AGENT-BASED DATABASE WITH GIS

Maré G. Sieling

A Capstone Project

submitted in partial fulfilment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2022

Reading Committee:

Prof. Munehiro Fukuda, Chair

Prof. Arnold Lund

Prof. Min Chen

Prof. Erika Parsons

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

AGENT-BASED DATABASE WITH GIS

Maré G. Sieling

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
Computer Science

Geographic Information Systems (GIS) create, manage, analyse and maps data. These systems are used to find relationships and patterns between different pieces of data in a geographically long distance. GIS data can be extremely large and analysing the data can be laborious while consuming a substantial amount of resources. By distributing the data and processing it in parallel, the system will consume less resources and improve performance.

The Multi-Agent Spatial Simulation (MASS) library applies agent-based modelling to big data analysis over distributed computing nodes through parallelisation. *GeoTools* is a GIS system that is installed on a single node and processes data on that node. By creating a distributed GIS from *GeoTools* with the MASS library, results are produced faster and more effectively than traditional GIS systems located on a single node.

This paper discusses the efficacy of coupling GIS and MASS through agents that render fragments of feature data as layers on places, returning the fragments to be combined for a completed image. It also discusses distributing and querying the data, returning results by running

a query language (CQL). Image quality is retained when panning and zooming without major loss of performance by rerendering visible sections of the map through agents and parallelisation. Results show that coupling GIS and MASS significantly improves the efficiency and scalability of a GIS system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATION

To my family

Chapter 1.
INTRODUCTION

1.1 PROBLEM DEFINITION

Geographic Information Systems (GIS) are databases containing geographic data with tools to visualise and analyse the data. The data set is based on spatial locations [1]. The data's locations, such as roads, forests and cites (features), are defined as polygons, lines or points [2]. GIS is extremely resource intensive. Large images and complex data sets are slow or even impossible to analyse and display if the system does not have enough resources. It makes GIS-capable systems expensive and not very portable as such large systems are also power-hungry and cumbersome.

This capstone integrates an agent-based modelling (ABM) with a GIS, *GeoTools*, that can be processed for data by using the Multi-Agent Spatial Simulation (MASS) library and its Agents. The MASS library provides a method for organising data using the distributed Places objects [3]. There is more flexibility to adapt the GIS as needed by using an open-source GIS. This includes implementing the migration to the GIS (*GeoTools*) and completing an interface between MASS and the GIS. The project scales up the GIS database over a cluster system (MASS Places), creating a distributed array. This facilitates agent-based inquiries into the multi-dimensional database, allowing agent-based GIS data retrieval and computation across the distributed system. The GIS consists of spatially referenced datasets in the database that are separated into vector and raster formats. Raster represents surfaces while the vector data represent edges and vertices.

Mapping the datasets to MASS over distributed systems allows for larger datasets that can be accessed by agents. This improves scalability for the GIS data as the dataset size is not limited by having to fit on one machine to be visualised. The interface between *GeoTools* and the MASS distributed database allows *GeoTools* to access data as before, while the backend returns the data through MASS processes, with the interface simulating access to the hard drive of the computer GeoTools is running on.

This project adds various contributions to the MASS project.

- It creates a scalable distributed database as an abstract data source, with loose coupling and an interface hiding the implementation from the GIS.
- Agents render parts of a map, with each agent executing a piece of work on a place on a node, allowing for parallelisation.
- The completed GIS will perform as a single system image across a distributed system.

1.2 GeoTools

The project creates an implementation of the *GeoTools* data source that uses MASS to retrieve data from the distributed database.

A survey was done of the various GIS open-source software options before deciding on using *GeoTools*. The *GeoTools* open-source library can be used for the implementation of geographic information systems (GIS). It provides methods to manipulate geospatial data [4]. It was chosen for this integration as it forms the basis of many of the most popular GIS tools on the market currently. The code base is currently stable, follows Open Geospatial Consortium specifications and is consistently being supported.

GeoTools supports various raster and vector formats used in GIS such as shape files and geotiff (see A.2.1 GeoTiff) with the option of using plugins to support new formats. It also includes database support, such as *postgis* (see A.3 PostGis).

GeoTools' various downstream projects include *uDig*, *Geomajas* and *GeoServer*. It is designed to allow for integration with custom content. In this project, MASS and *GeoTools*' raw data is loosely coupled (see 2.3.1 Loose Coupling), however, the implementations for rendering the map and reading the data is tightly coupled to the *GeoTools* API (see 2.3.2 Tight Coupling).

The TIFF and shape files are an industry standard and not specific to *GeoTools*. The libraries and APIs in *GeoTools* are used to perform the agent-based rendering, creating an implementation that is tightly coupled to the *GeoTools* libraries. The same is true in the process for data retrieval, as the DataStore implements GeoTools APIs.

1.3 GOALS

This project is part of a larger group project, MASS, a parallelizing library for multi-agent spatial simulation, under the direction of Professor Fukuda at the University of Washington Bothell.

- The goal of the current project is to prove the GIS database can be implemented as a MASS application. For a full description of the MASS team project, see the Appendix (A.1).
- Connecting MASS to GIS shows the efficiency of using a distributed system coupled with a multi-agent spatial system. This connection improves scalability compared to a GIS system that is limited to one computing node. The implementation of agents increases efficiency, performance and scalability by performing actions and computations in parallel on computing nodes instead of on the main node, allowing for larger datasets and more computations.
- Agents allows the system to perform as a single system image (SSI).
- A simple graphical user interface presents the program as an SSI while improving usability compared to running command line arguments. This improves testing and makes it easier for the developer to visualise new features added to the database. It also serves as a basis for the development of a fully-fledged GIS for users.

1.4 CONSTRAINTS, RISKS, AND RESOURCES

This project runs successfully on the University of Washington Linux servers and on Amazon Web Services (AWS). The University of Washington's Linux servers do have space constraints. There are no practical space constraints when running the project on AWS, however, non-free tiers must be used.

1.5 REPORT STRUCTURE

The report that follows discusses related research and implementations in chapter 2, with the implementation of the system in chapter 3. Chapter 4 evaluates the results and compares that to the goals of the project. The conclusions are discussed in chapter 5.

Chapter 2.
RELATED WORKS

Implementing Geographic Information Systems (GIS) as a distributed database is the subject of many reference works. There are various possible methods for integrating GIS and agent-based models. What follows is a discussion of agent-based parallel computing, the types of GIS and the theory surrounding integrating GIS with agent-based models.

2.1 AGENT-BASED PARALLEL COMPUTING

Multi-agent simulations provide an alternative method for modelling complex applications [5]. A multi-agent system contains multiple agents in one distributed system. Agents act as independent entities that can act together or individually [5]. They work in varied environments, including distributed and centralised environments. Each agent acts according to its own rules and protocols. A group of agents in a multi-agent system can perform a set of tasks or goals that would not be possible for a single agent [6].

2.2 GEOGRAPHIC INFORMATION SYSTEMS AND GEOTOOLS

Geographic Information Systems (GIS) rely on data that is based on spatial relationships between features [1]. As such, GIS data relies on object-oriented data models rather than process models. GIS uses tools to manage large set of data and complex modelling environments. In the data, the world gets abstracted to be represented as a set of different features as layers on a base. There are multiple open-source alternatives available, amongst others *OrbisGIS*, *uDig* and *GeoMaja*.

GeoTools is an open-source, Java code library for creating GIS tools that are compliant with Open Geospatial Consortium (OGC). It is released under the GNU Lesser General Public License (LGPL). It is a stable library that is consistently updated and well-documented. It is used as the basis for many other GIS, such as *uDig*. It was designed for developers building GIS products with spatial, not temporal scalability, by simplifying the construction of data processing applications as well as abstracting data sources, feature models and coordinate system map

projections [7]. *GeoTools* is thus not aimed at general users, but on developers who implement the library.

2.3 Integrating GIS and Agent-Based Models

Coupling agent-based process models with data-based GIS models require planning conceptually and functionally [1]. Processes and objects need to be represented and their interactions defined. Traditional GIS use spatial data models (location) at the cost of temporal dimensions (time). The process models use time and behaviour, so sacrifice space and spatial relationships. They require adaptation to integrate with the data-based models of GIS, since GIS contains spatial relationships.

GIS can be coupled with agent-based models either through loose coupling, tight coupling or as complete integration [6].

Complete integration requires spatial features and the encapsulation of the agent-based model in the GIS. It would also require temporal features, which is not usually a part of a GIS. It is thus not a recommended approach.

2.3.1 Loose Coupling

As discussed by Peng et al., in general, loose coupling is preferable to tight coupling and full integration.

Loose coupling makes testing simpler and the system more maintainable [9]. Loosely coupled code that is built on abstractions, adheres to the Dependency Inversion Principle (DIP). DIP requires objects to rely on abstractions, not concrete implementations [10]. The code is reusable, independent and more easily maintained [11]. Adding new features becomes simpler. Overall, the system is more resilient than tight coupling or full integration [12]. A very loosely coupled system can however become overly complex with performance deterioration.

The loosely coupled system illustrated in Figure 1 creates transitional files for data exchanges [6]. These files are in a format understandable to both the GIS and the Multi-Agent System. The systems stay independent. The extra files do consume extra resources and slows down performance. It is thus not an ideal solution for merging MASS with GIS.

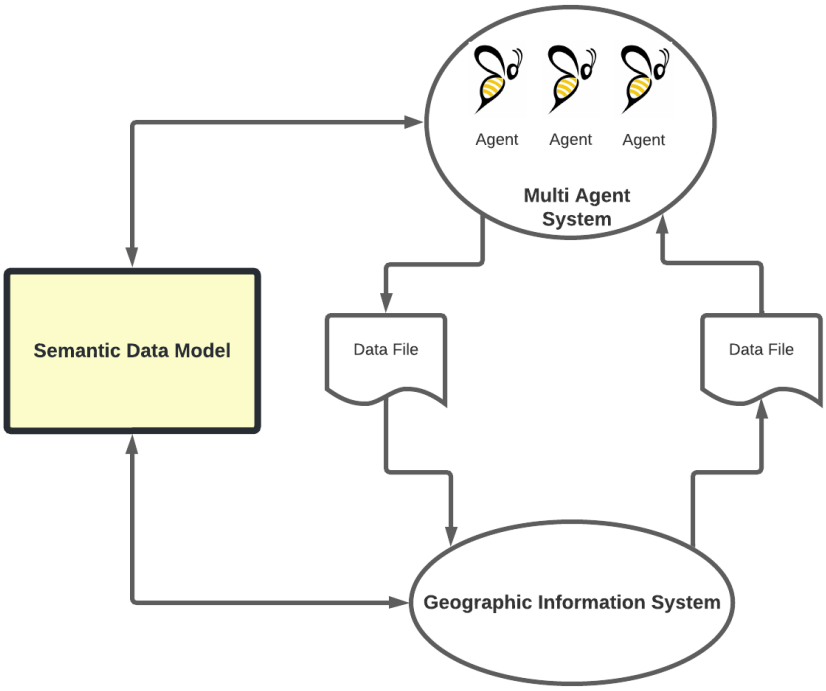

Figure 1 Loosely coupled system [6]

2.3.2 Tight Coupling

Many researchers advocate for tight coupling to improve performance and efficiency, which are high priorities in GIS [6], [1]. Tight coupling can be achieved by integrating MASS into GIS or the opposite, embedding GIS into MASS [13], as illustrated in Figure 2.

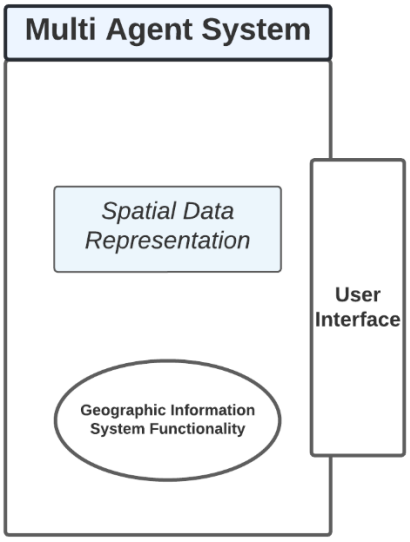

Figure 2 Tight coupling: GIS integrated into a multi-agent system [13]

Early integration approaches were ABM-centric. Geographic objects with features and attributes can be implemented as the agents, allowing for a state and behaviour [8]. This is an expensive approach as GIS code is developed to be efficient, while AMB systems are not designed for spatial interactions [1].

Using a middleware approach can couple a multi-agent system with existing GIS software. Creating an interface or middleware between a GIS and ABM minimises the need to provide full functionality in a single system [1]. This approach also alleviates the need to build a completely new system.

Tight coupling hides the implementation from the user, creating a better user experience through a singular user interface. This is considered a more complex implementation than loose coupling [13]. Unlike loose coupling, there is no need for transitional files.

2.3.3 Dynamic vs Static Coupling of Data

The geographic data for the GIS can be coupled during the execution of the model. This would be considered dynamic coupling. In contrast, static coupling requires the data to be imported before simulation [2]. This project uses static coupling.

Chapter 3.
IMPLEMENTATION

A Geographic Information System (GIS) database needs specific elements to be functional. It needs a mechanism to load into the database, a mechanism to query the data as well as a mechanism to perform processing of the data.

This implementation functions as an interface between the GIS and MASS. The GIS provides the business logic while MASS provides the scalability and parallelisation of data and processing. The interface being implemented provides the user with an interface to the GIS, both from the command line and through a Graphic User Interface (GUI). The data is distributed through MASS. GIS queries are run through this implementation and maps are rendered and displayed in the GUI. MASS provides the network infrastructure for data retrieval and processing on nodes.

Figure 3 illustrates the design components.

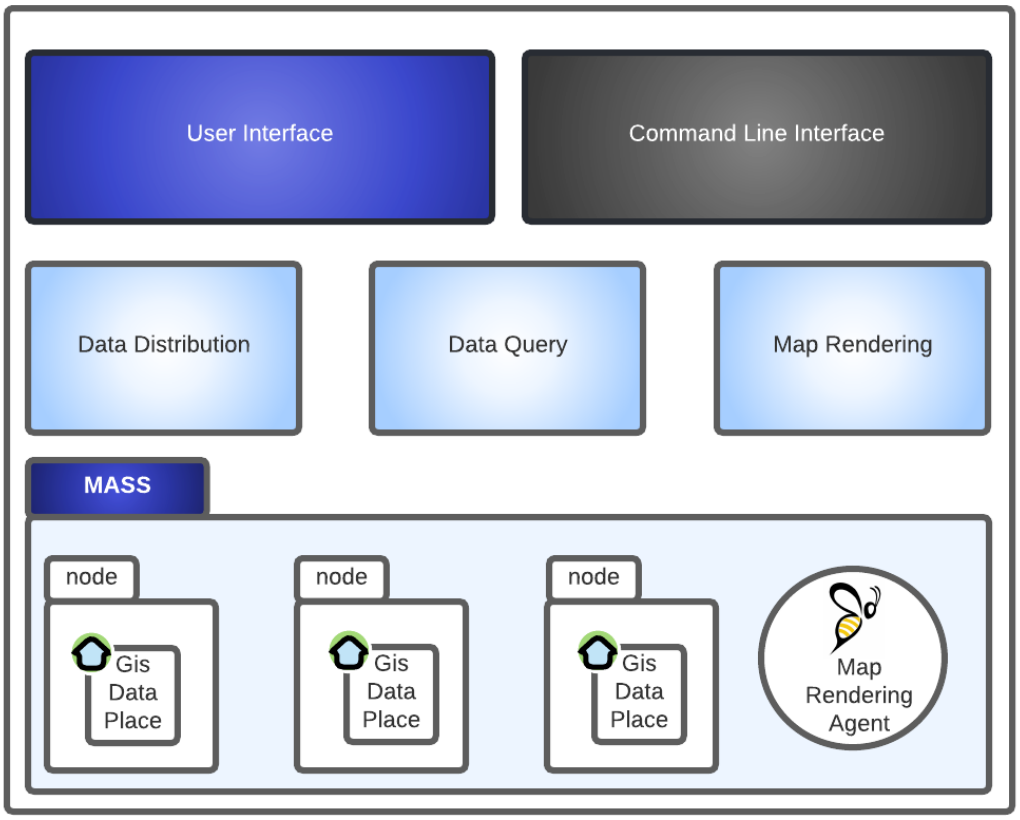

Figure 3 GIS MASS Database Design

3.1 System Implementation

This implementation illustrates the processing of the data in a shape file by either querying the raw data or rendering a map with vector GIS data superimposed over a base map raster image (Figure 4). It includes a user interface that allows users to add to and remove features (layers) from the map, zoom in and out of the map as well as pan across the map. The implementation also has the ability to query and sort feature data, then displaying the results in a table.

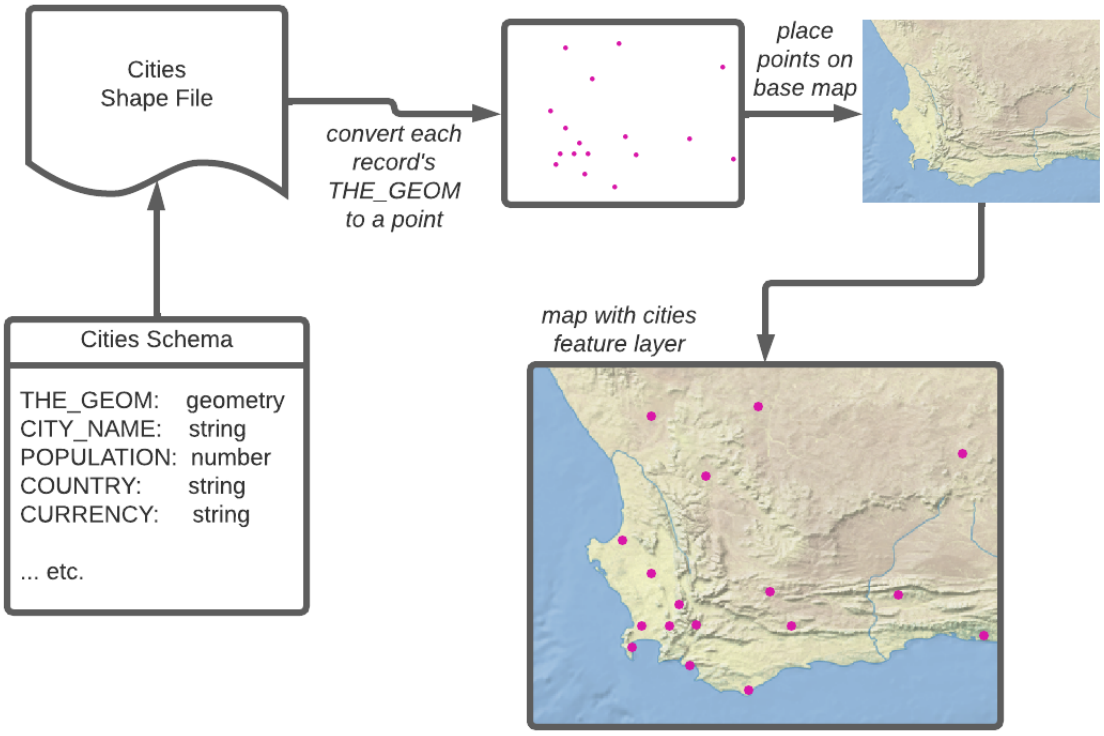

Figure 4 Rendering a Feature on a Base Map

3.1.1 Distributing GIS Data

All GIS vector data is tied to geometry, such as a location on a map, the outline of a country or the position of a manhole cover. This data needs to be broken up into tiles to be distributed across MASS Places. MASS Places is a distributed array that is distributed evenly across computing nodes over a cluster system (see 3.2 Parallelisation). The data is broken up geographically, making it easier to locate for retrieval and processing. The data is thus indexed by

location. Shape files are used as they are an industry standard. This allows processing to be independent of any specific library or implementation.

The base map tiles are divided according to the same coordinates as the feature files, ensuring that related data is stored on the same places. Each place will thus contain a section of the base map as well as its features over the same coordinates.

The base map is given a width in coordinates, with the x-coordinate ranging from -180 to 180 and the y-coordinate from -90 to 90 (see Figure 5).

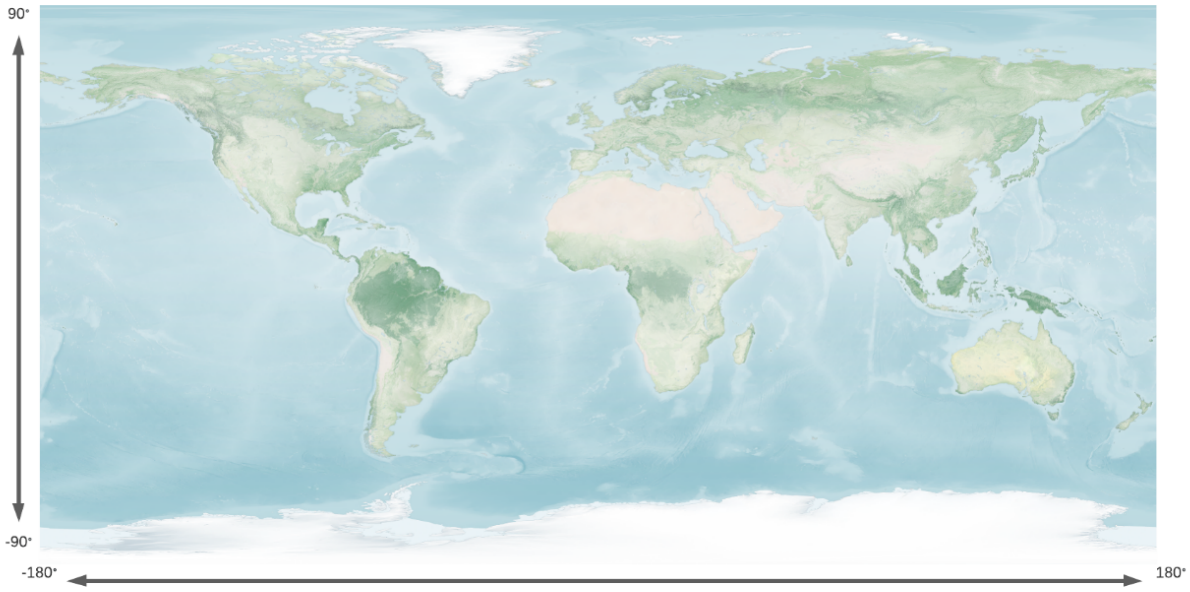

Figure 5 Map with axis [14]

Using the geometry of the vector and raster data as the key, the data is split up into different buckets and distributed to each place that represents a single geographic tile of the world. A simple example is to distribute the data across two places. All raster and vector data associated with latitude -90 to 90 and longitude -180 to 0 will be stored in the first place and all data associated with latitude -90 to 90 and longitude 0 to 180 will be stored in the second place. The simplified example in Figure 6 illustrates the division and storage of the tile files when the base image is divided into 12 tiles. There are two nodes, named cssmpi1h and cssmpi2h. These are the names of machines on the university network. In different implementations, it would be the names of the machines that are hosting the nodes. Each node contains six places. The 12 tiles are then each

stored in a place, with each place containing only one tile. The shape file feature data is divided up in a similar way using geography bounding boxes. The number of nodes can be changed as well as the number of tiles and Places (see B.1 Setting up Nodes).

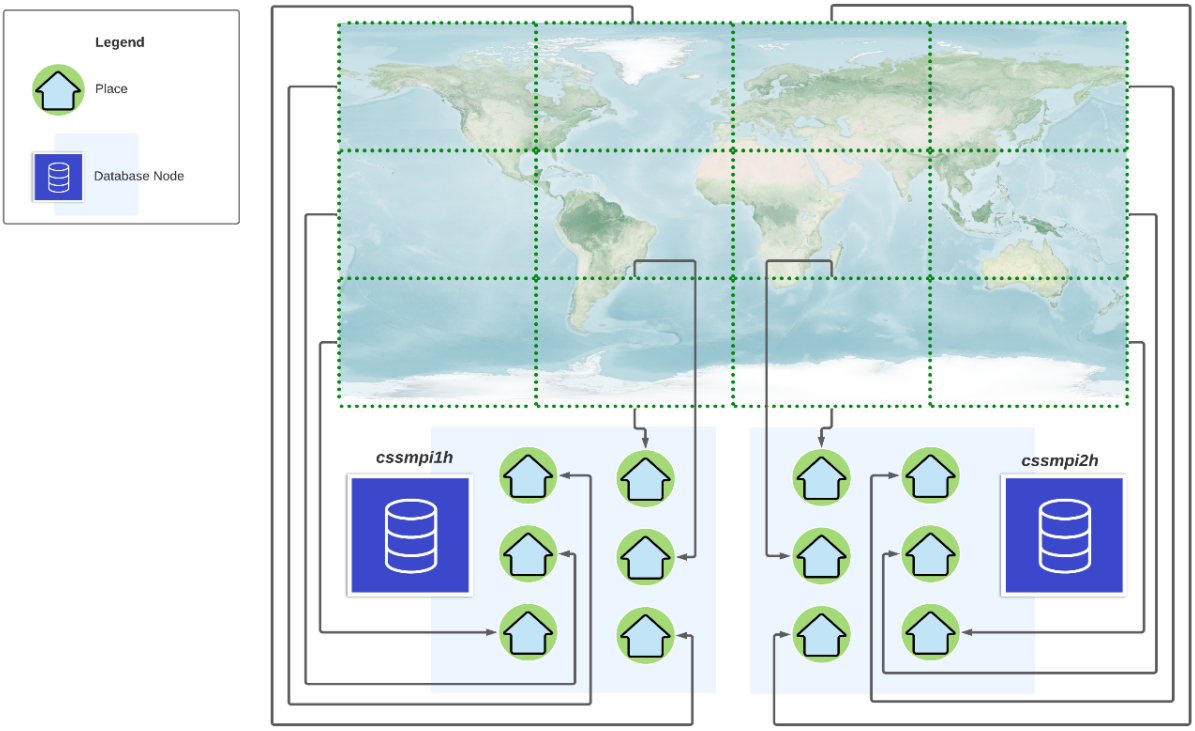

Figure 6 Saving each tile on a separate place

The main data distribution application is provided with an input folder that contains all the raster and vector data files that needs to be distributed to the Places. The application will iterate through the list of files in the input folder. The class UML for this feature application is shown in Figure 7, with the main function in `App`.

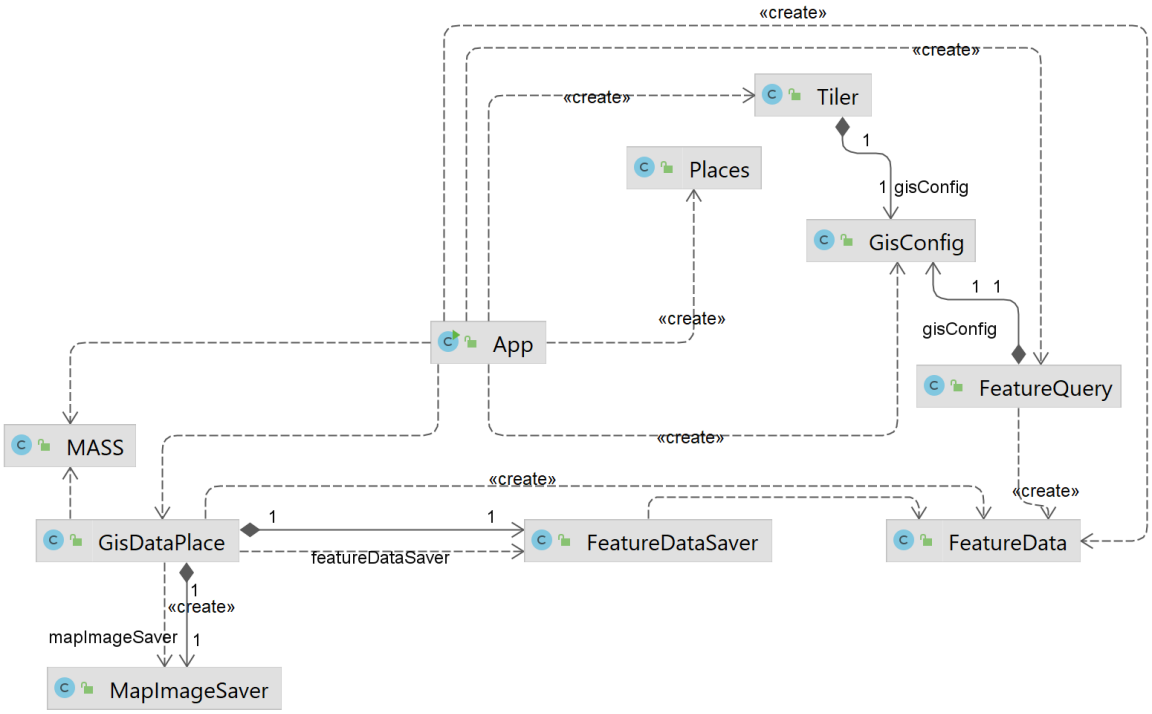

Figure 7 Data Distribution Application UML

If the current file being processed is a raster data file, the program will create a new instance of `Tiler` to produce the tiled raster data.

If the current file being processed is a vector data file, it will create a new instance of `FeatureQuery` to produce the tiled vector data. `GisDataPlace` is an extension of the MASS `Place` class. This implementation provides the ability to receive raster and vector data and store the data in an appropriate location (see Figure 8).

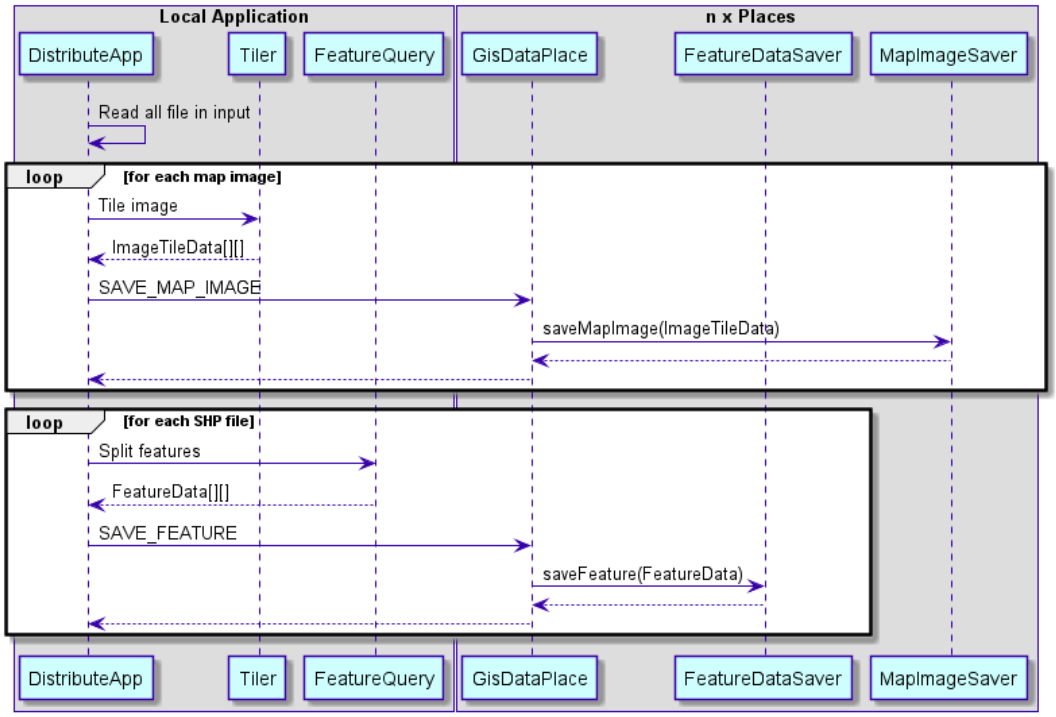

Figure 8 Sequence diagram for distributing data

Two different components have been developed to break up the data. The `Tiler` is responsible for splitting up raster data and the `FeatureQuery` is responsible for splitting up the vector data.

The `Tiler` starts by loading up the raster data (the base map). The raster data contains the image itself as well as the geographic information of the image expressed in the current Coordinate Reference System (CRS).

Definition The Coordinate Reference System (CRS) defines how three-dimensional spatial data, such as a real place on Earth, translates to two dimensions. It consists of a coordinate system, horizontal and vertical units, projection data and the datum [15]. Data needs to be converted to the same CRS to be analysed and processed.

The geographic information is used to break the image into smaller versions using the CRS (see Figure 9) while maintaining the geographic information for each tile. The axis needs to be reversed for the CRS, due to the quirk in geography that first has latitude, then longitude, versus mathematical models that first use the x axis, then the y axis [16].

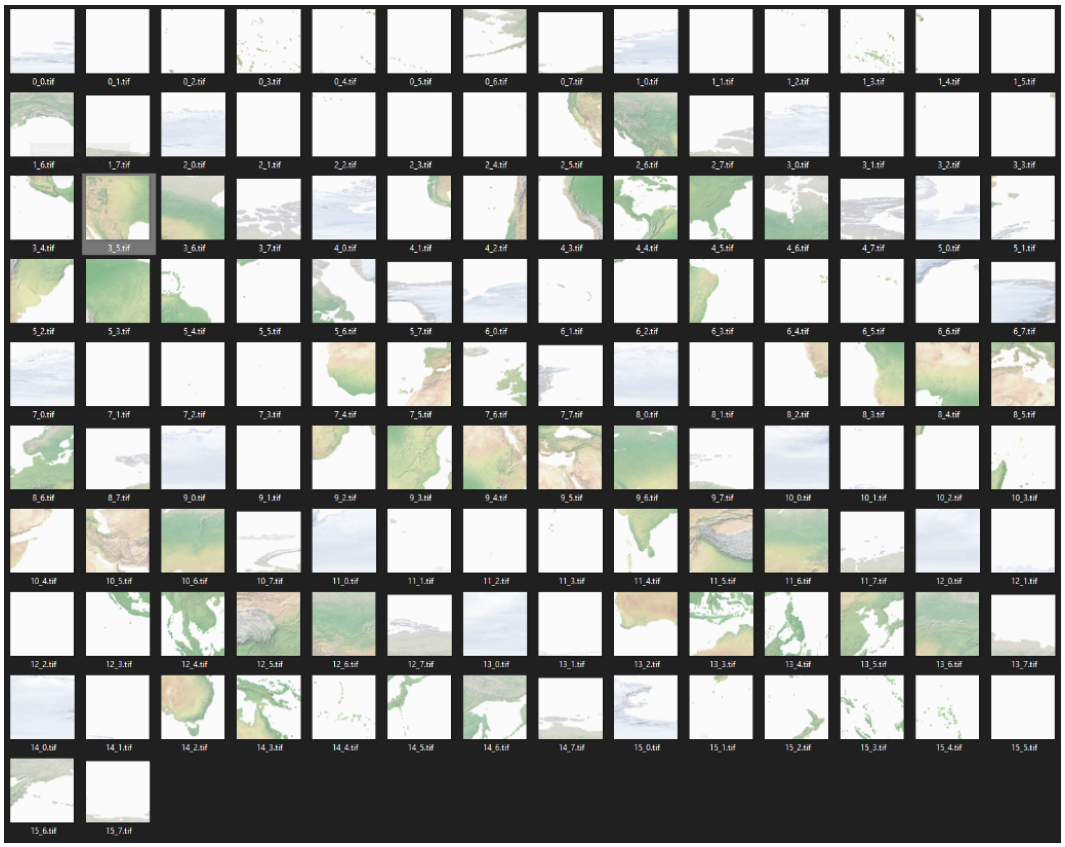

Figure 9 Map divided into tiles with numbered file names

This data will be distributed to each place using the MASS library, but it requires that the data be serialized to transmit over the network. Unfortunately, the *GeoTools* objects are not serializable, so a `GridCoverageWriter` is used to write the raster data for each tile to an array of bytes. This, along with the filename of raster data for each tile, is returned by the Tiler. Figure 10 shows the relationship between the Tiler and the *Geotools* API classes.

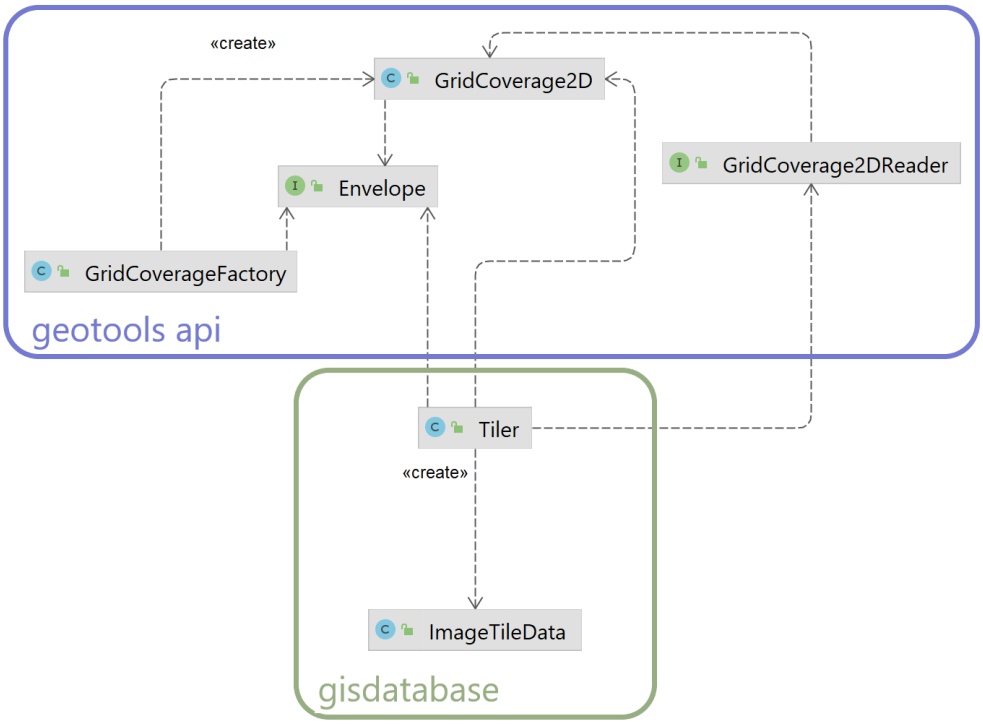

Figure 10 UML showing relationship between `Tiler` and `GridCoverage`

The `FeatureQuery` uses the *GeoTools* API to read each shape file in the input folder.

Definition Shape files are considered an industry standard for vector geospatial data files [17]. They contain the features of a geographic area. The `*.SHP`, `*.DBF` and `*.SHX` files are required with various optional additional files also possible.

Definition Features contain geospatial data, usually referred to as `the_geom`, that describe an entity using geometry to define the feature, for example with points, polygons, etc. Features can be anything from cities to manhole covers. The features contain a descriptive label as well.

The geometric bounds of each Place is determined by the number of rows and columns. The `FeatureQuery` filters the shape files to get all the records that fall within the bounds of each Place using a Bounding Box query.

Definition Filters use features to filter data in a map. It uses the common query language (CQL), part of the OGC Catalog specification, which functions similarly to SQL.

Definition A Bounding Box is a two-dimensional rectangle that is defined by minimum and maximum coordinates in the x and y directions on a map [18].

The *GeoTools* feature records are also not directly serializable, but unlike in the case of the raster data *GeoTools* does not have an implementation that can write the shape file data directly to an array of bytes. To work around the problem, the feature records for each tile is written to a staging shape file along with any sidecar files.

Definition Sidecar files contain the georeferencing data for the image in a separate file.

The results of this process will be a shape file and other sidecar files in a staging folder that represents the feature records associated with each Place.

The feature files are divided as shown in Figure 11.

Figure 11 Example of feature files divided into tiles

3.1.2 Querying the Database

The database requires the functionality to read the data that is distributed over multiple `Places` running on a set of nodes as a single system. The *GeoTools* library exposes a set of interfaces that can be implemented to plug new data stores into the library.

GeoTools uses the existing `DataStore` API (see Figure 12) to represent a file, database or service that contains the spatial data. The interface represents a physical source of feature data. Its implementation allows *GeoTools* to support many geographical data formats.

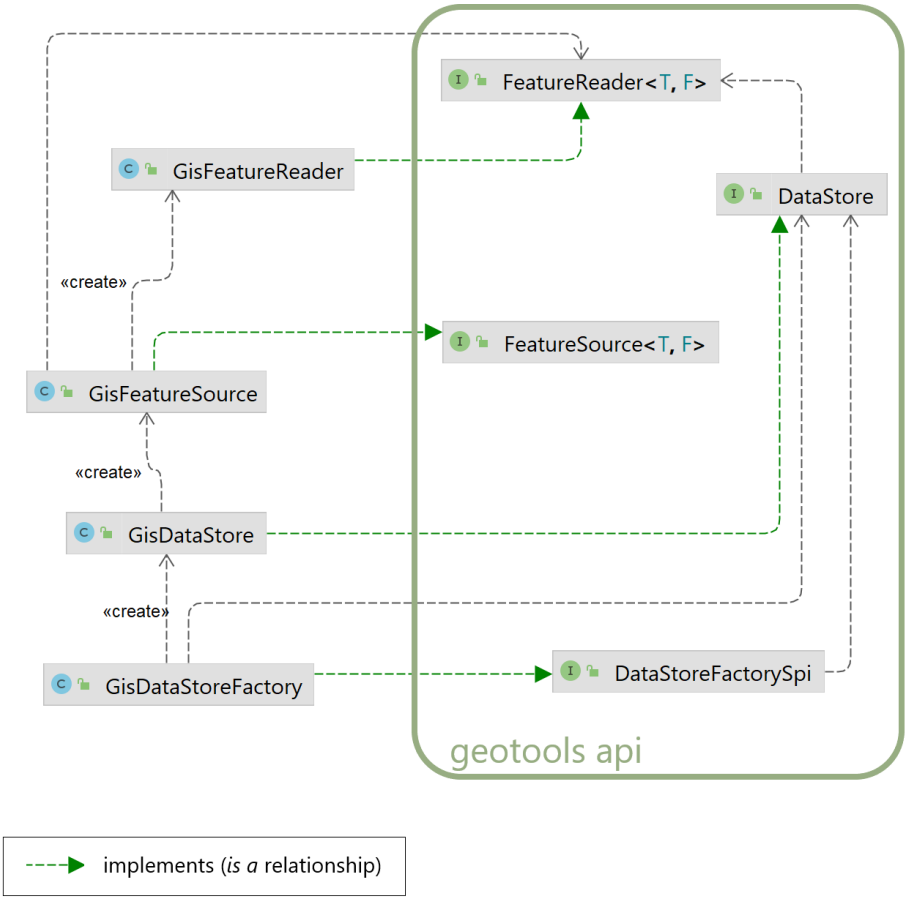

Figure 12 DataStore UML

The `GisDataStoreFactory` is responsible for creating new instances of `GisDataStore`. By creating a Factory plugin as an interface, customised content is integrated with *GeoTools*. This serves as pluggable point for implementing new data sources.

The `GisDataStore` is responsible for providing a list of the features that are supported as well as creating instances of `GisFeatureSource` for a selected feature.

It is assumed that the datastore will run on the same node with at least one Place, so a list of supported features is created by looking at all the shape files that have been distributed to the local node, extracting the name of the feature from the filename and returning that to the calling code.

The `GisFeatureSource` is responsible for providing the schema of the feature as well as creating instances of `GisFeatureReader` based on a provided `Query` object. It will provide

the schema by using the name of the feature to find the first shape file that matches the feature name and reading the schema from the file.

The `GisFeatureReader` acts like an iterator by providing implementation for `next`, `hasNext` and `close`. This way the client code can iterate through the feature records and process the records as needed.

The design required a way to create multiple instances of `GisFeatureReader` in a thread-safe manner while also performing the read operation across multiple nodes on the network. This implies that there will be open readers running on the Places (see `FeatureRecordReader` in Figure 13) that need to maintain their state on each node. That state is synchronized with the state of the `GisFeatureReader` that is requesting feature records from all the Places.

This coordination is done as follows:

- A UUID is created to uniquely identify an instance of the `GisFeatureReader` (client-side). This UUID is the key to synchronizing the state of the `FeatureRecordReader` instances running in the Places with the `GisFeatureReader`. It will be used in all network calls to the Places from these instances of the `GisFeatureReader`.
- The `GisFeatureReader` makes an `OPEN_READER` call to `GisDataPlace` instance running in each Place. The `GisDataPlace` will make a call to the `FeatureRecordReaderFactory` to create a new instance of `FeatureRecordReader`. The factory will store the newly created instance in a map using the UUID provided in the arguments as the key. This allows the factory to retrieve the instance in future interactions. The `FeatureRecordReader` will open the appropriate shape file using the provided query.

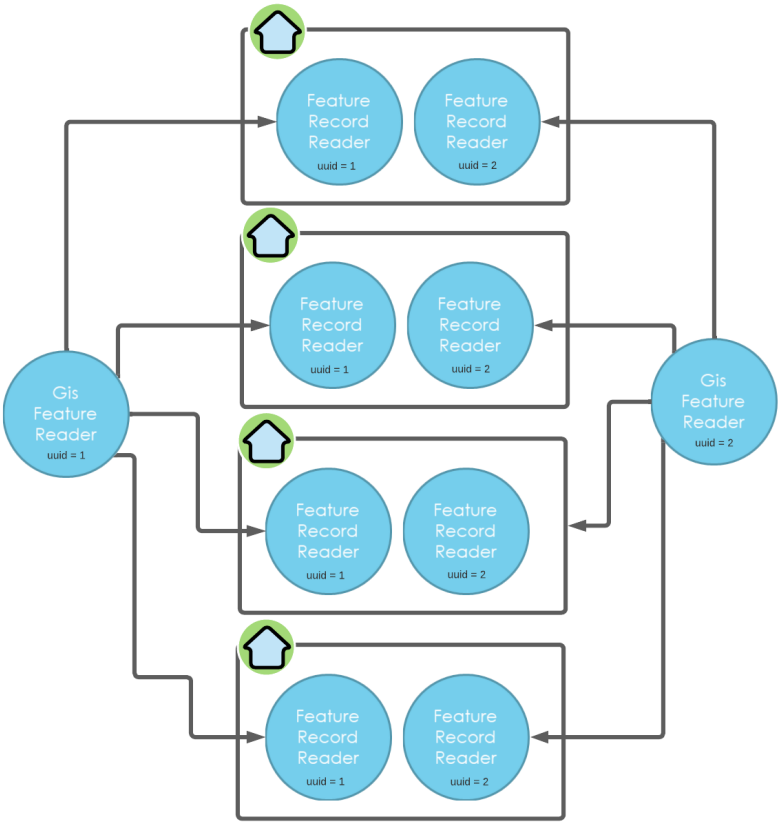

Figure 13 Multiple instances of Reader

- The `GisFeatureReader` next establishes a set of buffered records by making a `READ_NEXT_RECORD` call to each of the instances of `GisDataPlace` using the UUID established in the constructor as argument in the call.
- The `GisDataPlace` uses UUID to retrieve the appropriate `FeatureRecordReader`. It reads a set of records and returns them to the `GisFeatureReader`.

The `GisFeatureReader` will now have a set of buffered records and it will return records from the buffer to the calling code. If the buffer is empty, it will repeat these steps until no results are returned.

This is illustrated in the sequence diagram in Figure 14.

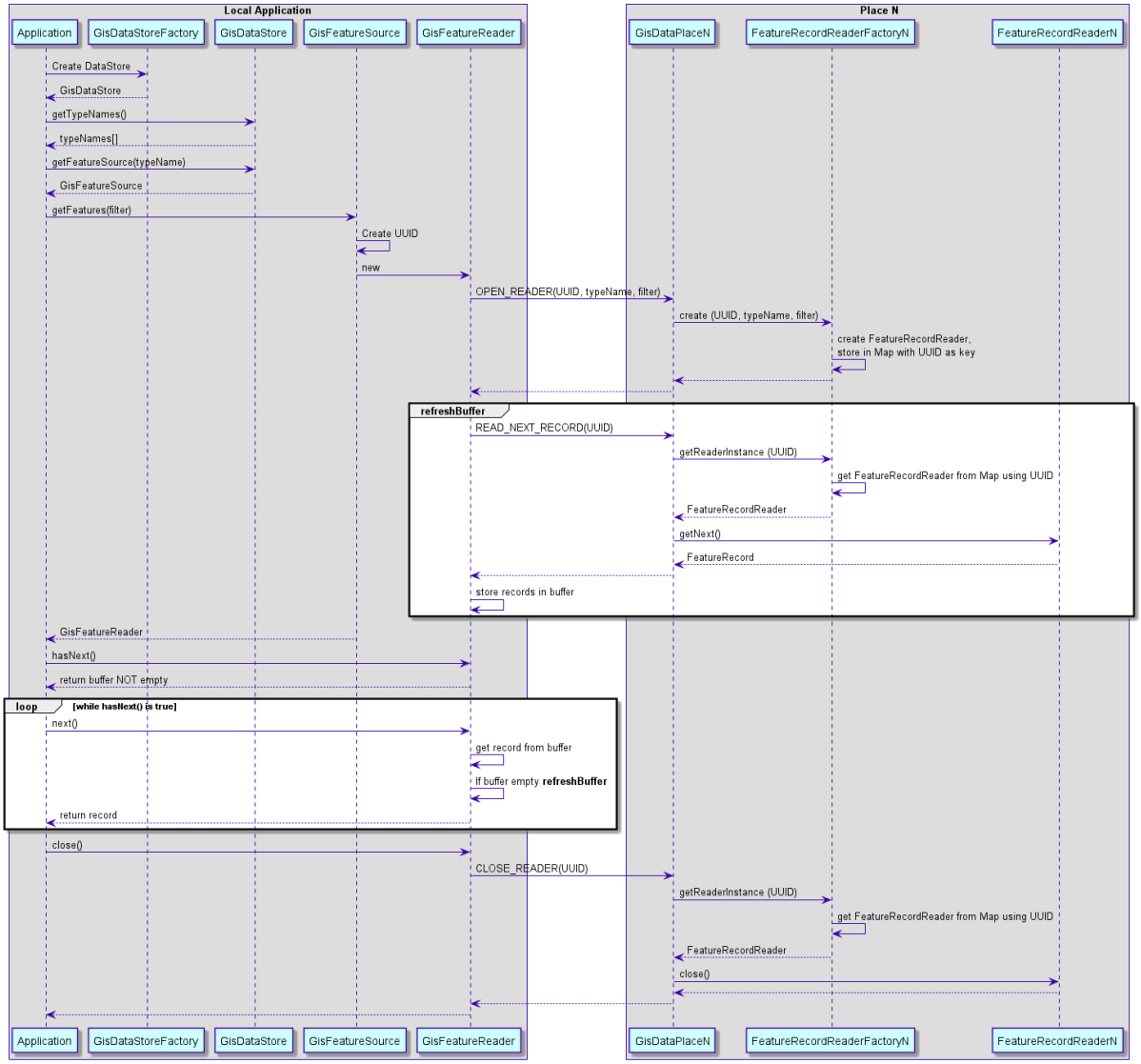

Figure 14 Sequence diagram for querying data

3.1.3 Agent based Rendering

Rendering vector-based features on a base map (raster data) is one of the main features of a GIS system. Rendering data is a data- and compute-intensive process, so the system makes use of agents to render fragments of the map in Places and send the fragments back to the main application where it is assembled into a complete map (Figure 15).

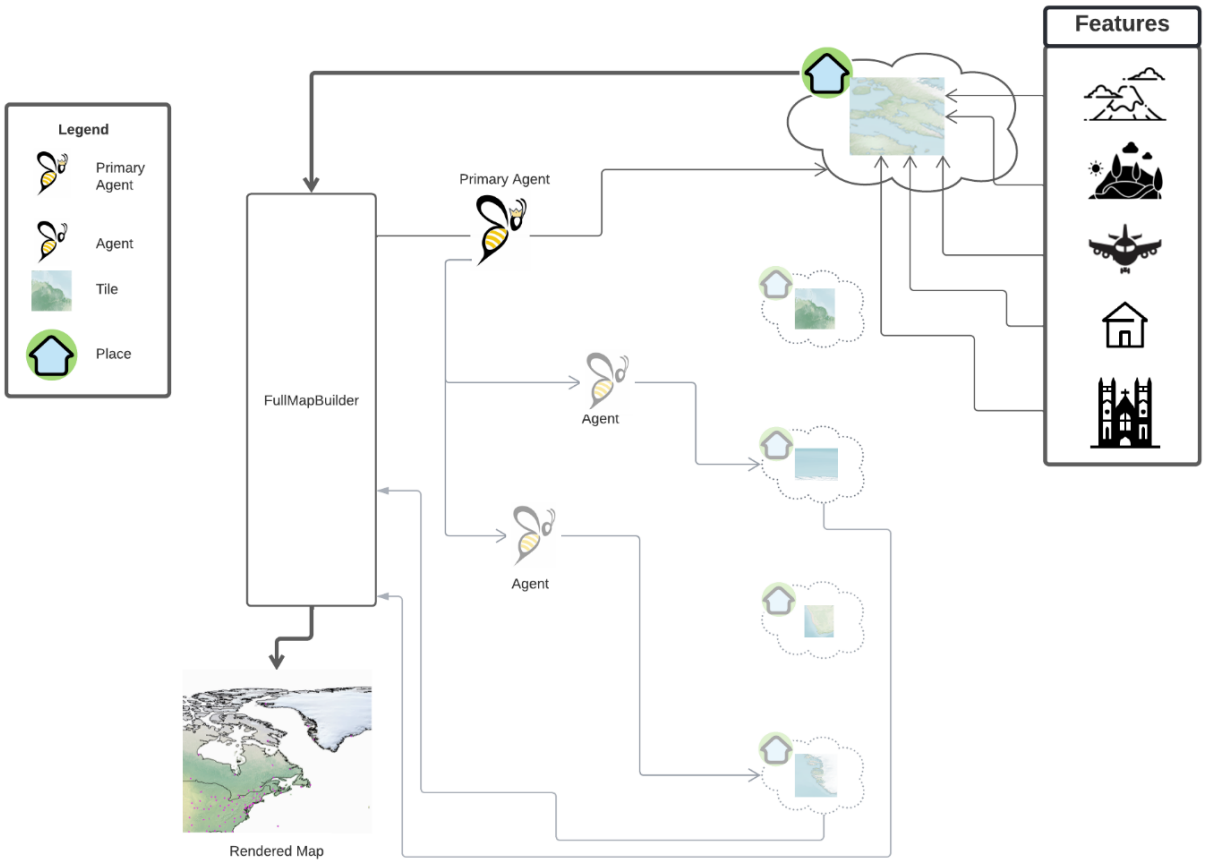

Figure 15 Lifecycle of primary agent: spawning agents and rendering map fragment

The application creates a single agent that is the primary agent from which all the worker agents are spawned.

The application establishes a utility class (`WorldGrid`) that provides the implementation with the latitude and longitude range for each of the tiles in the system. This is used to help set up the arguments that will be used in the calls to the agents to render the relevant fragments correctly.

The input parameters to the application includes the latitude and longitude bounds of the map image that should be rendered as well as the base map image and details of any features that should be rendered on top of the map. The application uses the bounds to identify the tiles that will be involved in rendering the map (Figure 16).

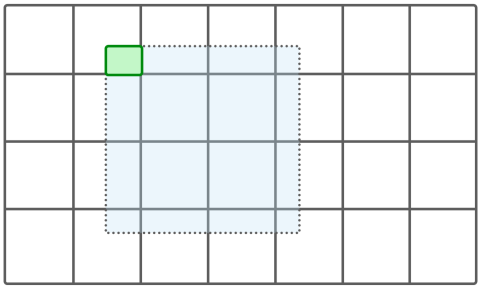

Figure 16 Finding the coordinates of a fragment

The application creates a two-dimensional array of arguments that represent the arguments that need to be sent to each of the agents once they have migrated to the correct places. The argument (`MapRenderingArgs`) encapsulates the base map image file name, the pixel width of the map image fragment to render, the latitude and longitude bounds of the fragment and a list of the features that need to be rendered on the map.

The application calculates the fragment coordinates in `FullMapBuilder`. These coordinates are passed as arguments to the agents (lines 37 – 61 in A.2.10 FullMapBuilder.java). Figure 17 shows the relationship between these classes in the UML diagram.

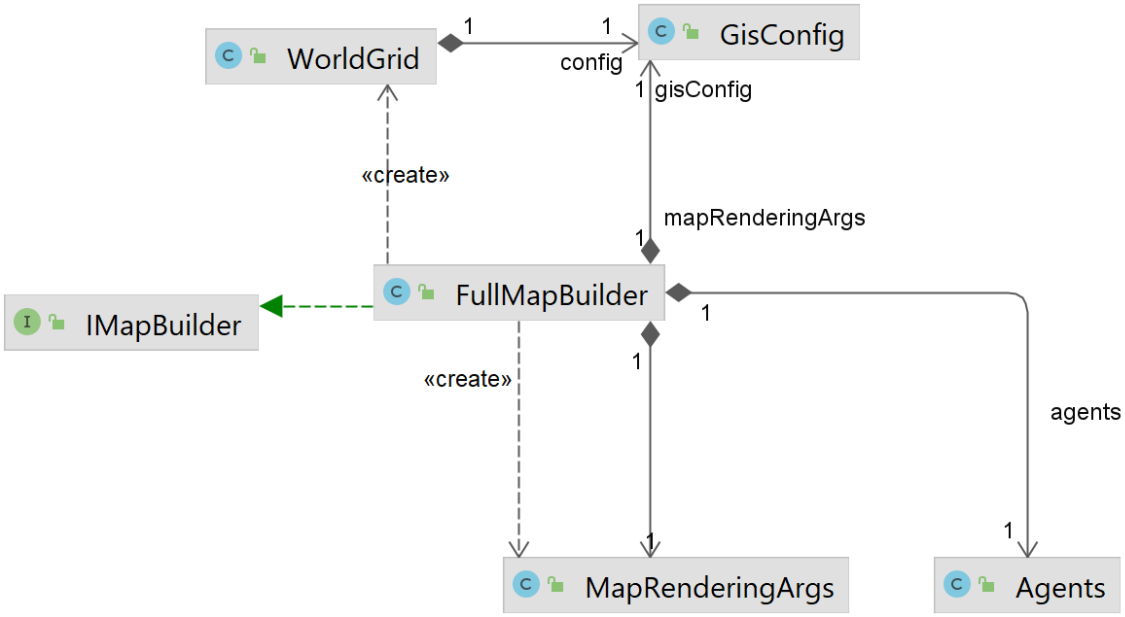

Figure 17 UML for classes for map rendering

The application makes a `SPAWN` call to the primary `RenderMapAgent` to spawn N-1 new agents (see Figure 18). The primary agent will also be creating a fragment. The provided input arguments are passed for each newly spawned agent in its constructor, where N is the number of tiles identified as taking part in the process.

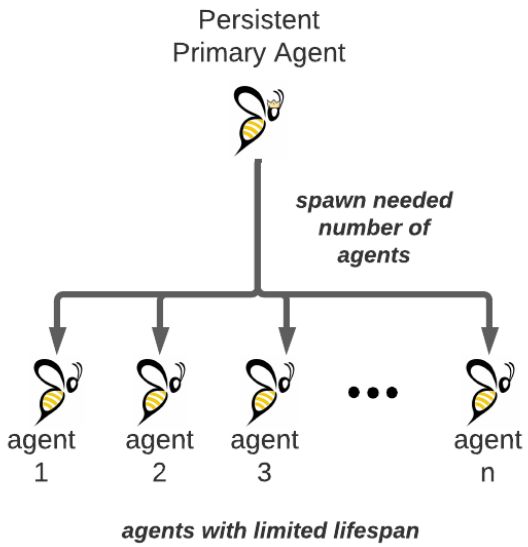

Figure 18 Primary agent will spawn other agents that have a task-limited lifespan.

The application makes a `MIGRATE` call to all agents. The input argument provided during the construction of each agent includes the indices of the Place that they need to migrate to.

The application will make a `RENDER` call to all agents. Each agent will proceed to do the following (Figure 19):

- Create a new `MapFragmentBuilder`.
- Render the selected vector data on top of the base image map fragment.
- Return the results of the map rendering process as an array of bytes.
- Terminate itself if it is not the primary agent.

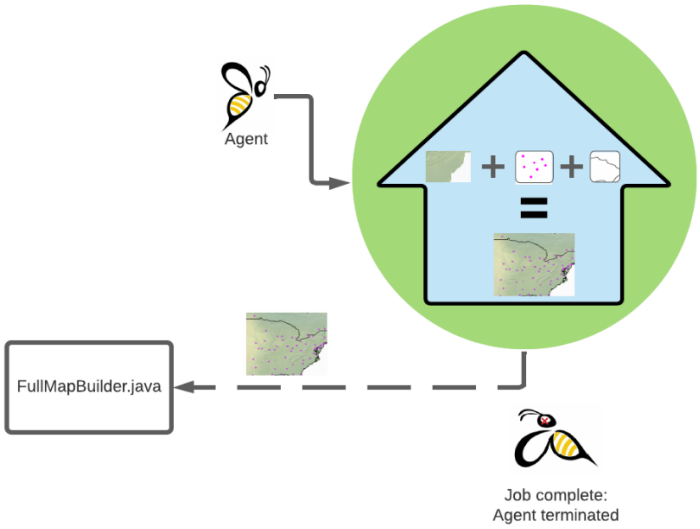

Figure 19 Drone Agent Lifecycle

Once the application has the various map fragments it can construct a complete image by stitching the map fragments together.

Figure 20 is an example of a map fragment returned by the agents that includes country borders and cities as the features.

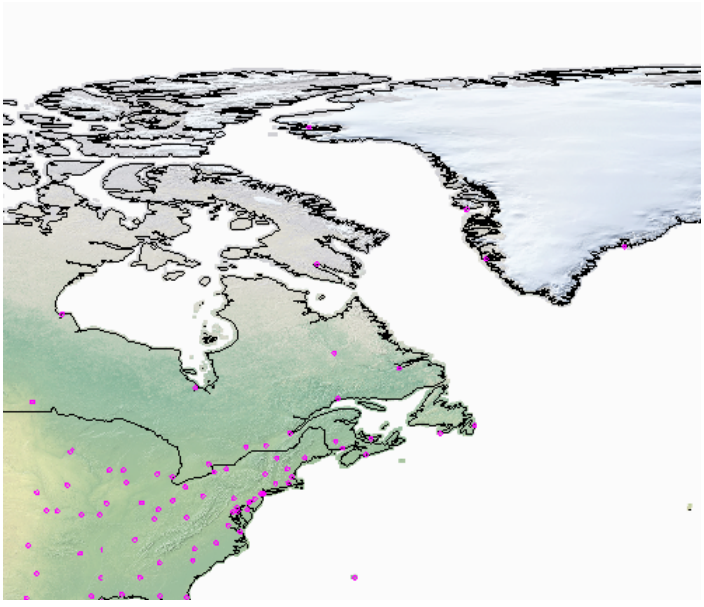

Figure 20 Section of map with country borders (black lines) and cities (pink dots)

This process is visualised in the sequence diagram in Figure 21.

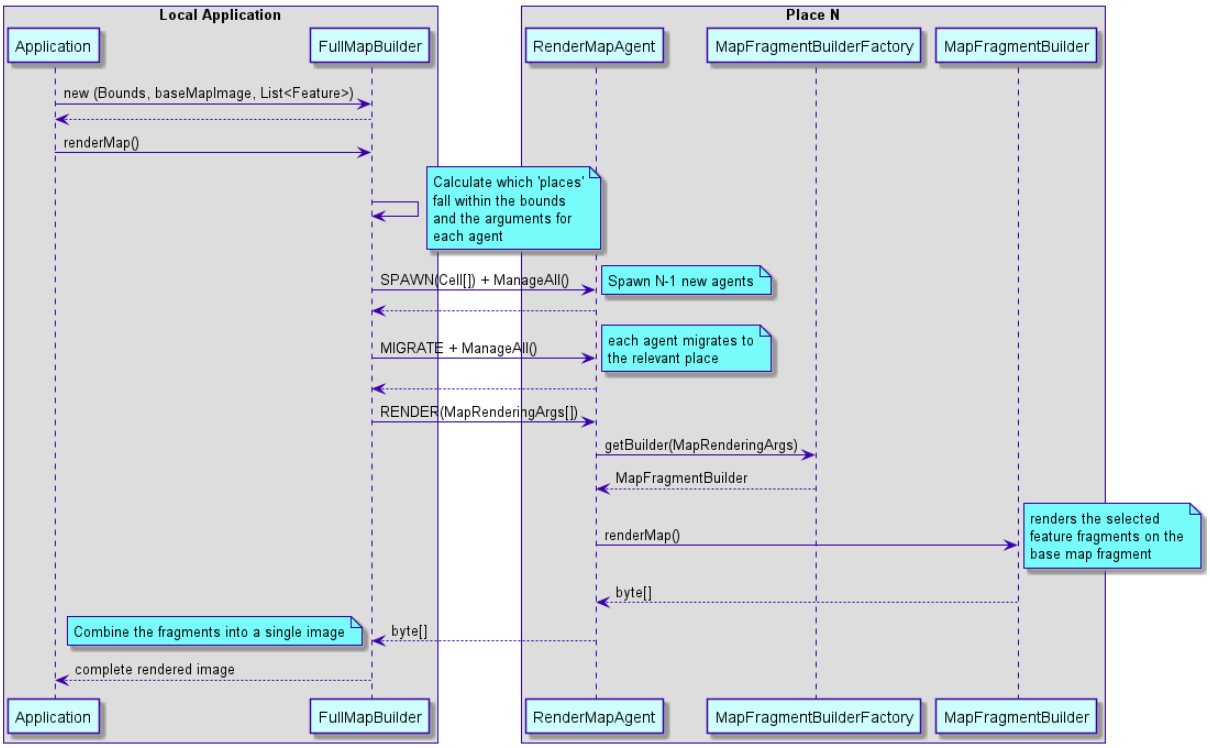

Figure 21 Sequence diagram for rendering

3.2 PARALLELISATION

The Multi-Agent Spatial Simulation (MASS) performs the parallelisation functions using the modelling objects Places and Agents (see Figure 22) [19].

Threads are spawned when MASS is initialised. The number of threads per computing node are defined when the program is run. The multi-threaded processes are managed with the message passing technique [19].

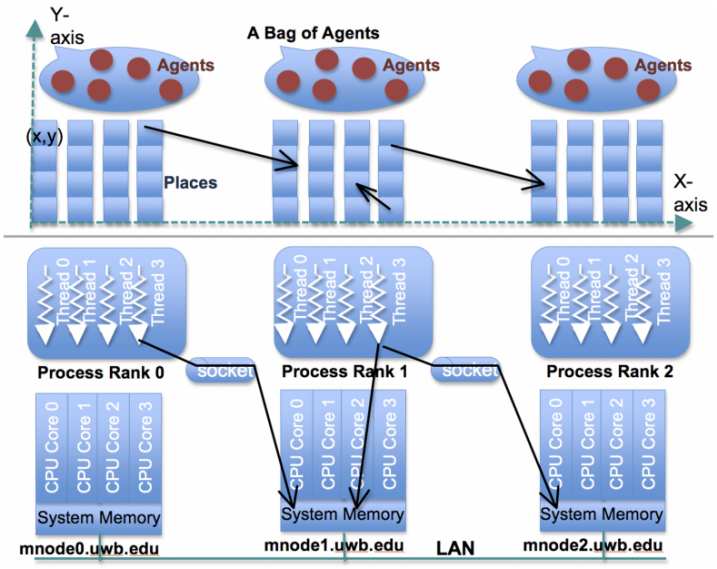

Figure 22 MASS Programming Model [20]

Data is saved to dedicated Places, so Agents can retrieve and analyse the data on demand [20].

3.2.1 Places

Places are distributed evenly over computing nodes in a cluster as a two-dimensional array of elements. In this implementation, a Place represents a section of the raster and vector data sets. The queries on the data are parallelised by distributing the data sets evenly across the Places.

3.2.2 Agents

The map rendering arguments (`MapRenderingArgs`) in this application-specific Agent are serialised upon migration [21]. Each agent is allocated to the processing of data of a single Place. The Agents migrate to Places. They can spawn new agents and control their own behaviour. Parallelisation of data processing is achieved by migrating Agents to the Places where the data is located. The compute-intensive processing is performed by the Agents on the Places in parallel. The results are returned to the original object that initiated the process.

3.3 User Interface

Multiple steps must be followed to generate a map with features or to query the data when running the components from the command line. Each time the pom file needs to be changed and

different parameters must be entered. If a mistake is made or the results retrieved are not adequate, then the steps all need to be repeated. Integrating all the components and functionality into one user interface simplifies the process and should make it easier for a developer to add and test new components.

A graphic user interface (GUI) was developed with JavaFX and FXML to integrate all the functionality in one place. Each component has an associated controller class with a top-level controller that contains the frame in which the components will run.

This GUI is a prototype.

Definition JavaFX is an open-source platform to develop user interfaces in Java. It serves as an alternative to Swing. It uses FXML to define the user interface visual components.

Definition FXML is a markup language to define the look and feel of a user interface. It defines containers and objects as well as the size of the object. Program logic is thus kept separate from the design.

Example: The following line defines a label component in FXML.

```
<Label fx:id="queryLabel" layoutX="25.0" layoutY="80.0" text="Query" />
```

The `UiLauncher` class is a workaround for executing a Java application with JavaFX without using modules. The class that extends Application (`GisApplication`) cannot be directly called for execution in the pom.xml file, thus this class is used as the entry point and calls the user interface (Figure 23).

`GisApplication` sets up the environment before calling the first controller, `PrimaryUiController`, to show the user interface. From there, listeners perform the logic to call the related windows and required functionality.

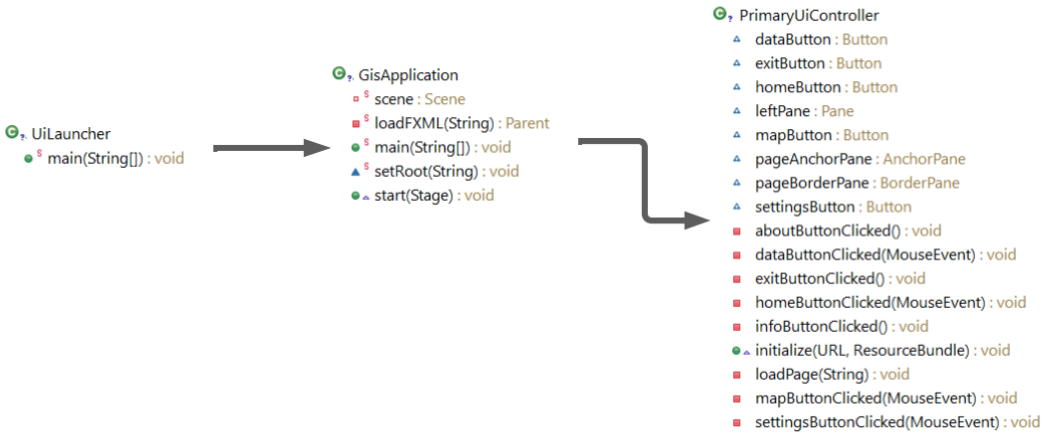

Figure 23 Start-up classes for the UI

The `SettingsController` sets the location of the base map and feature files (Figure 24). These values are persistent between instances of the program. It will also distribute the files across the nodes. Every time a new data location is set, the data will need to be redistributed. In this implementation, a base map is required. The base map needs to be in raster format.

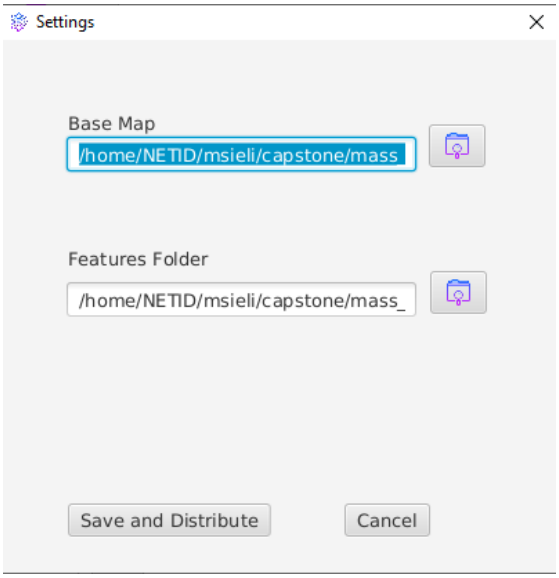

Figure 24 Settings Panel

`DataUiController` is called when the **Data** button is clicked (Figure 25). It contains logic for updating the user interface and rendering the results of the query.

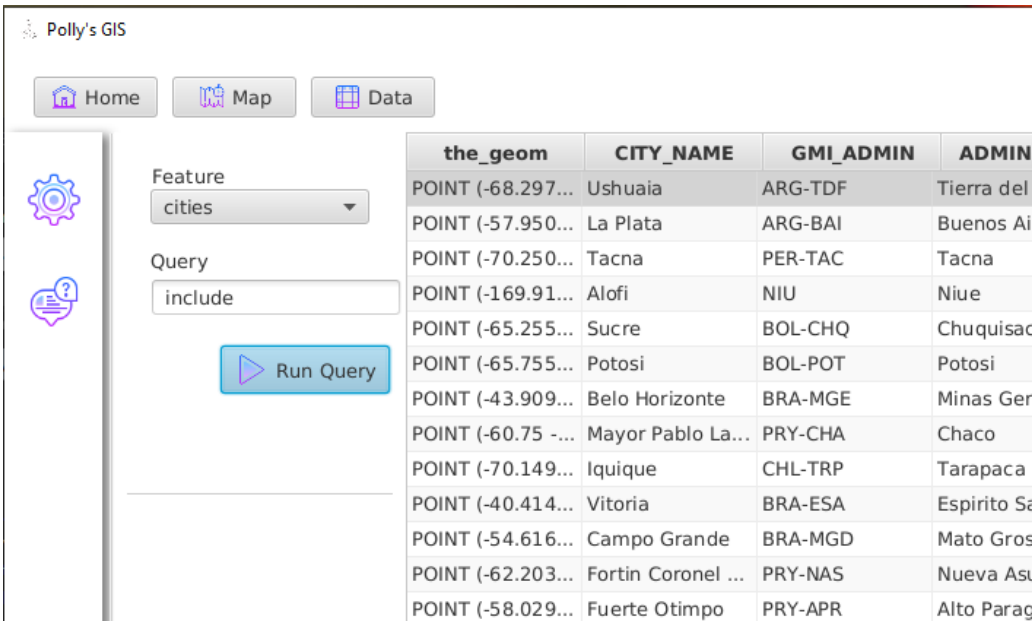

Figure 25 Data Query Panel with Cities Data

The `MapUiController` is called when the **Map** button is clicked. It contains logic for updating the user interface and renders the map with the class `MapFragmentBuilder` and the method `renderMap()`. The map is rerendered each time a feature is added or removed. The rerendering is done each time a change occurs on the map, whether the map gets resized or moved.

Panning moves the viewport of the map in the direction of the panning button that is clicked as shown in Figure 26. The map is rerendered each time to preserve the quality of the image. The panning buttons do not move the map if the full map is visible.

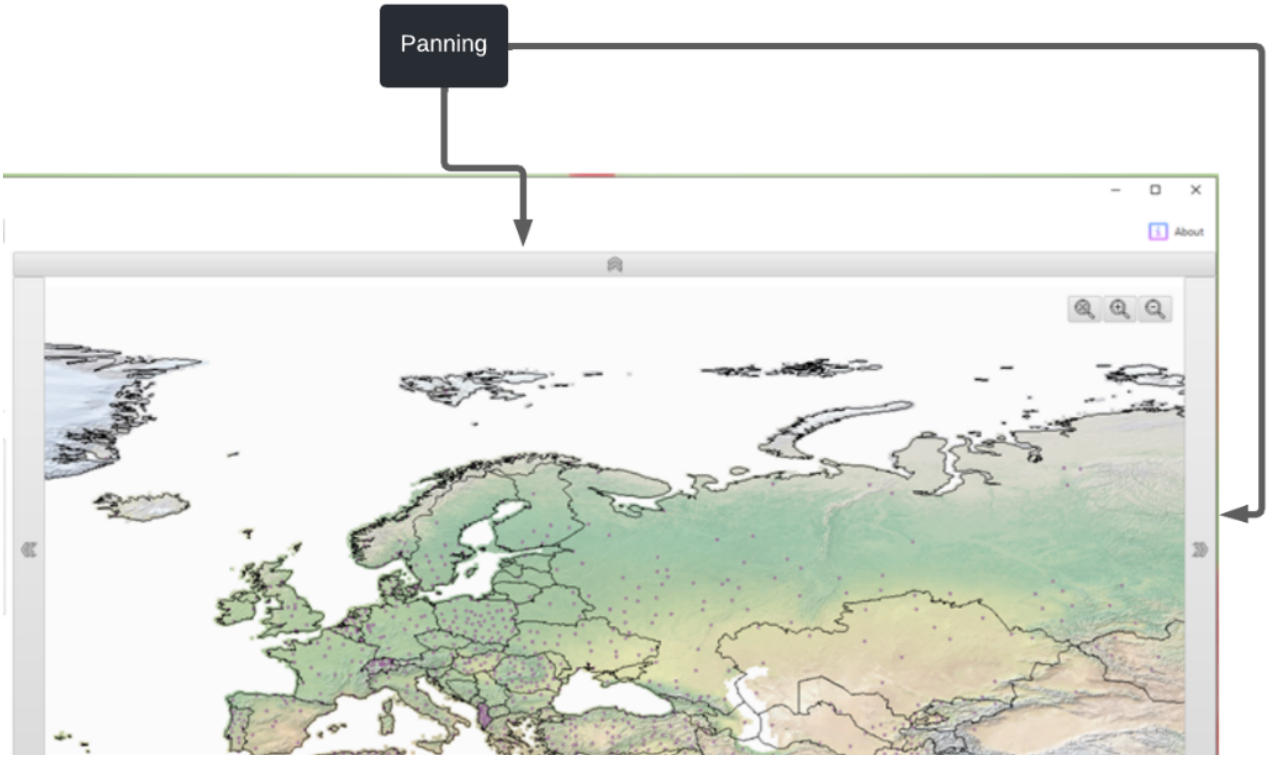

Figure 26 Panning Buttons

The user can zoom into the map with the appropriate magnifying glasses that are mapped to the functions `zoomReset()`, `zoomOut()` and `zoomIn()`. The function `zoomReset()` makes the full map visible. The function `zoomIn()` has a maximum level of zoom, while `zoomOut()` will not zoom out further once the full map is visible.

A change listener handles resizing of the window to redraw the map to fit the dimensions. The map is rerendered after the window is resized. There is a minimum size for the window. A timer is built in to delay resizing, to prevent continuous rerendering as the window is dragged bigger or smaller.

Chapter 4.
VERIFICATION

This chapter shows the benchmark results of both the data query and the map rendering modules. The benchmarking was completed using multiple thread combinations over multiple nodes, comparing the runtime of the modules.

4.1 EXECUTION ENVIRONMENT

The benchmarks were performed using up to eight computing nodes on AWS. AWS was set up using EC2 m5.xlarge instances that run on Ubuntu Linux. These instances use Intel Xeon Platinum 8175 3.1 GHz processors with 16 GB of memory and four virtual CPUs. The MASS library 1.4.0 is used during benchmarking along with Java 11 OpenJDK.

A maximum of eight nodes was used as AWS limits the number of on-demand virtual cores to 32 unless more are specifically requested. This would have accrued further costs to this project.

4.2 INPUT DATA SETS

The base map raster data for demonstration and benchmarking was retrieved from Natural Earth Data [22]. A very large vector data set is required for proper benchmarking of the program in map rendering and data querying. This was artificially generated as a shape file with *GeoTools* by creating a random vector set with a million records. The records in the file use the schema shown in Table 1.

Table 1 Benchmark Test File Schema

Name	Type	Notes
`the_geom`	Point	All shape files require this field. It can be a point, line or polygon.
`UUID`	String	Unique identifier
`number`	Integer	Randomly allocated value from 0 to 1000

4.3 Querying the Database Benchmark

The large artificial data set is distributed across 24 Places. For benchmarking, an arbitrary query is run to retrieve all records that have a numeric value greater than 990. This is the top 1% of the data. A small percentage of data was chosen to ensure that only a subset of records from each place is returned. This ensures that the benefit of having a distributed data set is realised. The time taken to iterate over all records is measured. The results are graphed in Figure 27.

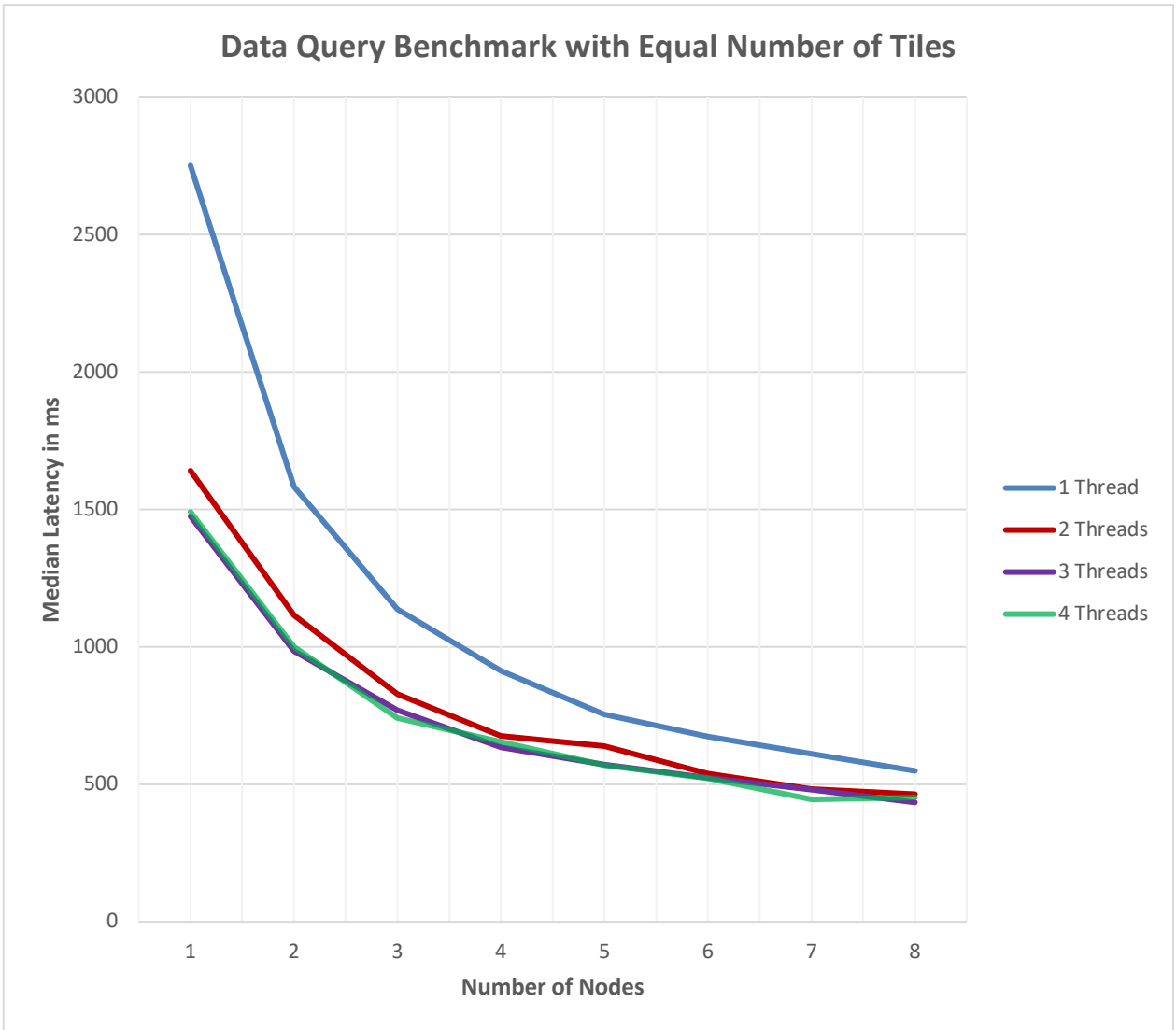

Figure 27 Data Query Benchmark Performance on AWS

The benchmarking was run with one to eight nodes, with each node run with one to four threads. A maximum of four threads per node were chosen as the AWS m5.xlarge EC2 instances only contain four virtual cores.

A second benchmark was run that kept the number of nodes and threads constant. It increased the number of places the data was distributed across. This was to determine if there is a benefit to using multiple Places per node, with only a single thread per node. The results with the trendline are graphed in Figure 28.

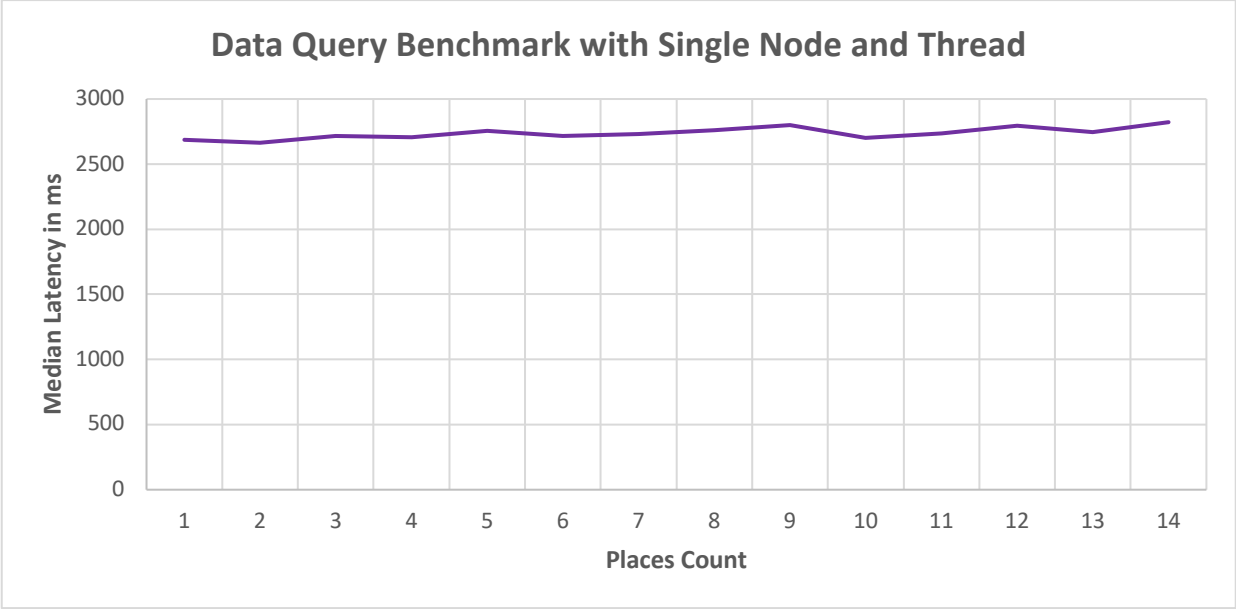

Figure 28 Data Query Benchmark with Single Node and Thread

There is no benefit, but rather a slow trend to increased latency with higher place counts.

Table 2 shows the results of the query run without aggregating the results. This data distribution is plotted in Figure 29.

Table 2 Data Query Benchmark Results with Single Node and Thread

Cells	1	2	3	4	5	6	7	8	9	10
1	2742	2701	2697	2714	2684	2686	2692	2715	2690	2705
2	2720	2701	2721	2703	2711	2702	2776	2693	2689	2683
3	2755	2738	2722	2704	2697	2720	2718	2725	2720	2704
4	2691	2676	2675	2673	2675	2653	2663	2679	2664	2666
5	2695	2649	2652	2651	2653	2653	2655	2725	2648	2649
6	2676	2656	2665	2656	2674	2679	2646	2662	2756	2657
7	2859	2885	2731	2722	2732	2740	2721	2724	2712	2710
8	2748	2885	2731	2678	2763	2687	2691	2701	2687	2685
9	2798	2687	2676	2963	2740	2729	2736	2744	2757	2743
10	2755	2743	2740	2690	2679	2689	2696	2675	2675	2715
11	2699	2773	2722	2673	2634	2651	2655	2661	2696	2632
12	2680	2650	2650	2657	2649	2653	2689	2673	2644	2641
13	2748	2651	2645	2722	2722	2749	2797	2744	2755	2703
14	2789	2725	2803	2746	2733	2721	2723	2726	2771	2727

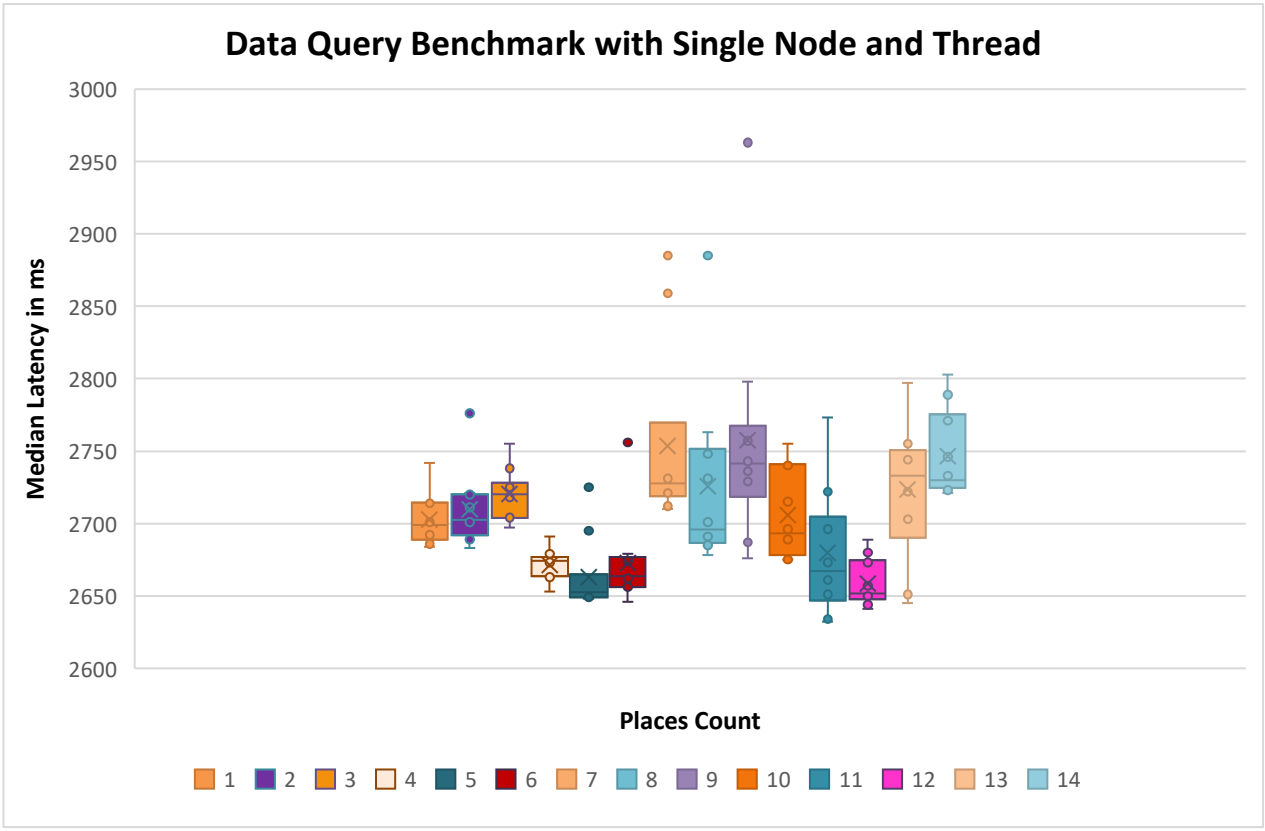

Figure 29 Data Distribution for Data Query with one Node and One Thread

4.4 AGENT-BASED RENDERING BENCHMARK

The large artificial data set as well as the raster data (map image) are distributed across 24 Places. Agent-based rendering is used to render the artificial vector data on top of the base map image. The time taken to render the full map with the artificial feature is measured. The results are graphed in Figure 30.

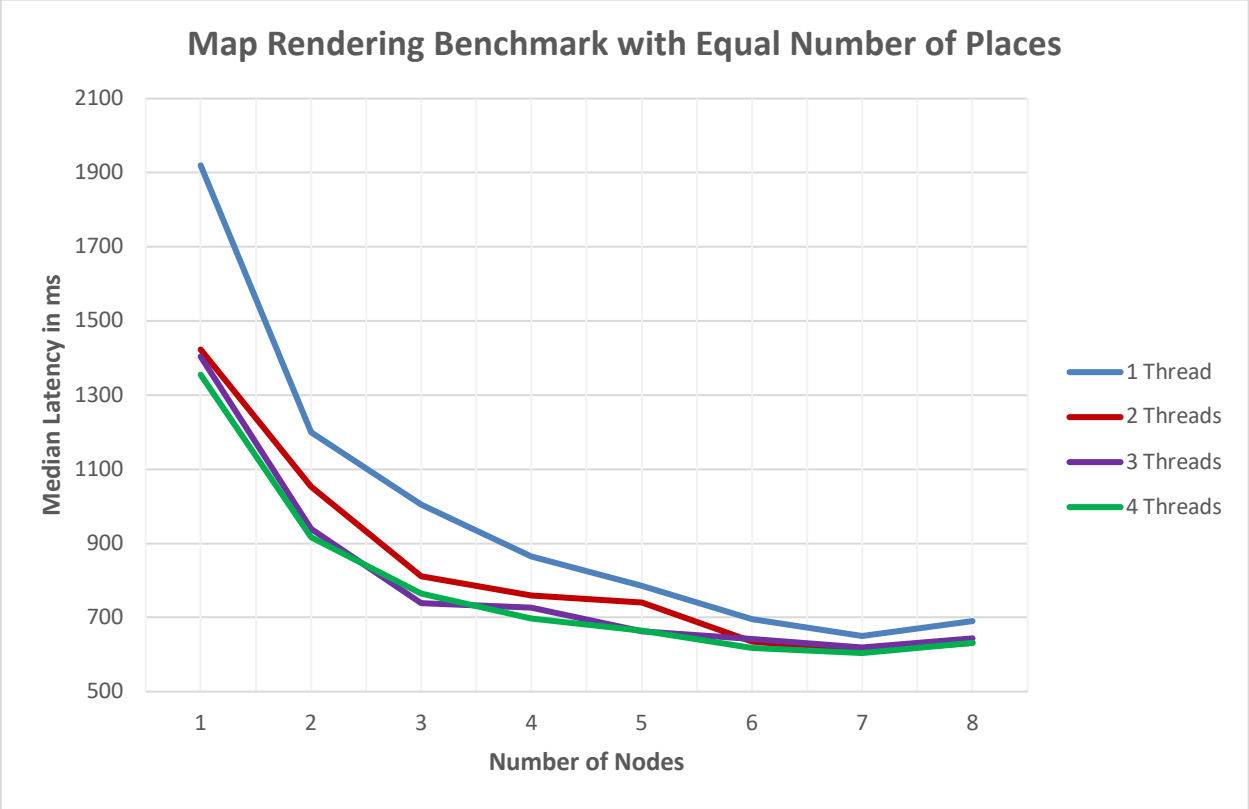

Figure 30 Map Rendering Benchmark Performance on AWS

The benchmarking was run with one to eight nodes, with each node run with one to four threads. A maximum of four threads per node were chosen as the AWS m5.xlarge EC2 instances only contain four virtual cores.

A second benchmark was run that kept the number of nodes and threads constant. It increased the number of places the data was distributed across. This was to determine if there is a benefit to using multiple Places per node, with only a single thread per node. The results are graphed in Figure 31.

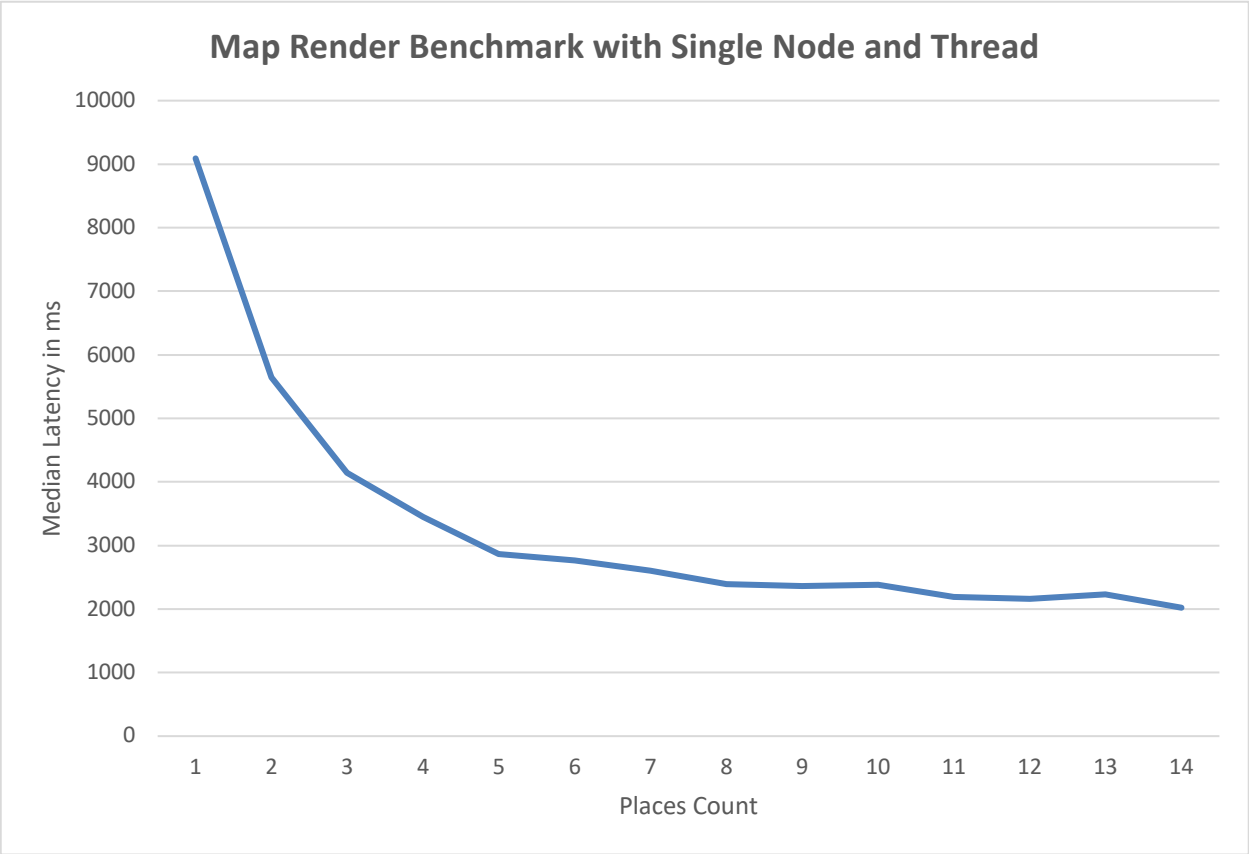

Figure 31 Map Rendering Benchmark with a Single Node and Thread

Map rendering shows a significant improvement in performance with the increase in Places. This improvement shows that the *GeoTools* API is more efficient when rendering small pieces of the map and stitching them together, than when trying to process one large map.

A benchmark test with a fixed number of Places, nodes and threads was done to measure the performance with different row and column combinations that add up to the total number of 12 Places. The combinations are shown in Table 3.

Table 3 Map Rendering Benchmark Results

Node Count	Thread Count	Row Count	Column Count	Median Latency(ms)
6	2	1	12	753
6	2	2	6	916
6	2	3	4	1107
6	2	4	3	1148
6	2	6	2	1055
6	2	12	1	1118

Figure 32 displays the results of the table, showing the large difference between minimal rows with maximum columns and other combinations of rows and columns. Each node has two threads, thus creating a total of 12 places.

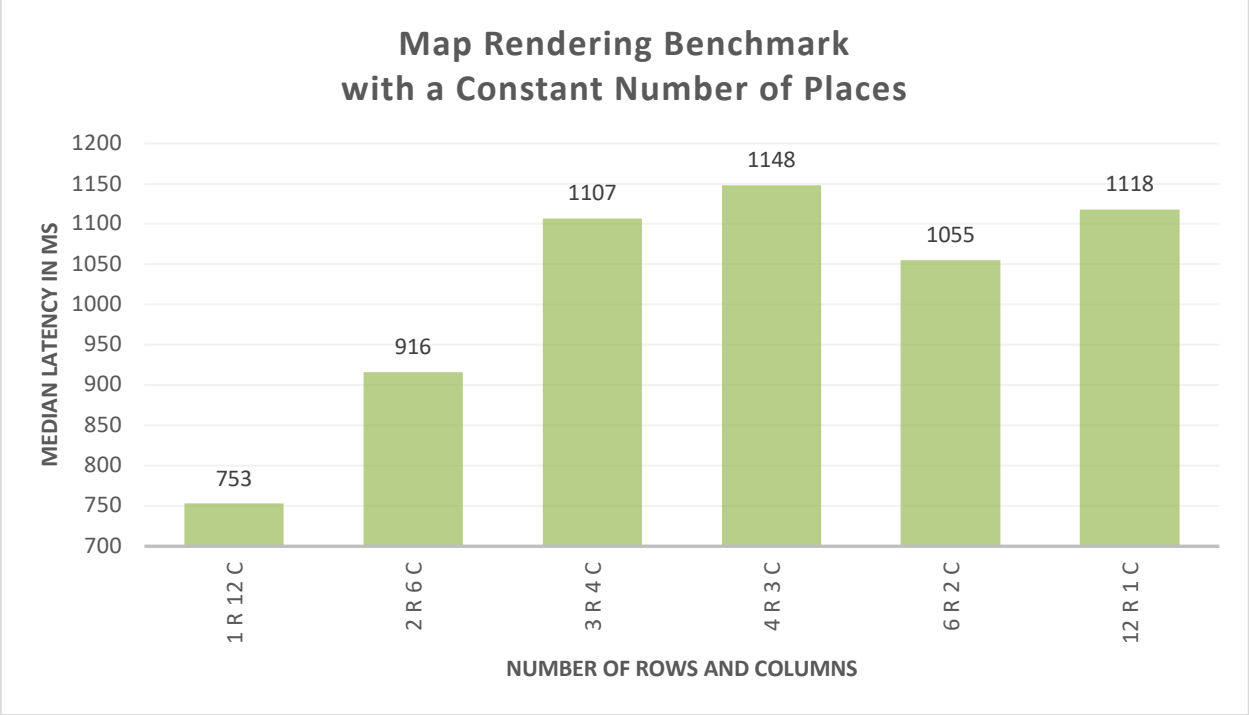

Figure 32 Map Rendering Benchmark with a Constant Number of Places

Different row and column combinations for dividing the image into tiles affect the performance of map rendering, with minimal rows proving to be the optimum for performance.

4.5 USABILITY

Distributing the data, querying the data and displaying the map with features can be accomplished with command line calls. This is however limiting as a separate command needs to be run for each functionality. Displaying a different section of the map or adding new features requires the command line call to be run again with new arguments. The user interface makes it clear what functionality is available to the user and allows the user to make changes to settings with ease.

The settings panel is a simple way for a user to select the relevant data through file choosers. The selected data set is distributed from the user interface in the settings. The base map can be selected in the settings as well, allowing for different types and styles of base maps.

The map section in the user interface allows visual variety, since the map can be resized, zoomed into or zoomed out of. It is also possible to pan around the map. This functionality is not available from the command line. It makes the user interface more efficient and simpler to use than the command line.

Dropdowns are an easy, efficient way to show the user which features are available to add to the base map (Figure 33). The features names are pulled from the data set that is linked in the settings, thus only offering the available features.

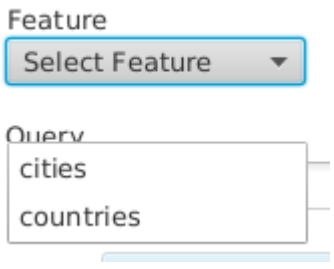

Figure 33 Dropdown for Selecting Feature

A legend is created from features added to the map. It allows the user to easily reference the features and their colours (Figure 34). Features can easily be removed as well, either individually or completely cleared.

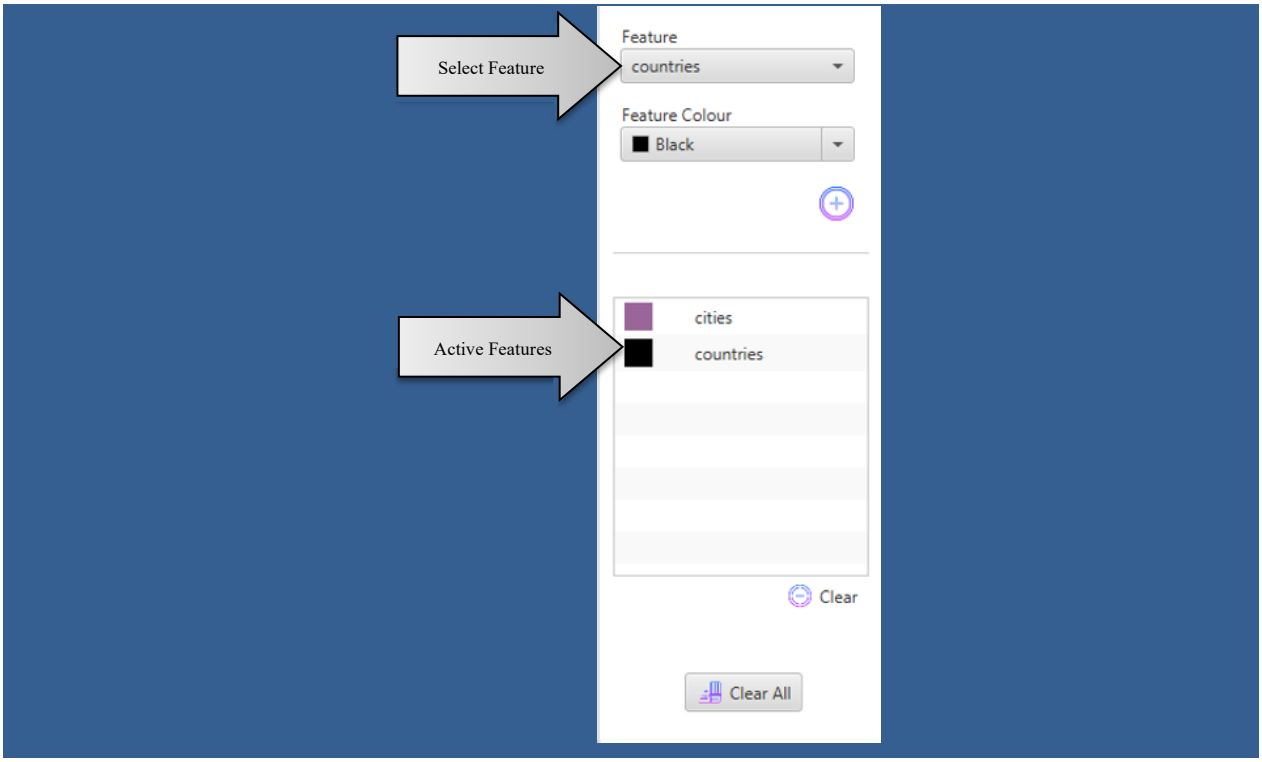

Figure 34 Adding Features to Map

As with the map functionality, the features names that can be queried, are pulled from the data set that is linked in the settings, thus only offering the available features. Subsets of data can easily be displayed with the use of queries. The query functionality for the data visually represents the resulting data set that the user can manipulate through sorting the columns.

4.6 SUMMARY

Benchmarking shows that there is significant performance improvement when running over multiple nodes, dividing the data across multiple places and by choosing a suitable tile division of the map for rendering. This illustrates the scalability of this system compared to running this GIS on a single machine. All benchmarks show that there is a flattening of improvement at between seven and eight nodes. As the number of agents increase, there is a latency increase for performing actions such as spawn and migrate. This is where the overhead of MASS becomes higher than the

benefit of multiple nodes. This point will move to higher node counts for a much larger dataset. A limit to the number of nodes has not yet been reached.

Shape files have a 2 GB inherit limit to their individual file size as they follow the dBase file standard [23]. The current implementation splits up shape files, so the same limit applies. Implementing a writable data store will allow the system to move beyond this limitation by adding more places with each place having its own 2 GB limit. It would thus add further scalability.

The `FullMapBuilder` and the `GisDataStore` abstract away that using MASS is making calls to multiple nodes. The callers of these classes only see a single system. This, along with the user interface create the impression that the user is working with a single system image (SSI) while hiding the implementation across a distributed system.

The graphic user interface improves usability by allowing the user to pick pre-determined options from an intuitive graphic interface. It helps minimise the potential for errors through visual cues and by using elements such as drop boxes. The user can only pick available features from the drop boxes since the features are taken from the available files. The user is prevented from trying to add a feature that does not have a corresponding data file. The user can see exactly which features are being used and can easily make changes to pick another feature or to remove features. The user also has more options for navigating through the data and around the map. This makes the user interface less restrictive than generating maps from the command line.

Chapter 5.
CONCLUSION

5.1 SUMMARY

This project serves as a proof of concept that a Geographic Information System (GIS) database can be implemented as a MASS application with a notable improvement in performance and scalability. The GIS would present as a single system image with a graphic user interface that is scalable over multiple nodes with large datasets. The simple user interface would improve usability, allowing the user to select features and functionality with point and click rather than having to repeatedly run the commands from a command line.

- A working basic GIS system was created that uses MASS for data distribution. It performs as a single system image across a distributed system.
- An interface hides the implementation with MASS from the GIS.
- MASS Agents render parts of a map, with each agent executing a piece of work in a place on a node, sending the resulting fragment back to be combined into the completed image.
- Improved performance when using more computing nodes is shown in various benchmarking tests.
- Usability is improved through the use of a simple graphic user interface prototype.

5.2 FUTURE DEVELOPMENT

Since the project serves as a proof of concept, there is a lot of scope in future development to build a fully-fledged GIS over MASS.

- Mathematical functions performed with *GeoTools* can be converted to be performed by Agents in MASS with MASS's optimised functions. This includes the calculation of distance between two points, the intersection between two line-segments and determining properties such as point-in-polygon, point distance to lines.

- The GIS data store can be implemented to be writable, for further scalability and to work around the shape file size limitation.
- The GIS can be connected to another type of agent-based system to compare performance with this implementation with MASS.
- In benchmarking, it was found that performance was improved during map rendering by having a minimum of rows and a maximum of columns. This behaviour should be further investigated.
- The GIS functionality along with the user interface can be expanded to perform more types of GIS analysis.
- Traditional GIS support multiple data formats. This implementation can be expanded to support more formats than shape files, using *GeoTools* plugins. This could also include removing the requirement for a raster base map so that the map can be rendered from a vector file.
- The GUI is currently a prototype. The interface can be improved by verifying and testing usability through user reviews.
- The user interface can be improved with navigation and zoom in the map rendering section. Mouse drag and clicks can be used as alternatives to buttons to make the interface more intuitive.

REFERENCES

[1] P. M. Fukuda, "MASS: A Parallelizing Library for Multi-Agent Spatial Simulation," 1 November 2021. [Online]. Available: https://depts.washington.edu/dslab/MASS/.

[2] "GIS Glossary," 2 Dec 2021. [Online]. Available: http://wiki.gis.com/wiki/index.php/GIS_Glossary.

[3] L. Wasser, "Introduction to Coordinate Reference Systems," 12 Dec 2021. [Online]. Available: https://www.earthdatascience.org/courses/earth-analytics/spatial-data-r/intro-to-coordinate-reference-systems/.

[4] S. Fellah, M. Desruisseaux, W. K. and A. Petkov, "Interface Coverage," 10 03 2022. [Online]. Available: https://docs.geotools.org/stable/javadocs/org/opengis/coverage/Coverage.html.

[5] "Feature Class," 13 Dec 2021. [Online]. Available: http://wiki.gis.com/wiki/index.php/Feature_class.

[6] GISGeography, "The Ultimate List of GIS Formats and Geospatial File Extensions," 11 Dec 2021. [Online]. Available: https://gisgeography.com/gis-formats/.

[7] "GeoTools Documentation," 2 Dec 2021. [Online]. Available: https://docs.geotools.org/.

[8] P. Taillandie, D.-A. Vo, E. Amouroux and A. Drogoul, "GAMA : bringing GIS and multi-level capabilities to multi-agent simulation," in *European Workshop on Multi-Agent Systems*, Paris, France, 2010.

[9] "OpenGIS FAQ," 10 03 2022. [Online]. Available: https://docs.geotools.org/stable/userguide/library/opengis/faq.html.

[10] apache.org, "Maven," Apache Maven Project, 2022. [Online]. Available: https://maven.apache.org/pom.html. [Accessed 24 March 2022].

[11] NASA, "Blue Marble," 12 03 2022. [Online]. Available: https://visibleearth.nasa.gov/collection/1484/blue-marble.

[12] "ArcMap," 2 Dec 2021. [Online]. Available: https://desktop.arcgis.com/en/arcmap/10.3/manage-data/coverages/what-is-a-coverage.htm.

[13] "Tutorials," 12 11 2021. [Online]. Available: https://docs.geotools.org/latest/userguide/tutorial/index.html.

[14] S. Gokulramkumar, "Agent Based Parallelization of Computational Geometry Algorithms," Seattle, Washington, 2020.

[15] X. Li, "Agent-Based Parallelization of a Multi-Dimensional Semantic Database Model," 2021.

[16] S. Paronyan, "Agent-Based Computational Geometry," Seattle, Washington, 2021.

[17] Z. Zhang, S. Zhang and C. Zhang, "An Agent-Based Hybrid Framework for Database Mining," *Applied Artificial Intelligence,* pp. 383-398, 2003.

[18] M. Batty and B. Jiang, "Multi-Agent Simulation: New Approaches to Exploring Space-Time Dynamics Within GIS," Centre for Advanced Spatial Analysis, London, 1998.

[19] Y. Peng, L. Meng and C. Lin, "Research on Coupling GIS and Multi-agent of Spatial Information," in *Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, 2010.

[20] D. G. Brown, R. Riolo, D. T. Robinson, M. North and W. Rand, "Spatial process and data models: Toward integration of agent-based models and GIS," *Journal of Geographical Systems,* pp. 25-47, 2005.

[21] R. C. Martin, Clean Code, Upper Saddle River, NJ: Pearson Education, Inc., 2009.

[22] S. Reges and M. Stepp, Building Java Programs A Basics to Basics Approach, Hoboken, NJ: Pearson, 2016.

[23] V. G. Cerf, "Loose Couplings," *IEEE internet computing,* p. 96, 03 2013.

ICONS

Nature Vectors by Vecteezy. (2022, 03 15). Retrieved from Vecteezy: https://www.vecteezy.com/free-vector/nature

APPENDIX A: DEFINITIONS

A.1. MULTI-AGENT SPATIAL SIMULATION

The MASS (multi-agent spatial simulation) library is an agent-based programming paradigm in Java, C++, and CUDA. It performs multi-agent and spatial simulation over a cluster of computing nodes using parallelisation. This parallel big data analysis is applicable to various scientific fields, including environmental data science, bioinformatics and space cognition. Places, which are mapped to threads, and Agents, which are mapped to individual processes, are key parts of MASS [3].

A.2. IMAGE FILE FORMATS

A.2.1. GeoTiff

Raster format image file with data contained in tiff. The data includes the coordinate reference system as well as the envelope of the image.

A.2.2. WorldFile

Tiff raster image file with sidecar files that contain data. The data includes the coordinate reference system as well as the envelope of the image.

A.3. POSTGIS

PostGIS is an open-source software program implemented as an external extender for PostgreSQL object-relational database. PostGIS adds support for geographic objects to the database by running location queries in SQL [24].

APPENDIX B: INSTALLATION AND TESTING

Currently the code is set up to run on the `cssmpi` servers at UW Bothell. The repository (https://bitbucket.org/mass_application_developers/mass_java_appl/src/master/) needs to be cloned on one of these servers. Create a folder on the primary server and clone the code to this folder. Currently the code exists in the `msieling-gis-database` branch.

Each functional element of the program can be run independently. Refer to the Maven and the pom.xml file section of this document for instructions on modifying the `pom.xml` file to run individual functionality.

Follow the next steps.

B.1. SETTING UP NODES

The servers and nodes are defined in the `nodes.xml` file. Note that there are various versions of this file. The specific one is located in the `..\mass_java_appl\Applications\gis_database` folder. Set up the required servers in this file. Example 1 has two nodes set up.

.ssh private key

To set up the file, the user first needs to define an `.ssh` key. In this example the key is of type `id-rsa`. Instructions on how to create a private key are in the MASS user guide.

hostname

Each host name refers to a server to be used.

masshome

This is the location of the project jar (after running `mvn package`).

username

The username needed to log into the host

Example: The following code shows a sample node.xml file.

```
<nodes>

  <node>
    <master>true</master>
    <hostname>cssmpi8h</hostname>
    <masshome>~/capstone/mass_java_appl/Applications/gis_database/target/</masshome>
    <username>msieli</username>
    <privatekey>~/.ssh/id_rsa</privatekey>
    <port>22123</port>
  </node>

  <node>
    <hostname>cssmpi7h</hostname>
    <masshome>~/capstone/mass_java_appl/Applications/gis_database/target/</masshome>
    <username>msieli</username>
    <privatekey>~/.ssh/id_rsa</privatekey>
    <port>22123</port>
  </node>

</nodes>
```

B.2. BUILDING THE PACKAGE WITH MAVEN

Run mvn package in the ..\mass_java_appl\Applications\gis_database folder. This will create a `target` folder.

B.3. DATA FILES

The data files include all the relevant feature and image files (`*.TIFF`,`*.SHP`, `*.DBF`, `*.SHX`, etc.). Copy these data files into the `input` folder. Java uses relative referencing. When running the program, it will look for this `input` folder relative to the `target` folder unless an absolute path is used.

B.4. MAVEN AND THE POM.XML FILE

Various changes had to be made to the pom file to include *GeoTools* dependencies.

Originally a very large jar would be created that included all the dependencies by unpacking their respective jars and putting their class and config files into the large jar. This caused conflicts since many files had the same names, specifically many of the metadata files. By leaving the dependency jars separate, this problem was avoided.

Example: The following code shows the changes to the pom.xml file.

```
<!-- Make this jar executable -->
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <executions>
      <execution>
         <phase>package</phase>
         <goals>
                <goal>jar</goal>
         </goals>
      </execution>
   </executions>
   <configuration>
      <archive>
         <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.App</mainClass>
                <classpathPrefix>dependency-jars/</classpathPrefix>
         </manifest>
      </archive>
   </configuration>
</plugin>

<!-- Copy project dependency -->
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-dependency-plugin</artifactId>
   <executions>
      <execution>
         <id>copy-dependencies</id>
         <phase>package</phase>
         <goals>
                <goal>copy-dependencies</goal>
         </goals>
         <configuration>
                <includeScope>runtime</includeScope>
                <outputDirectory>${project.build.directory}/dependency-jars/</outputDirectory>
         </configuration>
      </execution>
   </executions>
</plugin>
```

Refer to the section B.7 (Running the program) in this document for instructions on optional changes to the pom.xml file to prove different functionality.

B.5. DATABASE FUNCTION

The database works with flat files. It distinguishes between both image and feature (data) files. When the data is divided, files that are related (e.g. `*.SHP`, `*.DBF` and `*.SHX`) are stored together.

The data files (base maps and feature files) need to be stored in the input folder in the `gis_database` folder (Figure 35) for the data to be found to be divided amongst nodes. These files will only be referenced in the initial division of data. The tiles created after division will be located in the `tmp` folder on the respective nodes.

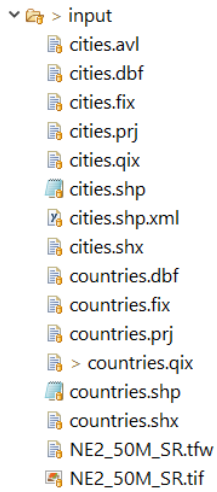

Figure 35 Feature files and base map files in the input folder

The build configuration (Figure 36) and run configuration differ with regards to the file structure. The pom.xml file to run the maven build in, is located in the `gis_database` folder. The build is completed with the command `mvn package` run in the `gis_database` folder.

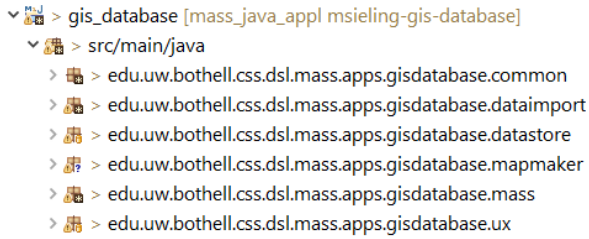

Figure 36 Structure of java file source

The build creates a **target** folder in `gis_database` if it does not exist and creates the necessary files for the run configuration. This folder includes the nodes.xml file. The nodes.xml file is not generated. The user needs to ensure that it exists.

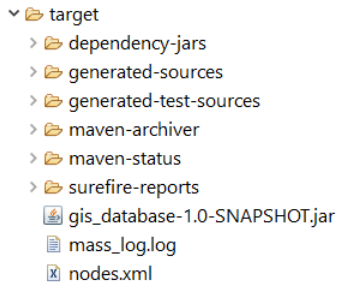

Figure 37 Run configuration: files created after build

The command to run the program needs to be run from the target folder:
`java -jar gis_database-1.0-SNAPSHOT.jar`

B.6. TESTING WITH MOCKITO

Unit tests are done with Mockito and JUnit.

Example: The following code uses mocks to test if the iterator returns a record if there is another record.

```
@Test
public void getNext_trueHasNext_returnRecord() throws Exception {
     SimpleFeature simpleFeatureMock = Mockito.mock(SimpleFeature.class);
     Mockito.when(simpleFeatureIteratorMock.hasNext()).thenReturn(true);
     Mockito.when(simpleFeatureIteratorMock.next()).thenReturn(simpleFeatureMock);
     Mockito.when(simpleFeatureMock.getAttributeCount()).thenReturn(1);
     Mockito.when(simpleFeatureMock.getAttribute(Mockito.anyInt())).thenReturn("test");
     FeatureRecordReader featureReader = new FeatureRecordReader(
                                              TestHelper.getFeatureQueryArgs(),
                                              1,
                                              2,
                                              "dummyFolder"
                                              );
     FeatureRecord featureRecord = featureReader.getNext();
     assertNotNull(featureRecord);
     assertEquals(1, featureRecord.getAttributes().size());
     assertEquals("test", featureRecord.getAttributes().get(0));
    }
```

B.7. RUNNING THE PROGRAM

The program can be run with the user interface as a complete system or with individual functionality from the command line for each software feature.

A basic user interface to show proof of concept allows a user to load a map, select an area and load selected features for this area. These features are rendered on top of the map and displayed to the user. Each functionality can also be tested separately by making the appropriate changes to the pom file in the root directory (by changing the `mainClass`).

As this Java application is a Maven project, it requires a Project Object Model (pom.xml) file to be compiled [25]. This file contains the configuration details so the project can access code it is dependent on that is part of other projects. MASS is also included in the project through the pom file as a dependency.

The number of nodes used in the program are defined by the node.xml file. This file is used by MASS and requires at least one node.

The base map and feature files for this program needs to be stored in the same folder.

B.7.1. Individual Functionality: Send Data to Places

Distributing the data amongst places is a separate process to retrieving the data. When the base map and data file locations are changed, the data is redistributed amongst the nodes. These locations need to be defined before running the application the first time.

The `mainClass` in the pom file needs to be changed to

```
<mainClass>
     edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.App
</mainClass>
```

B.7.2. Individual Functionality: Get Data from Places

A query utility is included to filter the feature data which the user can use to receive results as text in a table format. The query returns everything if the query term is `include`.

Example: The `mainClass` in the pom file needs to be changed to

```
<mainClass>
     edu.uw.bothell.css.dsl.mass.apps.gisdatabase.datastore.AppRetrieve
</mainClass>
```

The `GisDataStore` (C.1 GisDataStore.java) is responsible for the following:

- Instantiates places
- List of supported features
- Creates `GISFeatureSource` (C.4 GisFeatureSource.java)for a selected feature

The program supports any features depending on the shape files it is given. The currently included files are for features such as cities and country borders. The cities feature would contain attributes such as population size, city name, country, state and location. While the feature files (shape files) are in the correct folder, the program will recognise them and add them to the list. Further shape files can be downloaded from numerous online sources.

The `GISFeatureSource` (C.4 GisFeatureSource.java) constructs instances of the `GISFeatureReader` (C.7 GisFeatureReader.java) for the source according to the input query parameters (Table 4 Sample query parameters). It also provides an indicator for the number of records for a given query. Included is a schema of the feature.

Table 4 Sample query parameters

Query	Result
`include`	All data
`CNTRY_NAME='CANADA'`	All rows where the country name matches `*Canada*`
`BBOX(the_geom,110,-45,155,-10)`	Rows where the coordinates fall inside the bounding box

The schema of a feature is the names of the columns as well as their data types. This could include population size as a number, city name as a string, and location as a coordinate point.

B.7.3. Individual Functionality: Render a Subsection of the Base Map

Currently the values for this test are hard coded to render a subsection of the map. After running the program, a .tiff file is created with the features included on top of the subsection of the base map. This is stored in the root folder of `gis_database`. This functionality was kept in the example to show the contrast with agent-based rendering. It can thus be made to be variable. In B.7.4 when using agents to render the fragments, the functionality is updated and the subsections are determined by the user.

Example: The `mainClass` in the pom file needs to be changed to

```
<mainClass>
        edu.uw.bothell.css.dsl.mass.apps.gisdatabase.mapmaker.TestRender
</mainClass>
```

B.7.4. Individual Functionality: Render Subsection of the Base Map with Agents

The query values for this example are hard coded to render a subsection of the map by using agents. After running the program, a `.TIFF` file will be created with the features included on top of the subsection of the base map. The sample application will use this same functionality to allow the user to render the map they are interested in.

The `mainClass` in the pom file needs to be changed to

```
<mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.mapmaker.TestAgentsRenderMap</mainClass>
```

B.7.5. Full Functionality: User Interface

The interface is created with JavaFX and FXML.

Example: The `mainClass` in the pom file needs to be changed to

```
<mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.ux.UiLauncher</mainClass>
```

The user interface combines all the functionalities. The folder with the data can be selected in settings. The folder name is stored. The same can be done with the base map. This functionality includes dividing the data amongst the nodes. The data can be retrieved via a query. It is displayed in a table. The sub map can be retrieved and displayed. Further functionality includes customisable features, panning over the map and zoom in and zoom out functionality.

Definition JavaFX is an open-source platform to develop user interfaces in Java. It serves as an alternative to Swing. It uses FXML to define the user interface visual components.

Definition FXML is a markup language to define the look and feel of a user interface. It defines containers and objects as well as the size of the object (Figure 38). Program logic is thus kept separate from the design.

```
A <Label fx:id="queryLabel" layoutX="25.0" layoutY="80.0" text="Query" />
```

Figure 38 Label FXML definition

The tool SceneBuilder works with FXML to allow for some drag and drop functionality in building the interface. The interface created in SceneBuilder is saved as an FXML file. Controller settings (Figure 39) connects the relevant program logic to the FXML file.

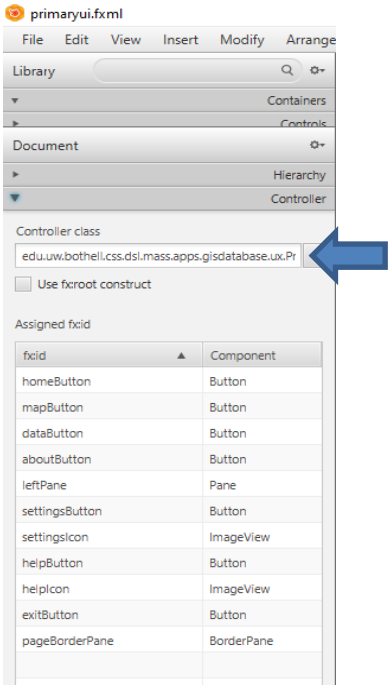

Figure 39 Controller class setting

B.7.6. Running the Command Line Arguments

Ensure that the package has been built with Maven. Navigate to the target folder.

Send Data to Places

Run `java -jar gis_database-1.0-SNAPSHOT.jar -f <folder name>`

Arguments

`-f <foldername>`	(required) set the name of the data folder/ file
`-htc <number>`	(optional) set the number of horizontal tiles
`-vtc <number>`	(optional) set the number of vertical tiles

Get Data from Places / Render Subsection of Map
(with and without Agents)

Run `java -jar gis_database-1.0-SNAPSHOT.jar`

Run java -Dprism.order=sw -Djdk.gtk.version=2 -jar gis_database-1.0-SNAPSHOT.jar

B.8. VALIDATING RESULTS

The database files are in the `/tmp/gisdata` folder on each server. As the `tmp` folders are unique to each `cssmpi` server, each server will contain different files, making it easier to validate the accurate division and distribution of the files.

The log files will show the execution steps. (The log file can be found by opening the ~ (tilde) folder inside the `target` folder and navigating through all the subsequent folders to the text file. Note the path in Figure 40.) Logs will indicate if the process completed successfully. It also logs the mapping of the data to the places.

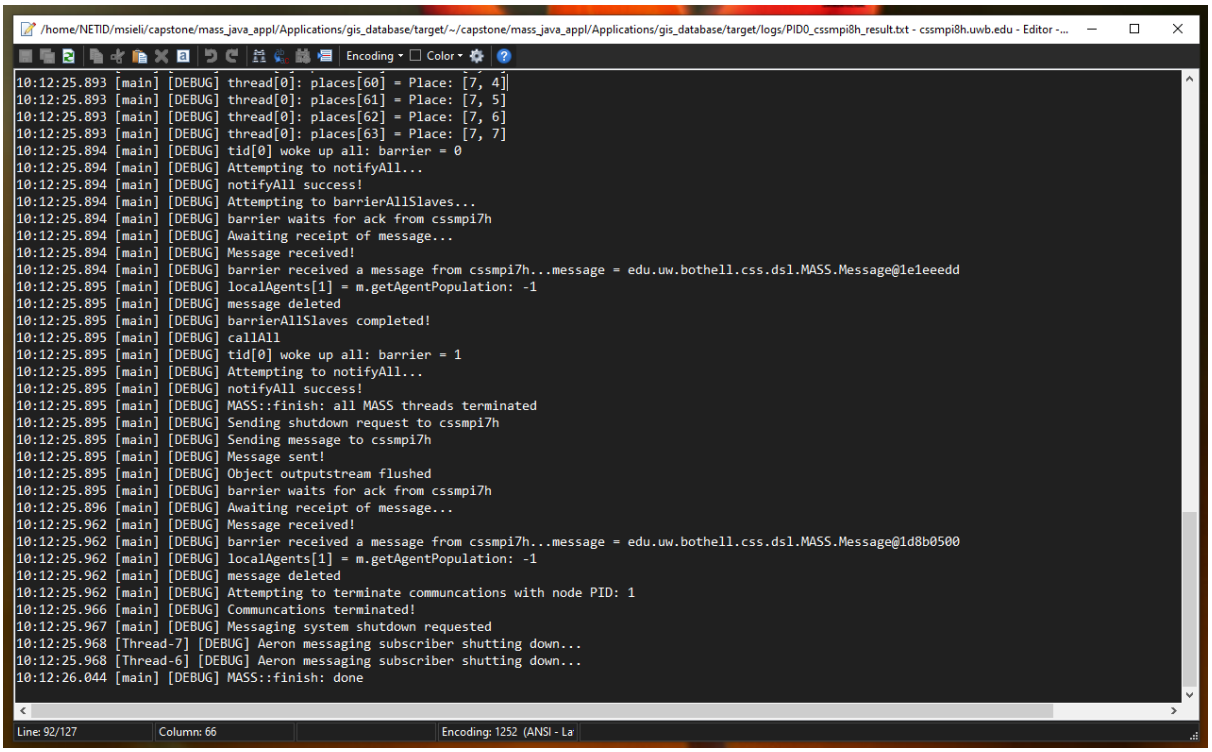

Figure 40 Example of a successful log file

APPENDIX C: CODE

C.1. GISDATASTORE.JAVA

```
public class GisDataStore extends ContentDataStore {

    private final Places places;

    // Get all the features - indicated by the file names
    public GisDataStore(Places places) {
       this.places = places;
    }

    /**
     * Remove everything from the first period.
     *
     * @param filename filename to trim
     * @return filename without extension / index
     */
    private String trimFileName(String filename) {
       return filename.split("\\.")[0];
    }

    /**
     * Get list of .shp files
     *
     * @return list of .shp files in source folder
     */
    private File[] getShapeFileList() {
       FileFilter fileFilter = new FileFilter() {
          public boolean accept(File file) {
             return (file.getName().endsWith(".shp"));
          }
       };
       File inputFolder = new File(Constants.SOURCE_FOLDER);
       return  inputFolder.listFiles(fileFilter);
    }

    /**
     * Get list of features
     */
    @Override
    protected List<Name> createTypeNames() throws IOException {
       Set<Name> featureSet = new HashSet<>(); // prevent duplicates

       File[] files = getShapeFileList ();

       for (File file : files) {
          featureSet.add(new NameImpl(trimFileName(file.getName())));
       }
       return new ArrayList<>(featureSet);
    }

    @Override
    protected ContentFeatureSource createFeatureSource(ContentEntry entry) throws IOException {
       return new GisFeatureSource(entry, Query.ALL, places);
    }

}
```

C.2. GISDATASTOREFACTORY.JAVA

```
private void createOrGetPlaces(GisConfig gisConfig) {
     if (this.places == null) {
          synchronized (lock) {
                if (this.places == null) {
                     this.places = new Places(
                           1,                                          // handle for places
                           GisDataPlace.class.getName(),               // places class name
                           null,
                        // constructor arg
                           gisConfig.getNumberOfHorizontalTiles(),     // size of first dimension
                           gisConfig.getNumberOfVerticalTiles());      // size of second dimension
                     GisLog.logMessage("Created Places!");
                }
          }
     }
}
```

C.3. GISDATAPLACE.JAVA

```
public class GisDataPlace extends Place {

     private static final long serialVersionUID = -3581731224030296274L;
     public static final int OPEN_READER = 0;
     public static final int SAVE_MAP_IMAGE = 1;
     public static final int SAVE_FEATURE = 2;
     public static final int QUERY_FEATURE = 3;
     public static final int READ_NEXT_RECORD = 4;
     public static final int CLOSE_READER = 5;
     public static final int RENDER_MAP_FRAGMENT = 6;

     private FeatureDataLoader featureDataLoader;
     private FeatureDataSaver featureDataSaver;
     private MapImageSaver mapImageSaver;
     private FeatureRecordReaderFactory featureRecordReaderFactory;

     /**
      * Set to print to logs

     static {
         System.setErr(new PrintStream(new StreamLogger(MASS.getLogger(), true)));
         System.setOut(new PrintStream(new StreamLogger(MASS.getLogger(), false)));
     }*/

     /**
      * Default constructor
      *
      * @param arg
      */
     public GisDataPlace(Object arg) {
         super();
         this.featureDataLoader = new FeatureDataLoader(Constants.SOURCE_FOLDER);
         this.featureDataSaver = new FeatureDataSaver(Constants.SOURCE_FOLDER);
         this.mapImageSaver = new MapImageSaver(Constants.SOURCE_FOLDER);
         this.featureRecordReaderFactory = new FeatureRecordReaderFactory(Constants.SOURCE_FOLDER);
     }

     /**
      * Calls the correct method to save the file depending on the type of data the
      * file contains. (Image vs Feature)
      *
      * @param functionID which save function (image vs feature)
      * @param argument   data to save
      */
     @Override
     public Object callMethod(int functionId, Object argument) {
         switch (functionId) {
         case OPEN_READER:
           try {
```

```
            this.featureRecordReaderFactory
                  .create((FeatureQueryArgs) argument, this.getIndex()[0], this.getIndex()[1]);
          } catch (Exception e) {
            GisLog.logException(e);
          }
          break;
        case SAVE_MAP_IMAGE:
          this.mapImageSaver.saveMapImage((ImageTileData) argument, this.getIndex()[0], this.getIndex()[1]);
          break;
        case SAVE_FEATURE:
          this.featureDataSaver.saveFeature((FeatureData) argument);
          break;
        case QUERY_FEATURE:
          return this.featureDataLoader.getFeatureRecords(
                (FeatureQueryArgs) argument,
                this.getIndex()[0],
                this.getIndex()[1]);
        case READ_NEXT_RECORD:
          FeatureRecordReader featureRecordReader = this.featureRecordReaderFactory
                .getReaderInstance((UUID) argument);
          return featureRecordReader.getNext();
        case CLOSE_READER:
          FeatureRecordReader featureRecordReaderToClose = this.featureRecordReaderFactory
                .getReaderInstance((UUID) argument);
          featureRecordReaderToClose.close();
          break;
        case RENDER_MAP_FRAGMENT:
          try {
            if (argument == null) {  //place doesn't have what I need
                GisLog.logMessage("Null args at " + Arrays.toString(this.getIndex()));
                return null;
            }
            GisLog.logMessage("Building fragment at " + Arrays.toString(this.getIndex()));
            IMapBuilder mapFragmentBuilder = new MapFragmentBuilder((MapRenderingArgs) argument);
            return mapFragmentBuilder.renderMap();
          } catch (Exception e) {
            StringWriter stringWriter = new StringWriter();
                PrintWriter printWriter = new PrintWriter(stringWriter);
                e.printStackTrace(printWriter);
                return stringWriter.toString();
          }
        default:
          break;
        }
        return null;
    }
}
```

C.4. GISFEATURESOURCE.JAVA

```
protected SimpleFeatureType buildFeatureType() throws IOException {
    String featureName = this.entry.getName().getLocalPart();
    FileFilter fileFilter = new FileFilter() {
        public boolean accept(File file) {
            return (file.getName().startsWith(featureName) && file.getName().endsWith(".shp"));
        }
    };

    File inputFolder = new File(Constants.SOURCE_FOLDER);
    File[] files = inputFolder.listFiles(fileFilter);

    FileDataStore fdStore = null;
    try {
        fdStore = FileDataStoreFinder.getDataStore(files[0]);
        SimpleFeatureSource featureSource = fdStore.getFeatureSource();
        return featureSource.getSchema();
    }
    finally {
        if (fdStore != null) {
            fdStore.dispose();
        }
```

```
22              }
23      }
24
```

C.5. IMAGETILEDATA.JAVA

```
1  public class ImageTileData implements Serializable {
2
3      private static final long serialVersionUID = 3306801116392230899L;
4      public byte[] image;
5      public String fileExtension;
6      public String filename;
7
8      /**
9       * Default constructor
10      */
11     public ImageTileData() {
12     }
13
14
15     /**
16      * Parameterised constructor
17      *
18      * @param image
19      * @param fileExtension
20      * @param filename
21      */
22     public ImageTileData(byte[] image, String fileExtension, String filename) {
23        this.image = image;
24        this.fileExtension = fileExtension;
25        this.filename = filename;
26     }
27 }
```

C.6. FEATURERECORDREADER.JAVA

```
1  public class FeatureRecordReader {
2
3     private FileDataStore store;
4     private SimpleFeatureIterator simpleFeatureIterator;
5     private UUID id;
6
7     public FeatureRecordReader (FeatureQueryArgs featureQueryArgs, int row, int column, String
8  sourceFolderName) throws Exception {
9        this.id = featureQueryArgs.getId();
10
11       String fileName = getFileName(sourceFolderName, featureQueryArgs.getFeatureName(), row, column);
12       store = FileDataStoreFinder.getDataStore(new File(fileName));
13       SimpleFeatureSource featureSource = store.getFeatureSource();
14
15       SimpleFeatureCollection collection =
16 featureSource.getFeatures(CQL.toFilter(featureQueryArgs.getQueryCondition()));
17
18       //open file
19       simpleFeatureIterator = collection.features();
20    }
21
22    public UUID getId() {
23       return id;
24    }
25
26    public FeatureRecord getNext() {
27       if (!simpleFeatureIterator.hasNext()) {
28            MASS.getLogger().debug("No next record");
29            return null;
30       }
31       SimpleFeature simpleFeature = simpleFeatureIterator.next();
32       FeatureRecord featureRecord = new FeatureRecord();
33       for (int i = 0; i < simpleFeature.getAttributeCount(); i++) {
34            featureRecord.add(simpleFeature.getAttribute(i));
35       }
36       return featureRecord;
37    }
38
39    public void close() {
40       if (simpleFeatureIterator != null) {
41            simpleFeatureIterator.close();
42       }
43
44       if (store != null) {
45            store.dispose();
46       }
47    }
48
49    /**
50     * Gets full path of file name.
51     *
52     * @param featureName feature that defines file name
53     * @return absolute path of file
54     */
55    private String getFileName(String sourceFolderName, String featureName, int row, int column) {
56       File tileDirectory = new File(sourceFolderName);
57       String featureFileName = featureName + ".shp" + row + "_" + column;
58       File featureFile = new File(tileDirectory, featureFileName + ".shp");
59       GisLog.logMessage("Attempting to load: " + featureFile.getAbsolutePath());
60       if (!featureFile.exists()) {
61            System.out.println("File not found!");
62       }
63       return featureFile.getAbsolutePath();
64    }
```

```
65      }
```

C.7. GISFEATUREREADER.JAVA

```
public class GisFeatureReader implements FeatureReader<SimpleFeatureType, SimpleFeature> {

      /** State used when reading file */
      private final ContentState state;
      private final Places places;
      private final FeatureQueryArgs featureQueryArgs;
      private final Queue<FeatureRecord> buffer;

      /**
       * @param state
       * @param query
       * @param places
       */
      public GisFeatureReader(ContentState state, Query query, Places places) {
          this.state = state;
          this.places = places;
          this.featureQueryArgs = new FeatureQueryArgs(state.getEntry().getName().getLocalPart(),
CQL.toCQL(query.getFilter()));

          this.places.callAll(GisDataPlace.OPEN_READER, constructArgs(this.featureQueryArgs));

          this.buffer = new LinkedList<>();

          refreshBuffer();
      }

      private Object[] constructArgs(Object obj) {
          Object[] objects = new Object[places.getSize()[0] * places.getSize()[1]];
          for (int i = 0; i < objects.length; i++) {
              objects[i] = obj;
          }

          return objects;
      }

      private void refreshBuffer() {
          Object[] results = places.callAll(GisDataPlace.READ_NEXT_RECORD, constructArgs(this.featureQueryArgs.getId()));
          for (Object record : results) {
              if (record != null) {
                  this.buffer.add((FeatureRecord) record);
              }
          }
      }

      @Override
      public SimpleFeatureType getFeatureType() {
          return state.getFeatureType();
      }

      @Override
      public SimpleFeature next() throws IOException, IllegalArgumentException, NoSuchElementException {
          FeatureRecord next = this.buffer.poll();
          if (this.buffer.isEmpty()) {
              refreshBuffer();
          }
          SimpleFeatureBuilder featureBuilder = new SimpleFeatureBuilder(getFeatureType());

          next.getAttributes().stream().forEach(a -> featureBuilder.add(a));

          return featureBuilder.buildFeature(null);
      }

      @Override
      public boolean hasNext() throws IOException {
          return !this.buffer.isEmpty();
      }

      @Override
      public void close() throws IOException {
          places.callAll(GisDataPlace.CLOSE_READER, constructArgs(this.featureQueryArgs.getId()));
      }

```

C.8. FEATUREDATALOADER.JAVA

```
1  public List<FeatureRecord> getFeatureRecords(FeatureQueryArgs featureQueryArgs, int row, int column) {
2          List<FeatureRecord> featureRecordList = new ArrayList<FeatureRecord>();
3          String fileName = getFileName(featureQueryArgs.getFeatureName(), row, column);
4          FileDataStore store = null;
5          SimpleFeatureIterator simpleFeatureIterator = null;
6
7          try {
8                  Filter filter = CQL.toFilter(featureQueryArgs.getQueryCondition());
9                  store = FileDataStoreFinder.getDataStore(new File(fileName));
10                 SimpleFeatureSource featureSource = store.getFeatureSource();
11                 SimpleFeatureCollection collection = featureSource.getFeatures(filter);
12                 simpleFeatureIterator = collection.features();
13
14                 while (simpleFeatureIterator.hasNext()) {
15                         SimpleFeature simpleFeature = simpleFeatureIterator.next();
16                         FeatureRecord featureRecord = new FeatureRecord();
17                         for (int i = 0; i < simpleFeature.getAttributeCount(); i++) {
18                                 featureRecord.add(simpleFeature.getAttribute(i));
19                         }
20                         featureRecordList.add(featureRecord);
21                 }
22
23         } catch (Exception e) {
24                 GisLog.logException(e);
25         } finally {
26                 simpleFeatureIterator.close();
27                 store.dispose();
28         }
29         return featureRecordList;
30     }
```

Code runs on places

C.9. TILER.JAVA

```
public ImageTileData[][] getTiles() throws IllegalArgumentException, IOException {
     GisLog.logMessage("Attempting to process " + this.inputFile.getAbsolutePath());
     ImageTileData[][] imageArgs = null;
     AbstractGridFormat format = GridFormatFinder.findFormat(this.inputFile);
     String fileExtension = this.getFileExtension(this.inputFile);

     // working around a feature in geotiff loading via format.getReader which
     // sets the axis in reverse
     Hints hints = null;
     if (format instanceof GeoTiffFormat) {
          hints = new Hints(Hints.FORCE_LONGITUDE_FIRST_AXIS_ORDER, Boolean.TRUE);
     }

     GridCoverage2DReader gridReader = format.getReader(this.inputFile, hints);
     GridCoverage2D gridCoverage;

     gridCoverage = gridReader.read(null); // get everything
     Envelope2D coverageEnvelope = gridCoverage.getEnvelope2D();
     double geographicTileWidth = (Constants.MAX_X - Constants.MIN_X)
                / (double) gisConfig.getNumberOfHorizontalTiles();
     double geographicTileHeight = (Constants.MAX_Y - Constants.MIN_Y)
                / (double) gisConfig.getNumberOfVerticalTiles();

     CoordinateReferenceSystem targetCRS = gridCoverage.getCoordinateReferenceSystem();

     // iterate over the tile counts
     imageArgs = new
ImageTileData[gisConfig.getNumberOfHorizontalTiles()][gisConfig.getNumberOfVerticalTiles()];
     for (int i = 0; i < gisConfig.getNumberOfHorizontalTiles(); i++) {
          for (int j = 0; j < gisConfig.getNumberOfVerticalTiles(); j++) {

               GisLog.logMessage("Processing tile at indices i: " + i + " and j: " + j);
               // create the envelope of the tile
               Envelope envelope = getTileEnvelope(
                     Constants.MIN_X,
                     Constants.MIN_Y,
                     geographicTileWidth,
                     geographicTileHeight,
                     targetCRS,
                     i,
                     j);

               GridCoverage2D finalCoverage = cropCoverage(gridCoverage, envelope);

               if (this.gisConfig.getTileScale() != GisConfig.DEFAULT_SCALE) {
                  finalCoverage = scaleCoverage(finalCoverage);
               }

               ByteArrayOutputStream os = new ByteArrayOutputStream();

               // use the AbstractGridFormat's writer to write out the tile to the byte array
               // output stream
               format.getWriter(os).write(finalCoverage, (GeneralParameterValue[]) null);

               imageArgs[i][j] = new ImageTileData(os.toByteArray(), fileExtension,
this.inputFile.getName());
          }
     }
     return imageArgs;
}
```

C.10. FULLMAPBUILDER.JAVA

```
public BufferedImage renderMapToImage() throws IOException, InterruptedException {
            WorldGrid worldGrid = new WorldGrid(gisConfig);

            int startX = 0;
            int endX = gisConfig.getNumberOfHorizontalTiles() - 1;
            int startY = 0;
            int endY = gisConfig.getNumberOfVerticalTiles() - 1;

            // find cells that contain the corner points of the image to be rendered
            for (int i = 0; i < gisConfig.getNumberOfHorizontalTiles(); i++) {
                Bounds cellBounds = worldGrid.getCellBounds(i, 0);
                if (cellBounds.getMinX() < mapRenderingArgs.getBounds().getMinX()
                        && cellBounds.getMaxX() > mapRenderingArgs.getBounds().getMinX()) {
                    startX = i;
                } else if (cellBounds.getMinX() < mapRenderingArgs.getBounds().getMaxX()
                        && cellBounds.getMaxX() > mapRenderingArgs.getBounds().getMaxX()) {
                    endX = i;
                }
            }
            for (int j = 0; j < gisConfig.getNumberOfVerticalTiles(); j++) {
                Bounds cellBounds = worldGrid.getCellBounds(0, j);
                if (cellBounds.getMinY() < mapRenderingArgs.getBounds().getMinY()
                        && cellBounds.getMaxY() > mapRenderingArgs.getBounds().getMinY()) {
                    startY = j;
                } else if (cellBounds.getMinY() < mapRenderingArgs.getBounds().getMaxY()
                        && cellBounds.getMaxY() > mapRenderingArgs.getBounds().getMaxY()) {
                    endY = j;
                }
            }

            // build list of relevant cells that agents will migrate to
            Cell[] cellList = new Cell[(endX - startX + 1) * (endY - startY + 1)];
            int cellIndex = 0;

            MapRenderingArgs[] subMapRenderingArgs = new MapRenderingArgs[(endX - startX + 1) * (endY - startY + 1)];

            for (int i = startX; i <= endX; i++) {
                for (int j = startY; j <= endY; j++) {
                    // places agent needs to migrate to so it can find right cells (cells and places
                    // are equivalent)
                    cellList[cellIndex] = new Cell(i, j);

                    Bounds bounds = worldGrid.getCellBounds(i, j);
                    double minX = Math.max(mapRenderingArgs.getBounds().getMinX(), bounds.getMinX());
                    double minY = Math.max(mapRenderingArgs.getBounds().getMinY(), bounds.getMinY());

                    double maxX = Math.min(mapRenderingArgs.getBounds().getMaxX(), bounds.getMaxX());
                    double maxY = Math.min(mapRenderingArgs.getBounds().getMaxY(), bounds.getMaxY());

                    subMapRenderingArgs[cellIndex] = new MapRenderingArgs(
                            "/tmp/gisdata/" + mapRenderingArgs.getImageFilename() + i + "_" + j + ".tif",
                            (int) (degreeToPixelRatio() * (maxX - minX)),
                            new Bounds(minX, minY, maxX, maxY));

                    for (MapRenderingFeatureArg featureArg : mapRenderingArgs.getMapFeatureArgs()) {
                        subMapRenderingArgs[cellIndex].addMapFeatureArg(
                                "/tmp/gisdata/" + featureArg.getFeatureName() + ".shp" + i + "_" + j + ".shp",
                                featureArg.getColor());
                    }
                    cellIndex++;
                }
            }

            agents.callAll(RenderMapAgent.SPAWN, new Object[] { cellList });
            agents.manageAll();
            System.out.println("ManageAll after SPAWN called");

            agents.callAll(RenderMapAgent.MIGRATE);
            System.out.println("MIGRATE called");
            agents.manageAll();
            System.out.println("ManageAll after MIGRATE called");

            Object[] results = (Object[]) agents.callAll(RenderMapAgent.RENDER, subMapRenderingArgs);
            agents.manageAll();
```

```
75
76              int horizontalTiles = endX - startX + 1;
77              int verticalTiles = endY - startY + 1;
78              BufferedImage[][] fragments = new BufferedImage[horizontalTiles][verticalTiles];
79              int totalWidth = 0;
80              int totalHeight = 0;
81              int counter = 0;
82              for (int i = 0; i < horizontalTiles; i++) {
83                  totalHeight = 0;
84                  for (int j = 0; j < verticalTiles; j++) {
85                      fragments[i][j] = toBufferedImage(results[counter]);
86                      totalHeight += fragments[i][j].getHeight();
87                      counter++;
88                  }
89                  totalWidth += fragments[i][0].getWidth();
90              }
91
92              BufferedImage combinedImage = new BufferedImage(totalWidth, totalHeight, BufferedImage.TYPE_INT_ARGB);
93
94              Graphics2D g = combinedImage.createGraphics();
95
96              int pixelX = 0;
97              int pixelY = 0;
98
99              for (int i = 0; i < horizontalTiles; i++) {
100                 int columnWidth = 0;
101                 for (int j = verticalTiles - 1; j >= 0; j--) {
102                     if (fragments[i][j] != null) {
103                         g.drawImage(fragments[i][j], pixelX, pixelY, null);
104                         pixelY += fragments[i][j].getHeight();
105                         columnWidth = fragments[i][j].getWidth();
106                     }
107                 }
108                 pixelY = 0;
109                 pixelX += columnWidth;
110             }
111
112             return combinedImage;
113         }
```

C.11. FILEDATA.JAVA

```
1   public class FileDetails implements Serializable {
2
3           private static final long serialVersionUID = 765237785295914119L;
4           private String name;
5           private byte[] contents;
6
7           /**
8            * @param name
9            * @param contents
10           */
11          public FileDetails(String name, byte[] contents) {
12              this.name = name;
13              this.contents = contents;
14          }
15
16          /**
17           * @return the name
18           */
19          public String getName() {
20              return name;
21          }
22
23          /**
24           * @return the contents
25           */
26          public byte[] getContents() {
27              return contents;
28          }
29      }
```

C.12. FEATURERECORD.JAVA

```java
public class FeatureRecord implements Serializable {
	private static final long serialVersionUID = 7112321129853400467L;
	private List<Object> attributes;

	/**
	 *
	 */
	public FeatureRecord() {
		this.attributes = new ArrayList<>();
	}

	public void add(Object attribute) {
		this.attributes.add(attribute);
	}

	/**
	 * @return the attributes
	 */
	public List<Object> getAttributes() {
		return new ArrayList<>(attributes);
	}
```