

[1] G. Almasi and A. Gottlieb, *Highly parallel computing*. Redwood City, Calif. : Benjamin/Cummings.

[2] D. Rodgers, “Improvements in multiprocessor system design,” 1985, pp. 225–231.

[3] “Euclidean shortest path,” *Wikipedia*. Oct. 17, 2022.

[4] “Parallel Program - an overview | ScienceDirect Topics.” https://www.sciencedirect.com/topics/computer-science/parallel-program

[5] “MapReduce 101: What It Is & How to Get Started,” *Talend - A Leader in Data Integration & Data Integrity*. https://www.talend.com/resources/what-is-mapreduce/

[6] J. Emau, T. Chuang, and M. Fukuda, “A multi-process library for multi-agent and spatial simulation,” in *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, Aug. 2011, pp. 369–375. doi: 10.1109/PACRIM.2011.6032921.

[7] “Shortest path problem,” *Wikipedia*. Dec. 10, 2022. Accessed: Dec. 17, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Shortest_path_problem&oldid=1126599032#Single-source_shortest_paths

[8] “Point location,” *Wikipedia*. Jul. 24, 2022.

[9] F. Li and R. Klette, *Euclidean Shortest Paths*. London: Springer London, 2011. doi: 10.1007/978-1-4471-2256-2.

[10] “NetLogo 6.3.0 User Manual.” https://ccl.northwestern.edu/netlogo/docs/ (accessed Dec. 22, 2022).

[11] “Line–line intersection,” *Wikipedia*. Sep. 08, 2022.

[12] "Determining if a point lies on the interior of a polygon." https://www.eecs.umich.edu/courses/eecs380/HANDOUTS/PROJ2/InsidePoly.html (accessed Nov. 14, 2022).

[13] E. W. Weisstein, "Voronoi Diagram." https://mathworld.wolfram.com/ (accessed Dec. 22, 2022).

[14] "How Voronoi diagrams help us understand our world," *The Irish Times*. https://www.irishtimes.com/news/science/how-voronoi-diagrams-help-us-understand-our-world-1.2947681 (accessed Dec. 22, 2022).

[15] "Voronoi diagram," *Wikipedia*.

[16] T. Nickson and I. Potapov, "Broadcasting Automata and Patterns on Z^2." arXiv, Oct. 02, 2014. Accessed: Feb. 13, 2023. [Online]. Available: http://arxiv.org/abs/1410.0573

[17] I. Hong and A. T. Murray, "Assessing Raster GIS Approximation for Euclidean Shortest Path Routing," *Transactions in GIS*, vol. 20, no. 4, pp. 570–584, 2016, doi: 10.1111/tgis.12160.