Agent-based GIS Queries

Sahana Pandurangi Raghavendra

A Capstone Project

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science Software Engineering

University of Washington Bothell

2023

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington Bothell

Abstract

Agent-based GIS Queries

Sahana Pandurangi Raghavendra

Chair of the Supervisory Committee:

Professor Munehiro Fukuda

Computer Science

A Geographical Information System (GIS) is a vital software tool used across numerous domains to help store, manage, analyze, and visualize geospatial data. One of the core functions of the GIS is its ability to query, enabling scientists and researchers to analyze and discover underlying patterns and associations among various data layers. However, it is extremely time and computationally intensive to process complex spatial GIS queries on a single standalone system sequentially. Therefore, in this capstone project we parallelize GIS queries using an agent-based parallelization framework, Multi-Agent Spatial Simulation (MASS), and further explore the idea of incorporating computational geometry algorithms such as closest pair of points, range search and minimum spanning tree to GIS queries using agent propagation.

The major motivation behind integrating MASS library and GIS queries stems from the results of previous research in comparing MASS with other popular big data streaming tools. This research observed that agent-based computation using MASS yielded competitive performance and intuitive parallelization when introduced into data structures such as graphs. To verify this hypothesis of agent's superiority, we now would like to utilize MASS Agents in GIS queries where agents utilize computational geometry problems to find results of GIS queries through propagation over MASS Places spread across different computing nodes.

The significant contributions of this capstone project are to demonstrate GIS queries as a practical application of agent-based data analysis. Further, this project focuses on migrating the previous implementation of MASS-GIS system from Amazon Web Services (AWS) to the University of Washington Bothell computational clusters consisting of 24 computing nodes to achieve scalability and fine-grained partitioning of the GIS datasets suitable for agent-based parallel GIS queries. Sequential and parallel, attribute and spatial GIS queries are designed and implemented in this project using contextual query language (CQL) modules from GeoTools (open-source GIS package) and MASS. Additionally, we also extend and integrate the previous research on computational geometry algorithms using MASS to GIS queries. Algorithms such as the closest pair of points are incorporated into GIS queries to find the closest cities within a certain distance from a given city. Likewise range search is used to find all the cities in a given country given the range of geographical bounds of a country and minimum spanning tree is extended to find the shortest path between two points on a map. Lastly, we evaluate the performance of parallel agent-based GIS queries implemented using MASS. The results show that agent-based GIS queries using MASS-CQL and the closest pair of points algorithm are time efficient. Furthermore, MASS based GIS queries using computational geometry algorithms of the closest pair of points and range search provide 100% accuracy. However better optimization techniques need to be applied to improve the performance of agent-based GIS queries using the range search algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATION

To my husband, Abishek Prakash and my parents Mrs. Vani and Mr. Raghavendra P.K

Chapter 1

Introduction

1.1 Geographical Information System (GIS)

A Geographical Information System (GIS) is a software system that manages, visualizes, analyzes, and stores geographic data. GIS enables every object present in it to be connected to a location on the earth's surface which is further depicted on a map [3]. As shown in Figure 1, the GIS software depicts data on a map by overlaying multiple layers of different types of geographic data (such as water bodies, streets, and buildings) over one another on a base map.

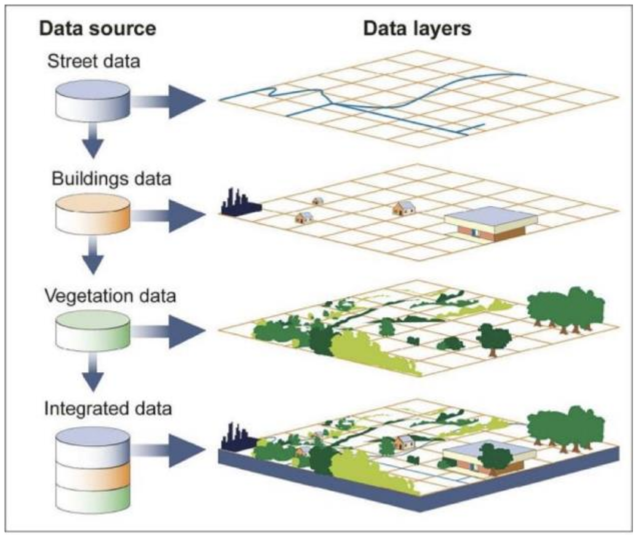


Figure 1: Overlaying different features of the real world on top of each other in the GIS environment [3].

One of the primary functions of the GIS is the ability to query, enabling the analysis and discovery of underlying patterns and relationships among various data layers. Hence, it finds applications in all location-centric disciplines such as transportation, meteorology, environmental studies, disaster management, satellite imaging, urban planning, etc.

Owing to the wide range of applicability of GIS, the sources of geographical data are enormous. Researchers and scientists analyzing these large-scale datasets using the GIS software are constantly challenged to query it in a time-efficient manner. Parallelizing GIS queries and executing them over a large dataset distributed across a cluster of computing nodes presents a compelling research opportunity with the potential to provide time-efficient GIS query processing [2]. Therefore, in this capstone project we parallelize GIS queries using an agent-based parallelization framework, Multi-Agent Spatial Simulation (MASS) [1] and also explore the idea of incorporating computational geometry algorithms to GIS queries.

1.2 Motivation

The major motivation behind parallelizing GIS queries with the MASS library stems from the previous research carried out in comparing MASS with other popular big data streaming tools by Prof. Munehiro Fukuda and his team at the University of Washington, Bothell. It was observed in this research that agent-based computation using MASS did not outperform conventional text data streaming but yielded competitive performance and intuitive parallelization when introduced into data structures such as graphs [4]. To verify this hypothesis of agent's superiority, the MASS research has been focused on solving basic computational geometry problems such as the closest pair of points, range search using KDTree and graph algorithms of minimum spanning tree using agents [5]. As a further step in this research endeavor, we now would like to practically implement this work in GIS where agents make use of some computational geometry problems (such as finding the shortest distance between two points on map) over GIS in response to user queries.

Some studies also indicated that parallel GIS queries using parallelization frameworks such as MPI, MapReduce and Hadoop showed faster performance on parallel spatial queries compared to sequential execution on a single node [6][7]. Thus, based on the fact that MASS provides competitive performance in comparison to MapReduce and Spark in graph-based applications [4], implementing MASS in GIS queries is a promising research direction.

1.3 Problem Definition

This capstone project builds upon the previous implementation of the MASS-GIS system carried out by M. Sieling [11]. The previous implementation focuses on GIS data division and parallel data retrieval using MASS across only eight Amazon Elastic Compute Cloud (EC2) instances. However, this system encountered issues and failed to execute on the University of Washington Bothell computational clusters.

In this project, we first migrate the previous version of the MASS-GIS system from eight (8) EC2 instances on Amazon Web Services (AWS) to the University of Washington Bothell computational clusters consisting of 24 computing nodes. The benefits of this migration are twofold: one we can evaluate the performance of the system with more servers and achieve fine-grained partitioning of the GIS datasets suitable for agent-based parallel GIS queries, and second that it will enable the execution of the MASS-GIS system in a heterogeneous environment across cloud and non-cloud clusters in future. Additionally, we also replaced the raster image backdrop of the world with a tiled map from OpenStreetMap (an open-source project for creating geographical databases) [14] to depict specific details of the geographic locations on the map as compared to just having terrain and land boundaries.

Thereafter, we implement sequential attribute, spatial, and aggregate GIS queries (combining attribute and spatial GIS queries). These queries are executed over multiple layers of GIS datasets at once using contextual query language (CQL) modules from GeoTools [12], an open-source library for GIS and the results are visualized on the map.

Our approach to parallelizing attribute and spatial GIS queries can be broadly divided into two categories: 1. using the CQL module and MASS Places (a distributed array assigned over a cluster, each element in the Places matrix represents a fragment of the GIS dataset). 2. using computational geometry algorithms and MASS Agent [1] (executable instance that moves parallelly across places and interacts with other agents and places) propagation on MASS Places.

In our first parallelization approach, we divide the large GIS data across MASS Places on different computing nodes using the GIS dataset division algorithm by M. Sieling [11]. We then execute spatial or attribute GIS queries using the CQL module and MASS [1], on each of the GIS data fragments. The results of the queries are collected from the remote nodes and sent to the master node using MASS and visualized on the map using GeoTools.

In our second approach to parallelization, we extend and apply computational geometry algorithms to MASS-GIS systems to execute spatial queries and gather responses back using MASS Agents. Algorithms such as the closest pair of points [15] that is used to find all the cities (represented as points) within a certain distance of the given city, range search [15] to find all the cities (represented as points) within a certain geographical latitude and longitude positions and minimum spanning tree [16] to find the shortest path between two geographical coordinates, are integrated as spatial queries in the MASS GIS system.

We evaluate these parallel attribute and spatial GIS queries using MASS with datasets obtained from various government websites consisting of 300K and 500K data points for CPU scalability analysis. Lastly, we present our inferences on the performance of agent-based GIS queries using MASS.

1.4 Report Structure

The report that follows provides a detailed overview of the important concepts that form the basis of this project in chapter 2. Chapter 3 discusses the related research. Chapter 4 details our design and implementation strategies in parallelizing GIS queries using MASS and GeoTools. We present our results and evaluate the performance of agent-based GIS queries using MASS in chapter 5. The conclusion and some insightful inferences drawn from our experiments along with potential research directions are discussed in chapter 6.

Chapter 2

Background

In this chapter we provide a comprehensive summary of the important concepts that form the basis of our project.

2.1 Geographical Information System Data

GIS stores data in either raster or vector formats as using these types would help represent the real-world data accurately. Raster dataset as depicted in Figure 2 is a collection of cells or pixels which make up a grid structure [17]. Each cell in the raster data represents a geographical location and values in it are the attributes and properties of that area. Raster data is used when the information to be displayed is continuous across a region like landscapes or terrains, rainfall trends.

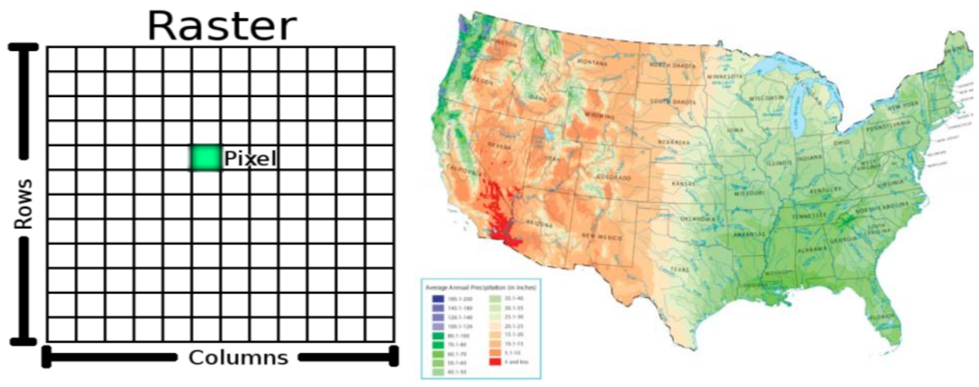


Figure 2: Raster data composed of pixels that display terrain information [17] [18].

On the other hand, vector data is used to depict real world features in the GIS like houses, roads, rivers (objects present on landscapes in the real world) [19]. Each object in vector data has a geometrical shape and a coordinate position as shown in Figure 3. For example, Seattle is represented by a point at a given x, y coordinate location in GIS using the cities vector data files. The vector object or feature can be represented using a point, line or polygon. GIS applications categorize vector features into layers. Each of the vector features / objects in the same layer have the same geometry and properties. Eg: Cities in the world that are represented using points. Thus in most of the GIS applications, the real world is decomposed into multiple layers of feature data (vector format) which are overlaid over one another onto the base map (raster format).

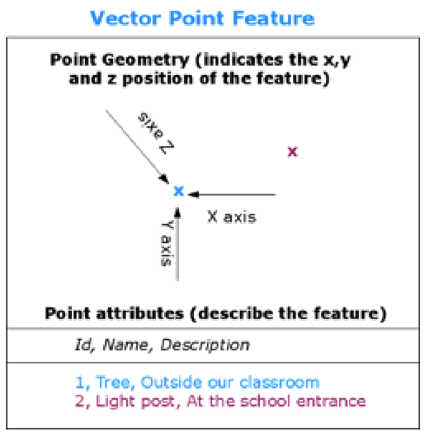

Vector Point Feature
Point Geometry (indicates the x,y and z position of the feature)
Z axis
X axis
Y axis
Point attributes (describe the feature)
Id, Name, Description
1, Tree, Outside our classroom
2, Light post, At the school entrance

Figure 3: A point object in GIS vector data represented by its X, Y and Z coordinates which can represent a city on a map [19].

2.2 Geographical Information System Queries

GIS queries can be broadly classified into two types, which are attribute based and spatial queries. Attribute queries select information from the data associated with the feature or from spatial databases integrated into the GIS software. Attributes that are queried can be a string, numeric or boolean data type. The results of an attribute-based query are highlighted on the map. Spatial queries select spatial features on the basis of their spatial relationships to other features in the GIS vector data. There are three different classes of spatial queries based on the spatial relationship that are used in the query, which are proximity, direction and topological spatial query [20]. Proximity spatial queries are when spatial relationships in the query are distance based. They can be expressed quantitatively as a metric and qualitatively as using verbal measures such as very far or near. Direction based spatial queries look for spatial features based on directions provided by the user as input. Lastly, topological spatial query is based on spatial relations that remain unaffected by transformations like stretching, shifting, rotating, or bending. They are based on topological relations such as adjacency, containment and intersection.

2.3 GIS Querying Languages

Query languages used to query the GIS data depend on the way in which it is queried. When GIS data is queried using an user interface provided by the GIS application like in QGIS, the UI translates the user query into the one that is understood by the GIS datastore like SQL. Another scenario would be when a spatial GIS database like PostGIS [21] is directly queried then Structured

Query Language (SQL) is used or when modules provided by libraries are used to query the shapefiles containing vector data then modules like Contextual Query Language (CQL) [12] are used.

2.4 GeoTools

GeoTools [22] is an open-source library written in JAVA that provides tools to manage GIS data. In our project we utilize GeoTools to read GIS vector data from shapefiles, manipulate this geospatial information from shapefiles as JAVA objects, query GIS data, serialize the GIS query results on remote, deserialize the GIS data on the master node and to visualize the query results on the map. We also use GeoTools to build large graphs from the GIS vector features representing streets or railroads of the United States and process the graph to implement the minimum spanning tree algorithm.

2.5 Multi-Agent Spatial Simulation (MASS)

Multi-Agent Spatial Simulation abbreviated as MASS [1] is an agent based parallel computing library for multi-agent and spatial simulation across a cluster of computing nodes. It was designed and developed at University of Washington Bothell by Prof. Munehiro Fukuda and his team. Many research projects have been undertaken to apply the MASS library to real world problems such as bioinformatics, climate analysis and influenza epidemic simulations that comprise of big data. MASS library has two major components called Places and Agents[1]. MASS Places represent a distributed matrix allocated over a cluster and MASS Agents are executable instances that move parallelly across MASS Places, collecting information by interaction with other MASS Agents and MASS Places. MASS library defines GraphPlaces, VertexPlaces and GraphAgents that are employed when data processed is maintained in a graph data structure. The distributed graph data structure is known as GraphPlaces which is an extension of MASS Places[4]. The MASS GraphPlaces are made up of MASS VertexPlaces. Vertices of the graph data structure are mapped to MASS VertexPlace which contain information about the edges from the vertex and vertex itself. MASS GraphAgents, extended from MASS Agents base class, move across MASS VertexPlace to execute computations or exchange information.

Chapter 3

Related Works

This chapter outlines the existing agent-based models that provide built-in support to GIS and discusses various research endeavors aimed at parallelizing GIS. It differentiates these efforts from our approach in this capstone project.

3.1 Sequential Agent-based Models Integrated with GIS and Their Implementation of GIS Queries

Agent-based modeling (ABM) is effective in simulating complex systems wherein agents interact with other agents and the environment under a given set of rules. The collective behavior of agents give rise to patterns that aid deeper analysis of the system. Therefore, it is important to study the GIS system and its queries from the perspective of ABM. This section describes some well-known agent-based models that have been integrated with GIS such as NetLogo [9] [23] [24], Recursive Porous Agent Simulation Toolkit (Repast J) [25], MASON [27] [28] and AnyLogic [29] in detail.

3.1.1 NetLogo

Netlogo is a popular agent-based modeling system that provides support for GIS integration. It uses a variant of Logo programming language and is based on a structured, hierarchical program model [9]. Two basic components in Netlogo are patches and turtles. Patches are cells in the NetLogo environment whereas turtles are agents (executable programs) that move across the patches. In order to work on GIS data in NetLogo, the first step is to import the vector and raster data into the NetLogo environment. Next, map vector features into the Netlogo patches components, draw polygons or appropriate geometrical objects representing various geographic regions in the Netlogo space [23]. Once these procedures are completed, agents move across and interact with patches and other agents to complete the task. Netlogo also provides modules to implement spatial and attribute queries in GIS wherein the user can query some parts of the vector feature (patches) or the complete dataset. One of its major drawbacks is that it doesn’t have code parallelization support and hence can cause performance issues when running simulations on large amounts of GIS data [24].

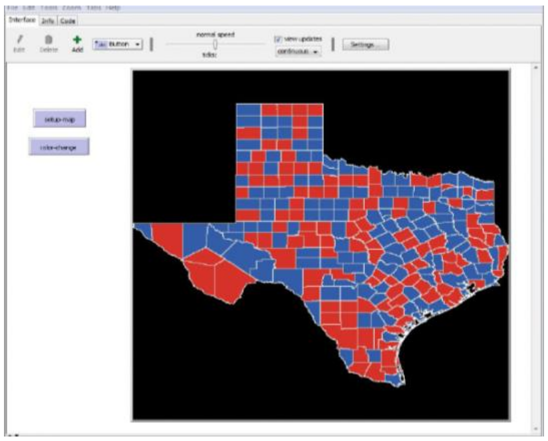


Figure 4: Texas map displayed on Netlogo GIS integrated system [23].

3.1.2 Repast Simphony

Repast Simphony [25] is an open-source library for agent-based simulations. It is built in Java. The concepts of contexts, projections, and queries in Repast Simphony provide a suitable environment for GIS integration. Context is a data structure used to organize agents representing GIS objects, it can be thought of as a bag full of agents based on set semantics. It contains projections that define relationships between agents. An in-built feature, Geography projection, is employed to group agents with the same spatial geometry (points, lines and polygons) into a geographic layer. This geography projection can be used to implement spatial and attribute queries. An example would be to query all agents with a specific spatial geometry [26]. Additionally, Repast Simphony provides support for continuous GIS agent space where the agent locations are represented using real numbers which is well suited for the GIS. The Repast HPC extension of Repast Simphony supports distributed and parallel processing.

3.1.3 Multi Agent Simulation of Neighborhood (MASON)

Multi Agent Simulation of Neighborhood (MASON) is an open-source Java library for agent-based modeling. MASON has a GIS extension called Geo-MASON to support GIS functionalities. GeoMASON is designed to have the utility, model, and visualization layer. The utility layer provides support for importing GIS data into MASON. Model layer contains the attribute information of the GIS vector data (geometry of the underlying features) and their metadata. Visual layer contains modules for displaying the GIS data. MASON supports network, discrete and continuous agent spaces. GeoMASON also provides good documentation to implement spatial queries in GIS data. In GeoMASON computation and visualization components are separated therefore the time required in processing GIS data is less. MASON can also be hosted in a distributed system across a cluster of computing nodes and can be run in parallel using the message passing interface with the distributed MASON extension.

3.1.4 AnyLogic

This is another agent-based modeling framework and provides in-built GIS support. The GIS implementation is using OpenMap which is a JAVA visualization toolkit for geospatial data. It provides support to Tiled and Shapefile maps. It also provides inbuilt modules to define different elements of GIS such as a point, route or a region. Further, it provides support to spatial and attribute queries. It provides inbuilt methods to place agents on a map and move them across the map along the defined routes in search of a specific destination. Modules that calculate the distance between two agents, return the current latitude and longitude position of the agent and also return status of the agent based on whether it is moving are implemented.

There have been research studies in implementing distributed and parallel agent-based models using Repast HPC [25][26] in geospatial data. However not much work has been found in the literature that implements a full-scale GIS software with all the functionalities of data loading, querying and processing, visualization and exporting, using the existing ABM frameworks on distributed parallel setup of multiple nodes. Table 1 summarizes and compares different ABM frameworks that support GIS. From Table 1 we can infer that AnyLogic [29] and distributed MASON can be considered to benchmark the performance of MASS-GIS queries

Table 1: Summary of Agent Based Frameworks integrated with GIS.

Comparison parameters	**NetLogo [9]**	**Repast Simphony [25][26]**	**MASON [27][28]**	**AnyLogic [29]**	**MASS [1]**
Implementation language	Scala, Java	Java	Java	Java	Java

Documentation	Detailed	Detailed	Detailed	Detailed	Detailed
Raster data support	GIS extension	Yes	GeoMASON extension	Yes	Yes
Vector data support	GIS extension	Yes	GeoMASON extension	Yes	Contribution of this project
Spatial query	Yes	Yes	Yes	Yes	Contribution of this project
Parallel and distributed platform support	No	Yes, using Repast HPC	Yes, using distributed MASON	Yes	Contribution of this project
Concurrent user support	No	No	No	No	No

3.2 Parallelization of GIS and Parallel GIS queries

This section presents a detailed study on some efforts to parallelize GIS and GIS queries.

3.2.1 Research Study Using MPI-Based Framework for Processing Spatial Vector Data in GIS On Heterogeneous Distributed Systems

In this research endeavor [6], the GIS data is distributed across multiple computing nodes in the cloud and local environment. Communication between the local and cloud servers take place through the Internet. The system has a master node that is responsible for scheduling tasks and predicting execution time. The master node and the worker nodes process GIS data stored in a distributed file system. The MPI framework is composed of the following components: execution time predictor, schedulers and C++ wrapper library that implements MPI. Each of the spatial vector files is considered as a task. The master node begins by processing the spatial files. Information such as file size and number of vertices is extracted from the spatial data files and imported into the distributed file system. Files (tasks) are arranged according to the number of vertices in the spatial data. The scheduler assigns the files to the MPI processes in the increasing number of spatial vertices (files with smaller number of vertices are assigned first). Each MPI process performs its assigned task in the master and worker nodes and calculates the execution time of the completed task in order to predict the execution time of the uncompleted tasks. The framework also creates an execution time prediction model. Next, the scheduler assigns tasks / files with the next lowest number of vertices and execution time to the MPI processes. The results of the experiments showed that the framework processed the spatial data 12.9 times faster than the sequential execution [6]**.**

3.2.2 Parallel Spatial Query Processing for Big Spatial Data using MapReduce.

This research endeavor implements a unique solution to process spatial queries over big spatial data named VegaGiStore [7]. In this solution the spatial data is first divided into multiple blocks based on a threshold size and geographic space and distributed over a cluster of nodes. In each node, each piece of spatial data is stored sequentially allowing geographically close spatial objects to be placed together. Thus, this organization would reduce the number of I/O operations for spatial queries and support multiple concurrent spatial user queries. Next the solution applies a distributed spatial index for pruning the search space. Lastly, an indexing and MapReduce data processing architecture is used for computation in spatial data. This experiment was carried out using the Hadoop distributed file system and MapReduce framework. The results showed that average spatial query performance increased by 10.3 -13.5 better than single node databases.

3.3. Differentiate our approach from other existing implementations of GIS.

The difference between our approach to GIS and GIS queries with the ones studied in the literature review in sections 3.1 and 3.2 are as follows:

- Though there have been many agent-based models integrated with GIS and state-of-the-art open-source GIS tools, many of them such as Swarm, NetLogo and Mesa are sequential in nature Swarm [8], NetLogo [9] and Mesa [10]. Our approach using MASS library as described in chapter 4 implements a parallel solution using agents.
- The existing parallel integrations of agent-based models with GIS don't utilize the computational geometry algorithms for efficient spatial query processing. However, our approach utilizes the benefits of computational geometry algorithms such as, the closest pair of points, range search and minimum spanning tree.

- The current ABM libraries that provide support for parallel GIS operations do not include any optimizations for reducing the number of agent movements in query processing. The MASS based GIS system organizes the geospatial data based on geographical coordinates. This ensures that data is not randomly distributed across nodes and reduces the number of agent movements. For instance, if the GIS is queried for countries near the United States of America, then most of the data pertaining to the United States and countries near it will reside on one node or in the worst-case scenario on the next node. This will help in reducing agent movement as opposed to data being randomly distributed across all the computing nodes in the cluster where agents need to migrate to every node on the cluster to fetch for results.
- None of the agent-based libraries integrated with GIS support concurrent user queries. The MASS-GIS solution organizes geospatial data across distributed systems such that the underlying design will also support concurrent user queries when it is implemented in the future. For example, two users can query for data pertaining to different regions of the world concurrently as the data for these regions reside on different nodes.
- Hadoop is a batch processing system. Therefore, the current implementation of GIS parallelization using Hadoop and MapReduce is not suitable for GIS queries involving real time data processing.
- MPI based parallel GIS, requires the use of low-level designer code to parallelize it. However, our implementation uses JAVA which is a widely used high level programming language (easier to code).

Chapter 4
Implementation

In this chapter, we discuss the bug fixes employed in migrating the basic version of the MASS-GIS system from Amazon Web Services (AWS) cloud to the University of Washington Bothell (UWB) internal lab cluster in section 4.1. We provide details about our implementation of sequential GIS queries in section 4.2. In section 4.3, we outline a summary of our approach to parallelize GIS queries. Further in sections 4.4, 4.5, 4.6 and 4.7 we focus on the design and implementation strategies used in our approach to agent-based attribute, spatial, and aggregate GIS queries using MASS.

4.1 Migrate the basic version of the MASS-GIS system from AWS to the UWB cluster

The basic version of MASS-GIS implementation by M Sieling [11], was carried out on 8 AWS EC2 instances. It implemented the basic functionalities of distributing the GIS data and rendering map fragments in parallel from 8 EC2 instances. However, this implementation is quite primitive, and the partitioning of the GIS data was too coarse. To achieve time-efficient performance for large scale complex GIS queries, it is important to have fine grained distribution of GIS datasets over a bigger cluster system. Thus, given the monetary resource constraints in continuing our implementation in AWS on a larger cluster system we decided to migrate the basic version of the MASS-GIS system into UWB internal cluster. This migration will enable us to evaluate the performance of the system on 24 servers and achieve fine-grained partitioning of the GIS datasets suitable for agent-based parallel GIS queries. Additionally, it will help us evaluate the MASS-GIS system on a non-cloud platform as well.

Following bug fixes were made while migrating the MASS-GIS system from AWS cloud to internal lab environment at UWB:

- Fixing MASS initialization errors: Compilation errors in initializing the MASS core library were encountered while executing the MASS-GIS system in the UWB lab environment. This error was because the mass library was not initialized using the appropriate constructor.

- Fixing hardcoded file path: File not found exceptions we encountered as the file paths to read and save the map differed from the AWS environment to the internal UWB lab environment. Hardcoded file paths were fixed.
- Missing dependencies in documentation: Documentation didn't mention XQuartz, an open-source graphics framework, as a dependency when executing the code. Errors caused by missing XQuartz software were fixed once it was installed.
- Error in displaying distributed map images: The code to visualize the map, distributed across MASS Places displayed an empty screen. This was because the map fragments retrieved from MASS Places were getting corrupted in subsequent sections of code that combined the map fragments. This bug was fixed by making another copy of the map fragments retrieved from MASS Places and using it to combine and display the map.
- Incomplete map image: Some parts of the map were missing when the MASS-GIS system was executed on multiple nodes. This was because the code to spawn MASS Agents and migrate them to MASS Places was not successful in migrating MASS Agents to all the MASS Places containing map fragments. As a result, when subsequent sections of the code tried to display map fragments, some of them were missing and displayed null pointer exceptions. This problem was remedied by creating new agents and assigning one agent per MASS Place. Agents in each of the MASS Place containing map fragments returned the map fragment to the master node and a complete map was displayed.
- We replaced the raster image backdrop of the world, depicting only borders of continents and water bodies, with a tiled map from OpenStreetMap. The OpenStreetMap is an open-source project for creating geographical databases. It provides tile service which is a web-service for map tiles. These map tiles are raster images that have specific details of a geographic location for instance country and state boundaries with labels, roads in a specific neighborhood, highways, etc. Thus, this would provide more information about the geographic locations to the user as opposed to just having terrain and land boundaries without labels depicted from the previously implemented raster image.

In conclusion, the migration from 8 AWS EC2 instances to the UWB cluster helped in scaling the MASS-GIS system across 24 nodes. This effort also helped in dividing the large GIS datasets into 400 fragments and distributing it across 24 nodes within a small amount of time. Moreover, the user can now utilize the GIS map and retrieve extremely specific details of location. Figure 5 depicts the previous and current MASS-GIS UI after migration (zoomed version with cities, national parks, US states).

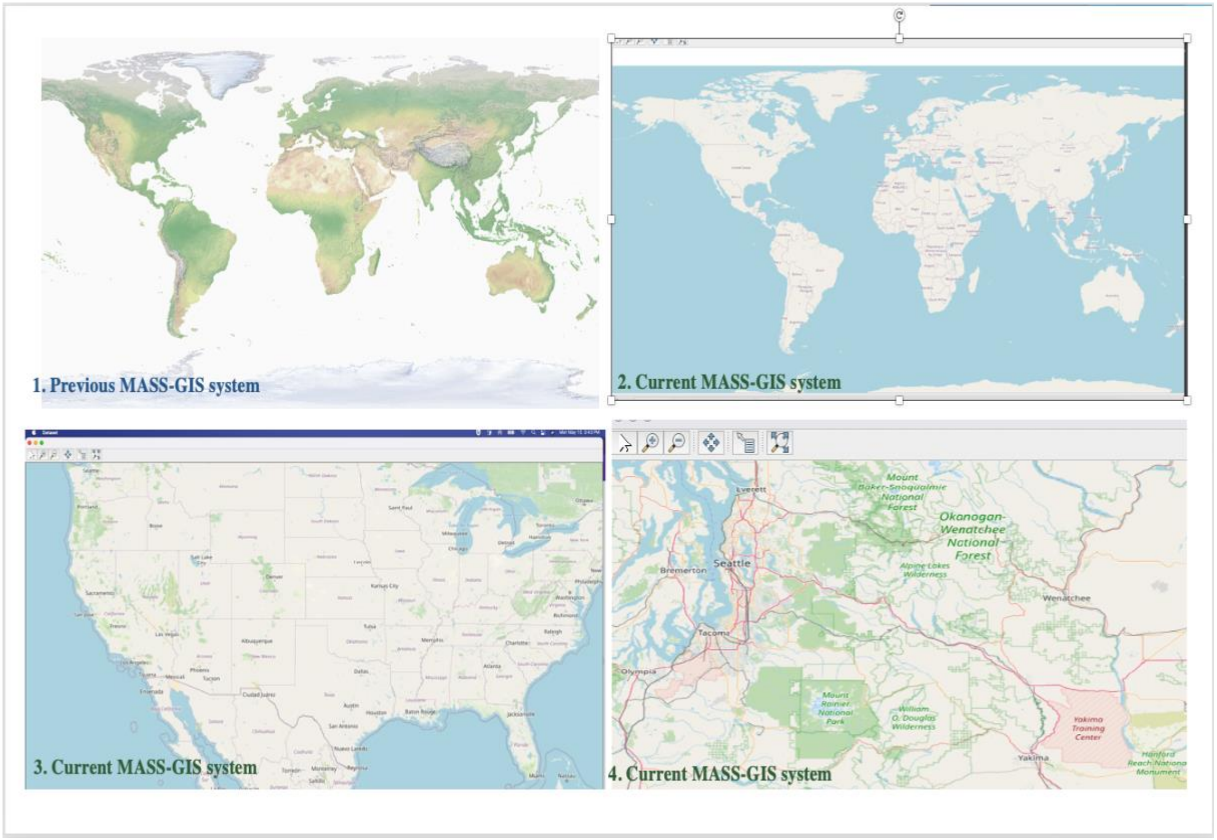

1. Previous MASS-GIS system
2. Current MASS-GIS system
3. Current MASS-GIS system
4. Current MASS-GIS system

Figure 5: Previous (no labels and details) and current MASS-GIS UI (can zoom to view details).

4.2 Sequential GIS queries

The sequential GIS queries are implemented using the contextual query language module from GeoTools. The shapefile is first read using an interface *FileDataStore* [31] provided by the GeoTools library. A *SimpleFeatureSource* object is created to access the geospatial information in the shapefiles as JAVA objects. Further, we create a *SimpleFeatureCollection* [34] object, which is a collection of vector features and provides support for executing spatial and attribute queries. Attribute and spatial CQL queries are defined and applied to the collection of vector features (*SimpleFeatureCollection*). The CQL query applies constraints to the collection of features and returns a subset of selected features in the *SimpleFeatureCollection* format. The resultant subset is added to the map as a layer with a specified geometry and color using the *org.geotools.map.FeatureLayer [37][38]* and *org.geotools.styling.Style* [39] libraries. The feature layer contents are displayed by the MapContent *object from org.geotools.map.Map* [35] [36] library.

4.3 Summary of our parallelization strategies for GIS queries

As illustrated in Figure 6, our approach to parallelizing GIS queries can be broadly divided into two categories: (1) using the contextual query language (CQL) module and MASS Places and (2) using computational geometry algorithms and MASS Agent propagation on MASS Places. In our first approach, we implement attribute and spatial GIS queries. However, in the second approach, we implement only spatial GIS queries using computational geometry algorithms of closest pair of points, range search and minimum spanning tree.

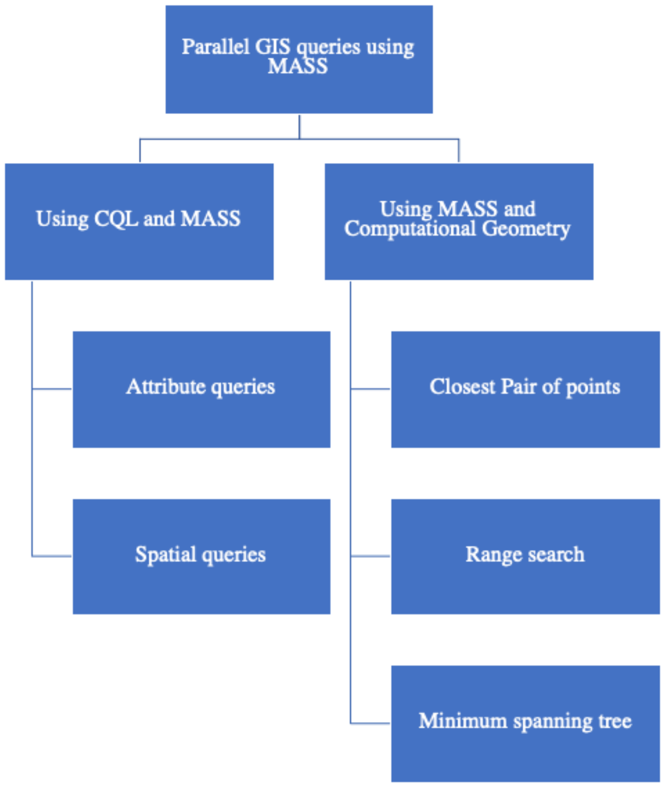


Figure 6: Taxonomy of parallel GIS queries implemented in MASS.

4.4 Parallel Attribute and Spatial GIS Queries using CQL Module and MASS Places

The design for attribute and spatial GIS queries in 2D space using MASS and CQL utilizes MASS *Places*, and the to*Filter()* method from GeoTools library that defines a constraint to be verified against an instance of a GIS feature vector data. The GeoTool's *toFilter()* method can be used to place spatial and attribute constraints on the GIS vector features.

The design comprises 5 stages which are i. import GIS vector data, ii. vector data division and distribution, iii. attribute / spatial query on MASS Places using CQL, iv. query results serialization, v. query results deserialization, vi. results visualization on GIS map as described in Figure 7. The sequence diagram in Figure 8 describes the implementation of design phases in detail. The boxes in the sequence diagram in Figure 8 represent the different classes implementing the code and arrows indicate the control flow across these classes.

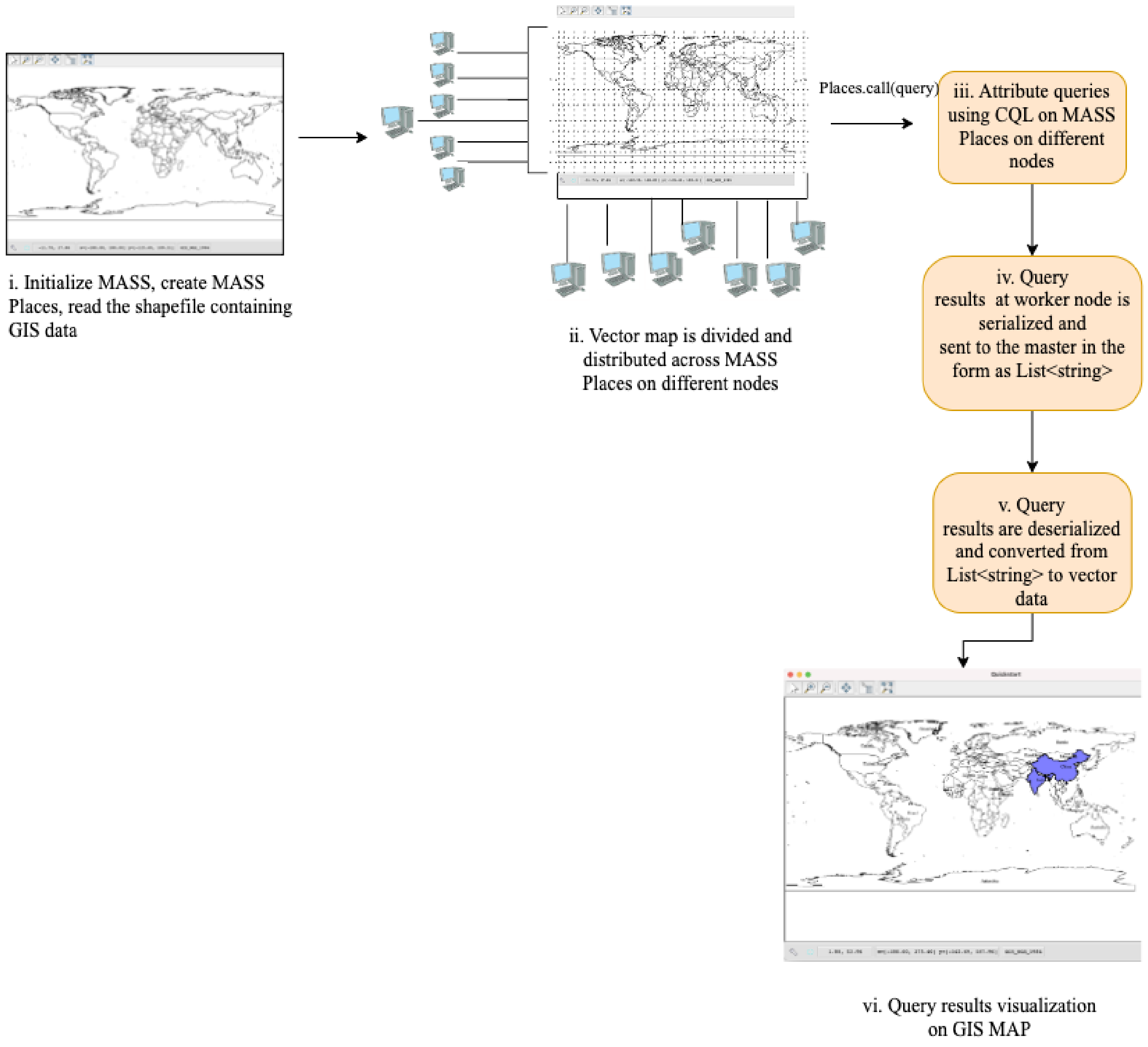

i. Initialize MASS, create MASS Places, read the shapefile containing GIS data
ii. Vector map is divided and distributed across MASS Places on different nodes
Places.call(query)
iii. Attribute queries using CQL on MASS Places on different nodes
iv. Query results at worker node is serialized and sent to the master in the form as List<string>
v. Query results are deserialized and converted from List<string> to vector data
vi. Query results visualization on GIS MAP

Figure 7: Design for parallel attribute and spatial queries using Contextual Query Language and MASS.

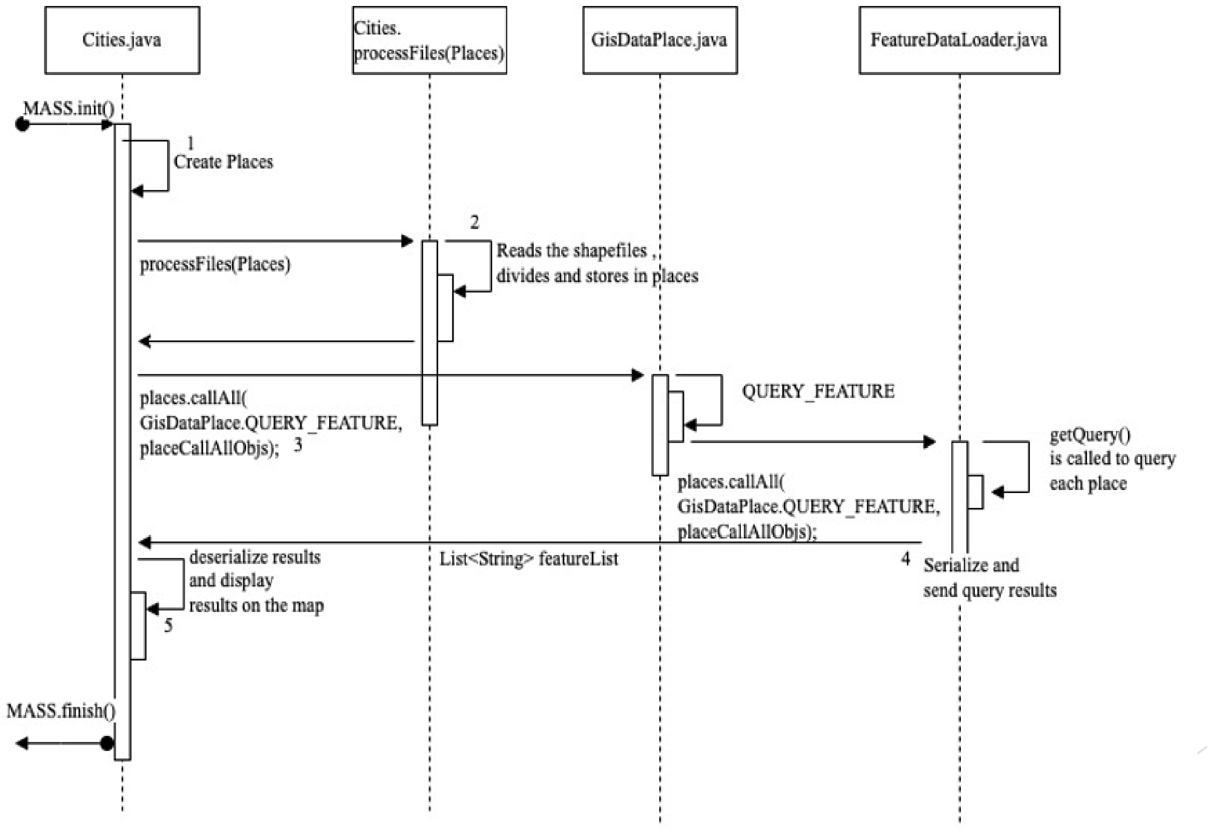

Cities.java
Cities. processFiles(Places)
GisDataPlace.java
FeatureDataLoader.java
MASS.init()
1
Create Places
2
Reads the shapefiles , divides and stores in places
processFiles(Places)
QUERY_FEATURE
places.callAll(GisDataPlace.QUERY_FEATURE, placeCallAllObjs); 3
getQuery() is called to query each place
places.callAll(GisDataPlace.QUERY_FEATURE, placeCallAllObjs);
List<String> featureList
4
Serialize and send query results
deserialize results and display results on the map
5
MASS.finish()

Figure 8: Sequence diagram for parallel attribute and spatial queries using contextual query language (CQL) and MASS.

The design phases along with implementation details in the sequence diagram from Figure 7 and Figure 8 respectively are detailed through steps i to vi.

i. Import GIS vector data:

This step represents phase i from the design diagram in Figure 7 and phase 1 from the sequence diagram in Figure 8. In this step, the main program initializes the MASS library and creates *MASS Places*. The configuration details provided by the user, containing the number of horizontal rows and columns to divide the map and location of input shape files to be processed are stored in the MASS-GIS system.

ii. Vector data division and distribution:

Phase ii from the design diagram in Figure 7 and phase 2 from the sequence diagram in Figure 8 is carried out in this step. In this step the GIS data in shapefiles is divided and distributed across *MASS Places* on different computing nodes.

iii. Attribute / spatial query on places using CQL:

Once the vector data is distributed across various nodes, CQL queries are executed on individual GIS data fragments maintained across *MASS Places* . Queries are defined as a *String* in the main function and passed as arguments to the *places.callAll()* function to execute in parallel across MASS Places. Figure 9 shows an example syntax of the parallel spatial and attribute CQL queries that were implemented in the MASS-GIS system. The phase iii from the design diagram in Figure 7 and phase 3 from the sequence diagram in Figure 8 is implemented in this phase.

```
// Displays Seattle on the map
String query ="CITY_NAME LIKE 'Seattle%'";
// Displays all the cities within 500 kilometers from the point (Bombay) . Note that distance
// conversion is not accurate in CQL. A potential bug in GeoTools
String query = ""DWITHIN(the_geom, POINT(72.82599639892578 19.07699966430664), 5,
kilometers)";
```

Figure 9: CQL queries executed on MASS-GIS system.

iv. Query results serialization :

The results obtained from the CQL query at each *MASS Place* on the remote computing nodes, take a similar format as if it was an output from Java Database Connectivity (JDBC) queries. Since they are not serializable, they are converted into a *List<String>* and thereafter sent back to the master node as shown in phase 4 and iv of the design and sequence diagram in Figure 7 and Figure 8 respectively.

v. Query results deserialization :

The query results obtained as a *List<String>* from different computing nodes in the master node needs to be converted into the vector data format as geometrical shapes to be displayed on the GIS map. This is achieved by using the GeoTools library in this step. The phase v and 5 in the design and sequence diagram from Figure 7 and Figure 8 represents this step.

vi. Query result visualization on GIS map:

The deserialized GIS vector data is added as a feature layer and visualized on the map using the libraries provided by GeoTools. A tiled map from OpenStreetMap (an open-source tool that exposes APIs for raster map images) is used as a backdrop to display labels / names of the location. The phase vi and 6 in the design and sequence diagram from Figure 7 and Figure 8 represents this step.

The results of attribute, spatial and aggregate (combining attribute and spatial) GIS queries implemented using CQL and MASS library are presented in Figure 10 through Figure 16. Particularly, Figure 10 and Figure 11 depict the findings of an attribute GIS query to search the city Seattle. This GIS query uses the “like” operator to query for the city with the name Seattle. Further, Figure 12 presents the results of an attribute GIS query aimed at finding all cities in the United States. Here, we observe that only the cities in the United States are marked on the map.

The results of spatial GIS queries are presented in Figures 13 through Figure 16. These spatial queries utilize the distance within and bounding box spatial operators. Figure 13 shows the outcomes of the spatial query to find all the mineral resources within 500 miles from the city Mumbai. Figure 14 depicts the results of the spatial GIS query to find cities within 100 kilometers from Seattle. The queries in Figure 13 and Figure 14 use the distance within operator. An illustrative example of the bounding box operator is shown in Figure 15 that depicts the spatial query to find all cities within the bounding box. Lastly, an example of aggregate query is presented in Figure 16 that applies the bounding box spatial operator and searches for the cities with attribute “CNTRY_NAME” as United States.

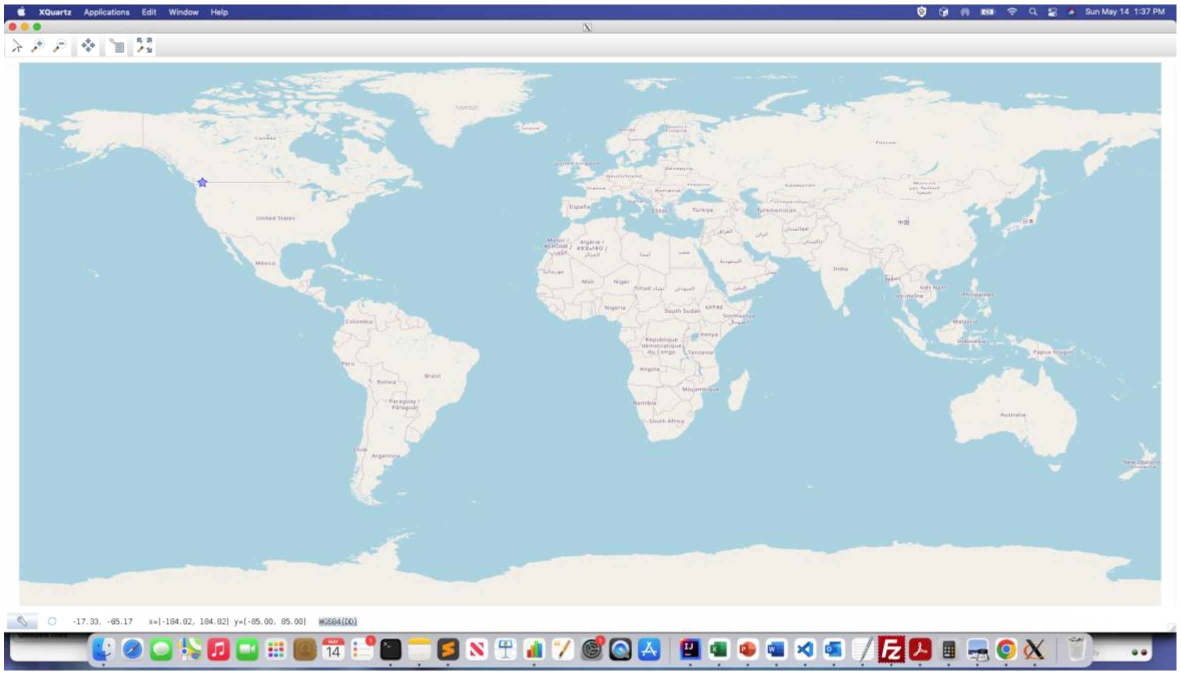


Figure 10: Attribute based GIS query to find the city Seattle.

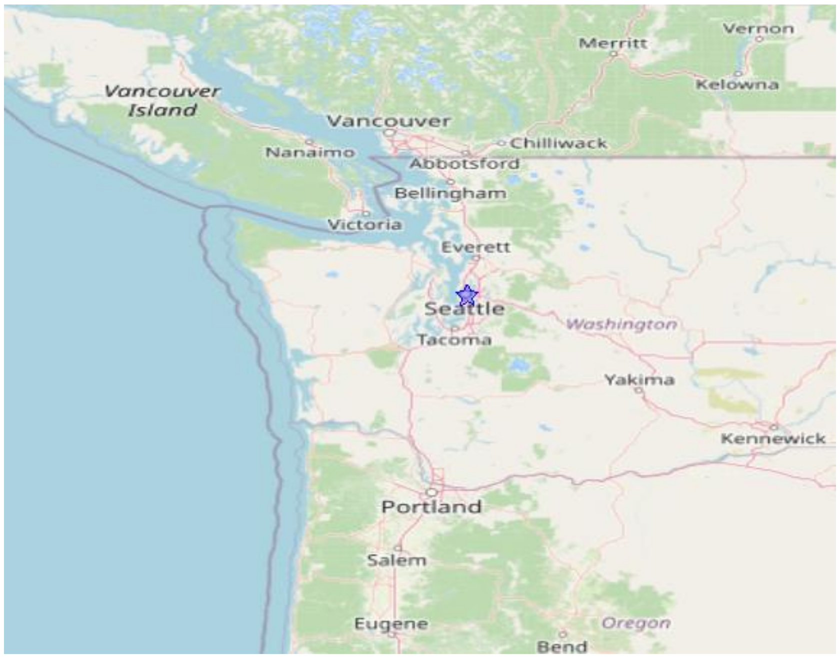


Figure 11: Attribute based GIS query to find the city Seattle (zoomed view).

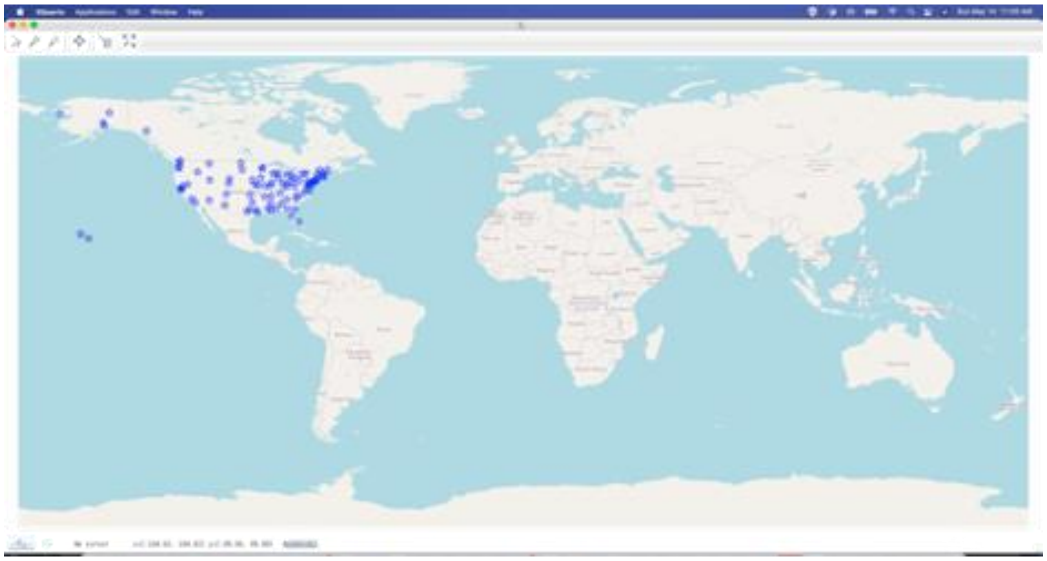


Figure 12: Attribute based GIS query to find all cities in the United States on the map.

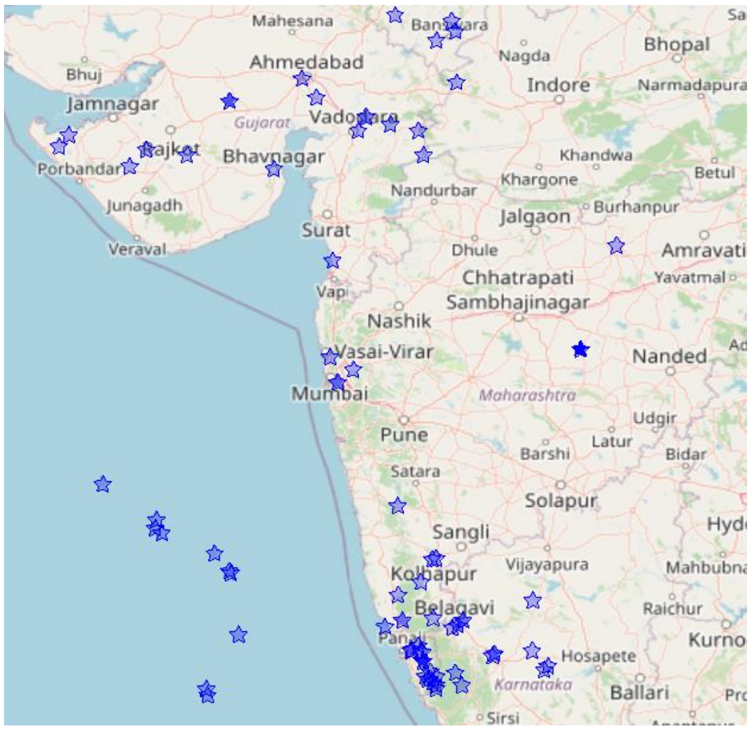


Figure 13: Spatial GIS query to find all minerals resources within 500 miles from the city Mumbai.

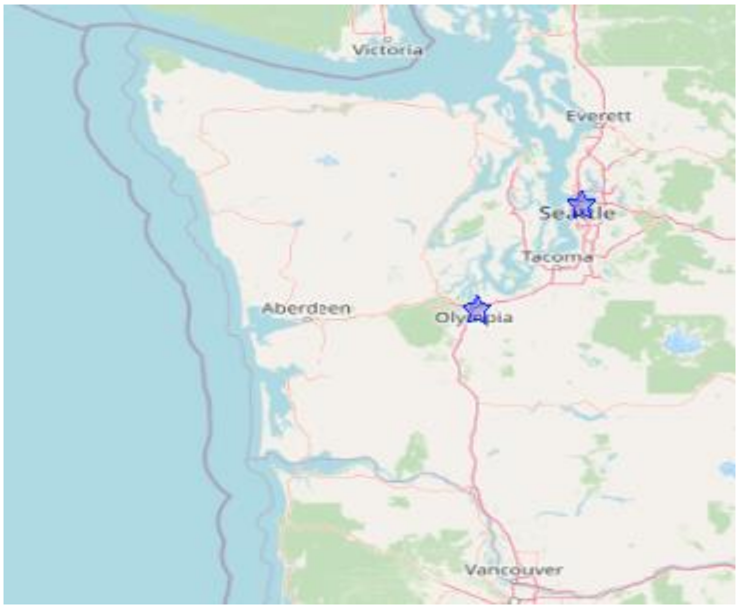
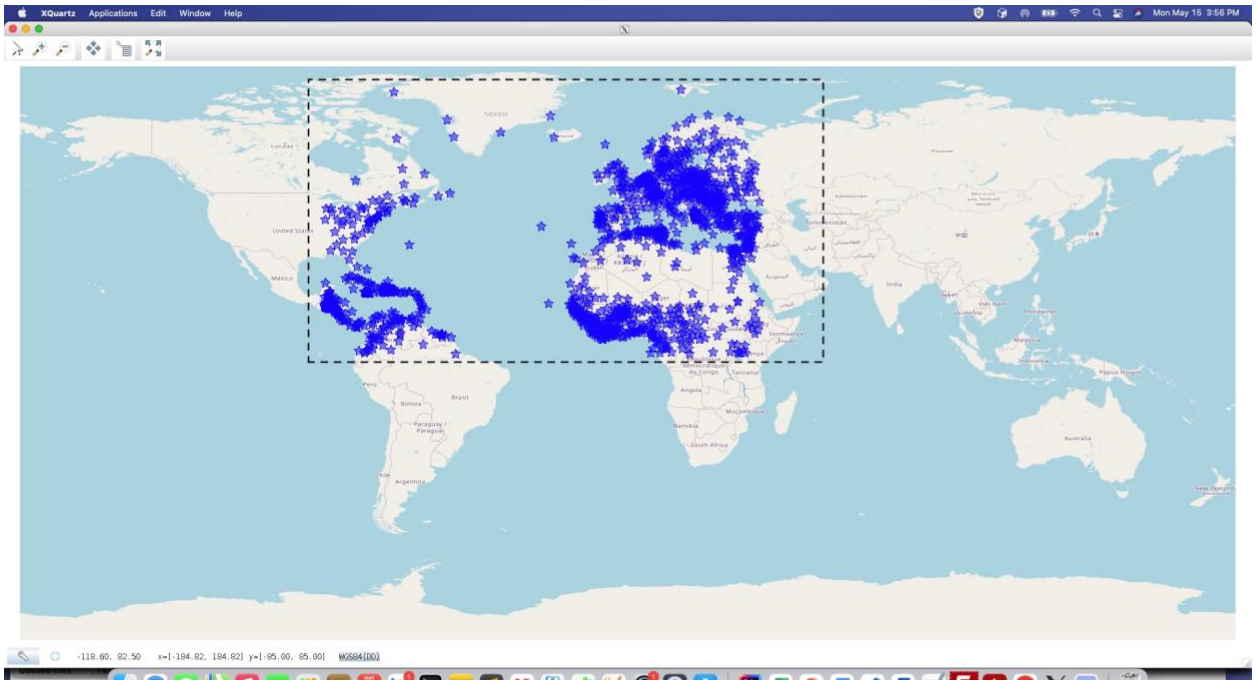


Figure 14: Spatial GIS query to find city within 100 kilometers from Seattle.



```
-118.60, 82.50    x=[-184.82, 184.82] y=[-85.00, 85.00]    WGS84(DD)
```

Figure 15: Spatial GIS query to find all the countries within the bounding box.

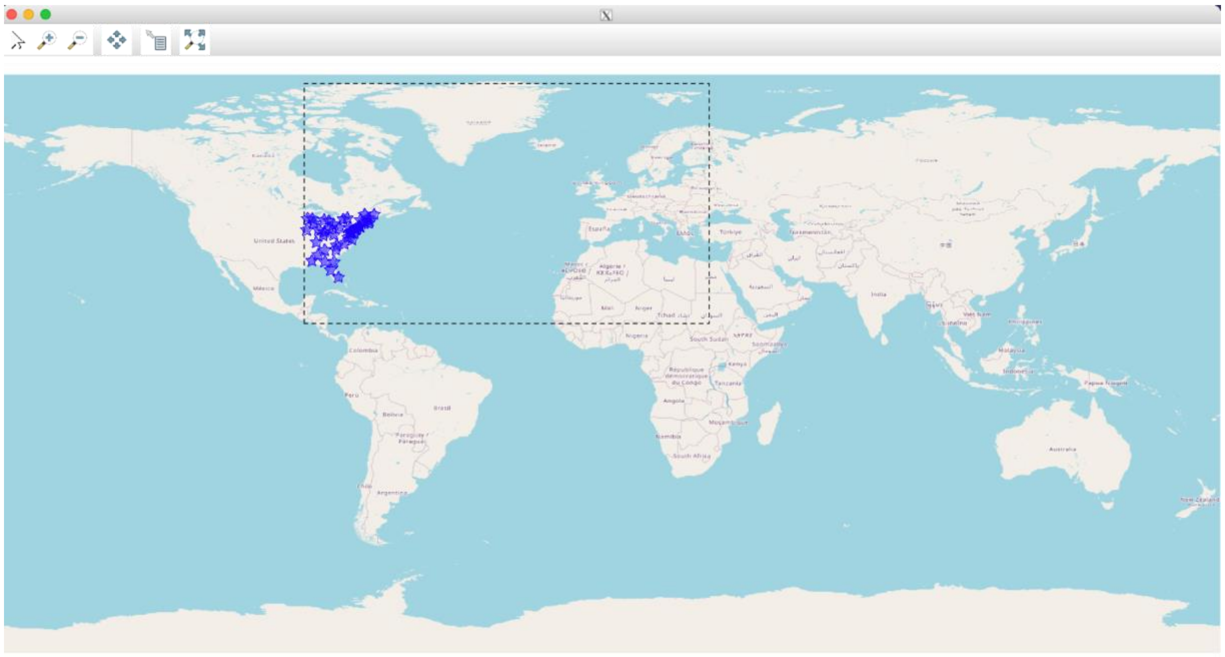


Figure 16: Aggregate GIS query combining attribute and spatial query to find all the cities within the bounding box and has the “CNTRY_NAME” attribute set to “United States.”

4.5 Closest Pair of Points: Parallel Spatial GIS Query using MASS Agent Propagation

In this GIS query we extend and modify the MASS based closest pair of points algorithms [15] to find the closest points (cities) within a certain distance of a given point (city). This algorithm demonstrates the practical applications of the MASS based closest pair points algorithm.

The design consists of 5 phases: i. import vector data in the MASS-GIS environment, ii. extract vector data within 1 latitude and longitude coordinate position from the given point, iii. compute all the cities with a given distance from a point /city using agent propagation, and iv. visualize the results on the map as shown in Figure 17. The sequence diagram in Figure 18 describes the control flow across the class implementations.

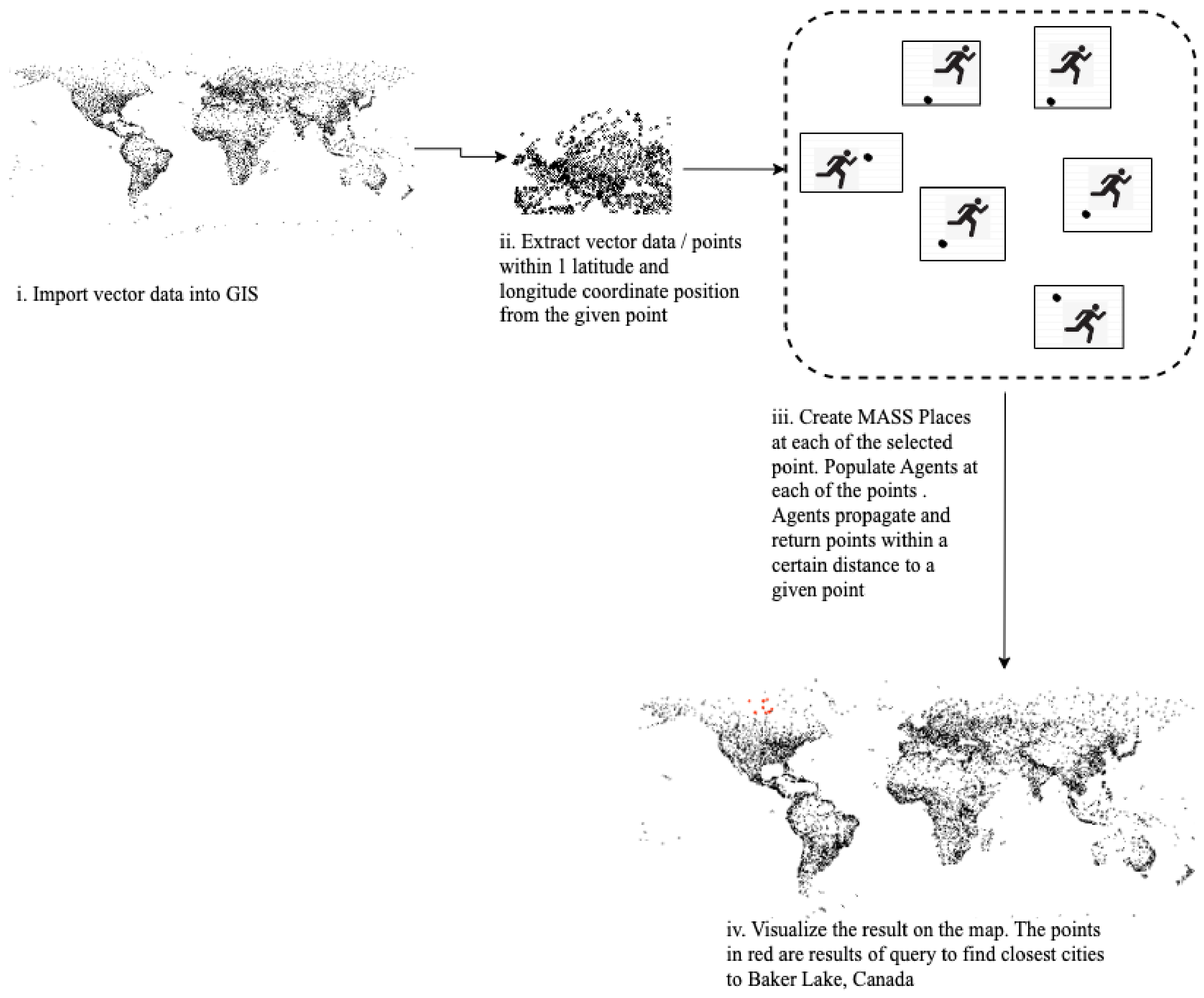

i. Import vector data into GIS
ii. Extract vector data / points within 1 latitude and longitude coordinate position from the given point
iii. Create MASS Places at each of the selected point. Populate Agents at each of the points . Agents propagate and return points within a certain distance to a given point
iv. Visualize the result on the map. The points in red are results of query to find closest cities to Baker Lake, Canada

Figure 17: Design for spatial GIS query to find the cities within a certain distance from a given city using agent propagation.

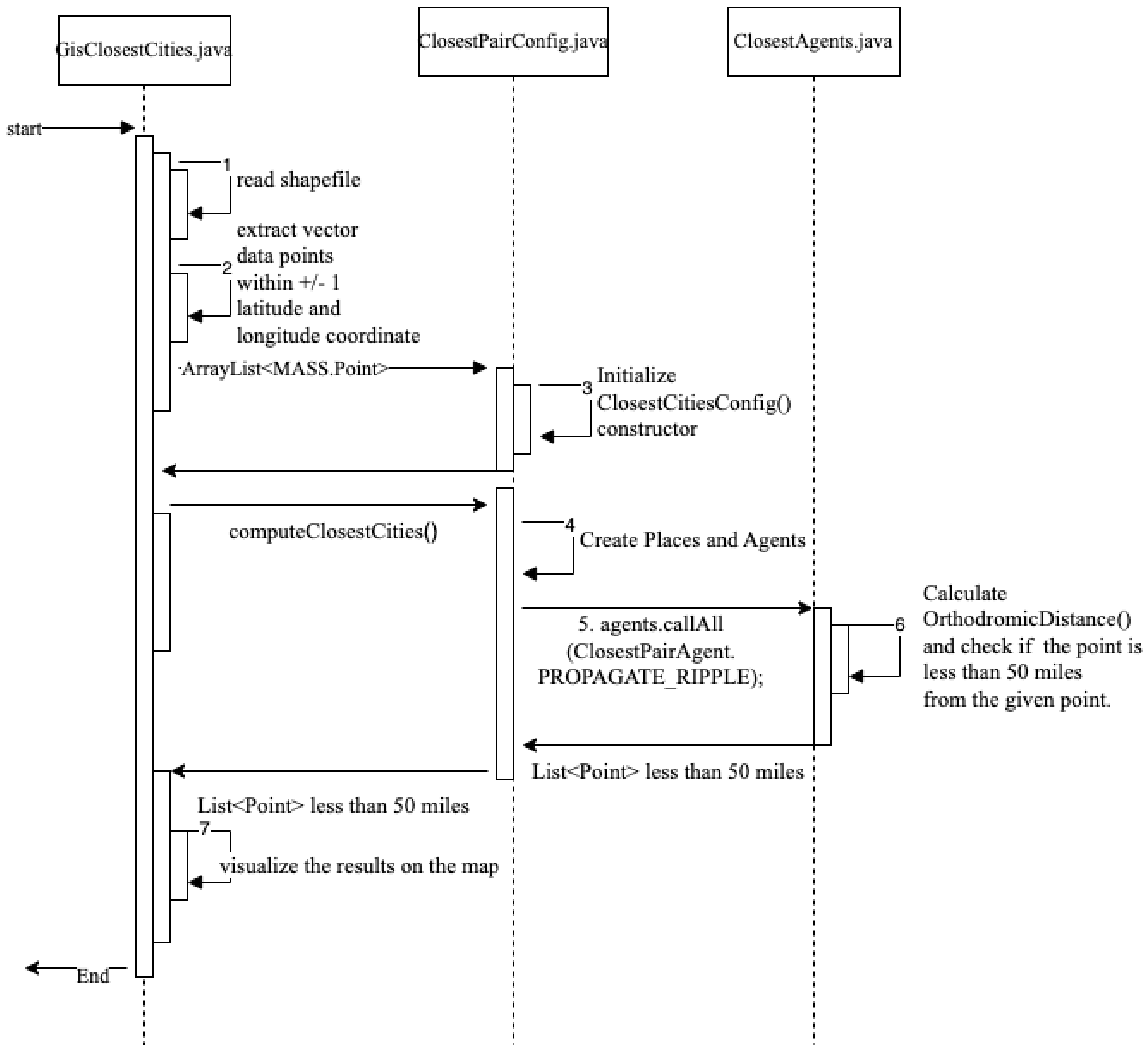

GisClosestCities.java
ClosestPairConfig.java
ClosestAgents.java
start
1 read shapefile
2 extract vector data points within +/- 1 latitude and longitude coordinate
ArrayList<MASS.Point>
3 Initialize ClosestCitiesConfig() constructor
computeClosestCities()
4 Create Places and Agents
5. agents.callAll (ClosestPairAgent. PROPAGATE_RIPPLE);
6 Calculate OrthodromicDistance() and check if the point is less than 50 miles from the given point.
List<Point> less than 50 miles
List<Point> less than 50 miles
7 visualize the results on the map
End

Figure 18: Sequence diagram for parallel agent-based query to find all the points /cities within a given distance of a point / city.

The design phases along with implementation details in the sequence diagram from Figure 17 and Figure 18 respectively are detailed through steps i to iv.

i. Import vector data in GIS environment:

The GIS vector data (cities information) present in the shapefile is read into the MASS-GIS system using the GeoTools library as represented in phase i and 1 in the design and sequence diagram in Figure 17 and Figure 18 respectively.

ii. Extract vector data within +/- 1.0 latitude and longitude coordinate position from the given point:

After reading the vector information from the shapefile, we iterate over each of the vector data and extract the X and Y coordinate information (latitude and longitude) of all the vector data points (cities). Next, we consider only those points (cities) whose coordinates are within +1 and -1 latitude and longitude coordinate positions of the given point. This is because cities that lie within 50 miles of a given city have latitude and longitude coordinate positions in the range of +/-1 coordinates. To find points that are at a larger than 50 miles from a given point we can increase the range of latitude and longitude coordinates. For example, if points within 100 miles from a given location need to be chosen then we can consider points within +/- 2 latitude and longitude coordinate position. The X and Y coordinates (latitude and longitude) of a point are converted to the type *edu.uw.bothell.css.dsl.MASS.Point* [1] and stored in a *List<MASS.Point>*. This step is depicted in phase ii and 2 in the design and sequence diagram in Figure 17 and Figure 18 respectively.

iii. Compute all the cities with a given distance (50 miles) from a point /city using agent propagation:

In this phase, *MASS SpacePlaces* are created at each of the selected points (cities) in the *List<MASS.Point>* (points / cities that lie within +/- 1 latitude and longitude coordinate positions from the given point / city). *MASS SpacePlaces* extend MASS.Places and have additional methods that support complex agent propagation methods [15]. Each *MASS SpacePlace* represents a city (point). *MASS Agents* are created at each *MASS SpacePlace*. These *MASS Agents* propagate on *MASS SpacePlaces* representing cities and calculate the distance between the original / given city and the city to which they propagated. If the distance is less than 50 miles, the point is added to the result which is a list of MASS points and returned to the main function. The orthodromic distance [42], shortest distance between two points on a sphere, is used for distance calculation between two points as we are using latitude and longitude coordinates. This step is depicted in phase iii and 3, 4, 5, 6 in the design and sequence diagram in Figure 17 and Figure 18 respectively.

iv. Visualize the results on the map :

The result, a list of *MASS points* that are within 50 miles of a specified point is converted to vector data format and visualized on the map using the GeoTools library. This step is depicted in phase iv and 7 in the design and sequence diagram in Figure 17 and Figure 18 respectively.

Figure 19 illustrates the results of the spatial GIS query to find all the cities within 50 miles from Drammen, Norway. The cities of Oslo, Moss, Skien, Tønsberg and Lillestrøm are highlighted as results on the map. On computing the air travel distance from Drammen to these cities we observe that all the cities are within 50 miles from Drammmen, Norway and the GIS query provides 100% accuracy. The results of the computation are summarized in Table 2.

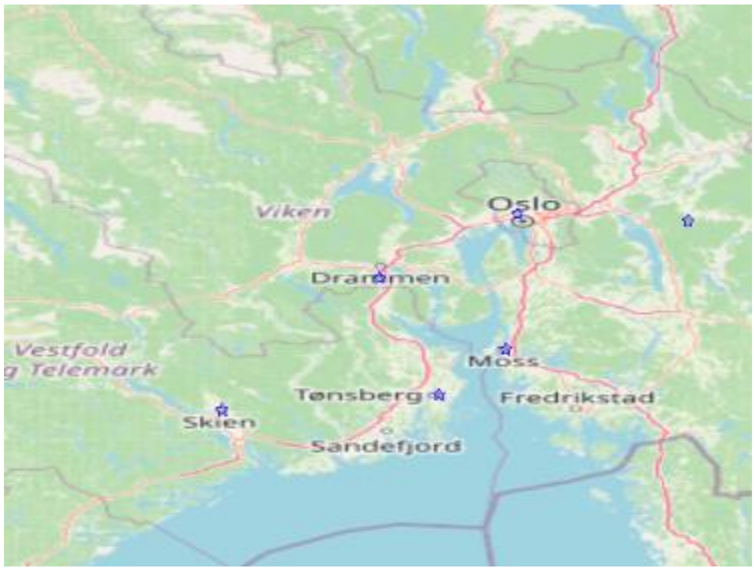


Figure 19: Results of the spatial GIS query to find all the cities within 50 miles from Drammen.

Table 2**:** Air travel distance from Drammen in miles to other cities displayed on the map, obtained from google search.

Sl.no	**City**	**Air travel distance from Drammen in miles**
1	Oslo	22
2	Moss	27
3	Skien	42
4	Tonsberg	34
5	Lillestrøm	32.8

4.6 Range Search: Parallel spatial GIS query using MASS Agent propagation to find all points (cities) within a given range

This spatial GIS query is a practical application of the range search computational geometry algorithm. We extend and modify the MASS based range search algorithm [15]. The range search algorithm returns a subset of points from a given set of points that lies within the specified range. In this GIS query, we query for cities within a country. This is done by supplying the geographical bounds of a country and using range search to determine all the cities represented as points that lie within this geographic bound. The geographic bounds of a country are determined by considering the country as a polygon and using the four coordinate points of the polygon to determine the minimum of all the X and Y coordinates and maximum of all the X, and Y coordinates. For instance, the geographic bounds of Australia are minX : 113.7751361 , maxX: 153.62521, minY = -42.9911371 and maxY = -12.5328931. We find all the cities in Australia within this range.

The design phases as described in Figure 20 are: i. import vector data in the GIS environment, ii. create a text file with x and y coordinates of all the cities in the world, iii. range search using agent propagation and KDTree, iv. display the results of range search on the map.

The design phases from Figure 20 are detailed through steps i to iv.

i. Import vector data in GIS environment:

As shown in phase 1 from Figure 20, the vector data / shapefile with the cities information is read and stored in the GIS environment using the GeoTools library.

ii. Create a text file with x and y coordinates of all the cities in the world: The SimpleFeature The vector data representing cities is converted into an object of type *org.locationtech.jts.geom.Point* [41]. The X and Y coordinates of the points (cities) are extracted and written into a file "points.txt". This step is represented by phase 2 in the design diagram in Figure 20.

iii. Range search using agent propagation and KDTree: The range search algorithm using KDTree and agents, implemented by V.Mohan [10] is run on these points / city coordinates with the minX, maxX, minY and maxY range. The algorithm outputs a list of points of type MASS.Point2D that lie within the specified range. For instance, the range minX : 113.7751361 , maxX: 153.62521, minY = -42.9911371 and maxY = -12.5328931 representing the geographical bounds of Australia is specified to find all the cities (points) within Australia. The only changes that were made to the agent-based range search algorithm was to change all the integer input points and range to type double. This step is represented by phase 3 in the design diagram in Figure 20.

iv. Display the results of range search on the map:

The result of the range search algorithm is a list of points of type ArrayList<Point2D> which is converted to vector data type using GeoTools library and displayed on the map. This step is represented by phase 4 in the design diagram in Figure 20.

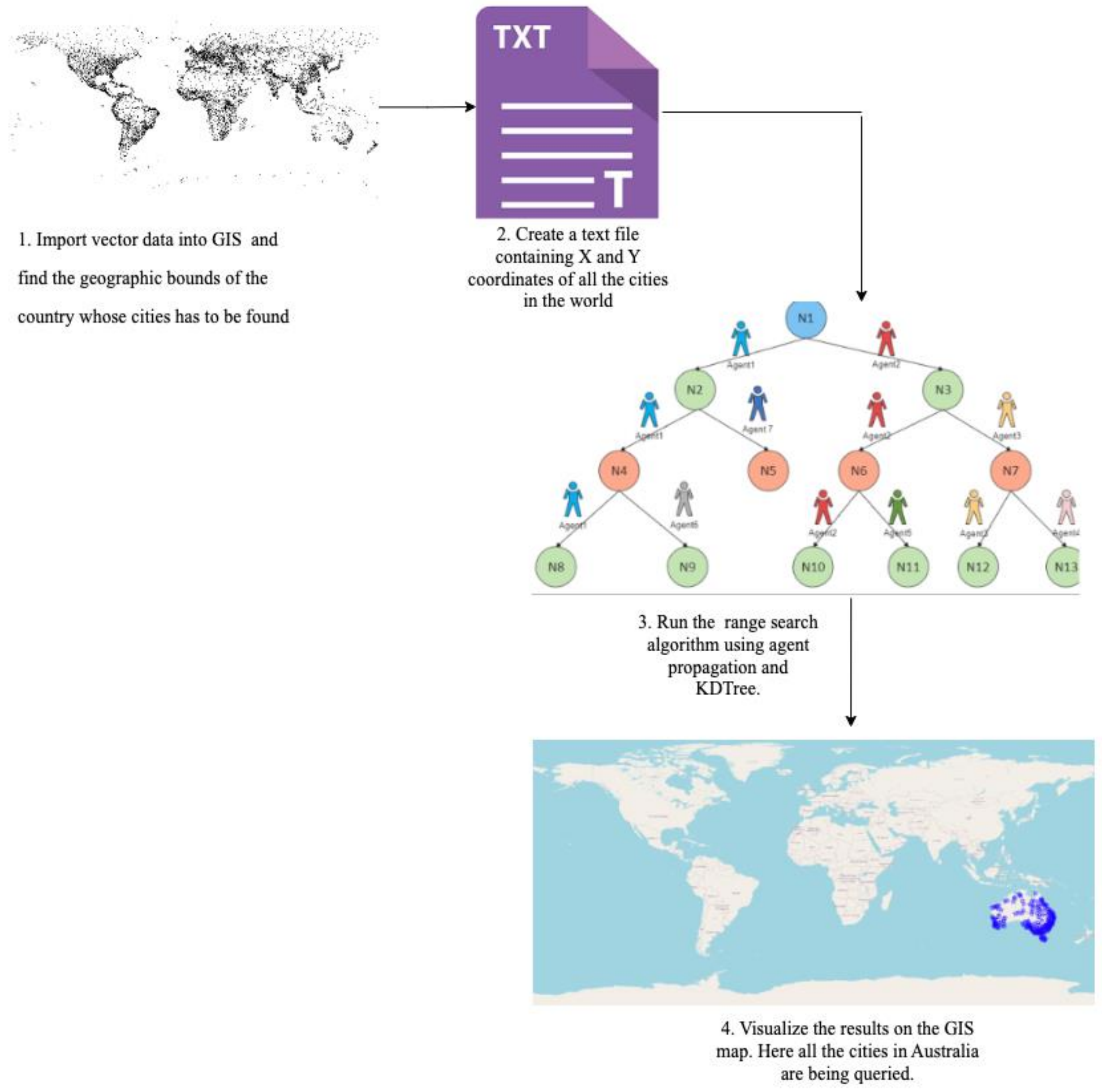

TXT
1. Import vector data into GIS and find the geographic bounds of the country whose cities has to be found
2. Create a text file containing X and Y coordinates of all the cities in the world
N1
Agent1
Agent2
N2
N3
Agent 7
Agent3
N4
N5
N6
N7
Agent6
Agent5
Agent4
N8
N9
N10
N11
N12
N13
3. Run the range search algorithm using agent propagation and KDTree.
4. Visualize the results on the GIS map. Here all the cities in Australia are being queried.

Figure 20: Design for the GIS query to find all the cities in Australia using Range Search algorithm that uses MASS Agents and KDTree.

The results of the range search GIS query to find all the cities given the geographical bounds of Australia are shown in Figure 21 and Figure 22. From the results we observe that the range search query provides 100% accuracy as no cities outside the boundaries of Australia are displayed on the map.

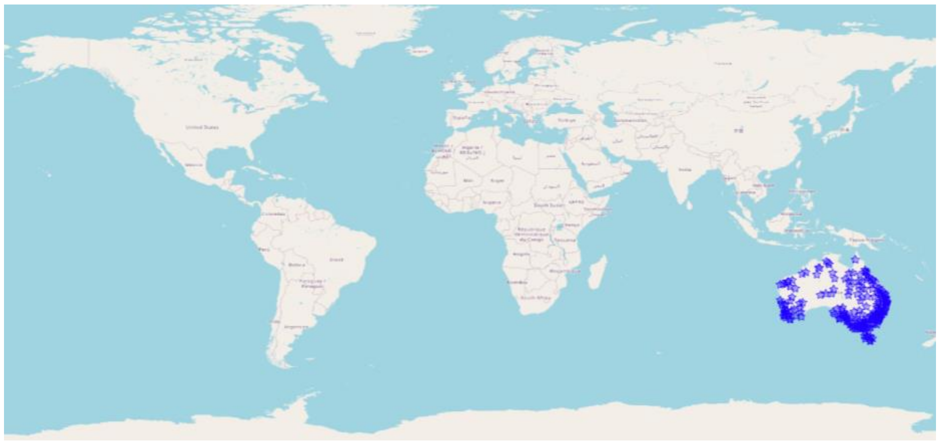


Figure 21: Results for range search using KDTree to find all the cities located in a country (Australia) using agent propagation (world map view).

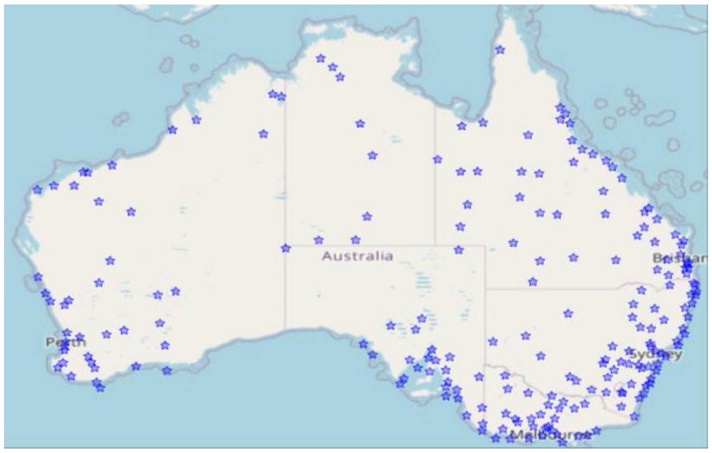


Figure 22: Results for range search using KDTree to find all the cities located in a country (Australia) using agent propagation (zoomed view).

4.7 Minimum Spanning Tree: Parallel spatial GIS query using MASS Agent propagation to Find the Shortest Path from a Source to Destination

In this GIS query we find the shortest path between the source and destination by modifying and extending the minimum spanning tree algorithm using MASS Agents [16]. This GIS query can be further modified to apply in domains like transportation and delivery where it can be used to find the minimum cost of laying roads, travel itinerary planning to visit all the tourist destinations in a city by traveling a minimum distance. The implementation of shortest path GIS query is designed to work on connected and disconnected graphs. However, it requires an existence of a path from the source to the destination. The different design phases as shown in the Figure 23 are import vector data, graph generation using GeoTools, create input file (DSL file), compute shortest path using minimum spanning tree algorithm, process the results of minimum spanning tree algorithm, and visualize the results on the map. The sequence diagram in Figure 24 describes the control flow across the class implementations.

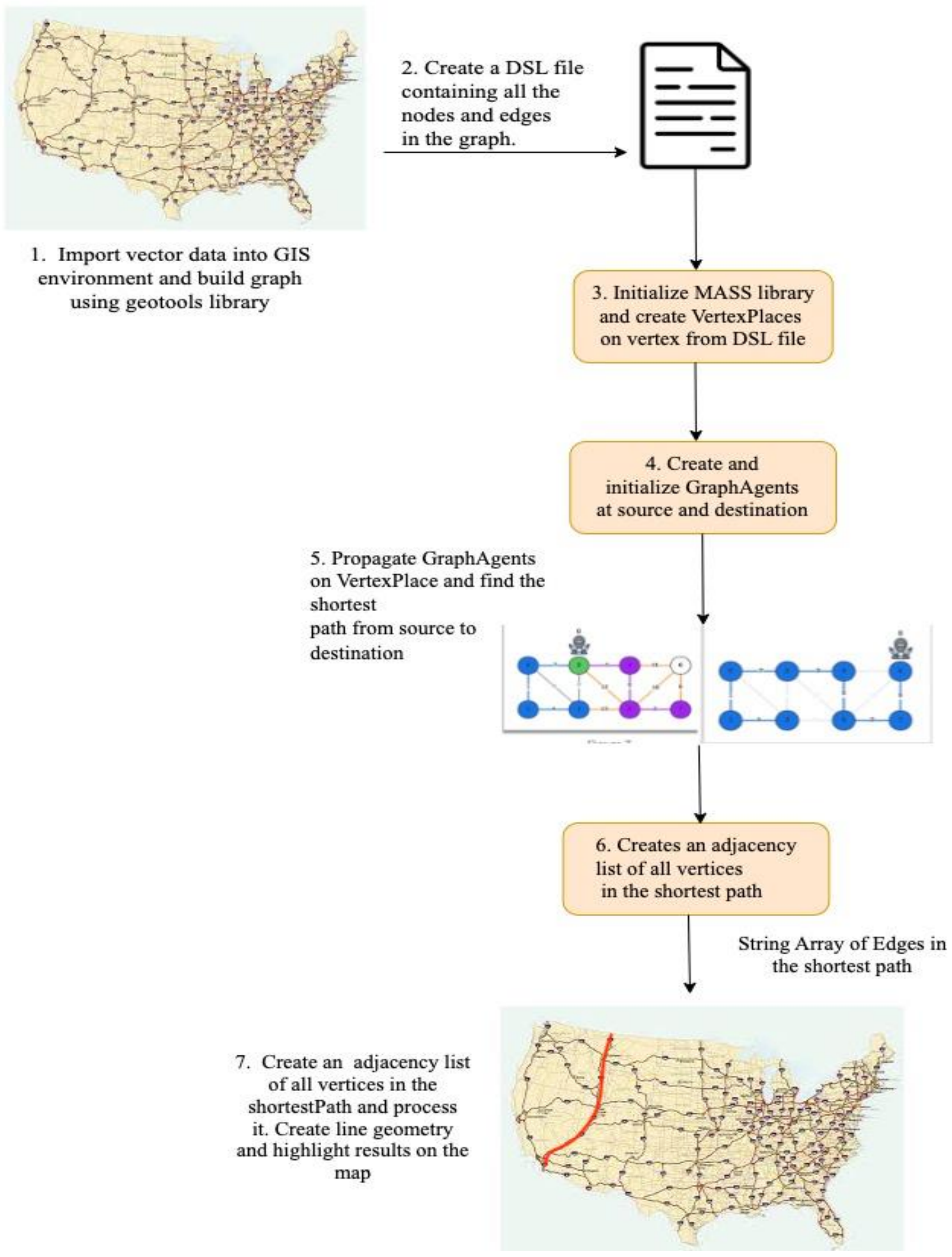

1. Import vector data into GIS environment and build graph using geotools library
2. Create a DSL file containing all the nodes and edges in the graph.
3. Initialize MASS library and create VertexPlaces on vertex from DSL file
4. Create and initialize GraphAgents at source and destination
5. Propagate GraphAgents on VertexPlace and find the shortest path from source to destination
6. Creates an adjacency list of all vertices in the shortest path
String Array of Edges in the shortest path
7. Create an adjacency list of all vertices in the shortestPath and process it. Create line geometry and highlight results on the map

Figure 23: Design for GIS query to find the shortest path from a source to destination by propagating agents over a graph using the minimum spanning tree algorithm.

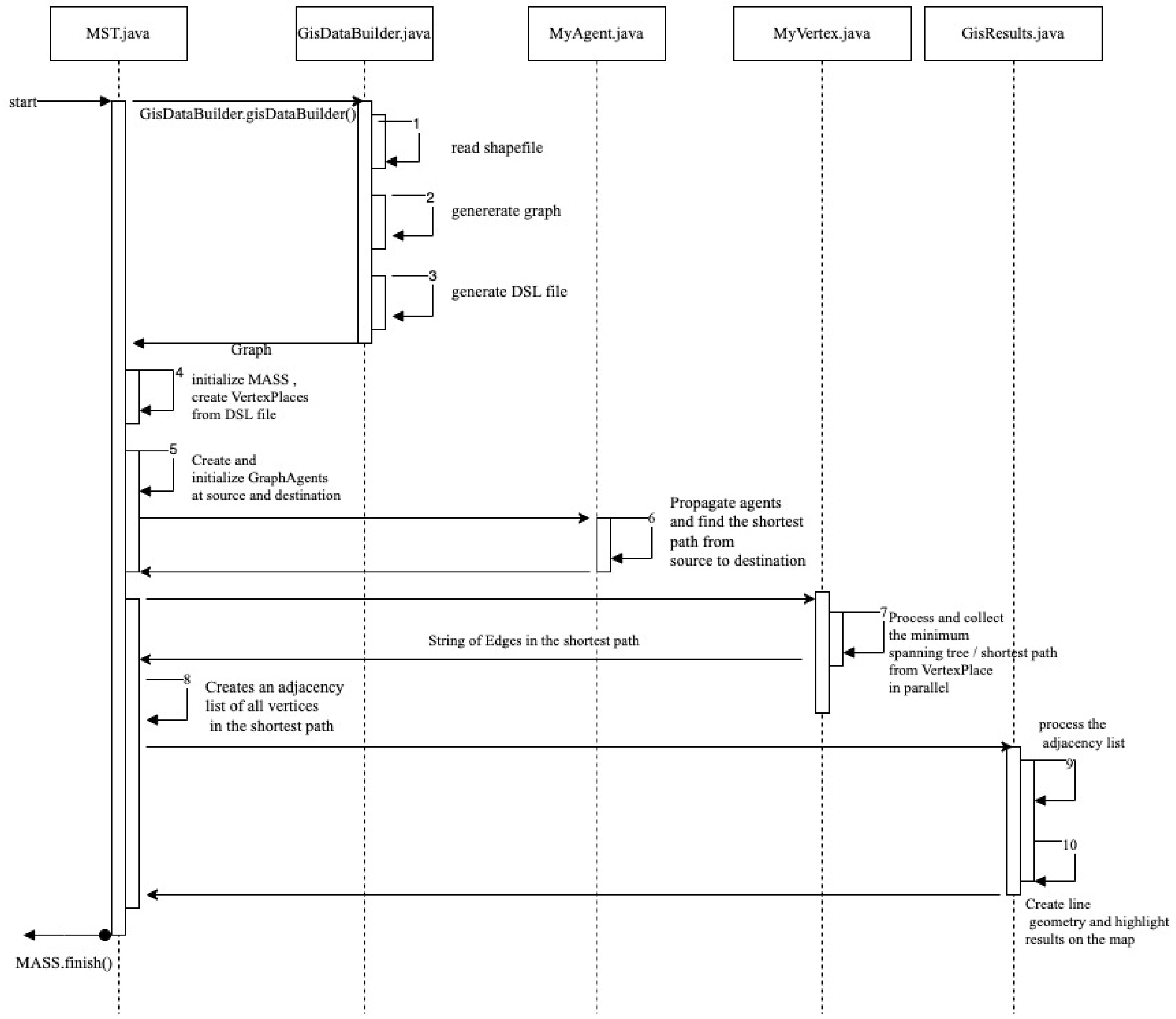

MST.java
GisDataBuilder.java
MyAgent.java
MyVertex.java
GisResults.java
start
GisDataBuilder.gisDataBuilder()
1
read shapefile
2
genererate graph
3
generate DSL file
Graph
4
initialize MASS , create VertexPlaces from DSL file
5
Create and initialize GraphAgents at source and destination
6
Propagate agents and find the shortest path from source to destination
7
Process and collect the minimum spanning tree / shortest path from VertexPlace in parallel
String of Edges in the shortest path
8
Creates an adjacency list of all vertices in the shortest path
process the adjacency list
9
10
Create line geometry and highlight results on the map
MASS.finish()

Figure 24: Sequence diagram for parallel spatial query to find the shortest path from source to destination using MASS based minimum spanning tree algorithm.

The design phases along with implementation details in the sequence diagram from Figure 23 and Figure 24 respectively are detailed through steps i to vi.

i. Import vector data:

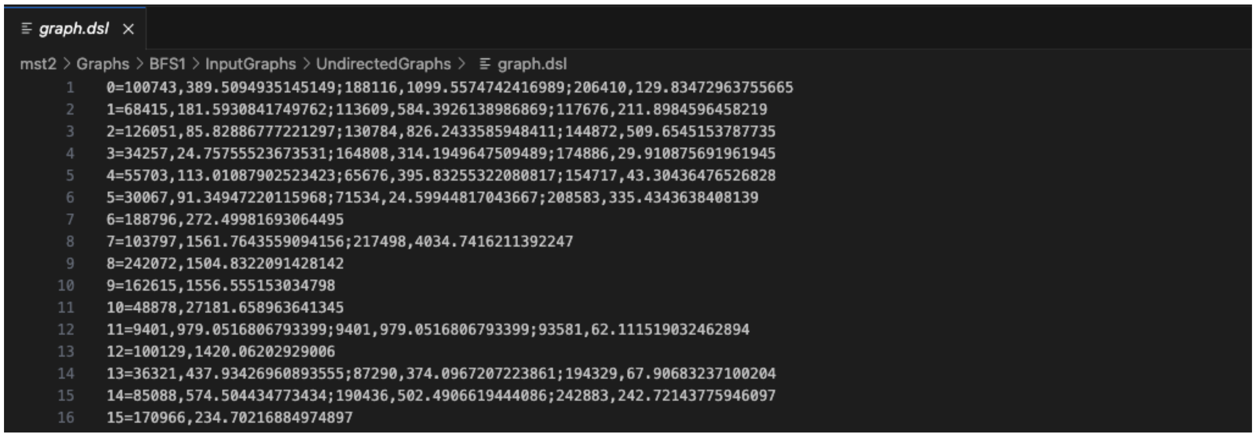
As depicted in phase 1 of Figure 23 and Figure 24, the program execution starts by first importing and storing the GIS vector data in shapefiles consisting of roads or railroads data into the MASS-GIS system.

ii. Generate graph using GeoTool:

Each instance of the imported vector data is of type *MultiLineString* [44] geometry (each multilinestring is made up of many lines) which represents paths. We iterate over each of the vector features and build a graph representing a network of roads / railroads. We utilize the GeoTools library to build a graph where roads represent the edges, endpoints of the roads form the vertices and length of the roads are the edges weights. This process is carried out in phase 2 of the design and sequence diagram in Figure 23 and Figure 24 respectively.

iii. Create input file (DSL file) :

Next, we traverse the generated graph to create an input file (DSL file) consisting of all the outgoing edges from the vertex along with their corresponding edge weights as shown in Figure 25. The input file is created by using two adjacency lists, one to store all adjacent vertices and the other to store edge weights of the adjacent vertex, and then traversing the adjacency lists to write all the adjacent vertices and their corresponding edge weights into an input file called *graph.dsl.* The input file has the format: <*vertex_number*>=<*adjacent_vertexNumber*>,<*edge_weight*> as shown in Figure 25. This step represents phase 3 of the design and sequence diagram in Figure 23 and Figure 24 respectively.


graph.dsl
mst2 > Graphs > BFS1 > InputGraphs > UndirectedGraphs > graph.dsl

```
0=100743,389.5094935145149;188116,1099.5574742416989;206410,129.83472963755665
1=68415,181.5930841749762;113609,584.3926138986869;117676,211.8984596458219
2=126051,85.82886777221297;130784,826.2433585948411;144872,509.6545153787735
3=34257,24.75755523673531;164808,314.1949647509489;174886,29.910875691961945
4=55703,113.01087902523423;65676,395.83255322080817;154717,43.30436476526828
5=30067,91.34947220115968;71534,24.59944817043667;208583,335.4343638408139
6=188796,272.49981693064495
7=103797,1561.7643559094156;217498,4034.7416211392247
8=242072,1504.8322091428142
9=162615,1556.555153034798
10=48878,27181.658963641345
11=9401,979.0516806793399;9401,979.0516806793399;93581,62.111519032462894
12=100129,1420.06202929006
13=36321,437.93426960893555;87290,374.0967207223861;194329,67.90683237100204
14=85088,574.504434773434;190436,502.4906619444086;242883,242.72143775946097
15=170966,234.70216884974897
```

Figure 25: Input graph DSL file.

iv. Compute minimum spanning tree:

MASS [1] library is initialized in this step and the input file (DSL file) is processed to create MASS VertexPlace at each vertex. GraphAgents are initialized at the source and destination VertexPlace. The GraphAgents propagate across VertexPlace and calculate the minimum spanning tree using the algorithm implemented by C. Tsui [16]. However, there is one change to this algorithm. GraphAgents stop computation as soon as the destination node is present in the minimum spanning tree. Thus, a minimum spanning tree from a given source to destination is the shortest path. The output of the minimum spanning tree algorithm is a *String Array* of edges from source to the destination with minimum weight (minimum distance). This step is pictorially represented in phase 4 and 5 of the design diagram and phases 4-7 of sequence diagram in Figure 23 and Figure 24 respectively.

v. Process the results of minimum spanning tree algorithm :

The *String Array* of edges obtained as results from the minimum spanning tree algorithm need to be sorted in the order that they occur in the path from source to the destination to be visualized on the map. An adjacency list containing the adjacent vertices and their corresponding weights is created. Breadth first traversal is applied on this adjacency list and sequence of vertices in the path from source to destination stored in a list. This step is pictorially represented in phase 6 of the design diagram and phases 8-9 of sequence diagram in Figure 23 and Figure 24 respectively.

vi. Visualize the results on the map:

We iterate through the list containing the ordered sequence of vertices in the path from source to destination and create the vector data and visualize it on the map using the GeoTools library in red color. The base map containing all the roads are also visualized in black to show the selected path from source to destination. This step is pictorially represented in phase 7 of the design diagram and phase 10 of sequence diagram in Figure 23 and Figure 24 respectively.

The results of the shortest path GIS query to find the shortest path from Everett Avenue to Hoyt Avenue in Everett, WA are shown in Figure 26. These results are verified using Google Maps as shown in Figure 27. From the results in Figure 26 and Figure 27 we observe that the shortest path query using minimum spanning tree provides 100% accuracy as the results from Google maps match the results obtained using this query in the MASS-GIS system.

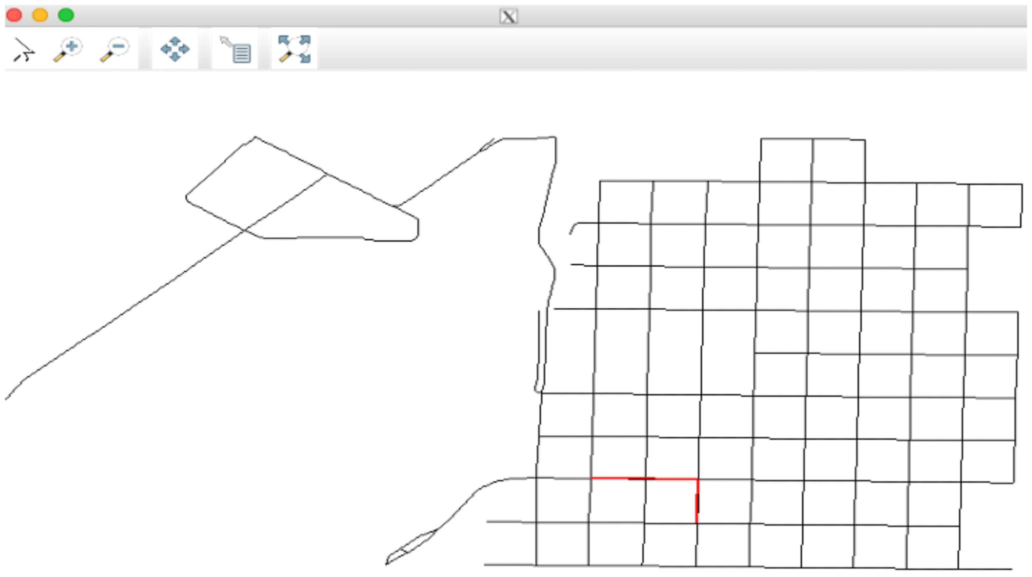


Figure 26: Results of the shortest path GIS query to find the shortest path from Everett Avenue to Hoyt Avenue in Everett, WA.

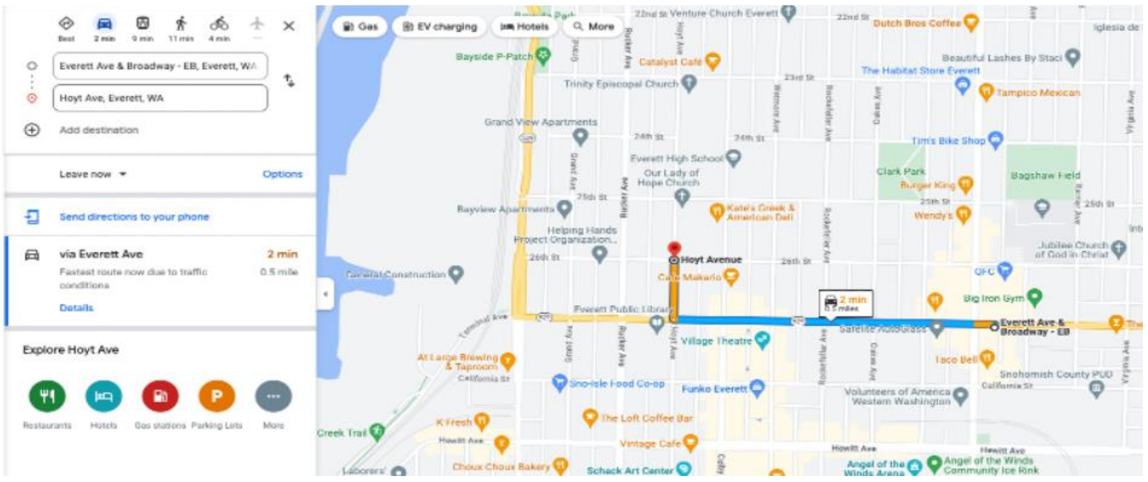


Figure 27: Results of the shortest path GIS query to find the shortest path from Everett Avenue to Hoyt Avenue in Everett, WA using Google maps.

Chapter 5

Verification

This chapter details the benchmark results of the MASS-CQL based GIS queries and computational geometry algorithm-based MASS GIS queries.

5.1 Execution Environment

The benchmark is carried out on 24 computing nodes in the University of Washington Bothell (UWB) internal lab environment. These computing nodes are 64-bit Linux servers mounted on a central filesystem. The java programs are run using OpenJDK version "11.0.18" and Apache Maven 3.6.3.

5.2 Input Datasets

The datasets from various government repositories were considered to test the performance of the agent-based GIS queries implemented in this project. Table 3 summarizes these datasets.

Table 3: Summary of the input GIS datasets used for performance evaluation.

Name of dataset	Description of the GIS query	Type of GIS query	Dataset size	Source
Cities	Attribute GIS query to find all the cities with “country” attribute equal to united states	MASS-CQL based GIS queries	2,533 instances (points)	MASS application bitbucket repository [48]
Cities	Spatial GIS query to find cities within 100 kilometers of Seattle	MASS-CQL based GIS queries	2,533 instances (points)	MASS application bitbucket repository [48]
Mineral resourc es	Spatial GIS query to find mineral resources within 100 kilometers of a given	MASS-CQL based GIS queries	304,632 instances (points)	U.S. Geological Survey government website. The dataset contains information

	coordinate location			about metallic and nonmetallic mineral resources throughout the world[49]
Fire occurrences	Closest Pair of Points: Spatial query to find the closest places of fire occurrences given a coordinate position.	MASS based computational geometry algorithm GIS query	572,834 instances (points)	US department of agriculture government website. The data points represent fire occurrence locations where wildland fires have historically occurred [50]
Populated Cities	Range search: Spatial query to find all the cities located within the range of geographical bound (geographical bound of Australia) using agent propagation	MASS based computational geometry algorithm GIS query	7,342	Natural Earth website. The data points represent major populated cities in the world [51].
Railroads	Minimum Spanning Tree: Spatial query to find the shortest path from a source to destination by propagating agents over a graph using the minimum spanning tree algorithm	MASS based computational geometry algorithm GIS query	250,411 vertices and 250411 + edges	United States department of transportation website [52]. Data consists of railroads from 50 states, the District of Columbia, Mexico, and Canada.

5.3 Performance Evaluation for CQL-MASS based GIS query

i. Attribute GIS query: The GIS dataset consisting of 2,533 cities is divided into 10 rows and 40 columns and distributed across 1, 2, 3, 4, 6, 12, 18 and 24 computing nodes. A CQL query to find all the cities with *CNTRY_NAME* attribute “United States” is executed on each of the fragments of data in parallel. From Figure 28 (and see Appendix B (B.1)) we observe that the execution time decreases as the number of nodes used to run the query increases. This pattern indicates that parallelization has helped achieve CPU scalability.

ii. Spatial GIS query: The GIS dataset of “cities” containing 2,533 cities is divided into 10 rows and 40 columns and distributed across 1, 2, 3, 4, 6, 12, 18 and 24 computing nodes. A spatial query to find all the cities within 100 kilometers distance from the coordinate positions of Seattle is executed in parallel using CQL module and MASS Places. Similar to the attribute GIS query, Figure 29 (and see Appendix B (B.2)) shows a reduction in execution time when the number of computing nodes increases.

To further verify this behavior of improved performance in parallel CQL-MASS GIS query, we execute a spatial GIS query to find all the occurrences of mineral resources within 100 kilometers distance from a coordinate position, on the mineral resources dataset. This dataset consists of 304,632 instances, much larger compared to the “cities” dataset. Figure 30 (and see Appendix B (B.3)) indicates a significant reduction in execution time when the number of computing nodes increases. From Figure 30**,** we observe that the time taken to execute the same spatial GIS query on a large dataset of 304,632 instances using 24 nodes (2492 milliseconds) is comparatively equal to the time taken to execute the query on 2,533 instances using 24 nodes (2,030 milliseconds) with difference a few milliseconds. This indicates that parallelization of GIS queries is suitable for large datasets and gives results in considerably less time. Additionally, from Figure 30, we observe that the time taken to execute the query sequentially on one node increases to 6000 milliseconds (304,632 instances) from 4,500 milliseconds (2,533 instances). This signals the need to parallelize GIS queries for large scale data. In conclusion, parallelization using MASS has made the spatial and attribute query using CQL time efficient.

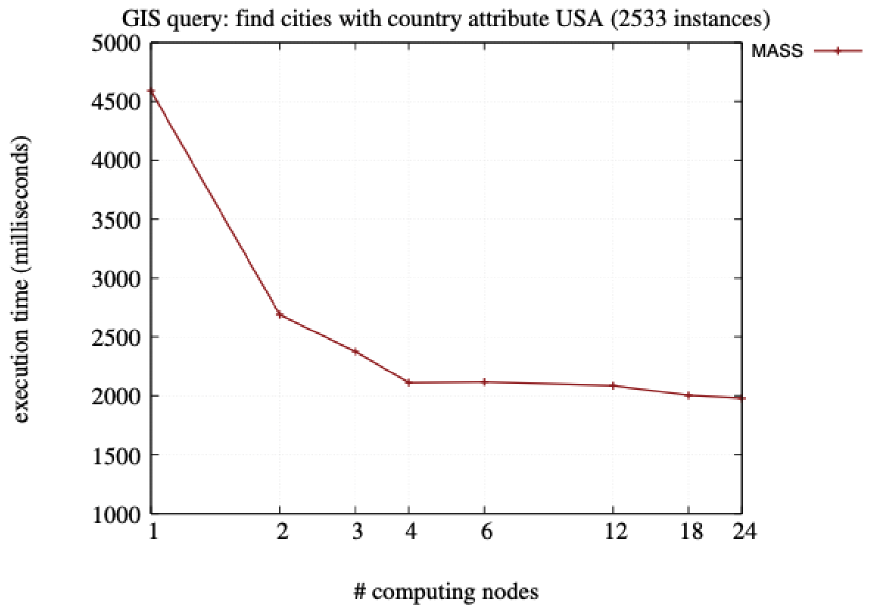


Figure 28: Graph depicting performance of attribute-based CQL-MASS GIS query to find cities with CNTRY_NAME attribute “United States”.

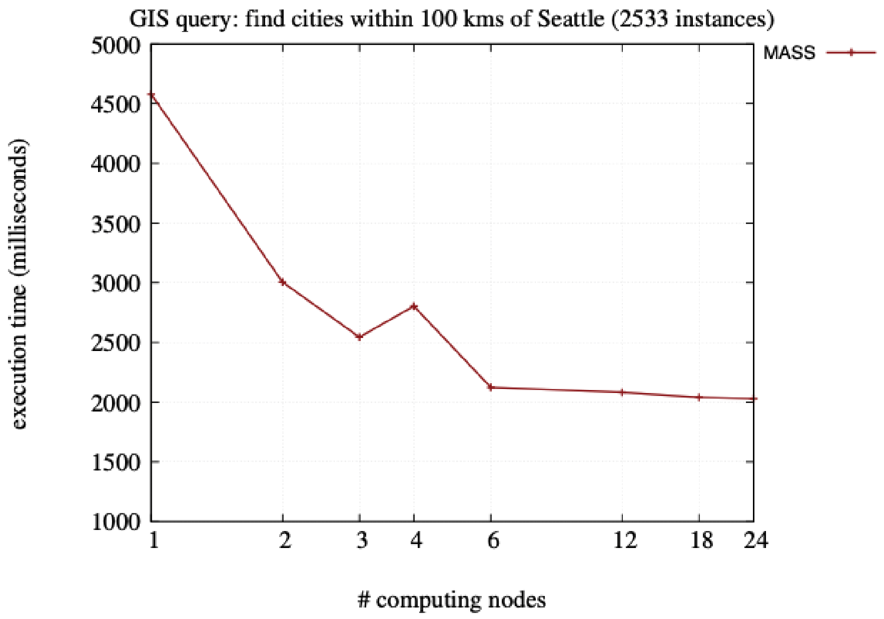


Figure 29: Graph depicting performance of spatial CQL-MASS GIS query to find all the cities within 100 kilometers distance from the coordinate positions of Seattle.

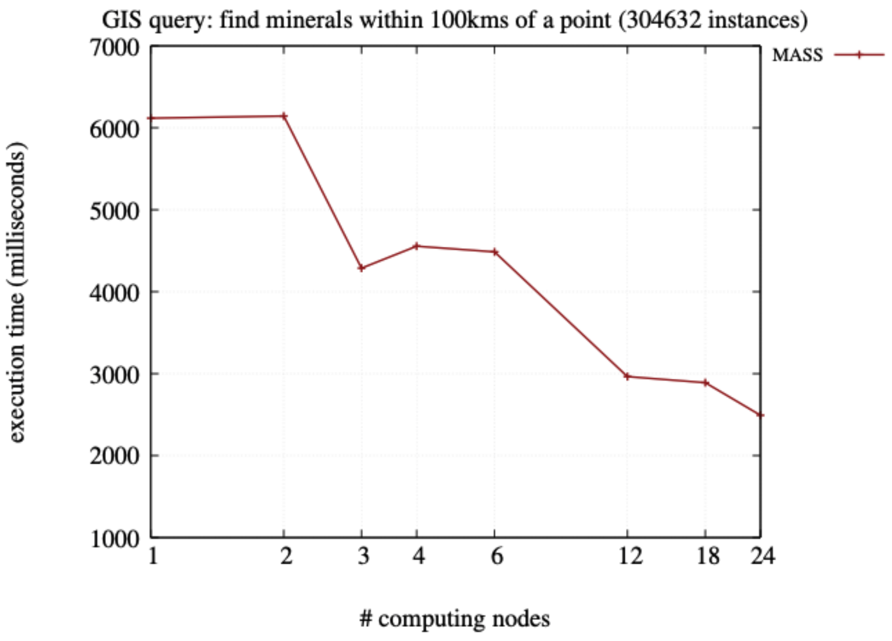


Figure 30: Graph depicting performance of spatial CQL-MASS GIS query to find all the occurrences of mineral resources within 100 kilometers distance from a coordinate position.

5.4 Performance Evaluation for GIS Query Using Closest Pair of Points

The GIS query to find the places of fire occurrence within 100 miles of a given coordinate location is executed on the fire occurrences dataset of 572,834 instances. The performance of the query is evaluated by executing it on 1, 2, 3, 4, 6, 12, 18 and 24 computing nodes. Figure 31 (and see Appendix B (B.4)) depicts a line graph indicating the query performance. From Figure 31**,** we observe that there is a significant decrease in query execution time from 27,657 milliseconds on a single node to 13,318 milliseconds on 24 nodes. Hence, this indicates that parallelization has improved the performance of the GIS query. However, the CPU and memory usage was very high while executing the query, this can be attributed to the large number of MASS Places created across 24 nodes. If we optimize the amount of MASS Places created to execute this query, we can obtain better results and further reduce the amount of time taken to execute the query.

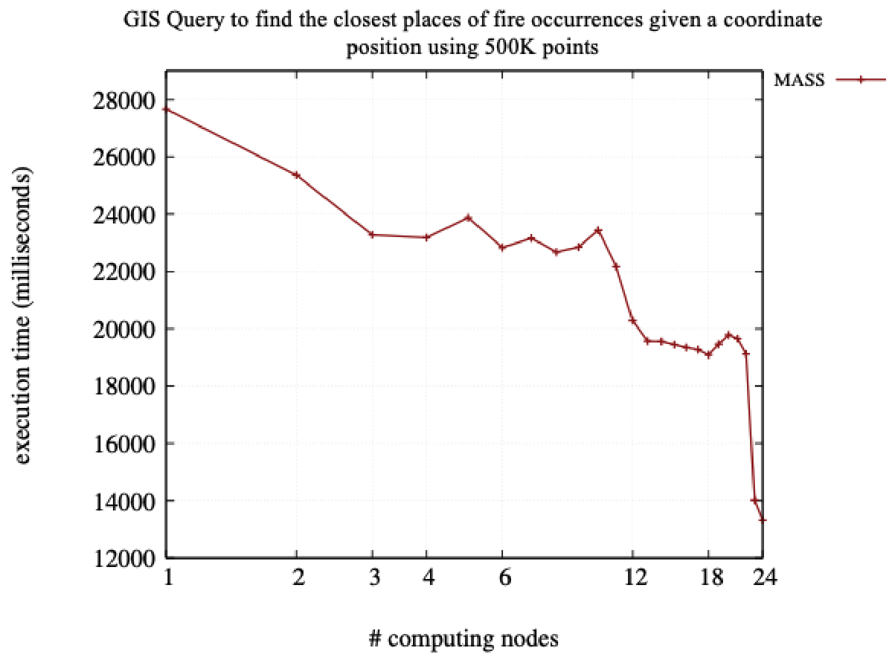


Figure 31: Graph depicting performance of spatial query to find the places of fire occurrence within 100 miles of a given coordinate location.

5.5 Performance Evaluation for GIS Query Using Range Search

The GIS query to find all the cities located within the given range of geographical coordinates i.e geographical bounds of the country Australia is executed on 7,342 cities in the world using 1 to 20 computing nodes. The benchmark was carried out on only 20 nodes rather than up to 24 nodes because the query took longer time to execute as the number of nodes increased. The range search GIS query constructs a KDTree and then propagates agents over it to find the results. The time taken to construct the KDTree, perform range search and the total time taken are measured and presented on a graph in Figure 32.

From Figure 32 (and see Appendix B (B.5)), we observe that the time taken to construct the tree exceeds the time taken to execute range search to a large extent. Therefore, there is a need to improve the performance of the tree construction algorithm which would in turn improve the execution performance of the range search query. The time taken to execute the range search algorithm increases with the increase in the number of nodes. Thus, this indicates that parallelization has not improved the performance of the query. The range search algorithm using MASS agents is equivalent to the bounding box query defined by the CQL library. The range search algorithm takes 93,067 milliseconds for 7,342 instances whereas the MASS-CQL query takes 2,492 milliseconds for 30,4632 instances. From these results we can infer that bounding box CQL query has better performance than range search. Efforts to measure the spatial scalability of the range search GIS query were not successful. Parallel range search queries failed to execute in reasonable time of 15 minutes on large datasets of mineral resources and fire occurrences on 2 nodes. The execution had to be stopped owing to the extremely large amounts of time taken to execute the query. However, this query executed in less than 15 minutes on one node. Therefore, GIS datasets larger than 572,834 instances are required to measure the spatial scalability which are not readily available. In conclusion, the range algorithm using MASS Agents is not suitable for GIS queries and must be further optimized given that the equivalent bound box query using CQL executes faster.

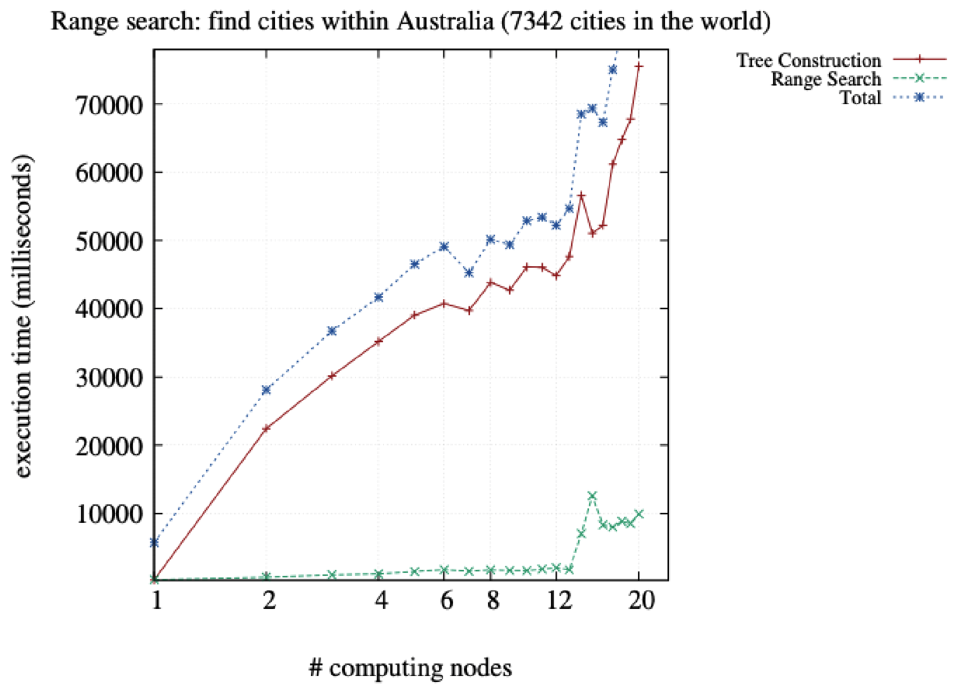


Figure 32: Graph depicting the performance of the range search GIS query to find cities in the range of geographical coordinates of Australia.

5.6 Performance evaluation for minimum spanning tree computational geometry algorithm-based MASS-GIS query

We execute the spatial GIS query to find a shortest path from a given source to destination on the railroads dataset consisting of 250,411 vertices and more than 250,411 edges on 1, 2, 3, 4, 5 and

24 computing nodes. The dataset consists of railroads from 50 states across the United States. Figure 33 (and see Appendix B (B.6, B.7)) provides a graphical representation of this result.

The shortest path GIS query using minimum spanning tree completed its execution in 1175.038, 57.769 ,130.189 , 159.598 seconds with 1, 3, 4 and 24 agents respectively on one computing node. Thus, it indicates that the query executes the fastest when run with 3 agents. The reason for this behavior is because 3 agents are optimal for finding the shortest path and when more than 3 agents are employed an additional amount of time is spent on exchanging information between agents and killing additional agents on a node. From Figure 33 we observe that the time taken to execute the query on one node using 3 agents (57.769 seconds) is less than the time taken to execute on one node with 1 agent (1,175.038 seconds). This indicates that parallelization on one computing agent using multiple agents provides an improvement in performance. The execution performance of the query decreases when run on more than 3 computing nodes. This is because the query is run in parallel on only 3 computing nodes at a time, an additional amount of time is required for agent migration when computing nodes are increased. The optimum result is achieved when the query is executed on 3 nodes in parallel with 3 agents executing independently taking advantage of the computing resources on all the three nodes.

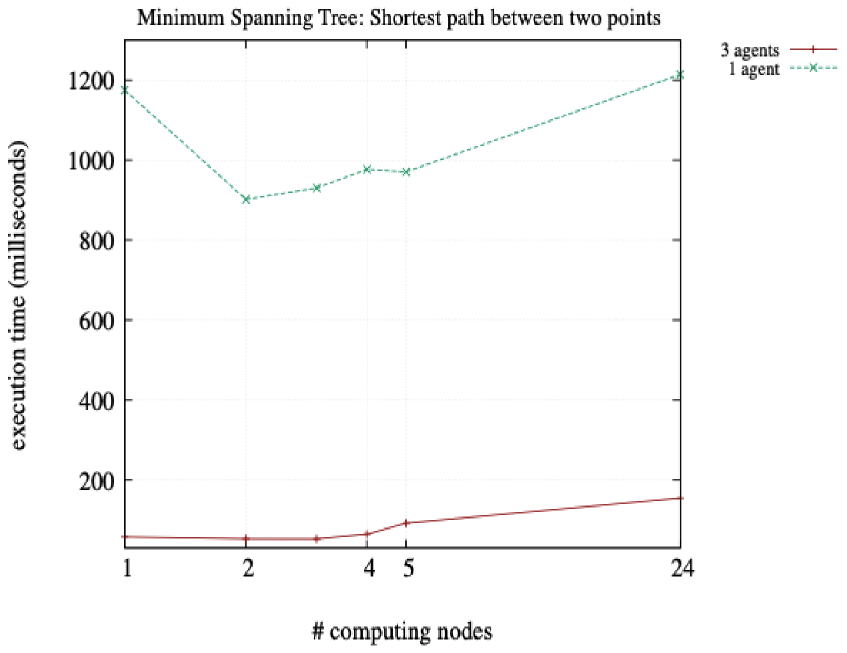


Figure 33: Graph depicting the performance of the shortest path GIS query.

5.7 Summary

From the performance measurements as conducted in section 5.3 through 5.6, we can summarize the strengths and challenges of agent-based GIS as follows:

- The results of benchmarking indicate that there is significant performance improvement in executing parallel attribute and spatial MASS-CQL queries across 24 computing nodes (section 5.4) on small and large datasets.
- Time taken to execute MASS-CQL queries sequentially on large datasets is considerably high and therefore this establishes the need for parallelization.
- The results indicate the time taken to execute the same spatial GIS query on a large dataset of 304,632 instances and 24 nodes (2492 milliseconds) is comparatively equal to the time taken to execute the query on 2,533 instances and 24 nodes (2030 milliseconds) with difference a few milliseconds. This parallelization has made GIS queries time efficient.
- Parallel GIS query using MASS agent propagation and closest pair of points algorithms across 24 nodes are faster compared to its execution on 1 node.
- Parallelization of GIS queries using range search algorithms degrade the performance of the GIS query.
- Parallelization of GIS queries using the minimum spanning tree algorithm provides fastest execution performance when run on 3 computing nodes using 3 agents.

Chapter 6
Conclusion

The achievements and future plans of this research are summarized below.

6.1 Summary

This project successfully demonstrates GIS queries as a practical application of agent-based data analysis. The literature review differentiates our work from other agent-based approach from the viewpoint of parallelization. A significant contribution of this capstone project was to migrate the MASS-GIS system to a scalable non-cloud cluster architecture of 24 nodes, implement spatial, attribute and aggregate GIS queries using CQL, computational geometry algorithms and MASS library, and benchmark their performance on large datasets obtained from government organizations.

The results of the performance evaluation showed that MASS-based GIS queries using the closest pair of points to find cities within a certain distance from a given city and parallel MASS-CQL achieved CPU scalability as the time taken to execute the query decreased with the increase in number of computing nodes. The parallel GIS query to find the shortest path using minimum spanning provided improved performance when run on 3 nodes using 3 agents compared to its sequential implementation using one agent. Furthermore, MASS-based GIS queries using computational geometry algorithms of the closest pair of points and range search provided 100% accuracy. However, agent-based GIS query using range search was not time efficient and took considerable time to execute. Optimization techniques must be applied to shortest path query using minimum spanning tree to provide improved performance on more than 3 nodes.

6.2 Future Work

The following four tasks are being planned:

- Though MASS based GIS queries proved to be a success, however the MASS-GIS system lacks a good UI that will be easy to use. This was because the focus of this project was on improving the efficiency of backend business logic of GIS queries. This could be a potential future work.

- A major drawback of the CQL library is that it doesn’t provide built-in support for direction based (east of a particular point) and complex temporal GIS queries (find the average sales in a particular geographical location in the last 30 days).   MASS-based GIS queries utilizing agents can be applied to address this limitation.
- The MASS-based GIS queries implemented in this project must be benchmarked against other commercial or open-source GIS services.
- The GIS queries using range search and minimum spanning tree implemented in this project should be optimized.

References

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

[2] P. M. Ogden, D. Thomas, and P. Pietzuch, “AT-GIS,” June. 2016, doi: https://doi.org/10.1145/2882903.2882962.

[3] National Geographic, “GIS (Geographic Information System), National Geographic Society,” *education.nationalgeographic.org*. https://education.nationalgeographic.org/resource/geographic-information-system-gis/ (accessed Sep. 13, 2022).

[4] J. Gilroy, S. Paronyan, J. Acoltzi and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA, 2020, pp. 2957-2966, doi: 10.1109/BigData50022.2020.9378298.

[5] Proceedings of the 15th International Conference on Agents and Artificial Intelligence , “Automated Agent Migration over Distributed Data Structures,” Feb. 22, 2023.

[6] K. Araki and T. Shimbo, "An MPI-based Framework for Processing Spatial Vector Data on Heterogeneous Distributed Systems," *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, Hiroshima, Japan, 2016, pp. 554-558, doi: 10.1109/CANDAR.2016.0101.

[7] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen, “Towards parallel spatial query processing for Big Spatial Data,” *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.

[8] N. Minar, R. Burkhart, C. G. Langton, and M. Askenazi, “The Swarm Simulation System: A Toolkit for Building MultiAgent Simulations.” Available at: https://www.santafe.edu/research/results/working-papers/the-swarm-simulation-system-a-toolkit-for-building (Accessed: December 10, 2022).

[9] U. Wilensky, “NetLogo Home Page,” *ccl.northwestern.edu,* 1999. https://ccl.northwestern.edu/netlogo

[10] B. Wang, V. Hess, and A. Crooks, "Mesa-Geo," *GeoSim '22: Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoSpatial Simulation*, no. 978–14503–95373, pp. 1–10, Nov. 2022, doi: https://doi.org/10.1145/3557989.3566157.

[11] M. Sieling , "AGENT-BASED DATABASE WITH GIS ," Jun. 2022. Accessed: Sep. 01, 2022. [Online]. Available: https://depts.washington.edu/dslab/MASS

[12] Open Source Geospatial Foundation, "CQL — GeoTools 29-SNAPSHOT User Guide," docs.geotools.org. https://docs.geotools.org/stable/userguide/library/cql/cql.html (accessed September 17, 2023).

[13] GRASS Development Team , "GRASS GIS - Bringing advanced geospatial technologies to the world," *grass.osgeo.org*, 1998. https://grass.osgeo.org/ (accessed Jan. 15, 2023).

[14] OpenStreetMap, "What is OpenStreetMap,| OpenStreetMap," *welcome.openstreetmap.org*. https://welcome.openstreetmap.org/what-is-openstreetmap/ (accessed Oct. 15, 2022).

[15] V. Mohan, "Automated Agent Migration Over Structured Data ," Jun. 2022. Accessed: Jan. 05, 2023. [Online]. Available: https://depts.washington.edu/dslab/MASS/

[16] C. Tsui, "Agentbased graph programming in MASS Java and comparison with graphic libraries," Dec. 2022. Accessed: Mar. 06, 2023. [Online]. Available: https://depts.washington.edu/dslab/MASS/

[17] QGIS, "Raster Data," *docs.qgis.org*. https://docs.qgis.org/2.18/en/docs/gentle_gis_introduction/raster_data.html (accessed May 17, 2023).

[18] GIS Geography, "US Precipitation Map," *GIS Geography*, Jan. 26, 2020. https://gisgeography.com/us-precipitation-map/

[19] QGIS project, "Vector Data," *docs.qgis.org*, May 17, 2023. https://docs.qgis.org/2.8/en/docs/gentle_gis_introduction/vector_data.html

[20] University Consortium for Geographic Information Science, "FC-13 - Spatial Queries, GIS & T Body of Knowledge," *gistbok.ucgis.org*. https://gistbok.ucgis.org/bok-topics/spatial-queries/ (accessed Sep. 25, 2022).

[21] PostGIS Developers, "PostGIS — Spatial and Geographic Objects for PostgreSQL," *Postgis.net*, 2020. https://postgis.net/ (accessed Sep. 25, 2022).

[22] Open Source Geospatial Foundation, "GeoTools Documentation — GeoTools Documentation," *docs.geotools.org*. https://docs.geotools.org/ (accessed Sep. 15, 2022).

[23] B. Walker and T. Phillips Johnson, "NetLogo and GIS: A Powerful Combination," *EPiC Series in Computing*, vol. 58, pp. 257–264, Mar. 2019, doi: https://doi.org/10.29007/w8gh.

[24] *Summary of NetLogo - static.cambridge.org* . Available at: https://static.cambridge.org/content/id/urn:cambridge.org:id:article:S2326376819000068/resource/name/S2326376819000068sup002.pdf (accessed: December 9, 2022).

[25] ARGONNE NATIONAL LABORATORY, "Repast Suite Documentation," *repast.github.io*. https://repast.github.io/docs.html (accessed Oct. 22, 2022).

[26] ARGONNE NATIONAL LABORATORY, "Repast Simphony Reference Manual," *repast.github.io*. https://repast.github.io/docs/RepastReference/RepastReference.html (accessed Oct. 12, 2022).

[27] S. Luke *et al.*, 'The MASON Simulation Toolkit: Past, Present, and Future', in *Multi-Agent-Based Simulation XIX*, 2019, pp. 75–86.

[28] K. Sullivan, M. Coletti, and S. Luke, "GeoMason: Geospatial Support for MASON", George Mason University, Fairfax, Virginia, *Technical Report,* 2010.

[29] J. E. Lane, "Anylogic on HPC clusters cons of anylogic on HPC," *Academia.edu,* 22-Apr-2016. [Online]. Available: https://www.academia.edu/24666575/AnyLogic_on_HPC_Clusters_Cons_of_AnyLogic_on_HPC. (accessed: 11-Dec-2022).

[30] A. Petrasova, "FUTURES v3.0.0 software for urban growth modeling." Zenodo, 02-Jun-2022, doi: 10.5281/ZENODO.6607097.

[31] "FileDataStore (Geotools modules 30-SNAPSHOT API)," *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/data/FileDataStore.html (accessed Mar. 15, 2023).

[32] "FileDataStoreFinder (Geotools modules 30-SNAPSHOT API)," *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/data/FileDataStoreFinder.html (accessed Mar. 16, 2023).

[33] Open-Source Geospatial Foundation, "FeatureSource — GeoTools 29-SNAPSHOT User Guide," *docs.geotools.org*. https://docs.geotools.org/stable/userguide/library/data/featuresource.html (accessed May 17, 2023).

[34] “SimpleFeatureCollection (Geotools modules 28-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/stable/javadocs/org/geotools/data/simple/SimpleFeatureCollection.html (accessed May 17, 2023).

[35] “MapContent (Geotools modules 28-SNAPSHOT API),” docs.geotools.org. https://docs.geotools.org/stable/javadocs/org/geotools/map/MapContent.html (accessed Mar. 14, 2023).

[36] “JMapFrame (Geotools modules 30-SNAPSHOT API),” docs.geotools.org. https://docs.geotools.org/latest/javadocs/org/geotools/swing/JMapFrame.html (accessed Mar. 14, 2023).

[37] “Layer (Geotools modules 30-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/map/Layer.html (accessed Mar. 14, 2023).

[38] “FeatureLayer (Geotools modules 30-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/map/FeatureLayer.html (accessed Mar. 14, 2023).

[39] “Style (Geotools modules 28-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/stable/javadocs/org/geotools/styling/Style.html (accessed May 17, 2023).

[40] “SimpleFeatureIterator (Geotools modules 28-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/stable/javadocs/org/geotools/data/simple/SimpleFeatureIterator.html (accessed May 17, 2023).

[41] “Point (org.locationtech.jts:jts-core 1.19.0 API),” *locationtech.github.io*. https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/Point.html (accessed May 17, 2023).

[42] “GeodeticCalculator (Geotools modules 30-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/referencing/GeodeticCalculator.html (accessed May 17, 2023).

[43] “Harvard Dataverse,” *dataverse.harvard.edu*. https://dataverse.harvard.edu

[44] “Geometries (Geotools modules 28-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/stable/javadocs/org/geotools/geometry/jts/Geometries.html (accessed May 17, 2023).

[45] “LineStringGraphGenerator (Geotools modules 30-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/graph/build/line/LineStringGraphGenerator.html

[46] “LineStringGraphGenerator (Geotools modules 30-SNAPSHOT API),” *docs.geotools.org*. https://docs.geotools.org/latest/javadocs/org/geotools/graph/build/line/LineStringGraphGenerator.html

[47] “GeoTools - Graph,” *Graph - GeoTools 29-SNAPSHOT User Guide*. [Online]. Available: https://docs.geotools.org/latest/userguide/extension/graph/index.html. (accessed: 13-Dec-2022).

[48]“Bitbucket,” *bitbucket.org*. https://bitbucket.org/mass_application_developers/mass_java_appl/src/master/ (accessed May 17, 2023).

[49] U.S. Geological Survey, “National Minerals Information Center | U.S. Geological Survey,” *www.usgs.gov*. https://www.usgs.gov/centers/national-minerals-information-center

[50] US department of Agriculture, “National USFS Fire Occurrence Point (Feature Layer) | Ag Data Commons,” *data.nal.usda.gov*. https://data.nal.usda.gov/dataset/national-usfs-fire-occurrence-point-feature-layer-0 (accessed May 17, 2023).

[51] “Natural Earth» 1:10m Cultural Vectors - Free vector and raster map data at 1:10m, 1:50m, and 1:110m scales.” https://www.naturalearthdata.com/downloads/10m-cultural-vectors/

[52] United States Department of Transportation, “North American Rail Network Lines,” *geodata.bts.gov*. https://geodata.bts.gov/datasets/d83e85154a304da995837889cc4012e3_0/about (accessed May 17, 2023).

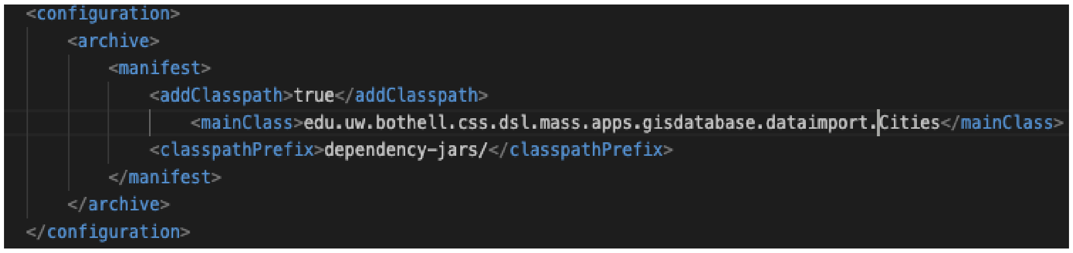
Appendix A: Installation

The code to setup and run the MASS-GIS queries is currently present in the mass_java_appl bitbucket repository under the branch gis_queries_sahana (https://bitbucket.org/mass_application_developers/mass_java_appl/src/gis_queries_sahana/Applications/gis_queries/).

The following steps need to be followed to install, build, and run the parallel spatial and attribute-based MASS-GIS queries.

1. Download / clone the MASS core library from mass_java_core bitbucket repository.
2. Navigate to the downloaded MASS core library folder and install it using the command: *"mvn -DskipTests clean package install"*.
3. Download / clone mass_java_appl repository and checkout the gis_queries_sahana branch using the command *"git checkout gis_queries_sahana"*.
4. Navigate to the pom.xml file under "/gis_quries/Applications/gis_database/" and add the appropriate path of the GIS query to be run under *mainClass* tag.

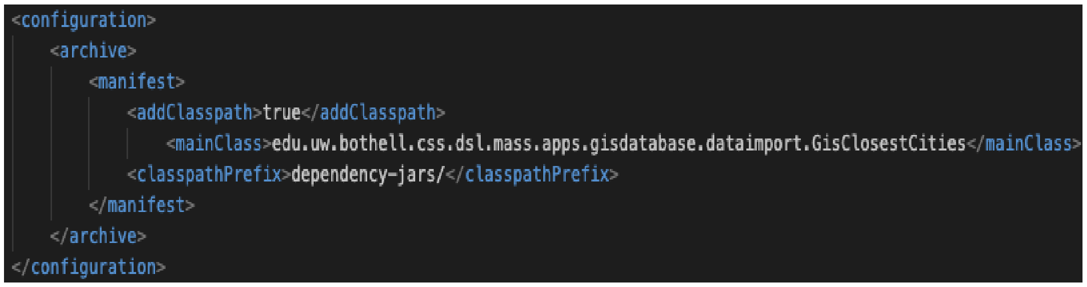
- To run the parallel attribute and spatial GIS query using MASS and CQL. Add the path of *Cities.java* class under *mainClass* tag in pom.xml as shown in Figure 34.



```
<configuration>
    <archive>
        <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.Cities</mainClass>
            <classpathPrefix>dependency-jars/</classpathPrefix>
        </manifest>
    </archive>
</configuration>
```

Figure 34: Pom.xml file configuration to run parallel GIS queries using MASS and CQL.

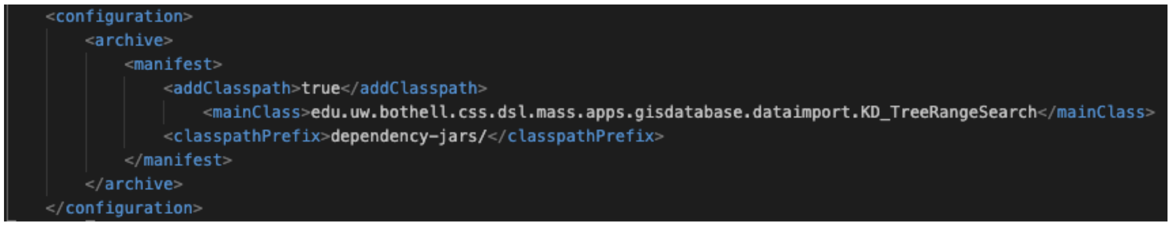
- To run the parallel GIS query using MASS-based closest pair of points algorithms. Add the path of *GisClosestCities.java* class under *mainClass* tag in pom.xml as shown in Figure 35.



```xml
<configuration>
    <archive>
        <manifest>
            <addClasspath>true</addClasspath>
                <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.GisClosestCities</mainClass>
            <classpathPrefix>dependency-jars/</classpathPrefix>
        </manifest>
    </archive>
</configuration>
```

Figure 35: Pom.xml configuration file to run the parallel GIS query using MASS-based closest pair of points algorithm.

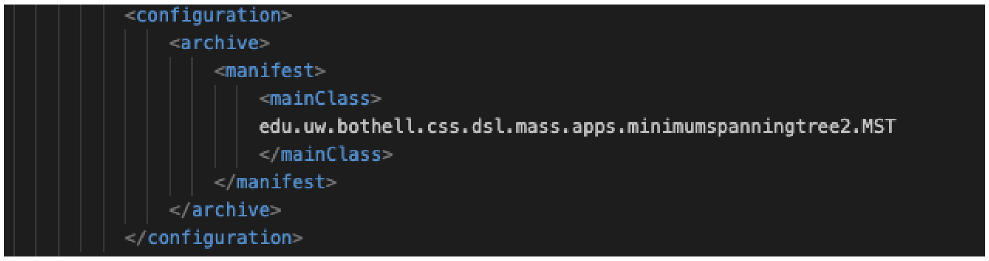
- To run the parallel GIS query using MASS-based range search algorithm. Add the path of *KD_TreeRangeSearch.java* class under *mainClass* tag in pom.xml as shown in Figure 36.



```xml
<configuration>
    <archive>
        <manifest>
            <addClasspath>true</addClasspath>
                <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.KD_TreeRangeSearch</mainClass>
            <classpathPrefix>dependency-jars/</classpathPrefix>
        </manifest>
    </archive>
</configuration>
```

Figure 36: Pom.xml configuration file to run the parallel GIS query using MASS-based range search algorithm.

- The code to run the shortest path GIS query using minimum spanning tree is present under *"/minimum_spanning_tree/Graphs/BFS1/"* folder. The code is present in a separate folder as the latest mass core version (1.4.3-SNAPSHOT) has bugs and displays thread failed exception while running code that uses MASS GraphPlaces (as in the case of minimum spanning tree GIS query). Currently, the code executes successfully with MASS core version 1.4.0-SNAPSHOT. Once the bug fixes in MASS core version 1.4.3-SNAPSHOT are complete the shortest path GIS query can be added to the same folder as other GIS queries. The pom.xml in *"/minimum_spanning_tree/Graphs/BFS1/"* folder can be compiled to run the shortest path GIS query.

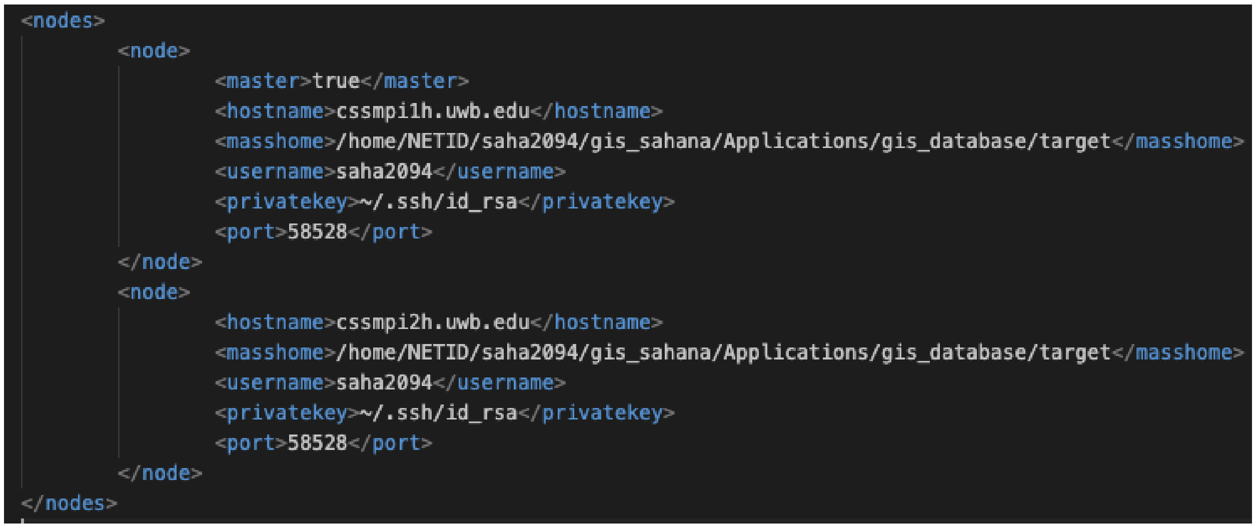


```
<configuration>
    <archive>
        <manifest>
            <mainClass>
            edu.uw.bothell.css.dsl.mass.apps.minimumspanningtree2.MST
            </mainClass>
        </manifest>
    </archive>
</configuration>
```

Figure 37: Pom.xml configuration file to run the parallel GIS query using MASS-based range search algorithm.

5. Compile the pom.xml file using the command *"mvn package"*.

6. Navigate to the target directory containing the jar file using the command: *"cd target"* and create the nodes.xml file with the appropriate .ssh private key, hostname, masshome and username. The detailed instructions to create the nodes.xml file is provided in the MASS JAVA developers guide in https://depts.washington.edu/dslab/MASS/. Figure 38 depicts a sample *nodes.xml* file.



```
<nodes>
        <node>
                <master>true</master>
                <hostname>cssmpi1h.uwb.edu</hostname>
                <masshome>/home/NETID/saha2094/gis_sahana/Applications/gis_database/target</masshome>
                <username>saha2094</username>
                <privatekey>~/.ssh/id_rsa</privatekey>
                <port>58528</port>
        </node>
        <node>
                <hostname>cssmpi2h.uwb.edu</hostname>
                <masshome>/home/NETID/saha2094/gis_sahana/Applications/gis_database/target</masshome>
                <username>saha2094</username>
                <privatekey>~/.ssh/id_rsa</privatekey>
                <port>58528</port>
        </node>
</nodes>
```

Figure 38: Sample nodes.xml configuration file.

7. Run the GIS queries using following commands in the target directory.

- Command for MASS-CQL GIS query:
  java -jar gis_database-1.0-SNAPSHOT.jar -f /home/NETID/saha2094/gis_sahana/Applications/gis_database/input1 -htc 1 -vtc 2
  Here in this command -f is a required parameter specifying the path of the input folder, -htc is the number of horizonatl divisions to divide the GIS map, -vtc is the number of vertical divisions to divide the GIS map.

- Command for GIS query using closest points and range search:
  java -jar gis_database-1.0-SNAPSHOT.jar
- Command for GIS query to find shortest path using minimum spanning tree:
  java -jar BFS2-1.0-SNAPSHOT.jar <source_node> <destination_niode>
  Example: *java -jar BFS2-1.0-SNAPSHOT.jar 1 57* .

Appendix B: Results

The results of parallel MASS-GIS queries in section 5.3 through 5.6 are detailed in a tabular format in this section.

B.1. The results of performance evaluation for CQL-MASS based GIS query to find cities with *CNTRY_NAME* attribute "United States" are presented in a tabular format in Table 4.

Table 4: Summary of the execution time taken by attribute based CQL-MASS GIS query to find cities with *CNTRY_NAME* attribute "United States".

Number of nodes	**Time in milliseconds**
1	4589
2	2688
3	2376
4	2113
6	2119
12	2088
18	2004
24	1980

B.2. The results of performance evaluation for CQL-MASS based GIS query to find all the cities within 100 kilometers distance from the coordinate positions of Seattle are presented in a tabular format in Table 5.

Table 5: Summary of the execution time taken by spatial CQL-MASS GIS query to find all the cities within 100 kilometers distance from the coordinate positions of Seattle.

Number of nodes	**Time in milliseconds**
1	4589
2	3006
3	2545
4	2804
6	2124
12	2088
18	2040
24	2030

B.3. The results of performance evaluation for CQL-MASS based GIS query to find all the occurrences of mineral resources within 100 kilometers distance from a coordinate position are presented in a tabular format in Table 6.

Table 6: Summary of the execution time taken by spatial CQL-MASS GIS query to find all the occurrences of mineral resources within 100 kilometers distance from a coordinate position.

Number of nodes	**Time in milliseconds**
1	6118
2	6144
3	4288
4	4556
6	4487
12	2964
18	2891
24	2492

B.4. The results of performance evaluation for MASS based GIS query using closest pair of points algorithm to find the places of fire occurrence within 100 miles of a given coordinate location are presented in a tabular format in Table 7.

Table 7: Summary of the execution time taken by the spatial GIS query to find the places of fire occurrence within 100 miles of a given coordinate location.

Number of nodes	Time in milliseconds
1	27657
2	25359
3	23274
4	23176
6	22831
12	20284
18	19086
24	13318

B.5 The results of performance evaluation for range search GIS query to find cities in the range of geographical coordinates of Australia are presented in a tabular format in Table 8.

Table 8: Summary of the execution time taken by the range search GIS query to find cities in the range of geographical coordinates of Australia.

Number of nodes	**Tree Construction (time in milliseconds)**	**Range search (time in milliseconds)**	**Total (time in milliseconds)**
1	262	307	5733
2	22437	659	28103
3	30145	1009	36730
4	35190	1136	41690

5	39072	1479	46526
6	40737	1769	49080
7	39737	1513	45274
8	43831	1737	50139
9	42705	1636	49392
10	46121	1611	52884
11	46061	1843	53395
12	44820	2039	52218
13	47643	1736	54663
14	56576	7037	68494
15	51020	12593	69341
16	52227	8319	67274
17	61192	8000	74973
18	64774	8842	80750
19	67807	8542	84314
20	75518	9916	93067

B.6 The results of performance evaluation for shortest path GIS query using 3 MASS Agents are presented in a tabular format in Table 9.

Table 9: Summary of the execution time taken by the shortest path query using 3 MASS Agents

Number of nodes	**Time in seconds**
1	57.769
2	53.231
3	52.936
4	64.705
5	92.476

24	154.978

B.7 The results of performance evaluation for shortest path GIS query using 1 MASS Agent are presented in a tabular format in Table 10.

Table 10: Summary of the execution time taken by the shortest path query using 1 MASS Agent.

Number of nodes	**Time in seconds**
1	1175.038
2	901.643
3	930.454
4	977.012
5	970.812
24	1214