A Parallel Implementation for non-nested and nested Voronoi Diagrams

MSc. High-Performance Computing

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Author: Patrick Prunty
Student ID: 15335866
Supervisor: Richard Morrin

School of Mathematics
Faculty of Engineering, Mathematics and Science
Trinity College Dublin, University of Dublin

August 23, 2020

Declaration

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

Acknowledgements

I would like to use this space to formally thank my supervisor Richard Morrin for his bountiful support throughout the summer, and with whom, I have learned so much from this past year. I would like to thank both my parents, my brother, and girlfriend for their endless support and gestures of kindness. My lecturers and classmates for their assistance and collaboration. I believe that all art is a collaboration. Finally, I would like to give a big thank you to the rest of my family and friends, without whom this degree would not have been possible.

Abstract

Nested Voronoi refers to a Voronoi tessellation which contains in each of its cells, k inner Voronoi tessellations. Until now, this phenomenon has been acknowledged due to its appearance in nature but left totally unexplored. Nested Voronoi suggests that there exists a hierarchical structure to some Voronoi tessellations, where given a point set P, a certain number of points $p_i, p_{i+1}, ..., p_k \in P$ are prioritized for the outermost tessellation.

Combining concepts from the k-means clustering analysis technique, which provides an indirect partition of the data set into Voronoi cells, nested Voronoi diagrams can be constructed by preforming a first-sweep Voronoi diagram for the point set's k cluster points, and later, k second-sweep nested Voronoi diagrams for the remaining points belonging to each cluster, or equivalently, the points enclosed in each of Voronoi cells in the first-sweep tessellation.

This paper provides a multi-threading parallel strategy for both Shamos and Hoey's divide-and-conquer paradigm for constructing non-nested Voronoi diagrams [1], and additionally, the first known serial and parallel implementation of nested Voronoi diagrams.

Shamos and Hoey proved that if the merge step for two Voronoi diagrams can be carried out in *O(n)* then the overall running time of this algorithm is *O(n log n)*, which is the best-case scenario for constructing a non-nested Voronoi diagram.

This paper provides a further investigation into this divide-and-conquer strategy by first dividing the original problem into three sub-problems rather than two in order to achieve an additional gain in speed-up, and then, parallelising the three sub-problems to further enhance this gain in speed-up.

The results suggest that further investigation into nested Voronoi diagrams be carried out, and that some of the traditional applications of non-nested Voronoi diagrams would benefit from using nested Voronoi diagrams over the traditional type.

Contents

Chapter 1

Introduction

Computational geometry is of practical importance because Euclidean space of two and three dimensions forms the arena in which real physical objects are arranged [1]. The main impetus for the development of computational geometry as a discipline is due to the progress in computer graphics and computer-aided design and manufacturing (CAD/CAM), but many problems in computational geometry are classical in nature and may come simply from mathematical visualization, or in other words, curiosity.

Moreover, solving these geometrical problems efficiently on a high-speed computer requires the constant development and maintenance of new geometrical tools, as well as the application of fast-algorithm techniques. This is because computational complexity is central to computational geometry, with great practical significance if algorithms are used on very large data sets containing tens or hundreds of millions of points. For such sets, the difference between a $O(n^2)$ and $O(nlogn)$ algorithm may be the difference between days and seconds of computation.

Voronoi diagrams, a field of computational geometry, has found applications in a multitude of different fields, with relevant material to be found in applications of astronomy, biology, cartography, crystallography, geology, linguistics, marketing, metallography, meteorology. operations research, physics, physiology, remote sensing, statistics, and urban and regional planning, to name a few. [3]

In more recent times, Natural Element Methods (N.E.M), or meshfree methods, have been used to solve partial differential equations by creating a collection of Voronoi cells and using these cells to decompose the grid over traditional grid types [7]. Pixar also have a long history of incorporating Voronoi diagrams in their animations to produce a "naturalistic" quality to the look of their characters, an example of this can be seen on the "dino-skin" in "The Good Dinosaur". [6]

All such applications of Voronoi diagrams require high-speed algorithms to handle the heavy computational pressures involved in solving a PDE using the N.E.M, or producing high-resolution animations of dinosaur skin.

This paper presents a thread-based parallel implementation of Shamos and Hoey's divide-and-conquer strategy for Voronoi diagrams that runs in an optimal time. As well as this, the paper produces the first known implementation of nested Voronoi diagrams both in serial and parallel.

Chapter 2

Nested Voronoi

This paper focuses on the core theoretical concepts necessary for the construction of a nested Voronoi diagram. This first chapter seeks to introduce readers to the concept of both Voronoi diagrams and clustering analysis, before combining both concepts into one in order to hypothesize the practical use of nested Voronoi diagrams.

2.1 Voronoi Diagrams

In mathematics, a Voronoi diagram is a partition of a plane into different regions or *cells* that enclose a given set of points, or *seeds*, such that any point enclosed in a given region is closest to its generating seed than to any other seed on the plane, as illustrated in Figure. 2.1.

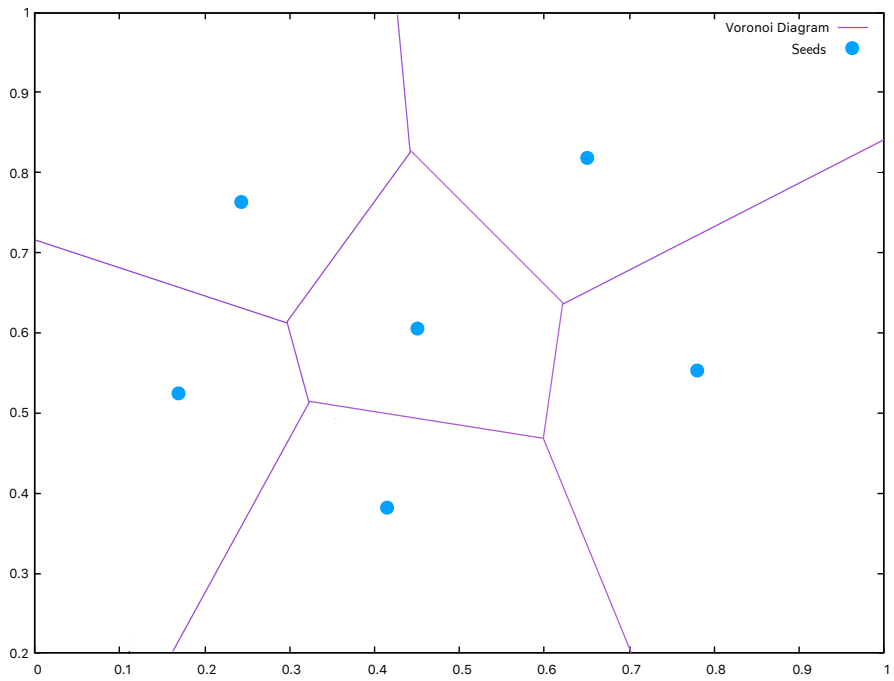

Figure 2.1: An example of a Voronoi diagram with six seeds.

Voronoi diagrams were first introduced by Georgy Voronoy who defined and studied the general n-dimensional case in 1908 [9]. However, practical applications of Voronoi diagrams can be traced back to Descartes in 1644 who employed these

tessellations to verify that the distribution of matter in the Universe forms vortices centered at fixed stars [8]. Peter Dirichlet also used Voronoi in his study of quadratic forms in 1850, as well as physician John Snow, who used Voronoi in his analysis of a cholera outbreak in London, 1854.

2.1.1 Formal Definition

Let X be a metric space with distance function d. Let K be a set of indices and let $(P_k)_{k\in K}$ be a tuple (ordered collection) of nonempty subsets (Voronoi seeds) in the space X. The Voronoi cell, or Voronoi region, R_k, associated with the site P_k, is the set of all points in X whose distance to P_k is not greater than their distance to the other sites P_j, where j is any index different from k. In other words, if $d(x, A) = \inf\{d(x, a)|a \in A\}$ denotes the distance between the point x and subset A, then

$$R_k = \{x \in X : d(x, P_k) < d(x, P_j), \forall j \neq k\} \tag{2.1}$$

Moreover, this simple means that given a point set $P = \{p_1, p_1...p_n\}$ in the plane, its associated Voronoi diagram, $V(P)$, divides the plane into n Voronoi cells with the following properties:

- Each point p_i lies exactly in one cell.
- If a point $q \notin P$ lies in the same region as p_i, then the Euclidean distance from p_i to q is less than the Euclidean distance from p_j to q, where $p_j \in P, p_j \neq p_i$.

2.1.2 Voronoi Algorithms

This section will briefly outline the algorithms used in this paper to construct a Voronoi diagram. Moreover, the naive approach for constructing a Voronoi diagram is combined with Shamos and Hoey's divide-and-conquer paradigm for merging Voronoi diagrams. Further extrapolation on the design and implementation of these algorithms can be found in Section. 3.2.3.

Naive approach

Given a point set $p = \{p_1, p_1...p_n\}$. The Voronoi cell, or region, for any point $p_i \in P$ is is defined as the inter-section of $n - 1$ half-planes formed by taking the perpendicular bisector of the segment $\overline{p_i p_j}$ for all $p_j \in P$ where $i \neq j$:

$$V(p_i) = H(p_i p_1) \cap H(p_i p_2) \cap ... \cap H(p_i p_n) \tag{2.2}$$

In this notation, $H(p_i p_j)$ refers to the half-plane formed by taking the perpendicular bisector of $\overline{p_i p_j}$. The intersection of any number of half-planes forms a convex region bounded by a set of connected line segments, as illustrated in Figure. 2.2.
These line segments form the boundaries of Voronoi cells and are called **Voronoi edges**. The endpoints of these edges are called **Voronoi vertices**.
The naive algorithm works by iteratively determining the convex region for each point p bounded by the intersection of its $n - 1$ half-planes by tracing the line of the closest bisector in an anticlockwise motion until it hits a Voronoi vertex, and repreating this process until a full rotation is complete.

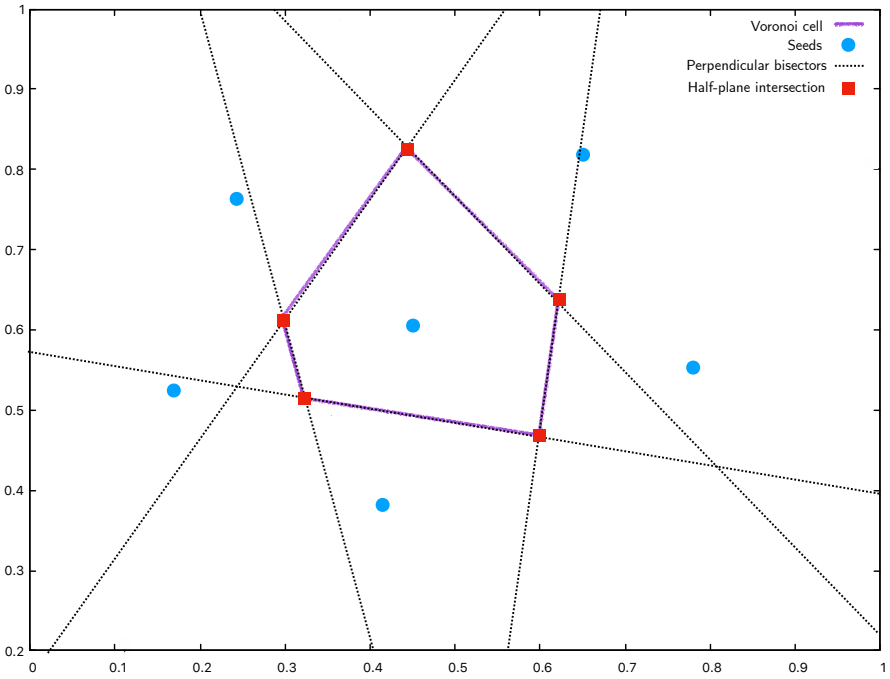

Figure 2.2: Voronoi cell $V(p_i)$ formed by half-plane intersection.

Divide-and-conquer

The first deterministic worst-case optimal algorithm for computing the Voronoi diagram has been presented by Shamos and Hoey [4]. In their divide-and-conquer approach, the set of point sites, P, is split by a dividing line into subsets L and R of about the same sizes. Then, the Voronoi diagrams $V(L)$ and $V(R)$ are computed recursively. The essential part is in finding the split line, and in merging $V(L)$ and $V(R)$, to obtain $V(P)$. If these tasks can be carried out in time O(n) then the overall running time is $O(nlogn)$.
[12]. This algorithm is significant from a theoretical stand point because it was the first to use a divide-and-conquer paradigm. The algorithm also boasts a $O(nlogn)$ time complexity equivalent to that of Stephen Fortune's sweep line algorithm [4]. Which, in recent times, due to the algorithm's elegance, has become somewhat of a standard in the literature for constructing Voronoi diagrams.
This paper focuses on implementing the divide-and-conquer paradigm for the construction of a Voronoi diagram. Its selection was due to the fact that it is naturally adapted for execution in multi-processor machines and the hypothesis that the problem could be divided into more than two sub-problems for an additional speed-up, as revealed in Section .

2.2 *k*-means clustering

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including pattern recognition, image analysis, information retrieval, bioinfor-

matics, data compression, computer graphics and machine learning.
k-means clustering is clustering analysis technique and method that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, known formally as the **cluster centroid**.
Reasons for popularity of k-means are ease of interpretation, simplicity of implementation, scalability, speed of convergence, adaptability to sparse data, and ease of out-of-core implementation [2].
The k-means algorithm is used for the discovery of k centroids for the first-sweep Voronoi diagram in the nested Voronoi algorithm. The k-means algorithm was chosen for this task because it provides an indirect partition the data set into Voronoi cells, as well as clusters.

2.2.1 Formal Definition

Suppose that we are given a set of n data points $p_1, p_2, ..., p_n$ such that each data point is in R^d. The problem of finding the *minimum variance* clustering of this data into k clusters is that of finding k points $\{m_j\}_{j=1}^{k}$ in R^d such that

$$\frac{1}{n}\sum_{i=1}^{n}\Big(\min d^2(p_i, m_j)\Big) \tag{2.3}$$

is minimized, where $d(p_i, m_j)$ denotes the Euclidean distance between p_i and m_j. The points $\{m_j\}_{j=1}^{k}$ are known as **cluster centroids** or **cluster means**. Informally, the problem in (2.1) is that of finding k cluster centroids such that the average squared Euclidean distance (also known as the mean squared error or M.S.E, for short) between a data point and its nearest cluster centroid is minimized; the euclidean distance, mean calculation and M.S.E routines will be covered in more depth in Section 3.2.
Unfortunately, the solution to this problem, finding when (2.1) is minimized, and the global optimum of the data set reached, is known to be NP-complete. The classical k-means algorithm [10], however, provides an easy-to-implement approximate to the solution in (2.1).

2.2.2 The *k*-means Algorithm

In the k-means algorithm, one chooses the desired number of clusters k, and their associated centroid values, both of which are chosen either randomly or through the use of some heuristic, covered in Section . The algorithm works by iteratively minimizing the within-cluster sum of squares, or variance. The discovery of these k cluster points is necessary for the construction of the first-sweep Voronoi diagram.
The above algorithm can be thought of as a gradient-descent procedure which begins at the starting cluster centroids and iteratively updates these centroids to decrease the objective function in (2.1) [2].
It is known that k-means will always converge to a local minimum. The particular local minimum found depends on the starting cluster centroids. Thus, endeavours must be made to maximize any local minima the algorithm may find itself in, or equivalently, choose better starting centroids. Heuristics to solve each of these problems will be discussed in the next section.

2.2.3 Heuristics

Heuristics play a crucial role in the initialization stage of the k-means algorithm. As previously mentioned, the user inputs the desired number of clusters k into the algorithm. But given a data set, how does one quantify the number of clusters that exist in it and choose the cluster's starting centroids? This section elucidates some heuristic techniques that seek to provide a solution to each of these problems.

The elbow method

A simple, yet effective way to approximate the number of clusters k in a data set is by using what is known appropriately as "the elbow method;;. The heuristic works by running the k-means algorithm for a different number of clusters k and plotting the variance as a function of the number of clusters.
The point on the graph which represents an "elbow" or "knee of a curve" is the point where adding another cluster would fail to give a better model of the data. Thus, the value of k for when the graph "elbows" is chosen as the appropriate number of clusters.
The intuition behind this is that when the graph "elbows", it is due to a very slight, or hardly noticeable change in the variance, which suggests, that the algorithm is "over-fitting".
Given a relatively simple data set consisting of three Gaussian distributions, as shown in Figure. 2.3, the elbow method preformed valiantly, where the value of k for where the graph "elbows" is in fact, at k=3, as shown in Figure. 2.4.

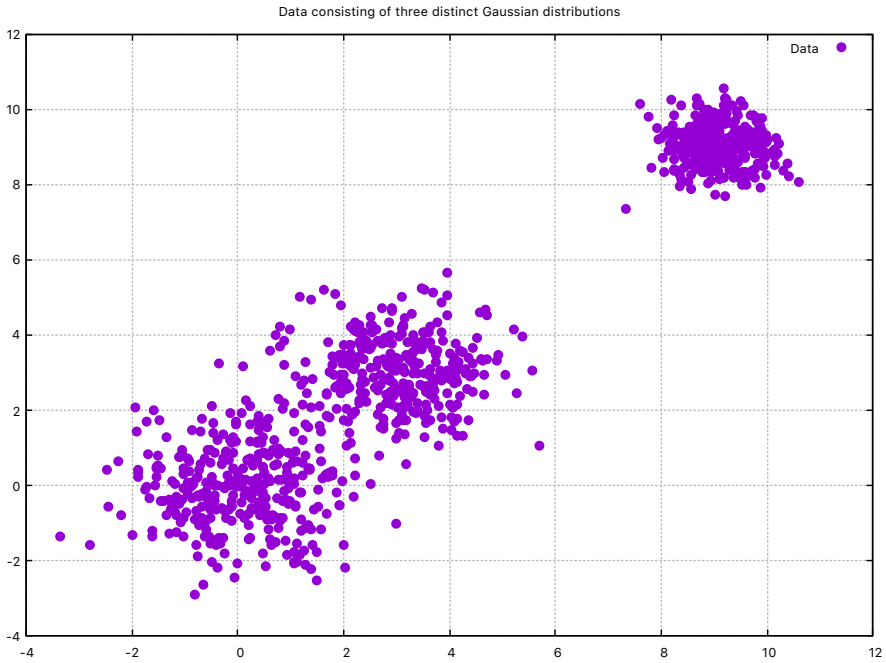

Figure 2.3: Dataset for three distinct Gaussian distributions.

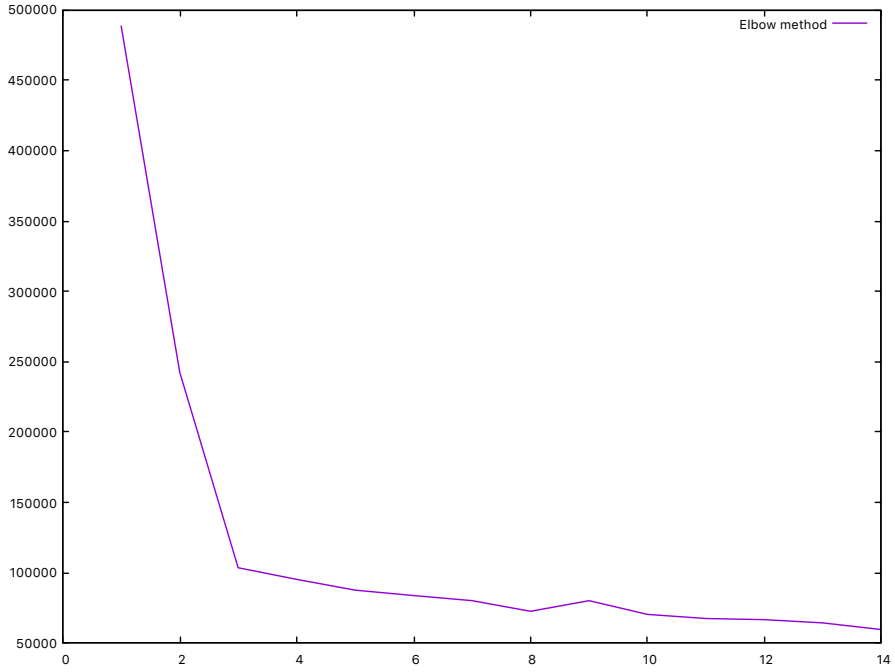

Figure 2.4: Elbow method for three distinct Gaussian distributions.

The heatmap method

An alternative, and experimental method to approximate the number of clusters k in the data set, as well as initial value centroids, is a heuristic which seeks to enhance ones visual perception of the data by visualising it in three dimensions
The heuristic works by creating a two-dimensional matrix for the data set to be representative of a histogram, where each element of the matrix represents a range for both the x and y-axis. Each element thus represents a square portion of the data set.

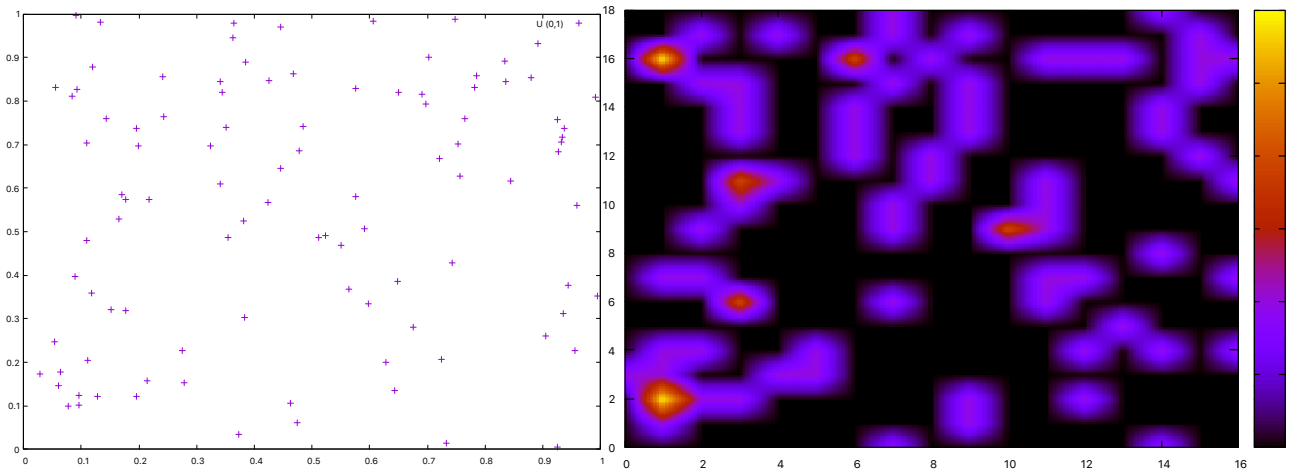

Figure 2.5: Heatmap of distribution for number of bins = 20.

For example, suppose we have a data set generated on a two-dimensional uniform distribution $U(0,1)$ and the number of bins = 20. Then the bottom left element on the matrix will have range, $0 < x, y < 0.05$. Thus, for any point in the data

set whose x and y-coordinate both fall between this range, $p = (0.002, 0.04)$ for example, the element of the matrix associated with this range will be incremented. The resulting heatmap of the matrix reveals "hot", or peak points in the data. Counting the number of visible peaks reveals the optimal number of clusters k in the data set, and sampling points in the range associated with these peaks give an optimal choice of initial centroids.
Since there are general rules for how many bins there should be in a histogram, such as the Freedman–Diaconis rule [11], the heatmap method provides a good means to approximate initial cluster centroids. Of course, this method works when combined with the elbow method to prove the validity of its results.

2.3 Nested Voronoi

Nested Voronoi diagrams are, as the name suggests, smaller Voronoi tessellations nested inside a larger Voronoi tessellation as shown in Figure. 3.1. When constructing a Voronoi diagram, the user has control only of its seeds, whereas in clustering analysis, the user has no control of its resulting centroids. Thus, nested Voronoi is an attempt to endow Voronoi diagrams with a naturalistic quality through taking the outputted centroids of the k-means algorithm and inputting them into the Voronoi algorithm as seeds for a Voronoi tessellation.

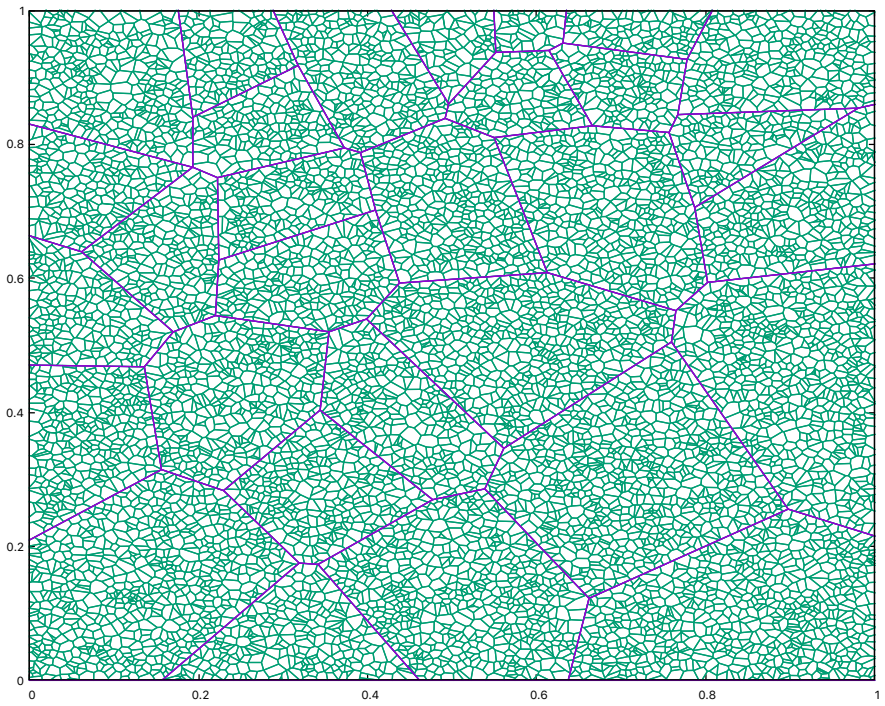

Figure 2.6: Nested Voronoi diagram for k=30.

2.3.1 Why combine Voronoi with Clustering Analysis?

As previously mentioned, the k-means algorithm provides an indirect partition of the data set into Voronoi cells, Fig. 2.7 . Therefore, each point in the data set is classified both by its cluster class, as well as the Voronoi cell that encloses it.

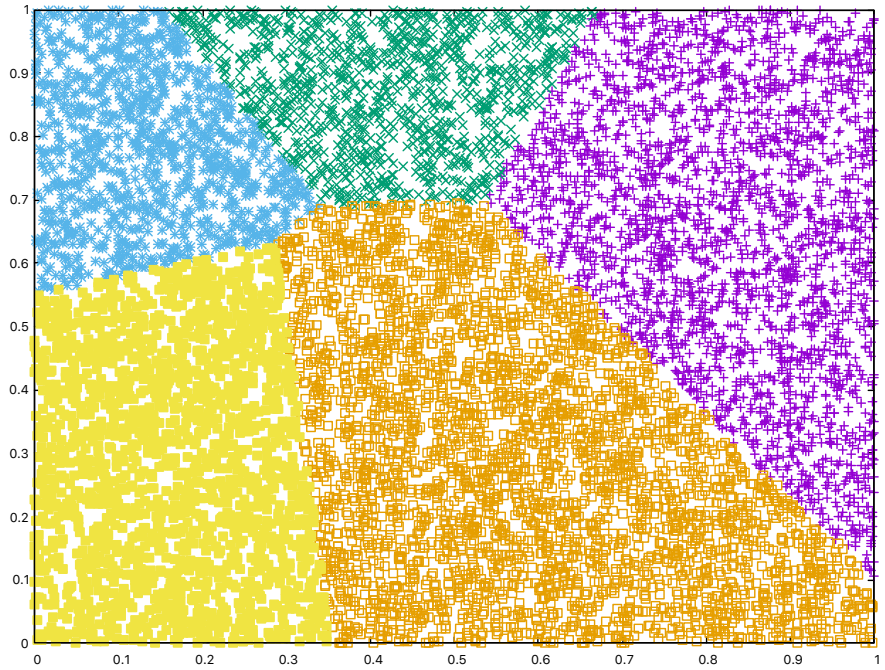

Figure 2.7: k-means convergence to Voronoifor k=5.

This classification becomes particularly useful when one wishes to produce nested Voronoi diagrams, since by the k-means algorithm, given an arbitrary point p and its corresponding cluster M and Voronoi cell $V(p)$, there should lie no point $p_i \in M_i$ such that $\exists\ p_i \in V(p_j)$.
Moreover, this simply means that for any nested Voronoi diagram, none of its edges will breach any of the edges of the larger Voronoi diagram which encloses it since $\forall p_i \in M_i, p_i \in V(p_i)$.

2.3.2 Applications

Nested Voronoi has a number of potential applications in fields such as data analytics, road-network planning and agriculture science.
In road-network planning, for example, nested Voronoi diagram can be used to establish an efficient and even flow of travel for motor vehicles across a landmass.
This derivation comes from the fact that given a point set P, and its associative Voronoi diagram $V(P)$, then for any point q on any of the Voronoi edges $e \in V(P)$, q is *always* equidistant between either two or three Voronoi seeds.
Now, letting these point in P represent cities, towns and suburbs on a continent, clustering analysis can identify k cluster locations, where population and/or trading is dense.
A first-sweep Voronoi diagram can then be generated for each of these cluster locations, providing motorway ring-roads around the locations on the continent where population and/or trading is dense.
Then, k second-sweep Voronoi diagrams can be generated for each of the cells in the first-sweep Voronoi diagram, providing ring roads around the locations that make up each cluster and consequently, break-points along the motorway.

2.3.3 Problem Statement

Given that this is the first implementation of nested Voronoi diagrams and there is no existing literature on the topic, this paper focuses its efforts on implementing a nested Voronoi diagram in a closed form, that is, a nested Voronoi diagram for the purposes of representation, and not combined with any real practical application.
Even though there exists no literature on nested Voronoi diagrams, there exists bountiful literature on both clustering analysis and Voronoi diagrams. Thus, this paper combines aspects of each literature to present not only a serial implementation for nested Voronoi diagrams, but a parallel implementation.
Indeed, the parallel implementation for nested Voronoi diagrams is derived from a parallel implementation of Shamos and Hoey divide-and-conquer strategy for constructing a non-nested Voronoi diagram.
All implementations of nested Voronoi diagrams in this paper are performed on a 2-dimensional uniform distribution $U(0, 1)$, where unbounded cells, whose seeds form an unbounded convex region are made bounded by "boxing" them in with either four simple line equations, in the case of the first-sweep diagram, or the edges of a larger Voronoi cell, in the case of a second-sweep nested Voronoi diagram.

Chapter 3

Serial Design & Implementation

Now that the fundamental theory behind Voronoi diagrams and clustering analysis has been covered, in this chapter, the general approach to implementing Nested Voronoi diagrams in serial will be discussed. The design structure of the code is explained, such as the data structures and classes used, as well as any of the core geometrical routines and algorithms illustrated.

3.1 Design Approach

When writing code in C++, steps must be taken to break the code up into as many logical partitions as possible. All the serial code for this paper was written in `C++17`, allowing use of object oriented programming and many of the newer features of the standard library (STL) such as lambda expressions and template functions. An endeavour was made to use as much functionality from the standard library (STL) as possible, not only because this is considered good practice, but because it also helps to make the code more portable.

3.1.1 Design Structure

Figure 3.1 illustrates the overall structure and design of the code - in both serial and parallel. The code is split into five main files.

- `main.cc` - This is the main C++ file and is used for invoking the clustering analysis and Voronoi operations. To do this, the output of the clustering analysis operation is taken and inputted into the Voronoi operation.
- `voronoi.cc` - Code for a `Voronoi` class and the construction of a Voronoi diagram, either nested or non-nested. It also contains code for merging of two `Voronoi` objects into one whole Voronoi diagram.
- `cluster.cc` - Code for a `Cluster` class and implements the k-means clustering algorithm to partition the data set into k different clusters.
- `geometry.cc` - Code for holding geometrical data structures and routines.
- `empirical.cc` - Code for preforming empirical analysis on a given data set.
- `heuristic.cc` - Code for heuristic techniques necessary for the discovery of the number k cluster points in the k-means algorithm.

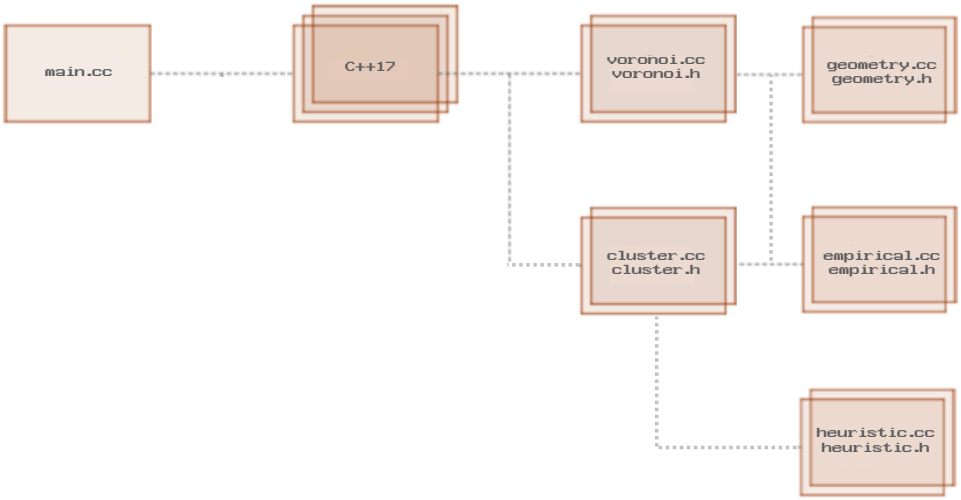

Figure 3.1: Structure for C++ code implementation.

`voronoi.cc` and `cluster.cc` are the core files. `voronoi.cc` depends on routine operations contained within the `geometry.cc` file. Likewise, `cluster.cc` depends on routine operations contained in the `geometry.cc`, `empirical.cc` and `heuristic.cc` files.

3.1.2 Data Structures

For this paper, two classes were written, `Voronoi` and `Cluster`, each of which are responsible for storing and generating the two core components of the paper - clusters and a Voronoi diagram for a given data set. All other data structures are geometrical data structures and are contained within the `geometry.cc` file. These geometrical data structures include `Point`, `Line`, `Bisector`, `Edge` and `Cell` and represent their namesake.

3.2 Implementation

In theory, the general approach for constructing a nested Voronoi diagram is as follows:

1. Import a 2-dimensional data set, or alternatively, sample from a 2-dimensional random distribution.
2. Invoke a heuristic routine in `heuristic.cc` to approximate the number of clusters, k, in the data set.
3. Create a `Cluster` object and generate k clusters by partitioning the data set into k distinct subsets.
4. Construct a first-sweep `Voronoi` diagram using the k centroid points outputted from the k-means algorithm as seeds for the tessalation.

5. Generate k second-sweep `Voronoi` diagrams for each of the k cells in the first-sweep tessellation, using the remaining points enclosed in each cell as seeds for the inner tessellations.

The next logical step is translating this theory into code. The next few section will walk through the selected algorithms and elucidate any core geometrical or empirical routines that were used in the construction of a nested Voronoi diagram.

3.2.1 Geometrical Concepts

Two of the core geometrical routines used throughout this paper for the construction of a Voronoi diagram are the cross product and euclidean distance routines and can be found in the `geometry.cc` file. Both routines are fairly simple to implement in code, but are considered extremely powerful tools in the field of computational geometry.

Euclidean metric

Euclidean space is the fundamental space of classical geometry and is of practical importance because Euclidean space of two and three dimensions forms the arena in which real physical objects are arranged.
Thus, given two points p and q in Euclidean n-space, one may recall from a first year basic linear algebra class that the distance between these two points is the length of the line segment connecting them, denoted $\overline{pq}$, and is given by the Pythagorean formula:

$$d(p,q) = d(q,p) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{3.1}$$

An illustration of the Pythagorean formula is shown in Figure 3.2.
Euclidean distance, despite its simplicity, is the only metric that is invariant under rotation. Moreover, this conforms with the general qualities of the universe, which is also rotation invariant. This is one of the core reasons why the Euclidean metric is so useful.

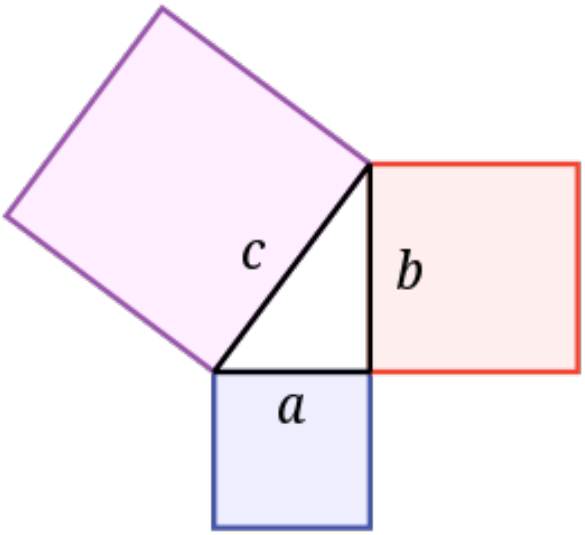

Figure 3.2: The Pythagorean theorem states that the sum of the areas of the two squares on the legs (a and b) equals the area of the square on the hypotenuse (c).

Orientation of Points

The orientation of something, *or somebody*, refers to their position relative to something else. Abstractly speaking, the position of anything in Euclidean space can only really be described relative to its orientation to something else. The cross product routine is a way to encode this sense of orientation into machines and is for this reason a core concept in computational geometry. Given two vectors $\overrightarrow{v_1} = x_1\hat{i}+y_1\hat{j}+z_1\hat{k}$ and $\overrightarrow{v_2} = x_2\hat{i}+y_2\hat{j}+z_2\hat{k}$, one may recall from a first-year class in linear algebra that the cross product of these two vectors $\overrightarrow{v_1} \times \overrightarrow{v_2}$ is orthogonal to both $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ and its direction is given by the "right-hand-rule".
Therefore, If we have two arbitrary vectors in the ij-plane such that $z_1 = z_2 = 0$ then the resulting $\overrightarrow{v_1} \times \overrightarrow{v_2}$ will be given by $\alpha\hat{k}$ and the sign of α will depend on the orientation of the vector. [5]
This can be generalized for points. Given three points $\overrightarrow{p_1} = (x_1, y_1)$, $\overrightarrow{p_2} = (x_2, y_2)$ and some other arbitrary point $\overrightarrow{p_i} = (x_i, y_i)$ in a point set P, the mnemonic for cross product in terms of determinants is used to determine the orientation of p_i with respect to the other two points, and is given by:

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ x_2 - x_1 & y_2 - x_1 & 0 \\ x_i - x_1 & y_1 - x_1 & 0 \end{vmatrix} = \begin{vmatrix} x_2 - x_1 & y_2 - x_1 \\ x_i - x_i & y_i - y1 \end{vmatrix} \hat{k} \tag{3.2}$$

where the sign of the determinant reveals whether the point is ordered clockwise (positive) or anti-clockwise (negative). Figure 3.3 illustratively demonstrates this logic.

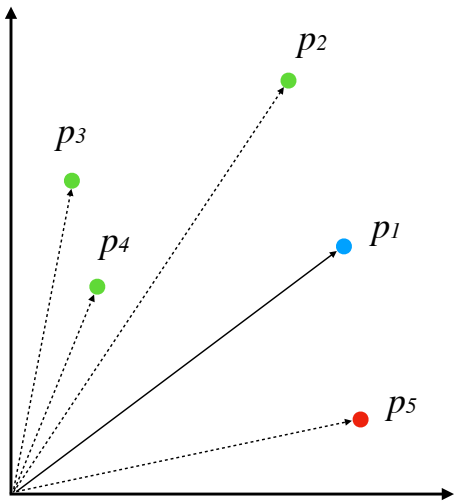

Figure 3.3: The right hand rule reveals that the cross product of $\overrightarrow{p_1}$ with $\overrightarrow{p_2}$, $\overrightarrow{p_3}$, $\overrightarrow{p_4}$ will be in the same direction, whereas the cross product of $\overrightarrow{p_1}$ with $\overrightarrow{p_5}$ will be in the opposite direction and hence opposite sign.

3.2.2 Empirical concepts

Whilst the construction of the Voronoi diagram relies only on geometrical concepts, the k-means algorithm uses empirical analysis to partition the data set into clusters, or Voronoi cells. This section will elucidate some of the core empirical concepts used in the k-means algorithm with retrospect to Voronoi diagrams.

Expectation

The expectation of a random variable X is one of the most useful concepts in probability theory. If X is a discrete random variable that takes on one of the possible values $x_1, x_2, ..., x_n$ then the expectation or expected value of X, also called the mean of X and denoted by $E[X]$, is defined by

$$E[X] = \sum_i x_i P\{X = x_i\} \tag{3.3}$$

[13]. Intuitively, given a number of darts on dart board, the expected value, or mean, approximates where on the dart board the throwers darts tended to, as illustrated in Figure 3.4. Formally, this mean point is known as the geometric "center of mass", and is a weighted average of the possible values that X can take on. As the name suggests, the k-means algorithm relies on the expectation values, or means, to partition the data into clusters.
In particular, the k-means algorithm uses a simple average in each dimension to approximate the mean point, or centroid, c_j of a given cluster M_j. In two-dimensions, this point is given by

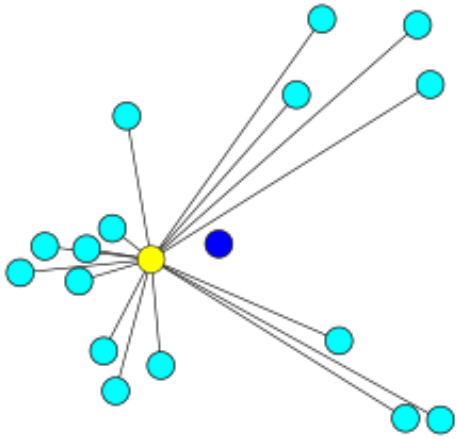

Figure 3.4: The blue dot illustrates the mean point, or "center of mass", and the yellow dot illustrates the median point, or "central tendency".

$$E[M_j] = \frac{1}{n}\Big(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i\Big) = (\mu_X, \mu_Y) = c_j \tag{3.4}$$

Thus, given a cluster of points $M_j = \{p_1, p_2, ..., p_m\}$, the mean, or centroid of this cluster, c_j, is a point for which all of these points "encircle".

Mean squared error

The mean squared error (M.S.E) is calculated by taking the average of the square of the difference between the original and predicted values of the data. In the context of the k-means algorithm, it is the averaged squared Euclidean distance between the centroid value and its cluster values and is a measure of the within-cluster variance. It is given by

$$\frac{1}{n}\sum_{i=1}^{n}\Big(\min d^2(c_j, p_i)\Big) \tag{3.5}$$

In geometrical terms, since the cluster is described geometrically as a circle in the 2-dimensional space, the variance approximates the diameter, or informally speaking, the size of the cluster. Thus, the k-means algorithm attempts to minimize the variance, and so minimize the diameter, or size, of each cluster through the use of the M.S.E metric.

3.2.3 Algorithms

Three core algorithms were used in this paper - the k-means algorithm, the naive Voronoi algorithm, and Shamos and Hoey's divide-and-conquer algorithm for Voronoi

diagrams. For the construction of a Voronoi diagram, the Naive approach is combined with the divide-and-conquer algorithm for Voronoi diagrams. Now that the core geometrical and empirical tools involved in these algorithms has been covered, this section will detail the implementation of these algorithms.

k-means algorithm

The k-means algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. The core concepts contained in the algorithm have already been covered. The pseudocode for the algorithm can be found at the end of this chapter.

Naive Approach

The naive approach for constructing a Voronoi diagram involves determining the convex region for each point p bounded by the intersection of n$-$1 half-planes. It uses two of the core geometrical concepts discussed in Section 3.2.1. The psudeocode for the algorithm can be found at the end of this chapter.
An illustration of what the algorithm looks like on the first iteration is shown in Figure 3.5. It is important to note that there may exist a number of intersections in the anticlockwise direction. Thus, the intersection point with minimal Euclidean distance from m on the first iteration, and the previously found vertex thereafter, is accepted as the new vertex for the Voronoi cell region, $V(p)$.

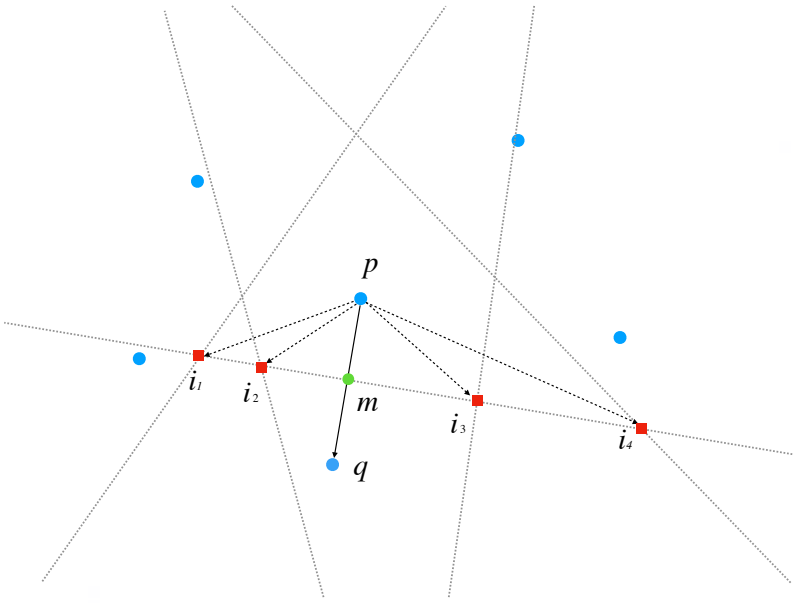

Figure 3.5: The right hand rule tell us that the cross product $\overrightarrow{pq}$ with $\overrightarrow{pi_1}$ and $\overrightarrow{pi_2}$ is in a clockwise direction and hence will have a positive determinant. Whereas, the cross product of $\overrightarrow{pq}$ with $\overrightarrow{pi_3}$ and $\overrightarrow{pi_3}$ is in an anticlockwise direction and hence will have a negative determinant.

Merging two Voronoi diagrams

Given a set of Voronoi seeds, S, the merging of two Voronoi diagrams involves computing the set of perpendicular bisectors of the sets P_L and P_R i.e., $B(P_L, P_R)$, of all Voronoi edges of $V(O)$ that separates the seeds in P_L from regions of sites in P_L. The idea of merging based on the fact that the edges of $B(P_L, P_R)$ form a single y-monotone polygonal chain, as illustrated in Figure 3.6, which reveals where one should add new and remove unwanted vertices.

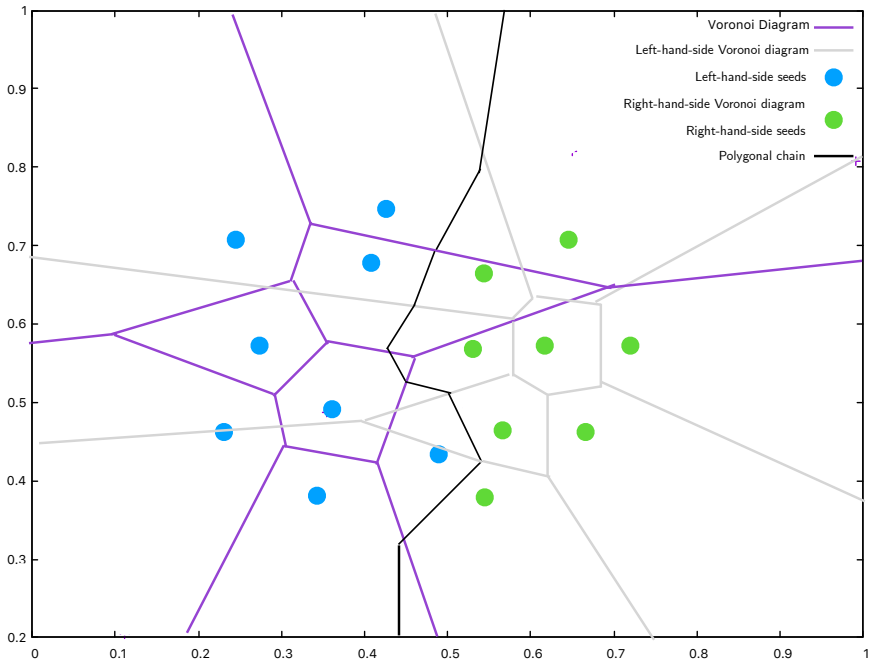

Figure 3.6: Polygonal chain indicates where on should add new and remove unwanted vertices.

Once this polygonal chain is found, it is a matter of 'splicing" in and out of the chain. The approach here is to add to the cell on the left, call it $V(p)$, and to the cell on the right, call it $V(q)$, any vertices of the chain which are enclosed within either of these cells. Once the vertices of the merged cell are added to both cells, the removal of the unwanted vertices and edges can be done on the fly, through removing any vertex in $V(p)$ that lies to the right of this polygonal chain (clockwise), and likewise, removing any vertex in $V(q)$ that lies to the left of this polygonal chain (anticlockwise). The psudeocode for the algorithm can be found at the end of this chapter.

3.3 Results

This section accounts the results for each of the prior algorithms stated in Section 3.2.3. The test bed architecture for these serial implementation results was done on a 1.80GHz Intel Core i5-5350U CPU, with 3Mb cache and 2 cores.
The optimized serial results accounts for any speed-up obtained by combining the naive approach with Shamos and Hoey's divide and conquer algorithm for the construction of a Voronoi diagram and preforming a "two-step" merge over the traditional single one-step merge in the literature.

3.3.1 Serial Implementation

k-means algorithm

The k-means algorithm was tested on a relatively simple data set containing three distinct Gaussian distributions and its performance is determined by two distinct

factors - its speed and the validity of its results.
For the initialization stage of the algorithm, the number of clusters k was approximated using the elbow method covered in Section.. and the initial cluster centroids was sampled using k random points in the data set. Since the k-means algorithm chooses these intial cluster centroids randomly, this paper uses the `std::random_device` from the C++ `<random>` library to obtain different results on different configurations in order to compare results and determine the algorithm's proficiency for different initial centroids.
Since each configuration of the algorithm produces different results the results for each test in this section is averaged over twenty different configurations.
In the context of algorithmic speed, the k-means algorithm, as expected, has a considerably large time complexity, Figure. 3.7 illustrates its quadratic scaling. This is due to the large number of mean calculations necessary in the update step. However, despite this large time complexity, the algorithm scales far beyond the Voronoi diagram, which stops scaling over 30,000 seeds.

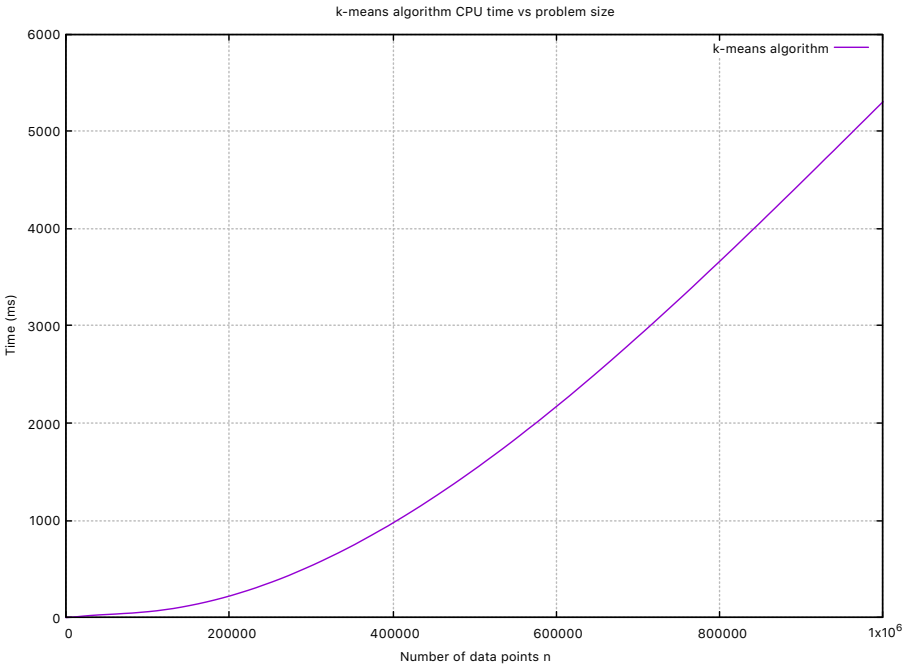

Figure 3.7: k-means algorithm for CPU time vs problem size.

In the context of results, the k-means algorithm produced valid results on 75% of its configurations. Figure. 3.8 illustrates a poor configuration of the k-means algorithm, whereas Figure. 3.9 illustrates a good configuration.

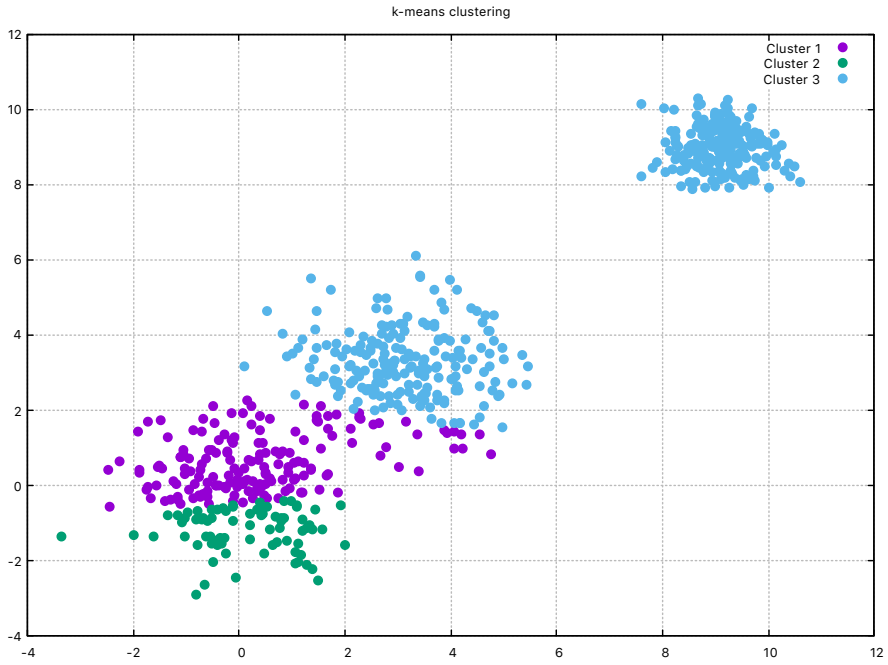

Figure 3.8: Poor configuration of k-means algorithm.

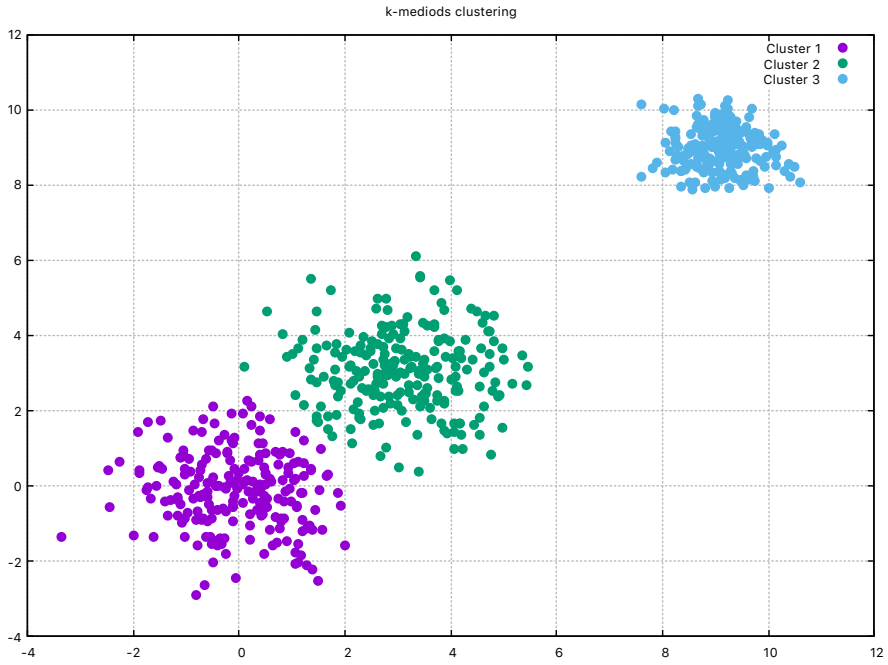

Figure 3.9: Good configuration of k-mediods algorithm.

These results suggest that improvements are necessary for both aspects of the algorithm's performance. There does not exist a go-to heuristic to determine good initial centroid values, however, a quick-and-dirty solution to this problem involves rerunning the algorithm, averaging the centroid results, and using these averaged results as input into the algorithm in order to maximize the quality of any local minima we find ourselves in. This simple strategy proved effective for reruns in

serial with as little as four reruns. Using the heatmap method covered in 2.2.2 to choose good initial starting centroids also proved equally effective despite the heuristic not being an automatic process.

Voronoi Diagram

As expected, the naive approach for constructing a Voronoi diagram has a considerably large time complexity and does not scale well for a large number of seeds; Figure 3.10, illustrates the algorithm's quadratic scaling.
This large time complexity is mainly due to the large number of computations that must be done in order to determine the convex region for each cell bounded by the intersection of its $n-1$ half-planes.
Each point must compute $n-1$ half-planes. Thus, repeating this process for the n number of seeds, we have a $O(n^2)$ algorithm for constructing a Voronoi diagram.

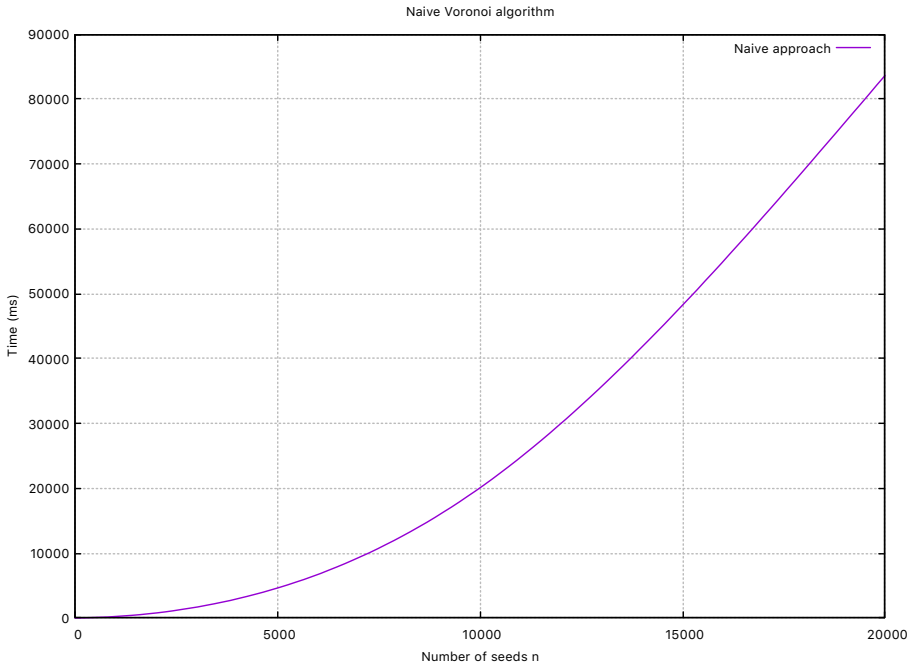

Figure 3.10: Naive approach for CPU time vs problem size.

3.3.2 Optimized Serial Implementation

Voronoi Diagram

Divide-and-conquer techniques are the basis of efficient algorithms and are used in a sundry of different problems. Thus, it is no surprise that the optimized method for constructing a Voronoi diagram using Shamos and Hoey's divide-and-conquer paradigm preforms considerably better than the naive approach.
Through dividing the problem recursively into sub-problems the issue of computing the $n-1$ bisectors is reduced enormously, as shown in Figure 3.11.

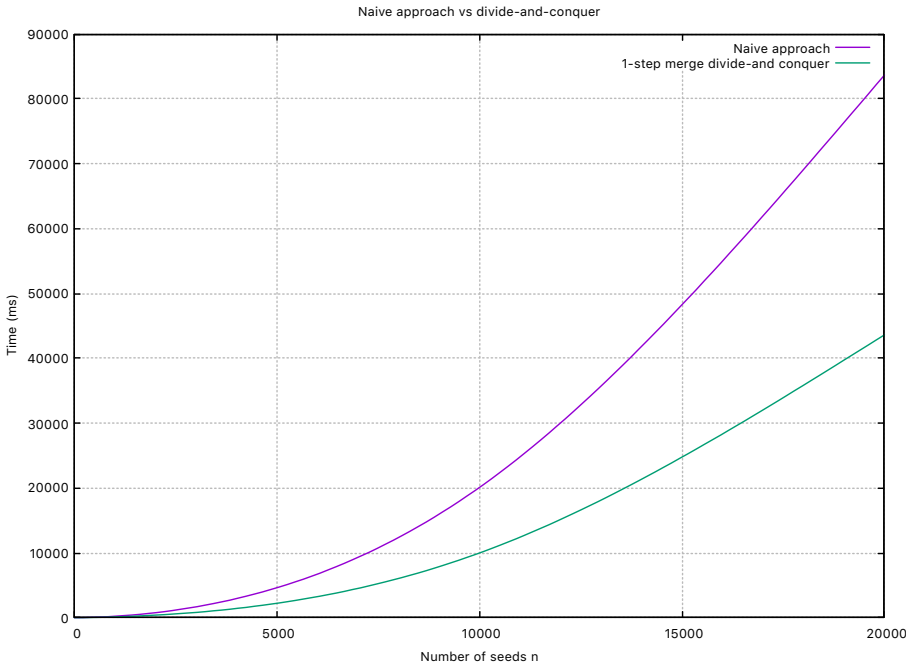

Figure 3.11: Comparison for divide-and-conquer approach for CPU time vs problem size.

Moreover, an additional speed-up can be achieved through dividing the problem into more than 2 sub-problems, and consequently calling the merge function more than once. This will from now on be referred to as a 'two-step' Voronoi merge. The additional speed-up of a two-step merge can be seen in Figure. 3.12

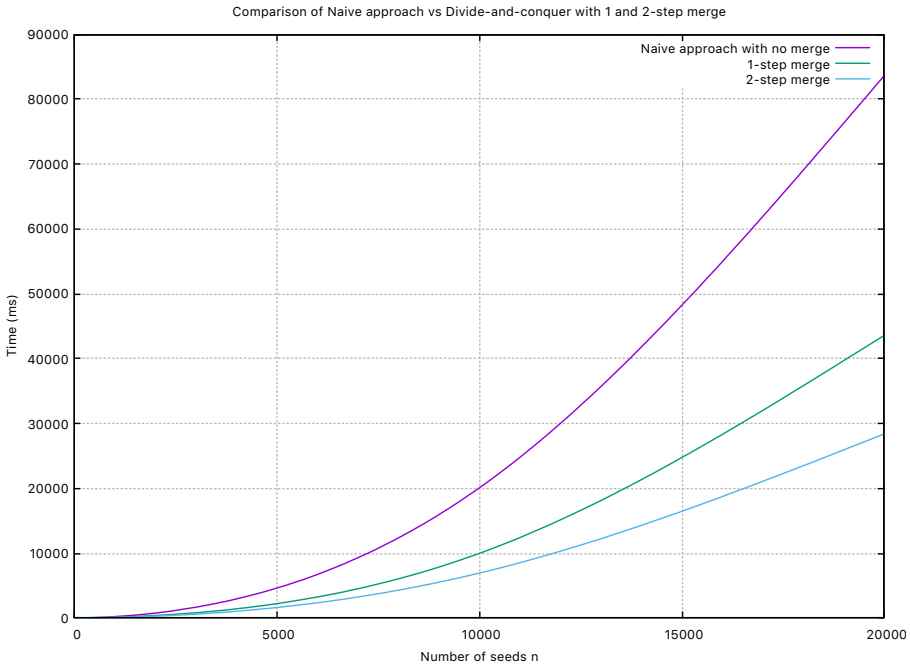

Figure 3.12: Comparison for divide-and-conquer approach with different number of merges.

As you can see, the construction time for a Voronoi diagram has been reduced

considerably. A major motivation for this paper was not only parallelizing Shamos and Hoey's divide-and-conquer method, but providing an implementation that could divide the problem into more than two sub-problems. The results suggest that a parallel implementation for the two-step divide-and-conquer algorithm would further enhance any speed-up obtained. For illustration purposes, the evolution of a two-step Voronoi merge process for a large scale Voronoi diagram is shown in Figure 3.13, 3.14 and 3.16.

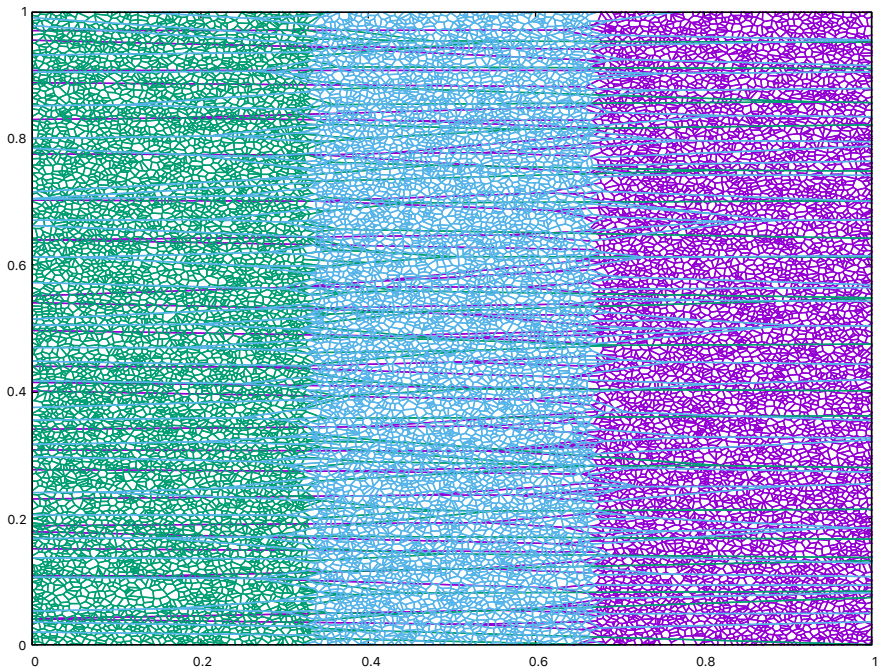

Figure 3.13: Two-step Voronoi merge before the first merge.

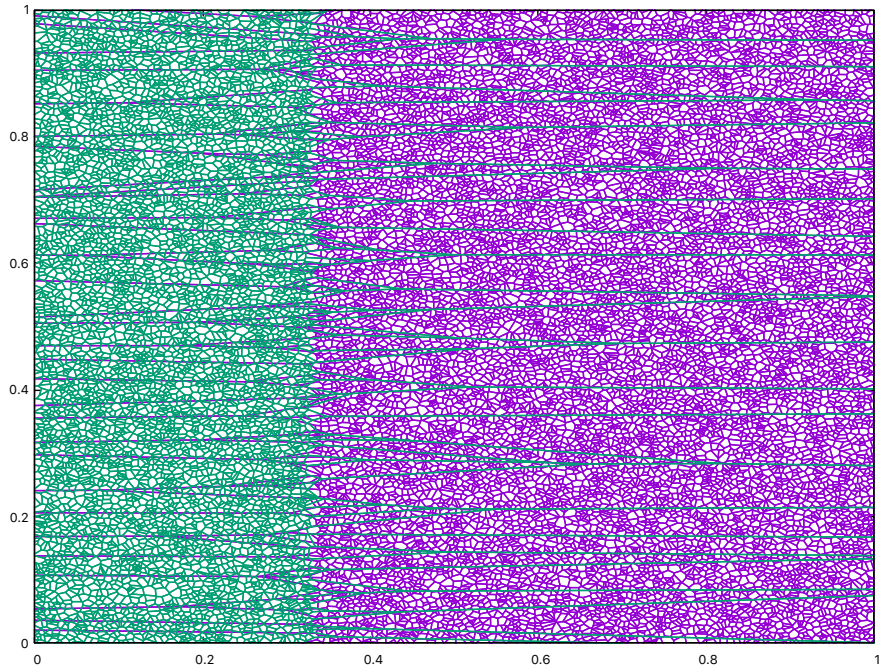

Figure 3.14: Two-step Voronoi merge after the first merge (merging right-hand-side Voronoi with middle Voronoi diagram from right to left).

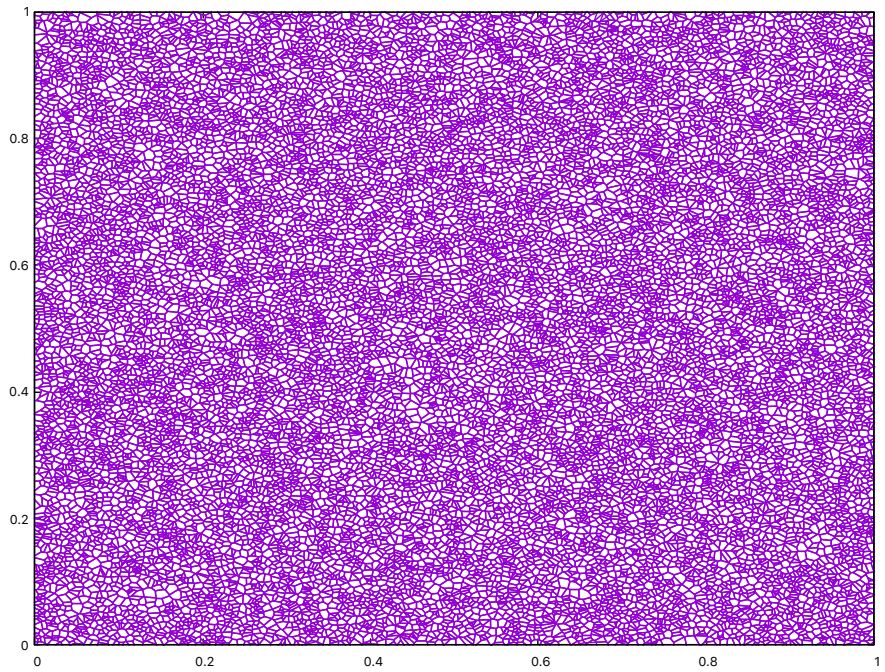

Figure 3.15: Two-step Voronoi merge after the second merge (merging middle Voronoi with left-hand-side Voronoi diagram).

Nested Voronoi Diagrams

When constructing nested Voronoi diagrams, the domain is in many ways being decomposed into k Voronoi cells. Moreover, this means that one should expect the

nested Voronoi diagram to preform noticeably better than a traditional Voronoi diagram. This is for obvious reasons, since the problem of the *n-1* half planes has been reduced to k sub problems.
The result is a nested Voronoi diagram that does not look too dissimilar to a standard Voronoi diagram, which is why nested Voronoi diagrams could be used as an alternative when constructing large scale traditional Voronoi diagrams.
Of course, one needs to factor in the time it would take to retrieve the k cluster centroids for the first-sweep tessellation, but this paper only uses clustering analysis as a means to endow a naturalistic quality to the Voronoi diagram, one could just as easily use a quasi-random number generator that would provide points that fill the data space well.

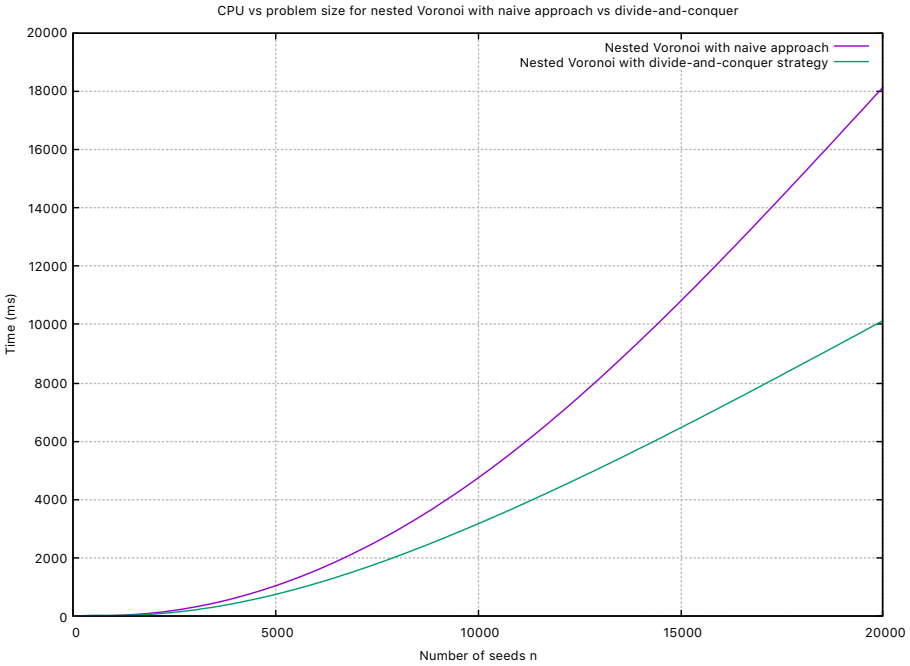

Figure 3.16: Results for the construction of a nested Voronoi diagram with different approaches.

Algorithm 1 k-means algorithm

MSE = LargeNumber

Select initial cluster centroids $\{c_j\}_{j=1}^{k}$.

while (MSE < OldMSE) **do**
    OldMSE = MSE;
    MSE' = 0;
    for $j = 1$ **to** k **do**
        $m_j = 0; n_j = 0;$
    end for
    for $i = 1$ **to** n **do**
        for $j = 1$ **to** k **do**
            compute the squared Euclidean distance $d^2(p_i, c_j)$;
        end for
        find closest centroid to point c_l to p_i;
        c'$_l = c'_l + p_i;$ $\quad n'_l = n'_l + 1;$
        MSE' = MSE' + $d^2(p_i, c_l)$;
    end for
    for $j = 1$ **to** k **do**
        $n_j = max(n_j, 1);\ c_j = \frac{c'_j}{n_j};$
    end for
    MSE = MSE';
end while

Algorithm 2 Naive Voronoi

for $i = 0$ **to** n **do**
    for $j = 0, j \neq i$ **to** n **do**
        Calculate and store perpendicular bisector of the line segment $\overline{p_i p_j}$;
        Find the point p_k with minimal Euclidean distance to p_i, i.e
        $p_k = \min d(p_i, p_j);$
    end for
    $m = \text{midpoint}(\overline{p_i p_k})$
    while new_vertex has not already been visited **do**
        For the perpendicular bisector of the line segment $\overline{p_i p_k}$, find its
        intersection with all other bisectors, call these intersection points $\{i\}_{j=1}^{m}$.
        for $k = 0$ **to** m **do**
            if $\overrightarrow{p_i m} \times \overrightarrow{p_i i_k} == -1$ and $i_k = \min d(p_i, i_k)$ **then**
                new_vertex = i_k
            end if
        end for
    end while
end for

Algorithm 3 Merge Voronoi

Input: Voronoi diagrams $V(P_L)$ and $Vor(S_R)$ of sets P_L and P_R.
Output: Voronoi diagram $V(P)$ of the whole set.

Construct the Convex hull of P_L and P_R.
Find the lowest common support line $\overline{pq}$ that connects the two hulls, call it $B(P_L, P_R)$
Draw the perpendicular bisector of the line segment $\overline{pq}$.

while $B(P_L, P_R)$ is not the upper common support line **do**
    Find the intersection point of $B(P_L, P_R)$ with the boundary of $V(P_L$,
    call it p'_L
    Find the intersection point of $B(P_L, P_R)$ with the boundary of $V(P_R)$,
    call it q'_R.
    if y-value of $p'_L < q'_R$ **then**
        $w_i = p'_L$
        P_L = the seed on the other side of the edge intersection.
    else
        $w_i = q'_R$
        P_R = the seed on the other side of the edge intersection.
    end if
    For the cell with generating the generating seed P_L, remove any vertex that lies anticlockwise to the line segment $\overline{w_i w_i + 1}$ connecting two points of the polygonal chain and add the vertices of the polygonal chain, w_i, and $w_i + 1$ to the cell.
    Likewise, for P_R, remove any vertex that lies clockwise to the line segment $\overline{w_i w_i + 1}$ connecting two points of the polygonal chain and add the vertices of the polygonal chain to the cell.
end while

Chapter 4

Parallel Design & Implementation

Now that results have been obtained for the serial implementation of the core algorithms in this paper, this chapter presents a parallel approach for the construction of non-nested and nested Voronoi diagrams. The initial intentions was to also parallelize the k-means algorithm, however, since the k-means algorithm scales far beyond the Voronoi algorithm, it made little sense to do this, and so, all parallel implementations are strictly based on the Voronoi constructions.

4.1 Design Approach

The parallel strategy in this chapter used multi-threading techniques from the C++17 standard library (S.T.L). Initially, the intention was to use the `boost.mpi` library to distribute k clusters to k different processes and use a mixture of process and thread based parallelism to construct k nested Voronoi diagrams using the divide-and-conquer method. However, this strategy proved extremely difficult to implement as it introduced a number of bugs. As well as this, any speedup was not expected to differ greatly compared to using just threads. This is because the nature of the divide-and-conquer strategy provides an ready-made division of the problem into sub-problems that are waiting to be processed concurrently.

4.2 Implementation

The setup for parallelizing Shamos and Hoey's divide-and-conquer strategy for Voronoi diagrams is relatively simple. This is because divide-and-conquer algorithms are naturally adapted for execution in a multi-processor, or multi-threading machine, as previously mentioned. In other terms, through dividing the problem into sub-problems and successfully merging their solutions, the hard work is already done.
The general approach to implementing a Voronoi diagrams in parallel is as follows:

1. Sort the points in the data set by x-coordinate and divide the points into two or three distinct subsets.

2. For each subset, concurrently construct the Voronoi diagram associated with its points.

3. Merge each process from right to left.

As you can see, the implementation is similar in design to the serial implementation. This is because the algorithm is already adapted to be parallelized.
The general approach for constructing a nested Voronoi diagram used a different approach. In order for each nested Voronoi diagram to be constructed it must have a copy of the first-sweep Voronoi diagram, in order to determine which Voronoi cell of the first-sweep diagram the nested its seeds are enclosed within. The first-sweep diagram decomposes the problem space into k subsets. Therefore, each nested Voronoi diagram can be constructed on different threads.
The general approach to implementing nested Voronoi diagrams in parallel is as follows:

1. Generate a first-sweep Voronoi for the k cluster centroids.

2. For each of the k cells in the first-sweep Voronoi tessellation, sort their corresponding cluster by x-coordinate.

3. Concurrently construct k nested Voronoi diagrams.

4.3 Results

This section will first detail the results obtained from the thread-based approach used to parallelise a single non-nested Voronoi diagram using Shamos and Hoey's divide-and-conquer paradigm. It will then detail the results obtained from the multi-threading-based approach used for constructing a nested Voronoi diagram.

4.3.1 Voronoi Diagram

It is no surprise that through parallelizing Shamos and Hoey's divide-and-conquer strategy for constructing a Voronoi diagram that a significant speed-up was achieved. Through dividing the sub-problem successfully, the CPU could concurrently compute their independent parts of the Voronoi diagram.
Two different thread-based policies, asynchronous and deferred, were used compared to determine which policy was superior. 4.1.

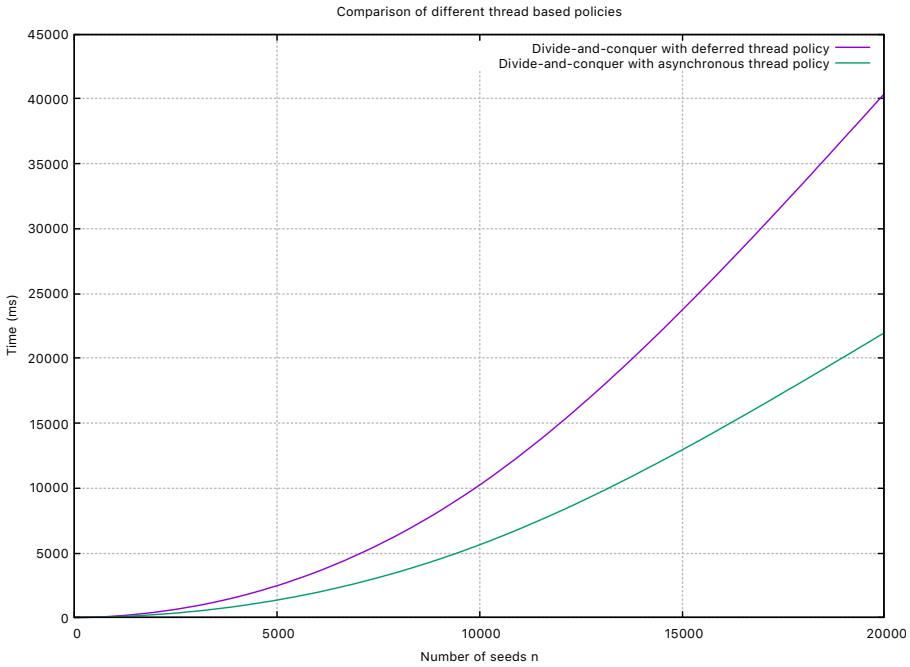

Figure 4.1:

As illustrated in Figure. 4.1, the asynchronous thread policy was optimal. This is because the problem of constructing each sub problem is independent of the other. Moreover, neither thread will need to be deferred until either a wait() or get() is called on the future. Thus, everything can be completed asynchronously.
Moreover, one of the focuses of this paper was to divide the problem of the Voronoi diagram into more than two sub problems. As seen in Section 3.3.2 , this focus was achieved. The desire was borne from the hope that parallelizing a two-step Voronoi merge would produce a significant speed-up compared to any other strategy.
This hope turned to reality, and the 3-step merge on asynchronous threads for a two-step merge produced an optimal solution, as shown here in Figure 4.2.

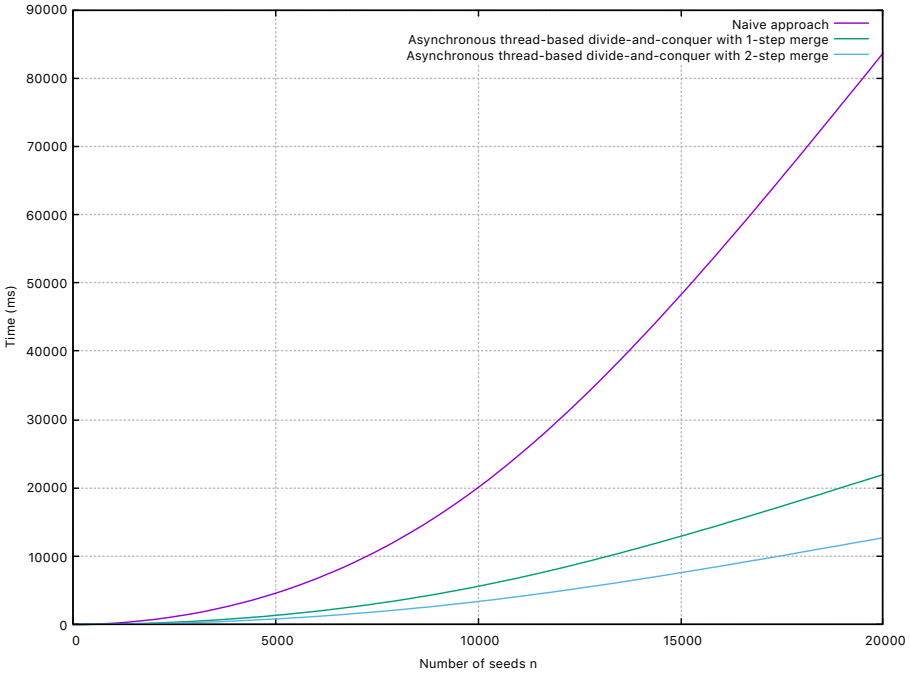

Figure 4.2:

To put these results into perspective. The naive approach for constructing a Voronoi diagram for $n = 20000$ took 83648ms = 83.648s to terminate, whereas the thread-based divide-and-conquer parallel strategy with a two-step merge took only 12711ms = 12.711s to terminate. Ultimately making for a speedup of 70.937s, translating to over one minute in waiting time.

Nested Voronoi Diagrams

The following results are obtained through each thread simultaneously constructing their nested Voronoi diagram. The results used a fixed number of nested Voronoi diagrams, k=4. As Figure. 4.3 suggests, a considerable speed-up is obtained through this multi-threading-based strategy.

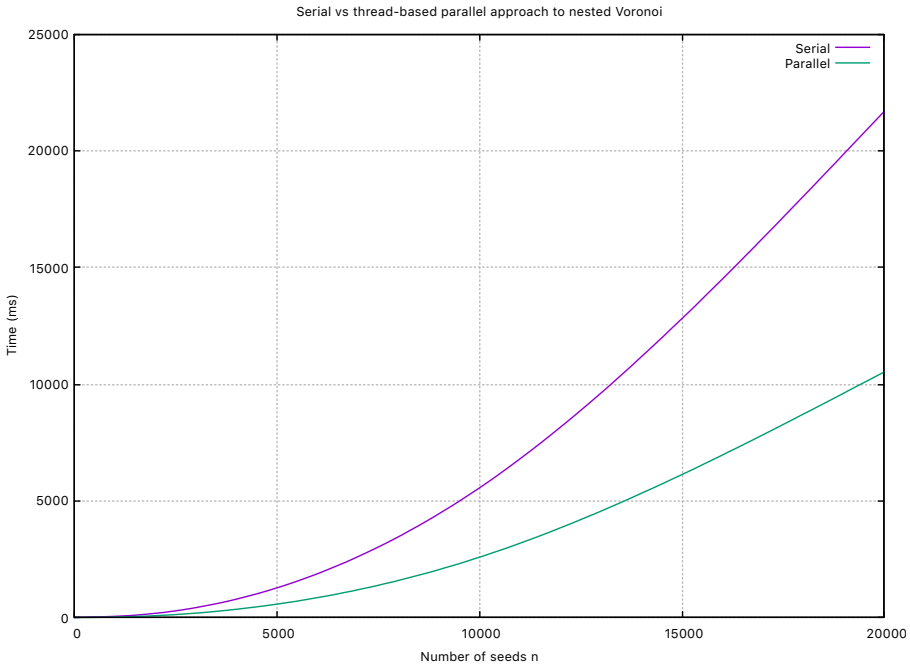

Figure 4.3:

This speed up here is considerable. This is for a similar reason to why there is a speed-up using the divide-and-conquer strategy. Through effectively decomposing the problem space into k Voronoi cells, each nested Voronoi diagram needs only to compute the $m-1$ bisectors for each of the m points enclosed in the larger Voronoi cell, as opposed to the $n-1$ bisectors for the total number of points in the problem space.

Chapter 5

Conclusion

5.1 Final Remarks

In this paper, a serial implementation for constructing a Voronoi diagram using C++17 was completed. This problem was optimized using Shamos-and-Hoey's divide-and-conquer approach and divided into three sub-problems to achieve an additional speed-up.
Furthermore, this strategy was parallelized using asynchronous threads to produce a near linear type scaling, and for large scale Voronoi tessellations, the waiting time was reduced enormously. Additionally, the first known serial and parallel implementation for nested Voronoi was produced.
The results suggest that if nested Voronoi diagrams are not to be used in the future for the purpose of their own application, they can be used as a substitute for the construction of large scale non-nested Voronoi tessellations. This is because the resulting tessellation is not too dissimilar to the non-nested Voronoi diagram for the same point set.

5.2 Future Work

In regards to fields of application, nested Voronoi diagrams should be used in any traditional Voronoi problem where a point set can be categorized into clusters, as per the example in 2.3.2.
The idea for producing nested Voronoi diagrams is borne from the desire to endow the Voronoi tessellation with a naturalistic quality. Through using k cluster points as initial seed values for the first-sweep Voronoi diagram, and all other cluster points within each cell thereafter, Voronoi diagrams are being produced as usual, but not in the traditional sense. Thus, any application of Voronoi diagrams that require the tessellation to inherit natural qualities should also use nested Voronoi diagrams.

With respect to further work on the code, the `geometry.cc` file in this paper provides a layman's library for the geometrical concepts necessary for constructing a Voronoi diagram. Encoding geometrical concepts into a machine proved to be extremely difficult at times during this project because of the large number of end-cases that need to be plugged in something as simple as a standard line equation. Thus, any future work on nested Voronoi diagrams should use a geometric library, such as the one provided by boost to make the core data structures more stable and

prevent the bugs that occur on some configurations of this program.
In order to achieve maximum performance, one should use the `boost.mpi` library to distribute k clusters to k different processes and use a mixture of process and thread based parallelism to construct k nested Voronoi diagrams in parallel using the divide-and-conquer method. As mentioned in Section 4.1, this approach proves extremely difficult to implement, however, rewriting some of the core geometrical data structures using the `boost.geometry` library, I believe that this speed-up could be achieved.

Bibliography

[1] Shamos M I, Hoey D. *Closest-point problems*. In Proc. 16th IEEE Annu. Symp. Found. Comput. Sci., Berkeley, US, 1975, pp.151–162.

[2] Inderjit S. Dhillon1 and Dharmendra S. Modha. *A Data-Clustering Algorithm On Distributed Memory Multiprocessors*, 'In Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence', p245–260, (2000).

[3] Okabe, Atsuyuki and Boots, Barry and Sugihara, Kokichi. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, John Wiley Sons, Inc. (1992).

[4] S. Fortune. *A Sweep-Line algorithm for Voronoi diagrams*, Algorithmica, Vol. 2 Issue 1-4, p153-174, 22p, (Nov. 1987).

[5] R. Morrin *5614: Assignment 5. Concurrency with Convex Hulls*. Assignment for TCD High-Performance Computing Class of 2020. May 2, 2020.

[6] Pixar in a Box. Website. Khan Academy. *Voronoi Partition, Patterns, Computer Animation* `https://www.khanacademy.org/computing/pixar/pattern/dino/v/patterns2_new`

[7] L. D. Libersky, et al. *High strain Lagrangian hydrodynamics: a three-dimensional SPH code for dynamic material response*, Journal of computational physics **109** (1), p67-75, (1993).

[8] R. Descartes. *Principia Philosophiae*. Ludovicus Elzevirius, Amsterdam, 1644.

[9] G. Voronoy (1908a). *Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire. Sur quelques propriétés des formes quadratiques positives parfaites* . Journal für die Reine und Angewandte Mathematik. 1908 (133): 97–178.

[10] Hartigan, J.A. *Clustering Algorithms*. Wiley (1975)

[11] Freedman, David; Diaconis, Persi. *On the histogram as a density estimator: L2 theory". Probability Theory and Related Fields*. 57 (4): 453–476. (December 1981).

[12] F. Aurenhammer, R. Klein. *Voronoi Diagrams*. Institut fur Grundlagen der Informationsverarbeitung Technische Universita t Graz Klosterwiesgasse 32/2 A-8010 Graz, Austria.

[13] Sheldon M. Ross. *Simulation*. Knovel Library. Academic Press, 2012.