Analysis and Improvement of MASS-based GIS

Shahruz Mannan

A white paper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2024

Reading Committee:

Dr. Munehiro Fukuda, Chair

Dr. Michael Stiber

Dr. Dong Si

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Analysis and Improvement of MASS-based GIS

Shahruz Mannan

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda
Computer Science

Geographical Information Systems (GIS) are important in several fields due to their ability to perform the functions needed to capture, manage, analyze, and visualize geographic data. However, the increasing complexity and volume of the geospatial data throws a challenge to traditional GIS processing techniques. Therefore, enhanced computational strategies should be investigated to meet demanding requirements for CPU and spatial scalability. This project focuses on improving the existing integration of the Multi-Agent Spatial Simulation (MASS) library with GIS, particularly computational geometry problems used within queries. The work includes a comprehensive analysis of the existing MASS-based GIS system identifying the inefficiencies. It proposes strategies for improvement, implementations, and benchmarks of existing and new computational geometry problems including Range Search, Convex Hull, Largest Empty Circle, and Euclidean Shortest Path using both Message Passing Interface (MPI) and MASS for parallel

processing, and conducting performance evaluations assessing CPU scalability, spatial scalability, and programmability between MASS and MPI. The findings revealed that MASS implementations have enhanced the organization and execution of spatial queries. Evaluations showed that MASS requires less boilerplate code and has a lower code complexity than MPI. The MASS implementations generally demonstrated better spatial scalability by effectively handling larger datasets. For two of four computational geometry problems, MASS outperformed MPI as more computing nodes were used. The evaluations identified potential optimizations to further improve the performance and applicability of the MASS-based GIS system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

Geographical Information Systems (GIS) [1] are software systems containing tools for storing, managing, analyzing, and visualizing geographical data. Geographical data [2] refers to attribute information that is associated with specific geographic locations on the Earth's surface. Essentially, GIS is a solution integrating hardware, software, and information to enable users to transform and present data that demonstrates correlations, patterns, and trends through maps, reports, and charts. GIS does not only help in the conversion of large amounts of data into more understandable information but also facilitates data utilization and interpretation in various fields such as urban planning, environmental management, and transportation.

One of the challenges with GIS is the system being resource intensive. Using large datasets of geographical data can slow down analyzing and visualizing the data significantly [3]. Therefore, enhanced computational strategies should be investigated to meet demanding requirements for time and spatial scalability. Parallelization of GIS is an appropriate approach which can address the challenges regarding time and spatial scalability. CPU scalability essentially refers to GIS's capacity to process decisions swiftly regardless of the volume of data. For real-time systems such as emergency response or navigation assistance, reducing response times as data continues to grow is critical. On the other hand, spatial scalability is imperative as the scope and detail of GIS applications increase, necessitating systems that can handle larger datasets without compromising the accuracy and performance.

Agent-based Modeling (ABM) [4] is a simulation modeling technique that consists of a population autonomous of agents. ABM is used to explore the behavior of agents in an attempt to understand the outcome of their interactions. These agents are entities with the ability to make

decisions and the power to act independently. An agent may interact with other agents and the environment based on a set of predefined rules. ABMs are especially advantageous to GIS because it helps to simulate social and environmental processes by simulating interactions and their implications on geographical space. As each agent can be modeled with unique characteristics and decision rules, the agents can show the diversity found in real-world scenarios [5].

Although ABMs show potential with GIS, using ABMs also has drawbacks. One of the drawbacks of using ABMs is that simulations can become computationally intensive as the number of agents and the complexity of their interactions increase. Another challenge is whether the models accurately represent the systems they are intended to simulate. Thus, the internal logic of the model, the parameters for the model and the accuracy of the model need to be constantly verified. Lastly, ABMs integrated with GIS might need to manage data from different sources and format types which can make data synchronization and data processing more complex [6].

This capstone builds on an existing GIS system integrated with an ABM framework, Multi-Agent Spatial Simulation (MASS) [7], with the objective of analysis and improvement of the system. We add various contributions to the existing MASS-based GIS system.

- Conduct a comprehensive analysis of the existing system to determine areas that need improvement. Based on the analysis, spatial queries using computational geometry [8] problems were notably inefficient.
- Improve existing and implement new computational geometry problems using Message Passing Interface (MPI) Java [9] to form a baseline for comparison.
- Based on how implementations with MPI behave, implement the computational geometry problems with MASS library focusing on improving CPU scalability or spatial scalability or both.

- Integrate the computational geometry implementations with MASS to the MASS-based GIS system to be used with spatial queries.
- Identify performance bottlenecks by performance evaluations assessing CPU scalability and spatial scalability.
- Draw insights on code complexity and maintainability by conducting a programmability comparison including Cyclomatic Complexity and Lines of Code (LoC) between MPI and MASS implementations.

Following this introduction, the report is organized as follows: Chapter 2 introduces the background of MASS library and existing MASS-based GIS system. Chapter 3 overviews known ABMs integrated with GIS and computational geometry libraries. Chapter 4 describes the implementation of this project. Evaluations are discussed in Chapter 5 and lastly, Chapter 6 concludes the project and presents directions for future work.

Chapter 2. BACKGROUND

This chapter introduces the reader to the key components that form the basis of this project. The chapter starts by introducing the MASS library. Next, the previous work and its challenges are reviewed. The section concludes with an introduction of MPI and discussing its role in parallel computing and in this project.

2.1 MULTI-AGENT SPATIAL SIMULATION (MASS)

The Multi-Agent Spatial Simulation (MASS) [7] is an agent-based parallel computing library for spatial simulation over a cluster of computing nodes. Development was initiated by Distributed Systems Lab (DSLab) at University of Washington Bothell in 2010. MASS is an intuitive programming framework to simulate real-world problems consisting of big data processes such as bioinformatics, social networks, and geographic Information system [10]. Remote nodes for a process at each node and the processes communicate with each other via TCP connection.

The MASS library has two key components: *Places* and *Agents* [7]. *Places* represent a distributed array of place elements over a cluster of computing nodes. The distributed array is managed by global indices and each place element can be distinguished with a unique index. Each place can save data which can also be exchanged between other places. Agents are a set of executable instances that can reside in places and migrate to other places through the cluster. Each agent can interact with other agents and multiple places. Figure 1 shows the MASS model. The places are assigned to threads and the agents are grouped in bags in each process.

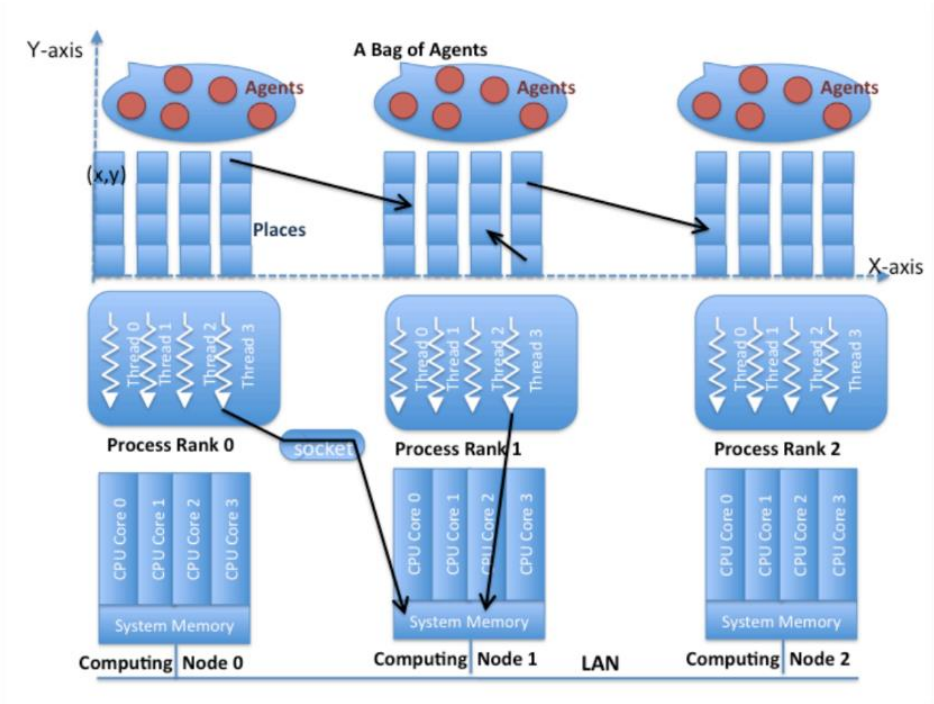


Figure 1: MASS Model [7].

2.2 Previous Work

The current development of MASS-based GIS systems has been greatly influenced by the early work of prior students concentrating on integrating MASS with GIS. These projects have used MASS's robust parallel processing and agent-based modeling abilities to address some of the challenges of traditional GIS.

M. Sieling [11] extended the capabilities of GIS by developing an agent-based approach to GIS databases that could effectively distribute and manage GIS data across multiple computing nodes. This approach reduced resource consumption and enhanced the overall performance through the parallel interaction between the computing nodes. Another of Sieling's approaches was agent-based rendering of GIS data. Essentially, the GIS data was handled in fragments by the agents. Agents would process independent fragments of a large dataset in different nodes, and then the entire dataset would be recompiled into an integrated image of the map. This was particularly

effective in ensuring high resolution and quality when interacting with the map. Furthermore, integrating the MASS-based GIS system with an open-source GIS toolkit, GeoTools [12] made it possible to perform complex spatial operations more efficiently.

S. Panduragi's [13] project was about parallelizing GIS queries to enhance the system's performance and scalability. She upgraded the GIS system to use distributed computing nodes. Previously the system used AWS cloud-based servers but later shifted to computational clusters in the University of Washington Bothell. This shifting enhanced better partitioning and distribution of spatial data across the multiple computing nodes promoting spatial scalability. Panduragi was also responsible for integrating GIS queries with computational geometry problems using the MASS library. This integration brought spatial analysis tools, including Closest Pair of Points (CPP) [15], Range Search (RS) [16], and Minimum Spanning Tree (MST) [17] directly into GIS. Lastly, Panduragi integrated Contextual Query Language (CQL) [14] from GeoTools. Particularly, CQL is a query language used to query information from the GIS system that involves spatial data or attribute data. Figure 2 shows the two categories of the parallelization of GIS queries.

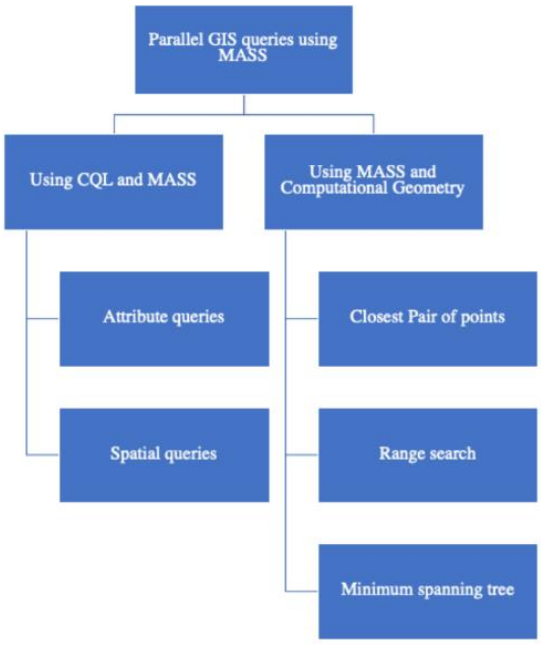


Figure 2: Previous GIS queries implemented in MASS-based GIS [13].

Both Sieling and Pandurangi's works offer improvements in computational efficiency, scalability, and performance for the MASS-based GIS system. However, several key challenges were identified as the previous work was reviewed, specifically related to the parallel GIS queries using computational geometry and MASS. The implementations of computational geometry problems with MASS encountered issues with CPU scalability and spatial scalability. As more computing nodes were used, the execution speed for Range Search and Minimum Spanning tree increased.

The Range Search problem also displayed issues with spatial scalability too. Previous performance evaluations showed RS had issues executing very large GIS datasets and the executions had to be stopped for taking extremely large amounts of time. Thus, indicating that parallelization had not improved the performance of this computational geometry problem. The Minimum Spanning Tree showed minimal CPU scalability but increasing the number of computing nodes further slowed down the execution performance. This computational geometry

problem was also implemented with MASS in C. Tsui's project [18]. The performance evaluations conducted by Tsui showed that this behavior is because of the memory overhead and the communication time between the computing nodes. These observations determined that MST might not be parallelizable with MASS and another approach was needed.

Another issue with the existing system was the scarcity of computational geometry problems. Although, three computational geometry problems have been integrated to be used with spatial queries, there is a lack of more advanced and complex computational geometry implementations in the system. This scarcity limits the applicability of the MASS-based GIS system in scenarios requiring in-depth spatial analysis and decision-making such as spatial pattern recognition and predictive spatial modeling.

Based on the observed challenges, we decided to focus on computational geometry problems with MASS portion of the system. This project will try to overcome these challenges by improving the existing and implementing new computational geometry problems to be used by the MASS-based GIS system. The selected computational geometry problems for further development are Range Search (RS), Convex Hull (CH) [19], Largest Empty Circle (LEC) [20], and Euclidean Shortest Path (ESP) [21]. The Range Search was chosen because the previous implementation was inefficient, and the MASS-based GIS needs an improved version of RS. CH and LEC were selected because these can be used for complex GIS queries and expand the applicability of existing system. The last computational geometry problem, ESP, can be used to solve similar problems as Minimum Spanning Tree. As MST is not parallelizable with MASS, Euclidean Shortest Path was chosen as a new approach potentially replace MST in the existing system. We implement these computational geometry problems with MPI first to observe the CPU and spatial

scalability. This approach will make the decision on what to focus on with the MASS implementations easier.

Chapter 3. RELATED WORK

This section covers agent-based models integrated with GIS and computational geometry libraries. At the end, it differentiates these related works from our approach.

3.1 AGENT-BASED FRAMEWORKS INTEGRATED WITH GIS

NetLogo [22] is a free, open-source, agent-based modelling system. It was developed with the aim of analyzing social and natural behaviors. The biggest advantage of NetLogo is its programming language which is a modified version of Logo programming language. Because of its simplicity, it allows beginners to create models with little programming skills. The ABM has a GIS extension which allows the users to import geographic data into NetLogo projects [23].

A study demonstrated a simulation of urban sprawl in the Waterloo region, Ontario, Canada [24]. The project utilized NetLogo to simulate the residential movement behavior of students based on their preferences on how far the housing locations are from necessary amenities and infrastructure. The agents made decisions on where to settle based on the time it would take to walk these locations. This study showed how NetLogo integrated with GIS offers a powerful system to analyze urban development patterns. Figure 3 shows the simulation environment created in NetLogo. The brown areas represent unsuitable locations for settlement, while the green areas indicate suitable locations where students can potentially settle.

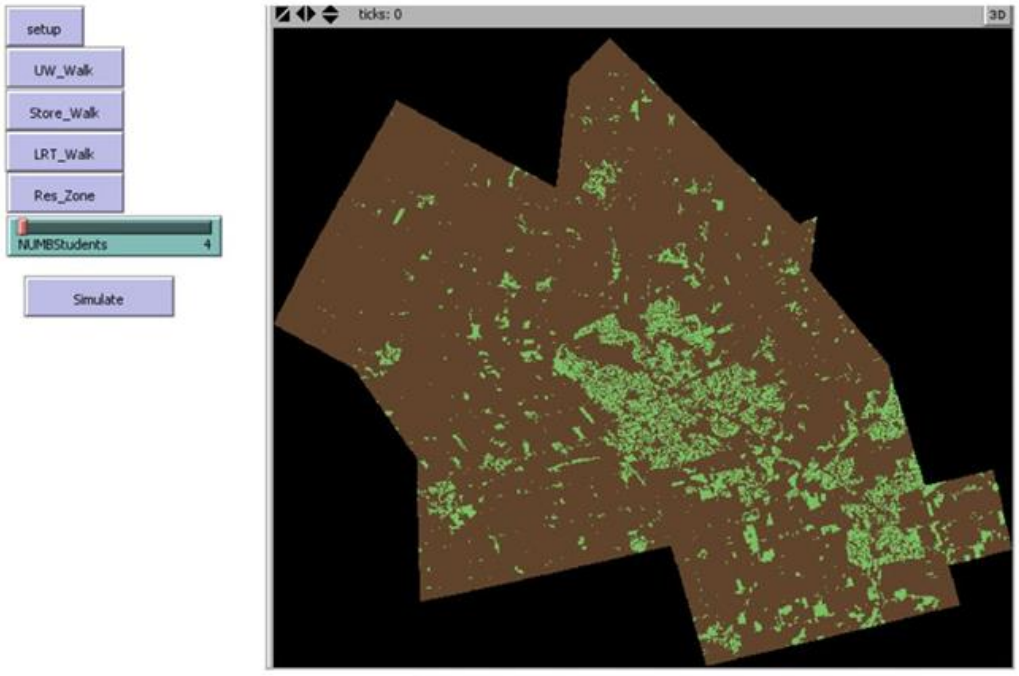


Figure 3: NetLogo simulation environment [24].

Repast Simphony [25] is a Java open-source framework which is designed for analyzing and building agent-based simulations. This ABM is a part of Repast Suite which is a family of ABM platforms. This framework uses geography projections for geographic referencing. This correlates to associating agents with specific positions in space. The agents in this space correspond to a particular geographical feature such as points or polygons. This geography projection relates to a connection with a coordinate referencing system (CRS) which can be used to execute Spatial and Attribute queries based on the agents' features. A study conducted by researchers at the Centre of Advanced Spatial Analysis (CASA) at University College London utilized Repast Simphony for simulating pedestrian movement patterns in a retail environment. Figure 4 demonstrates the simulation environment created for this study. The red dots represent the primary routes the agents have taken, highlighting the most frequently taken paths in the simulation environment.

Repast HPC [26] is an open-source, C++ based multi-agent simulation tool which enables parallel processing by using MPI and therefore making large-scale processes possible. In Repast

HPC, the agents are implemented as C++ classes, which allows the user to use C++ high performance libraries.

Figure 4: Pedestrian modelling in retail [25].

A more recent agent-based model framework that emphasizes the integration of GIS into simulation environments is GAMA [27]. This framework has similar GIS capabilities to Repast Simphony. GAMA treats every geographic object as an agent. This feature enables the dynamic management of geographic elements in the simulation. To simplify the process of including GIS data to simulations, a user-friendly modeling language GAML (GAMA modeling language) is provided.

A study used GAMA for a simulation of the local propagation and the persistence of avian flu in North Vietnam [27]. The simulation modeled the movement and the interactions between poultry populations from different farm types and landscapes. The agents represented the poultry populations. The interactions between the agents were guided by geographical features and human

management practices. Figure 5 shows the simulation model implemented with GAMA. The red dots represent the poultry flock agents within the simulation. The green areas indicate reachable rice fields which are the potential destinations for poultry flock agents.

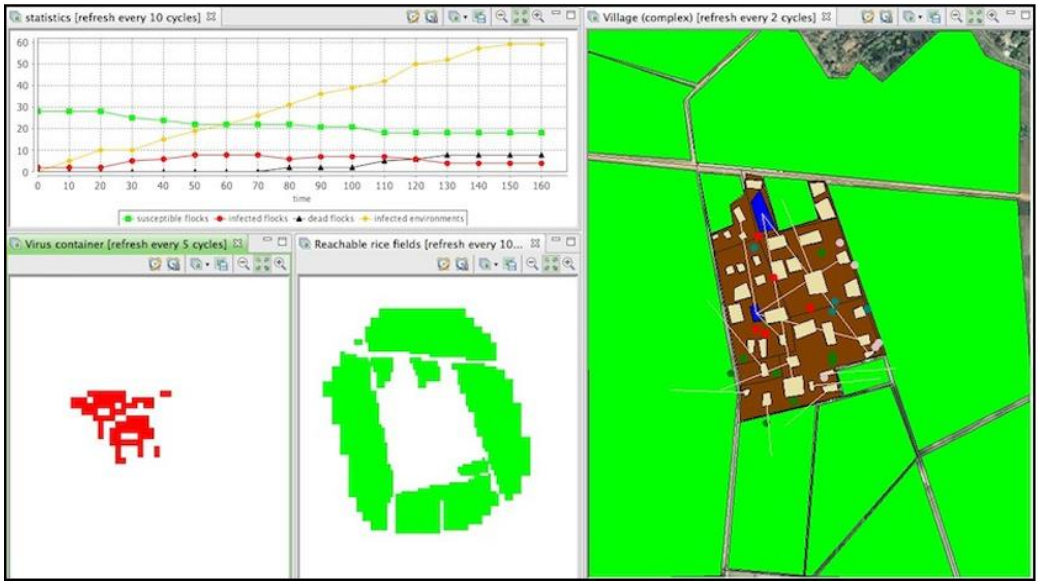


Figure 5: Avian Flu propagation and persistence simulation with GAMA [27].

Each of these ABMs has their advantages but also their challenges. NetLogo can have performance and scalability issues with larger or complex simulations due to its sequential execution. NetLogo also has a limited amount of debugging functionalities which might hinder the development process. Repast Suite has a steep learning curve which does not make it accessible for everybody. Repast HPC has outdated documentation, and the installation process is complex. For GAMA, its own modeling language GAML needs to be used which can be a learning curve and complicate the implementation of more complex simulation scenarios [28].

3.2 Parallelization of Computational Geometry Problems

The Computational Geometry Algorithms Library (CGAL) [29] is a C++ library of computational geometry algorithms. This library has algorithms for solutions such as 2D and 3D triangulations and surface mesh generations, searching neighbors for points and many other

geometric operations. This library is used in various fields such as medical imaging, CAD/CAM, and computer graphics. However, CGAL supports both commercial and open-source uses, thus making this library versatile.

ParGeo [30] is a multicore C++ library with parallel algorithms for computational geometry problems. This library is suitable for applications which require a large amount of data to be proessed on multicore machines. ParGeo contains modules for different tasks such as Kd-tree spatial search, spatial graph generation, and a collection of parallel geometric algorithms. In addition, ParGeo contains novel parallel algorithms for problems such as Convex Hull and batch-dynamic Kd-tree.

ParLeda [31] is a C++ library to make parallel implementations easier for computational geometry applications. This library integrates MPI (Message Passing Interface) for communication over distributed systems (a heterogeneous network of UNIX machines). Additionally, this library is utilizing data structures and algorithms from the existing computational geometry library LEDA. The MPI library enables the ParLeda to dynamically partition the data across the distributed system making this library suitable for environments where the computing power differs among the machines.

While these libraries make using computational geometry problems easier, they have limitations that can hinder the broader application. CGAL offers a wide range of algorithms, but only some of them are implemented in parallel. This limitation can restrict the library's effectiveness for real-time GIS applications. On the other hand, ParGeo is optimized for multicore systems but only for single machines. ParLeda presents a different set of issues. As this library uses MPI for communicating over the distributed system, the library inherits limitations associated with MPI such as lack fault tolerance and its low-level nature. In addition, ParLeda does not have

robust documentation which makes it challenging for developers to fully utilize the capabilities of this library.

3.3 Differentiate Our Approach from Related Works

The differences between our approach to other agent-based models integrated with GIS and computational geometry libraries are as follows:

- The ABM frameworks integrated with GIS don't use computational geometry algorithms for spatial queries. However, with MASS-based GIS, spatial queries can be executed by taking advantage of computational geometry problems such as Range Search, Convex Hull, Largest Empty Circle, and Euclidean Shortest Path.
- The sequential ABM, NetLogo, does not support parallel processing. Also, most of the CGAL library's algorithms are sequential. Our approach, parallel computational geometry algorithms with MASS which leverages parallel computing, specifically for distributed systems is a better approach for GIS applications which require manipulating and analyzing a large amount of spatial data.
- With GAMA, for model development purposes, GAML language needs to be used. This framework is better suited for quick prototyping because GAML being a simple and easy to learn modeling language. For more complex models and queries, MASS-based GIS is a better option. It can manage complex spatial data structures across multiple computing nodes which makes querying effective with large amounts of data.
- With MPI-based Repast HPC and ParLeda library, significant limitations include the lack of fault tolerance and synchronization challenges MASS-based GIS provides a better alternative. It uses low-level system calls and offers better fault tolerance and scalability, making it a more reliable choice for large-scale distributed GIS applications.

- The ParGeo library is designed for parallel computation within a single multicore machine. The library takes advantage of parallel schedulers like OpenMP [32], Cilk [33], or ParlayLib [34] which are used to manage and optimize parallel tasks on shared memory instead of in a distributed system. Our approach parallelizes computational geometry problems with MASS which assumes a cluster of multi-core computing nodes as the environment.

Chapter 4. IMPLEMENTATION

This section is organized to provide a detailed look at the implementation of four computational geometry problems: Range Search, Convex Hull, Largest Empty Circle, and Euclidean Shortest Path. The MPI implementations are explained first and then the MASS implementations. These computational geometry problems were selected based on the observations and limitations identified in the previous work section. Range Search can find features within a specific range which is a common requirement in geospatial analysis. Adding Convex Hull and Largest Empty Circle to MASS-based GIS will enable the existing system to be used for spatial pattern recognition and predictive spatial modeling. Lastly, Euclidean Shortest Path can be used in similar spatial queries as Minimum Spanning Tree for finding the shortest path between two coordinates.

4.1 MPI IMPLEMENTATIONS

The general process for implementing the computational geometry problems using MPI Java is first selecting what algorithm suits the best for these problems. After choosing the algorithm, the next part is splitting these algorithms into smaller tasks and identifying the specific tasks which can be executed in parallel. Then, these specific tasks are distributed to the other computing nodes using MPI, including the necessary data. Each node executes the task concurrently, which reduces the time spent on the computation. Lastly, the results from each computing node are collected to form the final solution for the original problem. The goal for implementing these computational geometry problems with MPI java is to observe their CPU and spatial scalability in order to form a baseline to compare with the MASS implementations. All these MPI implementations are implemented from scratch.

4.1.1 *Range Search*

Range Search (RS) is used to identify all points within a specified spatial boundary from a set of points. Figure 6 illustrates Range Search with a rectangular boundary. The red points are the output points which are in range. Several efficient data structures can be used for Range Searching including kd-trees [35], R-Trees [36], and Quadtrees [37]. The previous MASS implementation of RS used the kd-tree data structure to organize the points and query the tree for the points that fall in the specified boundary. We decided to use kd-trees for their simplicity and effectiveness in organizing and querying points in k-dimensional spaces.

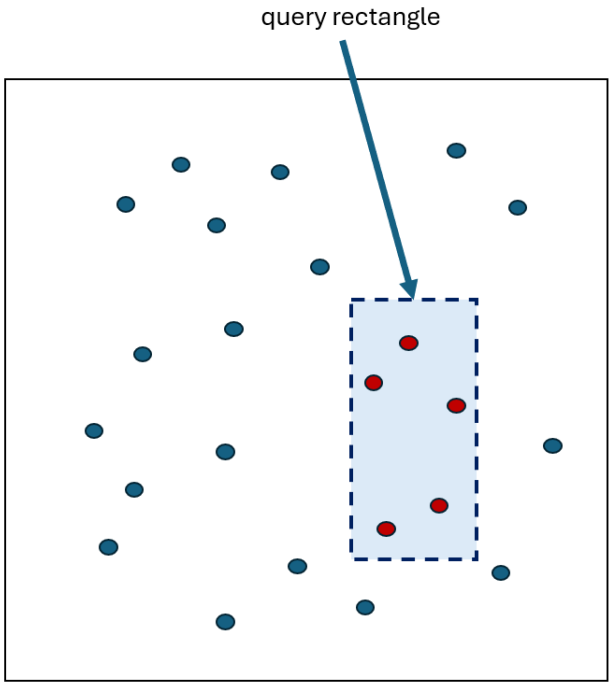


Figure 6: Range Search

This implementation follows the general process mentioned above. First, data points are read from a CSV input file and partitioning the data points equally to all the computing nodes. Each computing node constructs its own kd-tree and queries these trees to find the points in range. Kd-trees are binary trees where every node is a k-dimensional point (in our case, a 2-dimensional

point). The tree is built recursively by selecting a dimension, sorting the points with Quicksort [38] based on the selected dimension, and assigning the middle point as the node. The rest of the points are split into the node's left and right subtrees where smaller values go to left subtree and larger values to the right subtree. In this scenario, for every even layer (root being layer 0), the x coordinate was the dominating dimension and in every odd layer, the y coordinate was the dominating dimension. Figure 7 shows an example of how a kd-tree is constructed from a set of points. Every kd-tree is queried to find the points that are in the specified boundary. Once all the computing nodes have completed querying the trees, "Gather" MPI routine is called to gather all the points that are in range of the specified boundary back to the master node.

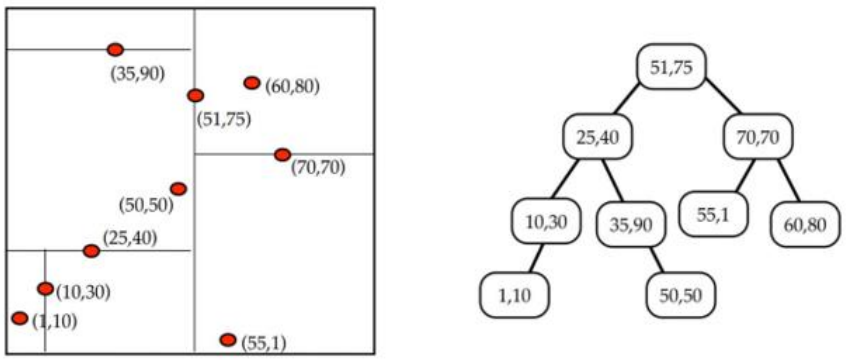


Figure 7: Example of a kd-tree [39].

4.1.2 *Convex Hull*

Convex Hull (CH) is a classic computational geometry problem that involves finding the smallest convex set containing all the points. As Figure 8 illustrates, the goal is to identify the least number of outermost points (red points) that form a convex polygon containing all the points. CH can be solved with many different algorithms such as Quickhull [19], Graham scan [40], and Monotone chain algorithm [41]. We chose to implement Monotone Chain algorithm because of two reasons. One is that all the algorithms above have the same time complexity as O(NlogN) where the N is the number of points. The second reason is that the other two algorithms are more

complex to implement. In the MASS-based GIS system, these two algorithms might complicate the execution of the spatial queries as coordinates (latitude, longitude) are introduced with spatial data.

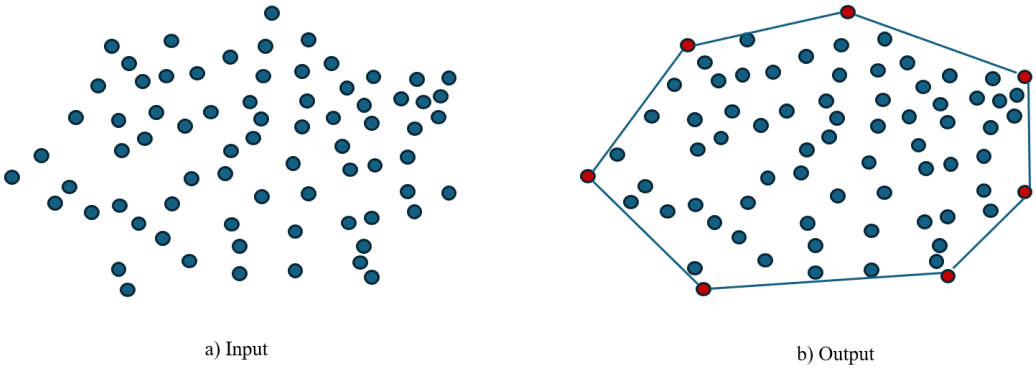


Figure 8: Convex Hull

The approach starts similarly by reading data points from an input file. However, before partitioning the data evenly for all the computing nodes, the data need preprocessing. The points are sorted based on the x coordinate. Now, partitioning the data and distributing the subsets to the computing nodes will result in the subsets having data points which are near each other. The data distribution can visualize as having the plane sliced into M vertical slices where M is the number of computing nodes.

Next, the Monotone Chain algorithm is used to compute the Convex Hull points in each computing node. The algorithm constructs the upper hull and the lower hull separately. Construction of the upper hull starts from the leftmost point and then checks iteratively for each point if the last two upper hull points and the current point make a left turn. If this left turn occurs, the second last point in the upper hull points is not an outermost point and needs to be removed. Once, no more left turns are found with the current point, this point is added to the upper hull points. The lower hull is computed in a similar manner but starting from the rightmost point and a

check for a right turn happens instead for a left turn. After both hull parts are constructed, the upper and the lower hulls are combined creating a complete convex hull. The right and left turns between three points are computed by calculating the determinant of the 3x3 matrix shown in Equation 1. L_i and L_{i-1} refer to the last two hull points and the p refers to the current point.

$$cp = \begin{vmatrix} L_{i-1}.x & L_{i-1}.y & 1 \\ L_i.x & L_i.y & 1 \\ p.x & p.y & 1 \end{vmatrix}$$

$$cp = (L_i.x - L_{i-1}.x)(p.y - L_{i-1}.y) - (L_i.y - L_{i-1}.y)(p.x - L_{i-1}.x)$$

Equation 1: Determinant formula [42].

After creating the partial hulls in every computing node, “MPI_Send” and “MPI_Recv” routines are called to send and receive hull points from other ranks to combine the hulls in a divide-and-conquer-like manner until the final global Convex Hull is in the master rank. Two partial convex hulls are merged by finding the upper and lower tangent lines. These tangent lines are calculated in a similar manner by checking for right and left turns until the outermost tangents that do not intersect the hulls are found. The data points between these tangent lines get removed. Figure 9 displays finding the two tangent lines between two different Convex Hulls.

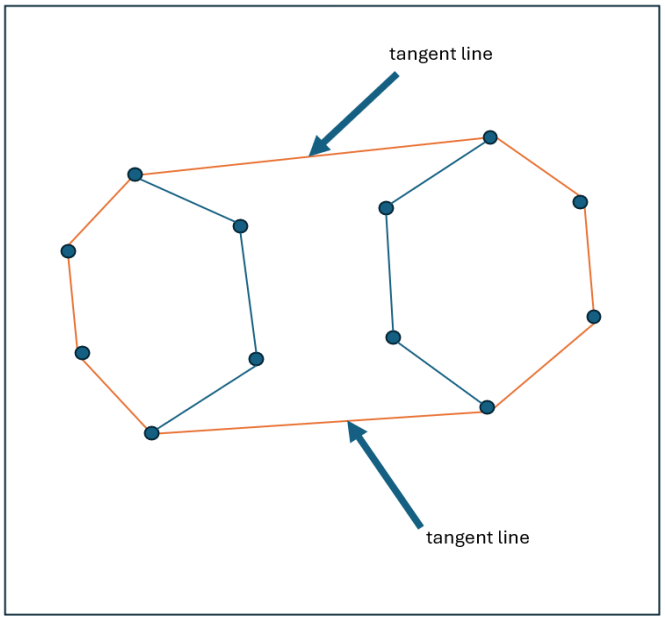


Figure 9: Tangent lines for partial Convex Hulls

4.1.3 *Largest Empty Circle*

This computational geometry problem, Largest Empty Circle (LEC), identifies the largest circle within a set of points that does not hold any of these points inside. Figure 10 visualizes a Largest Empty Circle from a set of points. The red point refers to the centroid of the Largest Empty Circle. For our approach, we decided to use a Voronoi Diagram to compute the LEC. The reason behind this decision is that LEC's centroid is either a Voronoi vertex or an intersection point between the input points' Convex Hull and a Voronoi Edge [20]. Creating a Voronoi Diagram will provide the vertices and the edges, and we have already implemented CH which can be used for this approach too, providing all the building blocks to retrieving every potential centroid for LEC.

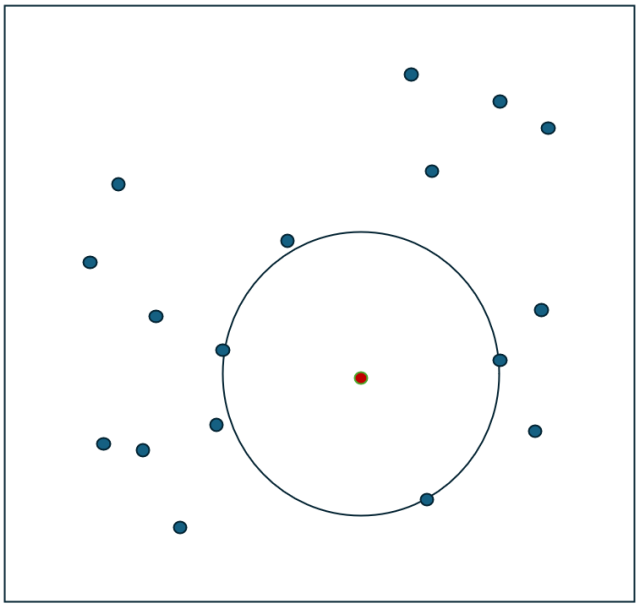


Figure 10: Largest Empty Circle

This approach starts similarly as Range Search MPI and Convex Hull MPI implementations by reading the data points from an input file. Next, a Voronoi Diagram from the input points is created using the Fortune Sweep algorithm [43] sequentially in the master rank. This algorithm is an efficient O(NlogN) algorithm where a sweep line progresses through the plane while maintaining an evolving beach line. While the beach line evolves, it creates the Voronoi vertices and edges. Once the sweep line has visited all the points, the Voronoi Diagram is completed. Figure 11 displays a complete Voronoi Diagram. A Voronoi Diagram represents partitioning of the plane into regions based on the distance to input points. Each region contains one input point and every point inside the region is closer to that input point than to any other.

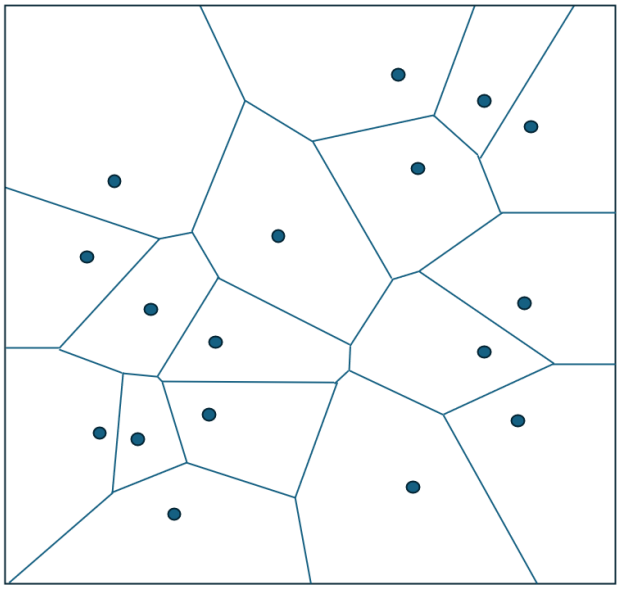


Figure 11: Voronoi Diagram

Once the Voronoi diagram is constructed, the Convex Hull points from the input points are computed with the Monotone chain algorithm we previously implemented. Next, the Voronoi vertices, Voronoi edges, and the Convex Hull points, are split into partitions and sent with MPI to the computing nodes. These nodes compute the intersection points between the subsets of Voronoi Edges and the Convex Hull edges to calculate the remaining potential center points. Next, the computing nodes iterate through these potential center points including Voronoi vertices and the intersection points to calculate the radius to their closest original data point. Finally, information of local LECs (radius and center point) are sent back to the master rank where the final LEC is selected. This part is the most computationally intensive part which is the reason the focus is on parallelizing this segment of the implementation. Figure 12 shows the Largest Empty Circle with Convex Hull and Voronoi Diagram constructed. The red dot refers to a Voronoi Vertex which is the centroid of LEC.

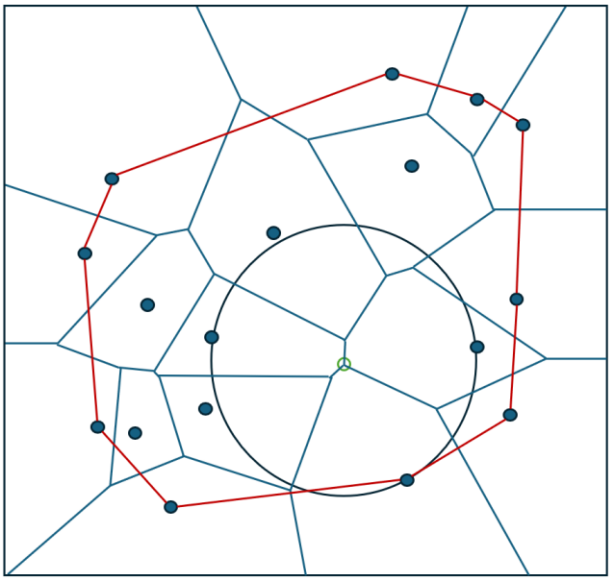


Figure 12: Largest Empty Circle where the centroid is on a Voronoi vertex.

4.1.4 *Euclidean Shortest Path*

The Euclidean Shortest Path (ESP) problem is one of the oldest and the most popular problems in computational geometry. This problem involves finding the shortest path between a start and destination point through polygon obstacles. This path must have the shortest traversal distance between these points without going through any of the obstacles. One of the common approaches for this problem is the visibility graph method where a visibility graph is constructed from the set of points and then running a graph traversal algorithm such as Dijkstra's algorithm [46] for finding the shortest path. This is the approach we chose to use for solving the ESP problem.

Input points for this implementation are the obstacle corners, source point and destination point. The data points are partitioned to the other computing nodes where visibility graphs are created from the subsets. A visibility graph is a graph of intervisible locations for a set of points. In other words, the vertices represent the data points, and the edges represent visible connections between the vertices which do not intersect obstacles. Figure 13 visualizes a visibility graph with two obstacles, source, and destination point. We implement the visibility graph with the naïve

approach [44]. This approach compares every pair of points from the input points whether a line segment between the pair of points intersects with obstacle edges. If a line segment does not intersect with any obstacle edge and does not go through an obstacle, the pair has a visibility edge.

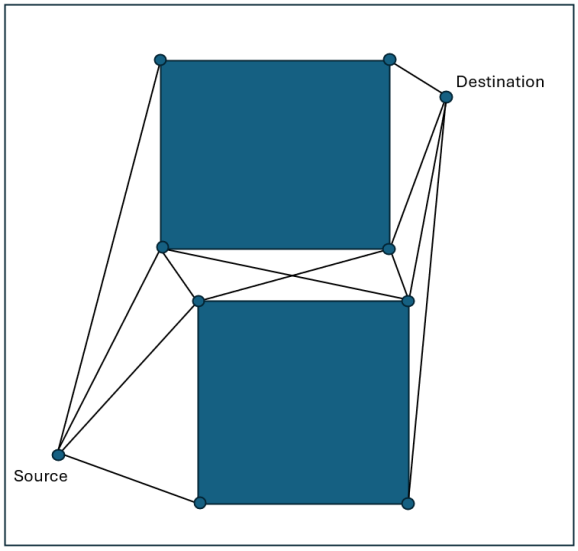


Figure 13: Visibility Graph

Once the visibility graphs are constructed, the information is saved as a HashMap, where the key is a vertex, and the value is a list of the vertices which can create a visibility edge with this specific vertex. Next, all the partial visibility graphs are sent back to the master rank and combined into complete visibility graph of all the data points. Lastly, Dijkstra's algorithm is used for finding the shortest path. Figure 14 shows the shortest path between the source point and the destination point with two obstacles.

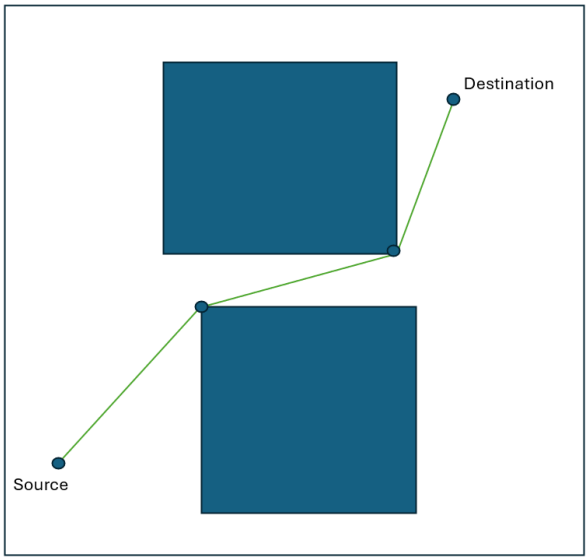


Figure 14: Euclidean Shortest Path

4.2 MASS IMPLEMENTATIONS

The general process for implementing computational geometry problems with MASS includes leveraging the library's capabilities for parallel processing and agent-based modelling.

The MASS implementation has two differences compared to MPI implementations. The first difference is that MASS enables partitioning and distributing the data across the computing nodes by using Places. The agents operate and interact by performing computational tasks in these places. The other difference is that GIS functionalities are added to these MASS implementations so that they would be compatible with the MASS-based GIS system to be used in spatial queries.

4.2.1 *Range Search*

Similarly to the Range Search MPI implementation, this approach uses the kd-tree data structure to organize the points and query the trees for the points which are in specified range. This implementation starts by reading the data points from a shapefile (.shp). Shapefiles offer a nontopological format to store the geometric location of geographic features. In addition to points,

shapefiles can store other complex geometries such as lines and polygons. Another reason for using shapefiles is because these can be easily visualized on maps. The next step is partitioning the data into subsets corresponding to the number of MASS Places. This ensures each subset is of a manageable size, thereby reducing the workload for every computing node. MASS gets initialized, and the Places and Agents are created. Each subset is assigned to a place where kd-trees are constructed from these points. In this implementation, the same recursive kd-tree construction algorithm is applied as in the MPI implementation. The previous MASS implementation created only one kd-tree with GraphPlaces, which is an extended class of Places. The vertices in the graph are distributed across the computing nodes and agents are spawned to query this single kd-tree. As this approach was not CPU and spatially scalable, creating multiple smaller kd-trees was a more appealing approach.

Once the kd-trees are constructed, the agents are responsible for querying the trees for the points in range. Each agent is associated with a Place and collects the points in range from that specific kd-tree. The agents return the points in range to the master node where the points are saved into the GIS database. There is a possibility that an Agent does not find a single point which falls under the specified range. In this scenario, the Agent gets terminated which leads to less agents returning the points in range and making the execution more efficient. Finally, the results are generated to a GIS map as coordinates.

GIS functionalities such as reading a shapefile and creating a map, need an external library. As the GeoTools library is already integrated with the MASS-based GIS system, I leverage this library for adding the GIS functionalities to this RS implementation. The GIS map is built by setting layers on top of each other to get the final map. This map has two layers: a tiled map from OpenStreetMap [47] and the points in range as dots on top of the map layer. Figure 15 presents a

spatial query of a mineral deposit dataset (MRDS) [48] to retrieve all mineral deposits in western Europe. The blue dots represent the mineral deposits.

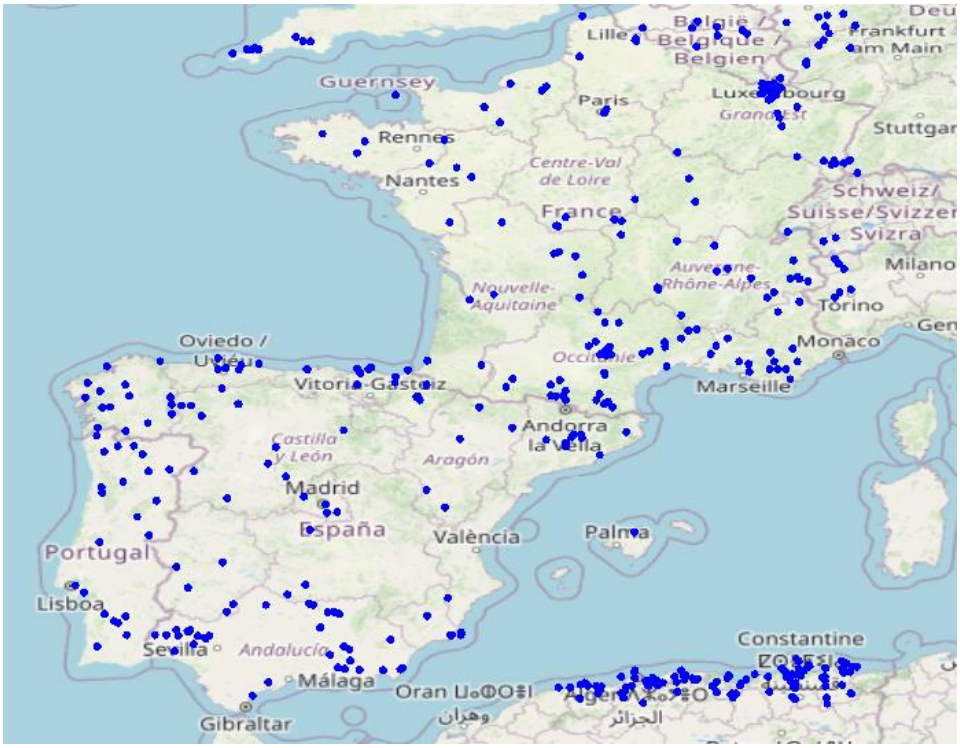


Figure 15: Map displaying spatial GIS query with Range Search.

4.2.2 *Convex Hull*

The Convex Hull MASS implementation uses the elastic band algorithm approach [49]. A. Potturi's [50] existing CH MASS application is used as a reference for this implementation. Similarly to the Range Search MASS implementation, CH MASS uses the MASS library to parallelize the computational geometry problem and adds GIS functionalities in order to be used with the MASS-based GIS system. Figure 17 displays a GIS map which the Convex Hull implementation generates with National USFS Fire Occurrence dataset [52]. Before using the Convex Hull on the dataset, Range Search is used to retrieve all the fire occurrences in Yellowstone National Park. The blue dots refer to the fire occurrences and the red line segments connect the outer hull points to create the Convex Hull.

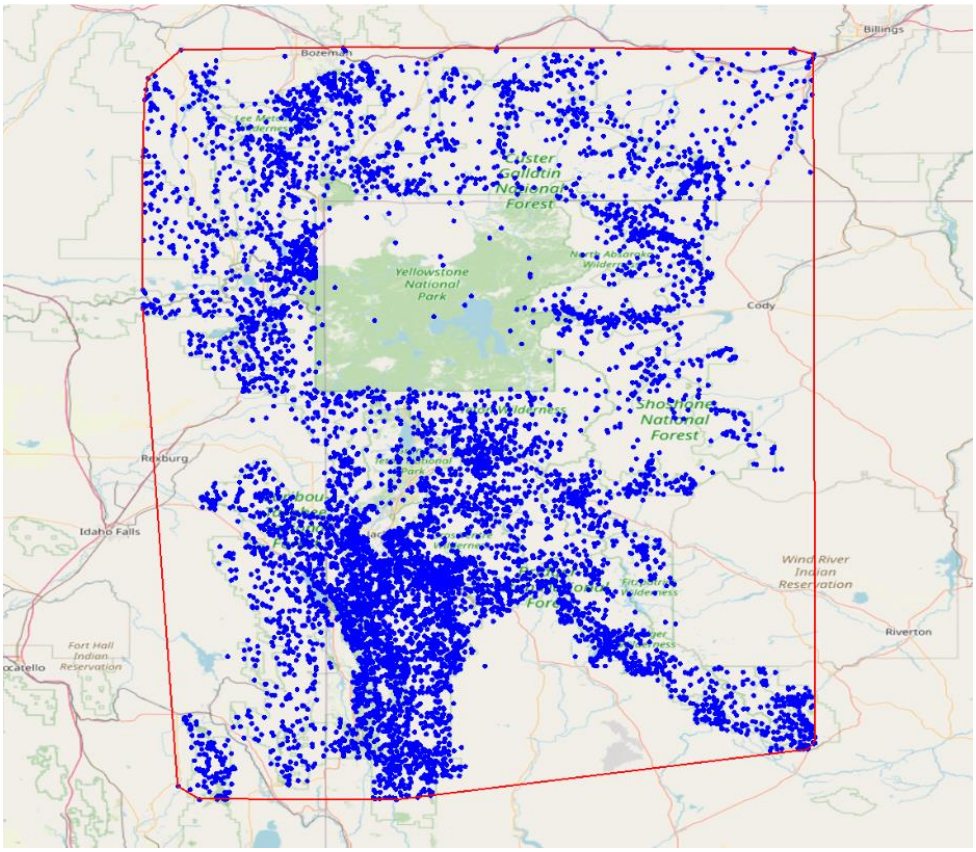


Figure 16: Map displaying spatial GIS query execution using Convex Hull.

In this implementation, the data points are read from a shapefile and at the same time the minimum and maximum values for both latitude and longitude are calculated to define bounding box to partition the data across the MASS Places. MASS initializes a two-dimensional grid of Places where each Place represents a portion of the overall space and contains a subset of the data points based on their spatial location. We make a modification to the approach regarding how each Place receives the points. In the referenced CH MASS application, each place reads the input file and collects the points which belonged to the specific place's space. Considering spatial scalability, using a large number of points may slow the performance as each place tries to read the same input file which can lead to overhead. We read the points only once and pass the points to the places when initialized.

Next, the agents are initialized and assigned starting positions at the boundary of the grid. We use agent propagation to find potential outer hull points which make the Convex Hull. Figure 16 shows the agent propagation process in a simplified manner. These agents traverse through the grid, and the strategy is to move inwards to identify potential points which might be part of the outer hull. If a place contains a potential outer hull point, the point is marked as visited and collected, while the current agent is terminated. After every agent is terminated, all the potential outer hull points are collected.

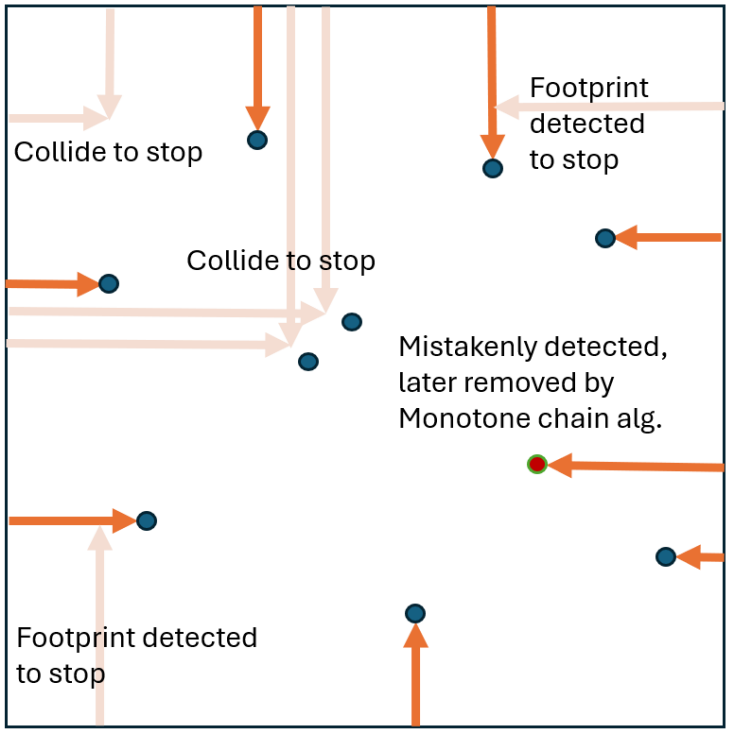


Figure 17: Agent propagation finding outer hull points.

From the reduced set of data points the final Convex Hull is calculated with Monotone chain algorithm to remove outliers which might have mistakenly identified as outer hull points. Lastly, the final CH points are written in the GIS database from where the points can be used for additional queries. In addition, a similar GIS map is created as in the RS MASS implementation. This GIS

map, however, has an additional layer for the map: line segments connecting the Convex Hull points which makes the convex hull complete on the map.

4.2.3 *Largest Empty Circle*

The first approach we attempted to use for the LEC MASS implementation was an agent propagation approach from a previously implemented LEC MASS application. The idea was that there are cities (input points) and dump sites (potential centroid of LEC), and the goal was to find the most suitable dump site farthest from all the cities. The strategy involved two sets of agents: static agents positioned at city locations and mobile agents starting from potential dump sites. The mobile agents propagated through the grid space trying to find a grid cell containing a city. Once a city was found, the distance from the original dump site was calculated aiming to maximize this distance. This approach, however, was unusable as the previous implementation was incomplete. It was missing multiple key functions which were called in the implementation making the code not executable. Another challenge was the implementation was not well-documented, especially for incomplete segments. Not understanding intended functionality made it difficult to debug. Thus, we decided to use a similar approach to the LEC MPI implementation.

The new approach for this problem creates a Voronoi Diagram sequentially from the read geographic data points. The same Fortune's Sweep algorithm used in the LEC MPI implementation is used for this approach too. Next, the diagram's vertices and edges are distributed among Places. Each place tries to compute the intersections between Convex Hull edges and Voronoi edges and the potential Largest Empty Circle. To determine the actual LEC, each Place sends their potential circles to the master node where the circles are compared to find the largest one among these circles. Lastly, a GIS map is created to visualize the output. For this implementation, four layers were needed to visualize the whole LEC problem: a tiled map layer,

an input points layer, a circle layer, and a centroid layer. The first two layers are self-explanatory. The circle layer contains the information about the circumference of the circle and the centroid layer contains the centroid of the LEC.

Figure 18 displays the results for finding a Largest Empty Circle from all nuclear power plants in Europe from 2023. This spatial query used World Nuclear Power Plant 2023 [54] dataset which contains all nuclear power plants around the world in 2023. To compute the LEC from just the power plants in Europe, Range Search had to be executed first to collect all the points located in Europe. In Figure 18, the blue dots represent the existing nuclear power plants, the red dotted circle refers to the Largest Empty Circle, and the blue cross refers to the centroid of the LEC and the location where a new potential nuclear power plant can be built. This Figure shows a perfect example as why using LEC is a good option for urban planning.

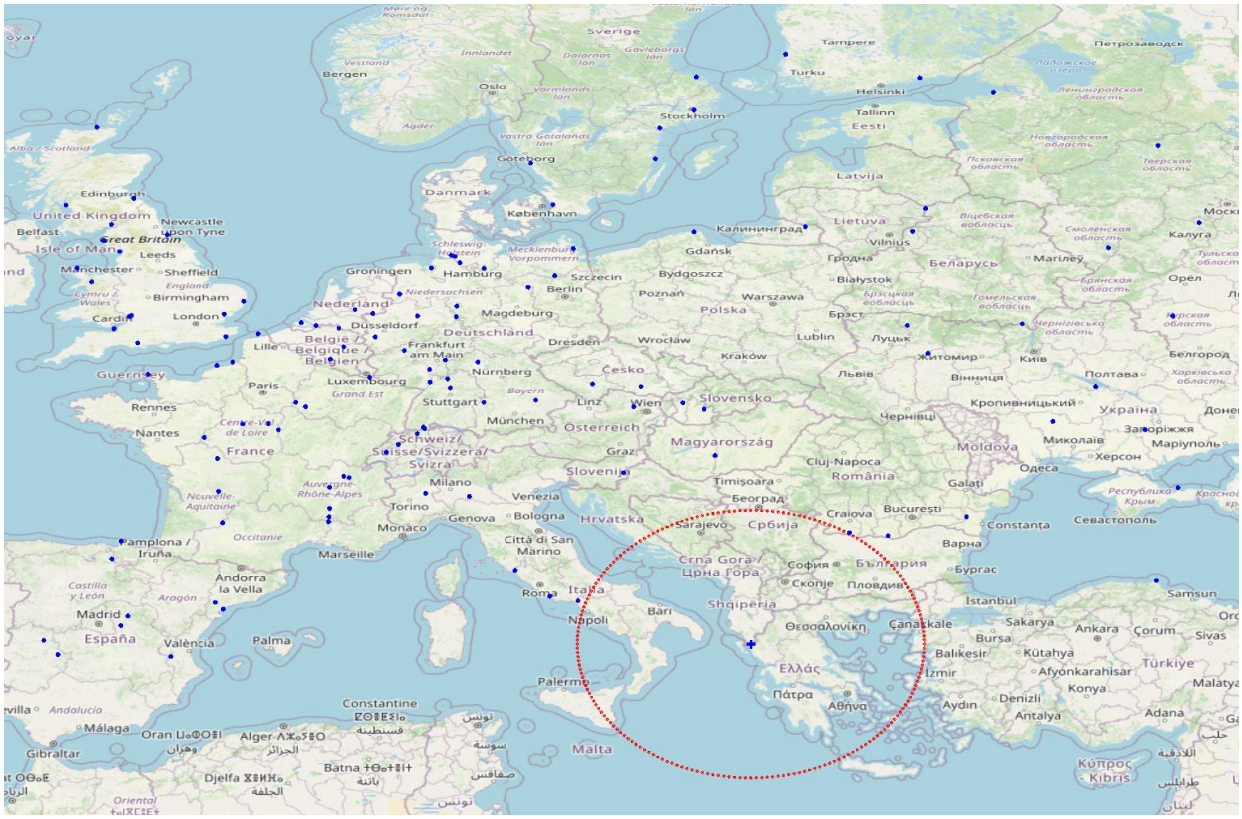


Figure 18: Map visualizing spatial GIS query execution using Largest Empty Circle.

4.2.4 *Euclidean Shortest Path*

For this Computational Geometry problem, A. Potturi [50] has already implemented a MASS application for ESP which we tried using as a reference for this implementation. However, this implementation was not suitable for the MASS-based GIS system. The approach would calculate the shortest distances between the points every time it was executed by using agent-propagation, which might make the spatial queries using ESP slow for larger datasets. However, as the MASS-based GIS system can save data, we thought of saving the shortest distances. This was not effective as the shortest distances saved would only be from a specific source point and would restrict users from computing ESP from other points. To take advantage of the MASS-based GIS system, this Euclidean Shortest Path MASS implementation needed another approach.

The approach taken was similar to the Euclidean Shortest Path MPI implementation. We created a visibility graph and used the Dijkstra's path finding algorithm to find the shortest path between a source and destination point. In this approach, the visibility graph is saved to the GIS database once the graph is created. This approach allows the users to query the shortest path between new start and destination points using the existing visibility graph which lowers the execution time considerably if a spatial query using ESP has been executed before with the same dataset. Figure 19 shows a GIS map that this ESP MASS implementation generates, illustrating the shortest path from Aden, Yemen, to Kuala Lumpur, Malaysia. The dataset this example execution used was National Oceanic and Atmospheric Administration's (NOAA) shoreline dataset [55]. The shorelines are constructed from hierarchically arranged closed polygons. In Figure 19, the blue dots refer to visibility graph vertices that create the shortest path and the red line represents the actual shortest path, avoiding obstacles.

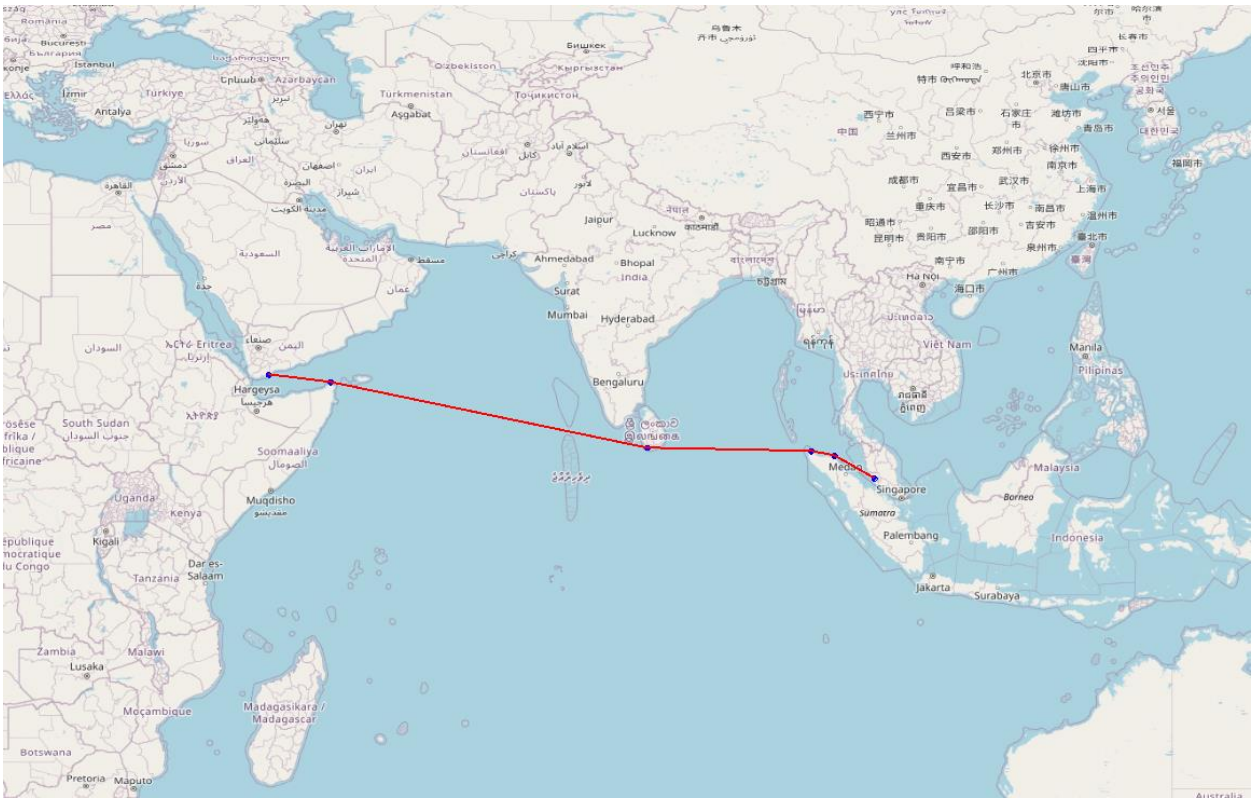


Figure 19: Map visualizing spatial GIS query execution using Euclidean Shortest Path.

The implementation reads obstacle points from a shapefile and creates a list of polygons objects. Each polygon object has a list of its corner points. If the dataset used has singular points instead, a list of points is created. The read data including the source point and the destination point are partitioned into subsets of points corresponding to the number of MASS Places to distribute the workload. The Places construct visibility graphs with their subset of points with the same naïve algorithm which is bound to $O(N^3)$. Once the visibility graphs are constructed, these graphs are returned to the master node to be combined into one big visibility graph. The final step for calculating the shortest path is to use the Dijkstra's algorithm on the combined visibility graph. Lastly, the implementation returns the shortest distance and the points creating this path which can be visualized with a GIS map.

Chapter 5. RESULTS

This section presents an analysis of benchmarks conducted on the MPI and MASS implementations. It discusses the execution performance in terms of CPU scalability and spatial scalability. At the end, this section offers a programmability comparison between the MPI and the MASS implementations.

5.1 BENCHMARKING PROCESS

The benchmarks were carried out on University of Washington Bothell's HERMES cluster. This benchmarking used 20 computing nodes from the cluster which are 64-bit Linux servers connected to a central filesystem. Detailed information about these computing nodes is available in Table 1.

Table 1: Benchmarking environment

Number of Computing Nodes	Logical Cores	CPU Model	Memory
3	4	Intel Xeon 5150 @ 2.66GHz	16 GB
4	8	Intel Xeon E5410 @ 2.33GHz	16 GB
5	4	Intel Xeon Gold 5220R @ 2.20GHz	16 GB
8	4	AMD EPYC 7252 @ 3.10GHz	16 GB

The evaluations of the computational geometry implementations with MPI and MASS utilized a diverse range of GIS datasets. Table 2 shows which dataset was used for which computational geometry problem. The datasets are in .csv format which made it possible to use the datasets for both MPI and MASS implementations. As the MPI implementations did not have support for GIS functionalities, the implementations could not read shapefiles. Using csv files allowed us to perform more accurate comparisons. To evaluate the spatial complexity for LEC, a dataset of

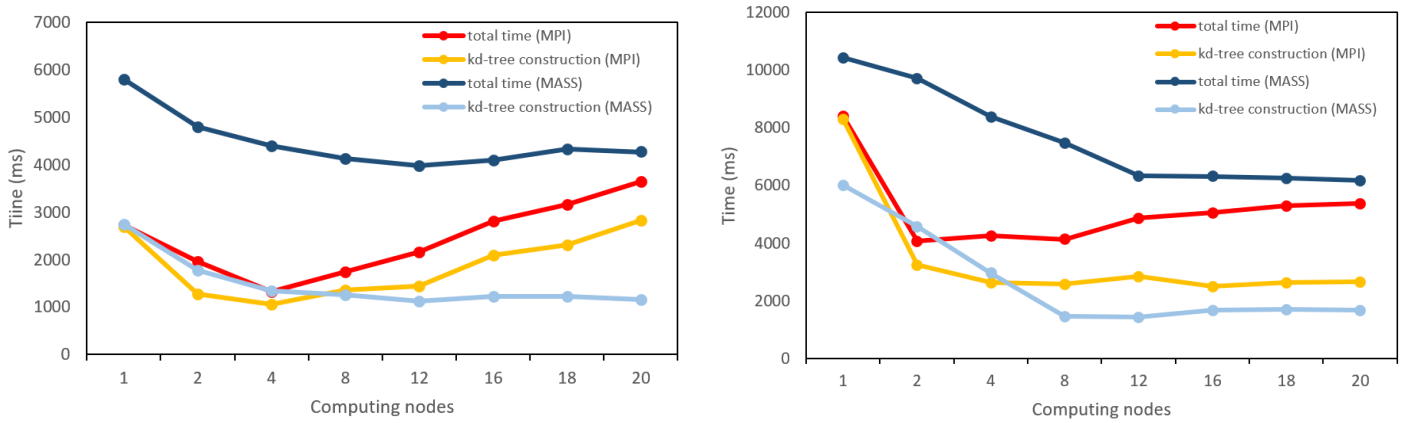
50,000 randomized coordinates is used. We could not find a GIS dataset in .csv format that contained obstacles as points for benchmarking the ESP implementations. Thus, we generated a dataset containing 500 polygons with randomized number of corners ranging from 4 to 8.

Table 2: Benchmarking Datasets

Dataset	Size	Computational Geometry problems
Crime Locations in LA, US [51]	938,458 points	Range Search, Convex Hull
National USFS Fire Occurrence [52]	581,541 points	Range Search, Convex Hull
US Private School Locations [53]	22,346 points	Largest Empty Circle
Randomized spatial points	50,000 points	Largest Empty Circle
Randomized 300 polygons with 4 to 6 vertices	1,200 – 1,800 points	Euclidean Shortest Path
Randomized 500 polygons with 4 to 6 vertices	2,000 – 3,000 points	Euclidean Shortest Path

Benchmarking the applications primarily focused on CPU scalability and spatial scalability. These benchmarks assessed how the implemented computational geometry applications performed with varying sizes of data. In this scenario, CPU scalability refers to how effectively can an application leverage multiple computing nodes. As for spatial scalability, it refers to how well an application can handle increasing sizes of spatial data. The execution times do not include the time taken to generate the GIS maps as the MPI implementations do not have this functionality implemented. All the execution times are an average of three runs.

5.2 EXECUTION PERFORMANCE



a) 581,541 points

b) 938,458 points

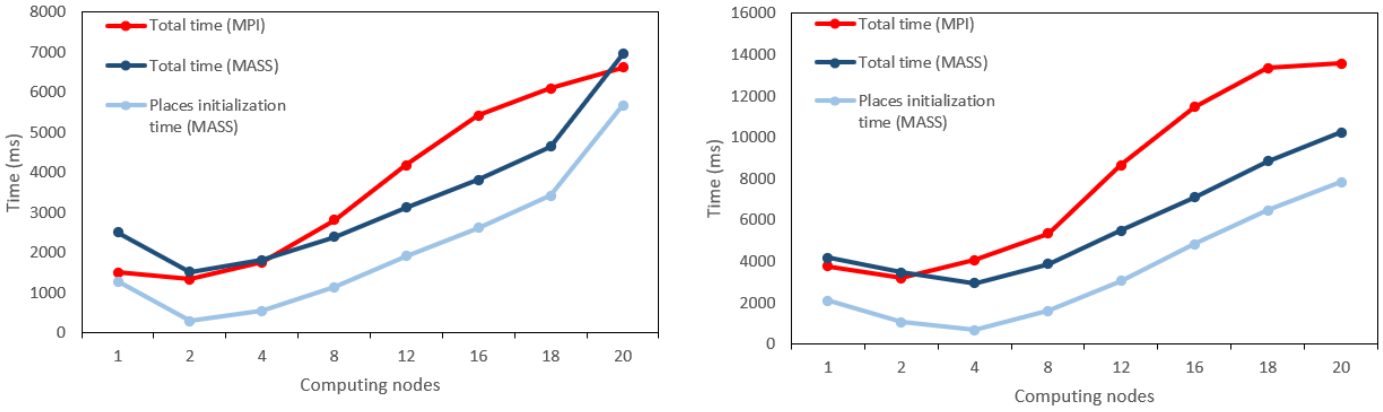
Figure 20: Range Search execution performance of MASS and MPI.

For RS benchmarking, the spatial query executed with the Fire occurrences dataset was: “Fire occurrences in Yellowstone national park”. With the larger dataset, the spatial query was: “Crime occurrences in Hollywood area”. Figure 20 showcases the execution performance of MASS and MPI for range search across two different sized datasets, provides insights into CPU scalability and spatial scalability. Both implementations visualize two metrics: kd-tree construction time and total time. Kd-tree construction time is the average time taken for the kd-trees to be constructed. The total time is the time taken to solve the computational geometry problem which excludes IO operations.

In general, MPI outperforms MASS as more computing nodes are used. However, MASS shows a decreasing trend during the whole benchmarking process, indicating good CPU scalability. In Figure 20 a), the most significant increase in performance is between moving from 1 to 4 nodes. With the larger dataset MASS started stabilizing after 12 nodes. This is because of

the communication overhead. Once the agents have queried the kd-trees for points which are in range, the agents return the points to the master node. As for the spatial scalability, MASS demonstrates good spatial scalability when the larger dataset is used. Kd-tree construction time for MASS starts outperforming MPI's kd-tree construction time after 7 nodes. In the previous RS MASS implementation, the kd-tree construction was the bottleneck, which is not the case anymore with the current implementation. This results in better CPU scalability and spatial scalability than the previous RS implementation.

MPI on the other hand, displays good CPU scalability in the beginning for both kd-tree construction time and total time but then the performance starts decreasing for both time metrics. In Figure 20 a), this behavior is noticed after 4 nodes for kd-tree construction time and 12 nodes for the total time. In Figure 20 b), both metrics for MPI start stabilizing after 2 nodes. This is also because of communication overhead. The master node calls "MPI_gather" which gathers all the result points from all the worker nodes. However, each computing node might have a different amount of result points, making each message size different. For spatial scalability, MPI maintains its performance advantage as the size of the dataset increases.



a) 581,541 points

b) 938,458 points

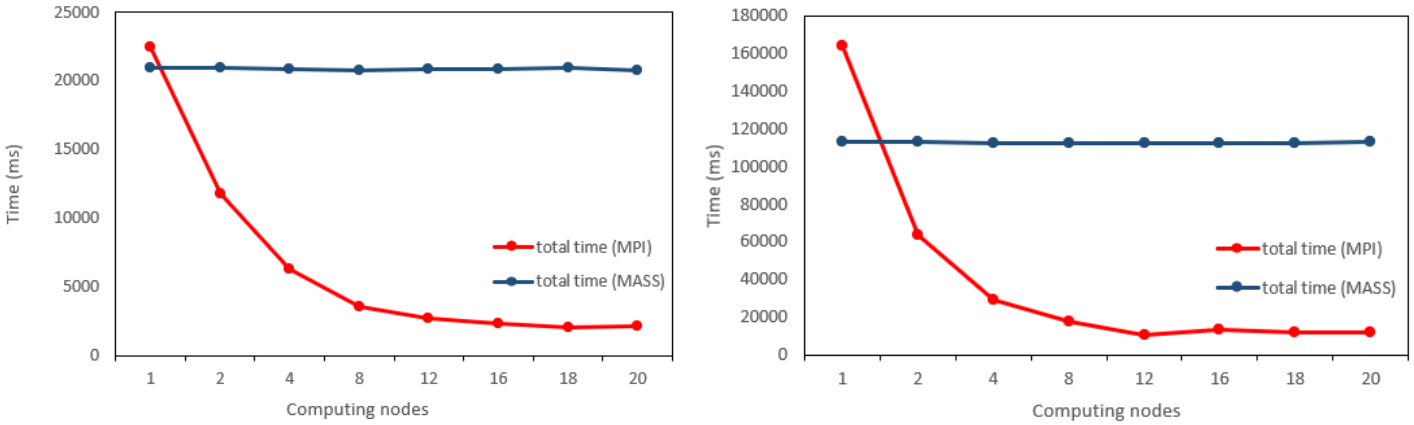
Figure 21: Convex Hull execution performance of MASS and MPI.

For benchmarking the CH applications, spatial queries are executed on the whole datasets. In a real-world scenario, this might not be the case as the spatial locations might be distributed to a large area. Point locations might be first gathered from a specific for example with RS and then CH would be used to query the filtered points. To get accurate benchmarks, all the points were used in both datasets. The three measurements in time are observed during the benchmarking process: total time for MPI, total time for MASS, and time it took for Places to be created and initialized in the MASS implementation.

Figure 21 indicates that MASS outperforms MPI for the most part. MASS demonstrates promising results from nodes 1 through 2 with the smaller dataset and 1 through 4. After that, the execution times start increasing. From both figures, we can observe that the Places initialization cost in time follows the same trend as the total time for MASS. Another observation is that initializing places take most of the time in the MASS implementation. The time difference between these two metrics includes the agent propagation for finding the outer hull points and the Monotone chain algorithm to remove the outlier points. The reason for the Places initialization taking the

most time is most likely because, each place iterates through every data point and collects the points which fall under the specific place's portion of the overall space. Regarding spatial scalability, MASS seems to support better spatial scalability than MPI by managing the larger dataset more effectively as seen in the execution times. Compared to a Convex Hull implementation from another MASS application which was used to compare different agent-based models in terms of performance and programmability [50], this implementation shows better execution performance with larger datasets, however this implementation can be improved further.

MPI's execution time is constantly higher than MASS's execution time with both datasets. This is due to needing multiple communicative calls. As MPI implantation uses a divide and conquer approach, the partial convex hulls in each computing node are merged by sending and receiving the partial hulls from the nodes. At every step, the partial convex hulls are merged in sets of two until one final convex hull remains. As more computing nodes are introduced, more partial hulls need to be merged, meaning more communication calls are needed.



a) 22,346 points

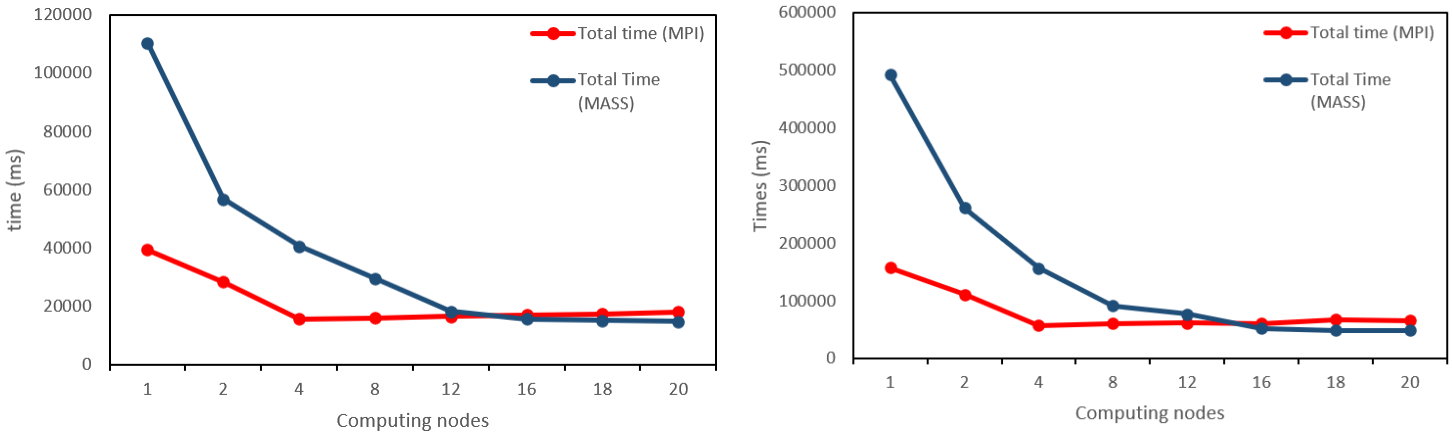
b) 50,000 points

Figure 22: Largest Empty Circle execution performance of MASS and MPI.

For benchmarking the Largest Empty Circle problem, only one time metric is considered, the total computation time excluding IO operations. From Figure 22 we can observe MASS does not show CPU scalability at all, and the execution times stay constant with all the computing nodes. This behavior is most likely due to the datapoints not being assigned correctly to the other computing nodes, leading to the application running only on one computing node. However, with one computing node, MASS is faster than MPI. Figure 22 b) shows the difference more clearly. MASS shows a slight performance improvement. in Figure 22 a) and b) which is due to more places being used when the number of computing nodes was used. The places are assigned to threads, leading to a small performance gain. As for spatial scalability, the implementation can be executed with larger datasets.

The MPI implementation shows great CPU scalability until 16 nodes with the smaller dataset and 12 nodes with the larger dataset. After that, MPI demonstrates the performance stabilizing up to 20 nodes. The stabilization in execution performance is because communication overhead. The

more computing nodes are utilized, the more partitions of Voronoi edges are created. Some of these might not have intersections with the Convex Hull edges which leads to an imbalance number of potential centroids for the Largest Empty Circle. As master node tries to receive the potential LEC, there might exist computing nodes which have not completed the computations. As the MPI implementation uses blocking communication calls, the receive buffer cannot be used until the receive buffer contains the intended message. In addition, MPI showed good spatial scalability by demonstrating a similar trend with execution performance in Figure 22 b).



a) 1,800 points

b) 3,000 points

Figure 23: Euclidean Shortest Path execution performance of MASS and MPI.

Figure 23 a) and b) demonstrate how well MASS can distribute computational tasks effectively across the cluster. The trend highlights MASS's great CPU scalability by showing a decrease in execution time as the number of nodes are increased.

With the smaller dataset both MASS and MPI start with high execution times with one node. However, as more computing nodes are utilized, MASS shows substantial improvement in execution time and caught up to MPI after 12 nodes. While MPI also expresses CPU scalability,

the decrease in execution time is not as substantial. The execution time starts stabilizing for MPI after four nodes.

Figure 23 b) describes the pattern better. MASS starts with a very high execution time, but the performance improves as more nodes are used. MPI demonstrates a similar trend to MASS, but it is not as efficient as MASS. In addition, Figure 23 b) displays that MASS can handle larger datasets efficiently while maintaining good execution performance indicating the implementation's great spatial scalability.

5.3 Programmability Comparison

We assessed the programmability of the MASS and MPI frameworks by analyzing their implementation complexities and overheads for various computational tasks. The goal was to clarify the trade-offs between development effort and maintainability within these frameworks when applied to algorithms such as range search (RS), convex hull (CH), largest empty circle (LES), and Euclidean shortest path (ESP). We investigated four metrics in this comparison: Lines of Code (LoC), Cyclomatic Complexity, Boilerplate Code, and Boilerplate %. LoC refers to the number of lines of code in the application excluding comments, blank spaces, and non-executable lines. Cyclomatic Complexity indicates the complexity of the program. Boilerplate code is code that is repeated in multiple places with little to no variation. Lastly, Boilerplate % refers to the ratio of how much from Lines of Code is boilerplate code.

Table 3: Programmability comparison: Range Search

	LoC	Cyclomatic Complexity	Boilerplate Code	Boilerplate %
MPI	233	3.1	20	8.5%
MASS	368	2.6	15	4.1%

From Table 3 we can see that MASS implementation has more Lines of Code with 368 compared to MPI's 233. This is because of the GIS functionalities such as generating the map and for MASS, Agents class and Places class were created which adds more code. However, MASS demonstrates a lower Cyclomatic Complexity of 2.6 than MPI's 31. MPI needs to have additional if statements for master node for example to create receive buffers to gather the points in range. For boilerplate code MPI has more than MASS resulting in a higher boilerplate percentage of 8.5% compared to 4.1%.

Table 4: Programmability comparison: Convex Hull

	LOC	Cyclomatic Complexity	Boilerplate Code	Boilerplate %
MPI	316	4.2	28	8.8%
MASS	710	3.4	27	3.8%

For convex Hull, the MPI implementation had less LoC compared to MASS as shown in Table 4. As MPI used the divide and conquer approach, not many lines of code were needed. Since MASS used an agent propagation approach to gather the outer hull points, Places class and Agents class needed many more lines of code. However, MPI had a higher Cyclomatic Complexity of 4.2 compared to MASS's 3.4, making the MASS implementation more maintainable. The MPI's merge for loop and the additional if statement the master and worker nodes add complexity to the implementation. As for boilerplate code, both implementations have roughly the same amount.

However, since MASS implementation is twice as long, the boilerplate percentage is much higher for MPI (8.8% to 3.8%).

Table 5: Programmability comparison: Largest Empty Circle

	LOC	Cyclomatic Complexity	Boilerplate Code	Boilerplate %
MPI	612	3.5	32	5.2%
MASS	767	3.1	19	2.5%

Table 5 shows that MASS had more lines of code than MPI even though both implementations were quite long. The reason is the same as for the RS implementations. Both used the same approach. However, MASS needed to implement a Place class resulting in having more code. Nonetheless, MPI had a higher Cyclomatic Complexity of 3.5 than MASS's 3.1. In the MPI implementation's main file, there are a lot of if statements for the different ranks and multiple for loops MPI communication. Similarly, to the other MASS implementations, LEC MASS has less boilerplate code than MPI (19 to 32) resulting in 2.5% MPI's boilerplate percentage and 5.2% for MPI.

Table 6: Programmability comparison: Euclidean Shortest Path

	LOC	Cyclomatic Complexity	Boilerplate Code	Boilerplate %
MPI	523	3.1	25	4.7%
MASS	692	4.1	35	5.1%

ESP is the only application where MASS had a higher Cyclomatic Complexity as shown in Table 6. Unlike the MPI implementation, MASS had the implementation where the user can use an existing visibility graph to find the shortest path between the source and destination node. This

implementation added more lines of code and boilerplate code, thus, MASS having a slightly higher boilerplate percentage of 5.1% compared to MPI's 4.7%.

5.4 DISCUSSIONS

The large-scale benchmarking and programmability comparison revealed several strengths and weaknesses for the computational geometry implementations using both MASS and MPI. These observations help will help us to identify potential improvements.

In general, MASS showed needing less boilerplate code and having better Cyclomatic Complexity than MPI, signifying better maintainability. These aspects make MASS more appealing for complex systems. However, MASS faced inconsistencies with the execution performance during benchmarking. These inconsistencies come from the computational geometry problems themselves and the agent-based approaches. For example, Range Search utilizes kd-trees and agents querying the trees which demonstrated good CPU and spatial scalability. In contrast, Convex Hull used agent propagation to identify outer hull points which showed to be more efficient than the divide and conquer approach for the MPI implementation. The other two MASS implementations did not use agents, leading to variations in performance. For the Largest Empty Circle MASS implementation, issues with data partitioning led to constant execution times, regardless of the number of computing nodes used. In contrast, the Euclidean Shortest Path implementation showed good CPU scalability and spatial scalability, but the execution times were high due to the naïve approach for the visibility graph constructions.

MPI demonstrated strong performance with smaller datasets and fewer nodes. Once the complexity and the data size increased, MPI started facing issues and the execution performance became inefficient. This brings out MPI's low-level control, while having strong initial

performance but as more complexity is introduced, synchronization and communication issues start to appear.

In terms of enhancing the computational geometry MASS implementations, various potential improvements can be carried out.

- Range Search: By extending the functionality to support 3D-space can enhance the range searching capabilities of the MASS-based GIS system and more complex spatial queries can be executed.
- Convex Hull: Improving the initialization process for Places can reduce the setup time and improve the execution times altogether.
- Largest Empty Circle: Correctly implementing the data partitioning across the cluster will allow us to make full use of MASS's capabilities. Additionally, integrating the incomplete agent propagation approach can potentially improve the LEC implementation.
- Euclidean Shortest Path: Implementing a more efficient algorithm for the visibility graph creation such as the Lee's Visibility Algorithm [45] which is bound to $O(N^2 \log N)$ can potentially reduce the overall time complexity of the ESP implementation and be used with larger datasets.

Chapter 6. CONCLUSIONS

6.1 OVERVIEW

This project was set out to enhance the existing integration of Multi-Agent Spatial Simulation (MASS) with Geographic Information Systems (GIS), primarily focusing on improving the existing and implementing new computational geometry problems to optimize spatial queries. This approach included conducting a comprehensive analysis of the existing MASS-based GIS system, implementing, and benchmarking computational geometry problems using Message Passing Interface (MPI) and MASS library, and carrying out a detailed programmability comparison to determine the implementation complexities and their relative efficiencies.

Our results reveal that the reimplemented Range Search showed significant improvements in execution time and scalability compared to the previous MASS implementation and the MPI baseline. The use of multiple kd-trees allowed for efficient querying and demonstrated good performance as data size and computing nodes increased. The Convex Hull implementation using MASS outperformed the MPI implementation in most cases. The implementation showed promising results by leveraging agent propagation to identify the outer hull points. However, the initialization time for places was a bottleneck. Despite this, the MASS implementation showed better spatial scalability with larger datasets than MPI. The Largest Empty Circle using MASS beat the MPI implementation using one computing node but was then outperformed by MPI and it did not show the expected improvements due to issues with data partitioning. The performance remained constant, indicating the need for better data partitioning. Euclidean Shortest Path was inefficient with large datasets. Despite that, the MASS implementation showed better CPU and spatial scalability than MPI, improving execution performance as more computing nodes were used. However, the naive algorithm's inefficiency limited the overall performance, emphasizing

the need for a more sophisticated approach. Lastly, the programmability comparison revealed that MASS applications require more lines of code in general due to the framework. However, MASS showed lower Cyclomatic Complexity and boilerplate percentage which indicates MASS having better maintainability and readability.

While the project achieved its main objectives, multiple limitations were encountered:

- Our approach only focused on vector data and excluded raster data. This was because the previous computational geometry problems utilized by spatial queries were intended to be used with vector data.
- The initialization of Places in the Convex Hull implementation took most of the execution time. This significantly impacted on the overall performance and needs optimization to reduce setup time and improve execution efficiency.
- The agent propagation implementation for Largest Empty Circle was incomplete which made us change our approach. In addition, partitioning the data partitioning faced issues which led to inefficient use of MASS and the implementation running on just one machine.
- The Euclidean Shortest Path implementation with naïve visibility graph construction algorithm proved to be too inefficient. This limited us to using very small datasets.

6.2 FUTURE WORK

Future works should address the limitations listed above and furthermore focus on enhancing computational geometry problems to further enhance the performance of the MASS-based GIS system. For Range Search, adding functionality to query in 3D space and adding the option to include additional filtering based on geographic data's attributes will broaden its applicability. In the Convex Hull implementation, optimizing the agent movement while agent-migration to

minimize capturing non-outer hull points will reduce execution performance. Also, adding new map layers that might suit different GIS queries will enhance the versatility and functionality of the system.

BIBLIOGRAPHY

[1] Maguire, D. J. (1991). An overview and definition of GIS. Geographical information systems: Principles and applications, 1(1), 9-20.

[2] Savinykh, V. P., & Tsvetkov, V. Y. (2014). Geodata as a systemic information resource. Herald of the Russian Academy of Sciences, 84(5), 365-368.

[3] Huang, F., Liu, D., Liu, P., Wang, S., Zeng, Y., Li, G., ... & Pang, L. (2007, August). Research on cluster-based parallel GIS with the example of parallelization on GRASS GIS. In Sixth International Conference on Grid and Cooperative Computing (GCC 2007) (pp. 642-649). IEEE.

[4] Wang, J., Xiong, J., Yang, K., Peng, S., & Xu, Q. (2010, June). Use of GIS and agent-based modeling to simulate the spread of influenza. In 2010 18th International Conference on Geoinformatics (pp. 1-6). IEEE.

[5] Carver, S., & Quincey, D. (2016). A conceptual framework of volcanic evacuation simulation of Merapi using agent-based model and GIS. Procedia-Social and Behavioral Sciences, 227, 402-409.

[6] Crooks, A., Castle, C., & Batty, M. (2008). Key challenges in agent-based modelling for geo-spatial simulation. Computers, Environment and Urban Systems, 32(6), 417-430.

[7] Emau, J., Chuang, T., & Fukuda, M. (2011, August). A multi-process library for multi-agent and spatial simulation. In Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (pp. 369-375). IEEE.

[8] Preparata, F. P., & Shamos, M. I. (2012). Computational geometry: an introduction. Springer Science & Business Media.

[9] Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. Parallel computing, 22(6), 789-828.

[10] Woodring, J., Sell, M., Fukuda, M., Asuncion, H., & Salathe, E. (2017, June). A multi-agent parallel approach to analyzing large climate data sets. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS) (pp. 1639-1648). IEEE.

[11] Sieling, M. (2022, June). AGENT-BASED DATABASE WITH GIS. Accessed on: June 4, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/.

[12] GeoTools. Accessed on: June 4, 2024. [Online]. Available: https://www.geotools.org/.

[13] Raghavendra, S. P. (2023, June). Agent-based GIS Queries. Accessed on: June 4, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/.

[14] Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. The VLDB Journal, 15, 121-142.

[15] Ge, Q., Wang, H. T., & Zhu, H. (2006). An improved algorithm for finding the closest pair of points. Journal of computer Science and Technology, 21(1), 27-31.

[16] Kakde, H. M. (2005, August). Range Searching using Kd Tree. Accessed on: June 4, 2024. [Online]. Available: https://users.cs.utah.edu/~lifeifei/cs6931/kdtree.pdf/.

[17] Zhong, C., Malinen, M., Miao, D., & Fränti, P. (2015). A fast minimum spanning tree algorithm based on K-means. Information Sciences, 295, 1-17.

[18] Cui, Y. (2022, December). Agent-based Graph Applications in MASS Java and Comparison with Spark. Accessed on: June 4, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/.

[19] Barber, C. B., Dobkin, D. P., & Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software (TOMS), 22(4), 469-483.

[20] Schuster, M. (2008). The largest empty circle problem. In Proceedings of the Class of 2008 Senior Conference, Computer Science Department, Swarthmore College (pp. 28-37).

[21] Kapoor, S., & Maheshwari, S. N. (1988, January). Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles. In Proceedings of the fourth annual symposium on computational geometry (pp. 172-182).

[22] Netlogo. Accessed on: June 4, 2024. [Online]. Available: https://ccl.northwestern.edu/netlogo/.

[23] Walker, B., & T. Johnson. (2019, March). NetLogo and GIS: A Powerful Combination. EPiC Series in Computing (pp. 257-264).

[24] Malik, A., & Abdalla, R. (2017). Agent-based modelling for urban sprawl in the region of Waterloo, Ontario, Canada. Model Earth Syst Environ 3 (1): 7.

[25] Crooks, A. T. (2007). The Repast Simulation/Modelling System for Geospatial Simulation. Tech. Rep. Paper 123, Centre for Advanced Spatial Analysis, University College London.

[26] Gorur, B. K., Imre, K., Oguztuzun, H., & Yilmaz, L. (2016, April). Repast HPC with optimistic time management. In Proceedings of the 24th High Performance Computing Symposium (pp. 1-9).

[27] Taillandier, P., & Drogoul, A. (2010). From GIS data to GIS agents, modeling with the GAMA simulation platform. In Technical Forum Group on Agent and Multi-agent-based Simulation: 1st meeting collocated with Eumas (Vol. 10).

[28] Raab, R., Lenger, K., Stickler, D., Granigg, W., & Lichtenegger, K. (2022, May). An Initial Comparison of Selected Agent-Based Simulation Tools in the Context of Industrial Health and Safety Management. In Proceedings of the 2022 8th International Conference on Computer Technology Applications (pp. 106-112).

[29] Fabri, A., & Pion, S. (2009, November). CGAL: The computational geometry algorithms library. In Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems (pp. 538-539).

[30] Wang, Y., Yu, S., Dhulipala, L., Gu, Y., & Shun, J. (2022, April). ParGeo: a library for parallel computational geometry. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 450-452).

[31] Ghodsi, M., & Sharifzadeh, M. (2003, July). ParLeda: A Library for Parallel Processing in Computational Geometry Applications. International Journal of Engineering, Volume 16, No. 2, pp. 123-132.

[32] OpenMP. Accessed on: June 4, 2024. [Online]. Available: https://www.openmp.org/.

[33] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., & Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. ACM SigPlan Notices, 30(8), 207-216.

[34] Blelloch, G. E., Anderson, D., & Dhulipala, L. (2020, July). ParlayLib-a toolkit for parallel algorithms on shared-memory multicore machines. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (pp. 507-509).

[35] Kakde, H. M. (2005, August). Range Searching using Kd tree. Accessed on: June 4, 2024. [Online]. Available: https://users.cs.utah.edu/~lifeifei/cs6931/kdtree.pdf/.

[36] Beckmann, N., Kriegel, H. P., Schneider, R., & Seeger, B. (1990, May). The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD international conference on Management of data (pp. 322-331).

[37] Tobler, W., & Chen, Z. T. (1986). A quadtree for global information storage. Geographical Analysis, 18(4), 360-371.

[38] Martínez, C., & Roura, S. (2001). Optimal sampling strategies in quicksort and quickselect. SIAM Journal on Computing, 31(3), 683-705.

[39] Kingsford, C. kd- Trees. Accessed on: June 4, 2024. [Online]. Available: https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf/.

[40] Wibowo, A., Santoso, H. B., Rachmat, C. A., & Delima, R. (2019, August). Mapping and Grouping of Farm Land with Graham Scan Algorithm on Convex Hull Method. In 2019 International Conference on Sustainable Engineering and Creative Computing (ICSECC) (pp. 121-126). IEEE.

[41] Alzubaidi, A. M. N. (2014, September). Minimum Bounding Containers of 2D Convex Polygon. European Academic Research Vol. II, Issue 6.

[42] Heineman, G. T., Pollice, G., & Selkow, S. (2016, March). Algorithms in a Nutshell: A Practical Guide. " O'Reilly Media, Inc.".

[43] Jin, L., Kim, D., Mu, L., Kim, D. S., & Hu, S. M. (2006). A sweepline algorithm for Euclidean Voronoi diagram of circles. Computer-Aided Design, 38(3), 260-272.

[44] Nguyet, T. T. N., Van Hoai, T., & Thi, N. A. (2011, October). Some advanced techniques in reducing time for path planning based on visibility graph. In 2011 Third International Conference on Knowledge and Systems Engineering (pp. 190-194). IEEE.

[45] Coleman, D. (2012). Lee's O(n2 log n) Visibility Graph Algorithm: Implementation and Analysis. Department of Computer Science, University of Colorado at Boulder: Boulder, CO, USA.

[46] Dijkstra, E. W. (2022). A note on two problems in connexion with graphs. In Edsger Wybe Dijkstra: His Life, Work, and Legacy (pp. 287-290).

[47] What is OpenStreetMap. OpenStreetMap. Accessed on: June 4, 2024. [Online]. Available: https://welcome.openstreetmap.org/what-is-openstreetmap/.

[48] Mineral Resources Data System (MRDS). U.S. Geological Survey. Accessed on: June 4, 2024. [Online]. Available: https://mrdata.usgs.gov/mrds/.

[49] Saadati, S., & Razzazi, M. (2022, December). Natural way of solving a convex hull problem. Accessed on: June 4, 2024. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/2212/2212.11999.pdf/.

[50] Potturi, A. (2023, June). Programmability and Performance Analysis of Distributed and Agent-Based Frameworks. Accessed on: June 4, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/.

[51] Crime Data from 2020 to Present. Los Angeles Open Data Portal. Accessed on: June 4, 2024. [Online]. Available: https://data.lacity.org/Public-Safety/Crime-Data-from-2020-to-Present/2nrs-mtv8/about_data/.

[52] National USFS Fire Occurrence Point (Feature Layer). U.S. Forest Service - Geospatial Data Discovery. Accessed on: June 4, 2024. [Online]. Available: https://data-usfs.hub.arcgis.com/datasets/6059c1a4dca749d393e33ee5f8a0cbaf_9/about/.

[53] Private School Locations – Current. ArcGIS Hub. Accessed on: June 4, 2024. [Online]. Available: https://hub.arcgis.com/datasets/nces::private-school-locations-current/explore/.

[54] World Nuclear Power Plant 2023. ArcGIS Hub. Accessed on: June 4, 2024. [Online]. Available: https://hub.arcgis.com/datasets/esriindia1::world-nuclear-power-plant-2023/about/.

[55] Shoreline / Coastline Resources. National Centers for Environmental Information (NOAA). Accessed on: June 4, 2024. [Online]. Available: https://www.ngdc.noaa.gov/mgg/shorelines/.

APPENDIX

A Range Search Benchmarking Results

Table 7: Range Search execution performance with 581,541 points (milliseconds)

Computing Nodes	Total time (MPI)	Kd-tree construction (MPI)	Total time (MASS)	Kd-tree construction (MASS)
1	2737	2692	5813	2742
2	1968	1280	4801	1772
4	1322	1059	4408	1338
8	1737	1361	4140	1260
12	2164	1440	3985	1119
16	2812	2088	4102	1228
18	3160	2305	4329	1220
20	3651	2830	4276	1162

Table 8: Range Search execution performance with 938,458 points (milliseconds)

Computing Nodes	Total time (MPI)	Kd-tree construction (MPI)	Total time (MASS)	Kd-tree construction (MASS)
1	8406	8306	10438	6014
2	4082	3248	9710	4577
4	4273	2644	8394	2969
8	4141	2576	7476	1454
12	4878	2840	6342	1440
16	5053	2492	6324	1684
18	5295	2631	6267	1694
20	5372	2673	6168	1674

B Convex Hull Benchmarking Results

Table 9: Convex Hull execution performance with 581,541 points (milliseconds)

Computing Nodes	Total time (MPI)	Total time (MASS)	Places initialization time (MASS)
1	1508	2511	1275
2	1330	1519	287
4	1754	1814	543
8	2818	2390	1142
12	4182	3124	1908
16	5425	3813	2623
18	6102	4652	3428
20	6628	6955	5673

Table 10: Convex Hull execution performance with 938,458 points (milliseconds)

Computing Nodes	Total time (MPI)	Total time (MASS)	Places initialization time (MASS)
1	3762	4169	2094
2	3197	3474	1056
4	4055	2952	673
8	5342	3870	1611
12	8660	5489	3047
16	11470	7110	4825
18	13357	8846	6479
20	13574	10251	7834

C Largest Empty Circle Benchmarking Results

Table 11: Largest Empty Circle execution performance with 22,346 points (milliseconds)

Computing Nodes	Total time (MPI)	Total time (MASS)
1	22498	20933
2	11747	20914
4	6307	20825
8	3592	20814
12	2708	20898
16	2325	20873
18	2079	20913
20	2144	20809

Table 12: Largest Empty Circle execution performance with 50,000 points (milliseconds)

Computing Nodes	total time (MPI)	total time (MASS)
1	163819	113027
2	63767	112991
4	29370	112389
8	17936	112318
12	10966	112577
16	13650	112232
18	12258	112228
20	11785	112801

D Euclidean Shortest Path Benchmarking Results

Table 13: Euclidean Shortest Path execution performance with 1,800 points (milliseconds)

Computing Nodes	Total time (MPI)	Total time (MASS)
1	39471	110414
2	28449	56817
4	15698	40728
8	16112	29643
12	16599	18395
16	17214	15762
18	17561	15261
20	18139	14926

Table 14: Euclidean Shortest Path execution performance with 3,000 points (milliseconds)

Computing Nodes	Total time (MPI)	Total time (MASS)
1	157770	492083
2	110970	261170
4	57738	157256
8	61398	91398
12	61691	76833
16	61331	52431
18	67658	49242
20	66435	49294

E Running GIS queries

To run GIS queries in the MASS-based GIS system, MASS library needs to be first installed. The steps are as follows.

1. Clone the MASS core library from mass_java_core BitBucket repository. The latest code is found in the "develop" branch.
2. Install the downloaded MASS library by navigating to the folder and using the command: *"mvn clean package install"*.
3. Clone the latest mass_java_appl BitBucket repository for MASS applications. The latest MASS-based GIS system is found in "shahruz/gis_improvements" branch.
4. Navigate to the MASS-based GIS folder located in path: "Applications/gis_database"
5. In pom.xml file, there are paths to GIS queries which use the four computational geometry MASS implementations. Uncomment the GIS query which should be executed. Only one GIS query should be uncommented. Figure 24 shows the GIS query path inside the pom.xml file.

```
<configuration>
    <archive>
        <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.rangesearch.RangeSearch</mainClass>
            <!-- <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.convexhull.ConvexHullDriver</mainClass> -->
            <!-- <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.largestemptycircle.Main</mainClass> -->
            <!-- <mainClass>edu.uw.bothell.css.dsl.mass.apps.gisdatabase.dataimport.euclideanshortestpath.EuclideanShortestPath</mainClass> -->
            <classpathPrefix>dependency-jars/</classpathPrefix>
        </manifest>

    </archive>
</configuration>
```

Figure 24: Pom.xml configuration file for MASS-based GIS.

6. Build the chosen GIS query in "gis_database" folder using the command: "*mvn package*". This will create a "target" folder.
7. Go to the "target" folder and create a nodes.xml file to define the computing nodes. Instructions for creating this file are found in the MASS Java Developers Guide. Figure 25 shows a sample nodes.xml file.

```
<nodes>
        <node>
                <master>true</master>

                <!-- Hostname is required, and may either be an FQDN or IP address -->
                <!-- Make sure the name/address specified is reachable by all other nodes -->
                <hostname>hermes01.uwb.edu</hostname>

                <!-- Masshome is where MASS.jar and any other dependencies of your application reside -->
                <masshome>/home/NETID/mannans1/mass_java_appl/mass_java_appl/Applications/gis_database/target/</masshome>
                <port>29050</port>

        </node>
         <node>
                <hostname>hermes02.uwb.edu</hostname>
                <masshome>/home/NETID/mannans1/mass_java_appl/mass_java_appl/Applications/gis_database/target/</masshome>
                <username>mannans1</username>
                <privatekey>~/.ssh/id_rsa</privatekey>
                <port>29050</port>
        </node>
        <node>
                <hostname>hermes03.uwb.edu</hostname>
                <masshome>/home/NETID/mannans1/mass_java_appl/mass_java_appl/Applications/gis_database/target/</masshome>
                <username>mannans1</username>
                <privatekey>~/.ssh/id_rsa</privatekey>
                <port>29050</port>
        </node>
```

Figure 25: Sample nodes.xml file.

8. Before the GIS query can be executed, X11 variable needs to be defined because the GIS query will create a GIS map. The execution will otherwise throw an error. If the X11 variable is defined, at the end of an execution the GIS map should be generated automatically.
9. The general command to execute the queries are: *"java -jar gis_database-1.0-SNAPSHOT.jar <parameter> <parameter> …"*.
    - Each query has a different set of parameters

    - For RS the full command is: *"java -jar gis_database-1.0-SNAPSHOT.jar <number of places> <minX><maxX><minY><maxY>"*, where
        - The *"number of places"* is how many place elements are used in the execution.
        - *"minX", "maxX", "minY", "maxY"* parameters define the spatial boundary where X refers to latitude and Y refers to longitude.

    - For CH the full command is: *"java -jar gis_database-1.0-SNAPSHOT.jar <filename> <grid size>"*

- *"filename"* parameter refers to the shapefile which wants to be used from "gis_database/input" path.
- The *"grid size"* refers to the size of the grid of places.

- To execute GIS query using LEC, the command is: *"java -jar gis_database-1.0-SNAPSHOT.jar <number of places> <filename>"*. The parameters are already explained above.

- ESP is executed by using the command: *"java -jar gis_database-1.0-SNAPSHOT.jar <number of places> <start latitude> <start longitude> <destination latitude> <destination longitude> <filename>"*

    - The*"number of places"* is how many place elements are used in the execution.
    - *"start latitude", "start longitude", "destination latitude"*, and "*destination longitude"* parameters refer to the start and destination point coordinates.
    - *"filename"* parameter refers to the shapefile which wants to be used from "gis_database/input" path. This parameter is optional. If this parameter is provided, the query will create a new visibility graph from the input file and save this graph for further queries. If a "visibilityGraph.txt" file exists in "gis_database/outputs", the execution can be run without providing the "filename" parameter and the execution will use the existing visibility graph.

10. The datasets used in this project are located in "gis_database/input" folder which can be used for these queries. The outputs for the queries are created in "gis_database/outputs" which can also be used for further analysis.