AGENT-BASED COMPUTATIONAL GEOMETRY

Satine Paronyan

Term Report

Part of CSS 595: Capstone Project

Master of Science in Computer Science & Software Engineering

University of Washington Bothell

Autumn 2020

Committee:

Munehiro Fukuda, Chair

Michael Stiber, Member

Min Chen, Member

Table of Contents

Chapter 1: INTRODUCTION

The agent-based computational geometry research project explores the applicability of agent-based modeling to designing efficient parallel solutions to computational geometry problems. The computational geometry applications are computationally complex and involve large datasets. The four computational geometry problems to be solved by agent-based algorithms using the MASS library. The MASS based applications will be compared against MapReduce and Spark implementations. The selected computational geometry problems are Range Search, Point Location, Largest Empty Circle, and Euclidian Shortest Path (Obstacle Avoiding Path). A great number of O(n log n) efficient sequential algorithms have been designed to solve geometric problems [1][2]. However, the sequential algorithms are bound to one machine, which limits the amount of data being processed. Distributed memory addresses the scalability concerns [3]. Spatial scalability enables to analyze large datasets more efficiently.

Multi-Agent Spatial Simulation (MASS) [4] is a parallel computing library based on multi-agents that behave as a simulation on a given virtual space. The collective and emergent group behavior of agents such as propagation, colliding, and occasional repelling makes it easier to discover attributes of structured and geometric datasets. In addition, the MASS library provides an ability to initialize data with different types of data structures such as 2D/3D space or graph over distributed memory. The data structure remains unchanged in memory while mobile agents collaboratively find the solution. This approach enables us to design more intuitive algorithms that provide competitive execution performance and programmability. In comparison with MapReduce and Spark big-data parallelization tools [3][5], MASS is better suited to computational geometry algorithms.

Several static problems are implemented for the MASS library by Distributed System Laboratory (DSL) at the University of Washington Bothell. Agent-based computational Geometry capstone project expands the work on agent-based computational geometry by parallelizing four additional geometric problems listed at the beginning of this section.

Chapter 2: CURRENT STATUS

In the Spring 2020 quarter as part of CSS 600, I completed the design and implementation of the parallel algorithms for Range Search [6] problem utilizing MASS, Spark, and MapReduce. This quarter I worked on two other problems: Point Location and Largest Empty Circle [7][8]. I completed the design and implementation of the algorithms for these two computational geometry problems (see Table 1).

Implementation and testing completed
Implementation completed, testing in progress
To be implemented and tested

	Spark	**MapReduce**	**MASS**
Range Search	Complete, Tested (10k points)	Complete, Tested (10k points)	Complete, Tested (10k points)
Point Location	Complete, Tested (500k trapezoids)	Complete, Tested (500k trapezoids)	Complete, Tested (500k trapezoids)
Largest Empty Circle	Complete, Tested (500k points)	Complete, Tested (500k points)	Complete, Testing (500k points)
Euclidian Shortest Path	Winter 2020	Winter 2020	Winter 2020

Table 1: *Current Status and Winter 2020 plan*

Table 1 shows the current status of my capstone project. The programs that are fully completed, specifically implemented and tested, shown with status in green. The programs that are implemented but still need comprehensive testing with large input data shown in orange. The status of the program in blue notes that the program will be implemented and tested in the next quarter.

Point Location program is fully implemented and tested with 500k input points. The collected results are presented in the next Results section. Largest Empty Circle (LEC) program is implemented and partially tested. I conducted execution performance tests for MapReduce and Spark versions of the program, but still need to finish tests for the MASS version. We are facing some issues with running the MASS version of LEC on a cluster while it runs successfully on a single computing node. We are troubleshooting this issue to determine the root of the problem to fix it. Next quarter, I plan to design a parallelized algorithm for my last application - the Euclidian Shortest Path problem [9]. The implementation and testing of the application are also planned for the next Winter 2021 quarter.

Chapter 3: RESULTS

The implementation of algorithms for the four computational geometry problems will be measured by programmability and execution performance metrics.

Programmability includes:

- *Boilerplate code* – number of lines of code required to set up the environment
- *Lines of code* – total number of lines of code in the implementation
- *Number of classes* – total number of classes in the implementation
- *Cyclomatic complexity* – number of linearly independent paths through an algorithm.

Execution Performance:

- Execution Performance of the applications will be measured by their run time.

This section presents the performance results for Range Search, Point Location, and Largest Empty Circle parallel applications. For the Point Location problem, I collected results in full:

programmability and execution performance. Yet, for the Largest Empty Circle, I present only programmability results, since execution performance tests are not complete.

3.1 Range Search

The range searching [6] problem consists of preprocessing a set of N points in the plane to determine which points reside within a query rectangle (range). The query range includes four values: x- minimum, maximum and y- minimum, maximum coordinates in a plane. The baseline of the range searching algorithms is the construction of a multidimensional binary tree (KD tree) [1][7]. KD tree for two-dimensional points is a modified two-dimensional binary search tree (BST), which alternates x- and y- coordinates as a key for inserting elements. The alternating sequence starts with the x-coordinate. The construction of KD tree consists of recursively partitioning the plane into two halfplanes, where the point positioned at the bisection line is the next point to be inserted into the tree with respect to x and y dimensions. Each bisection line is determined after sorting the points by x or y coordinate depending on the next dimension of the KD tree level. The bisection line is determined by dividing the number of points by two.

3.1.1 Programmability

Table 2 presents the programmability metrics for the Range Search implementation utilizing MASS, Spark, and MapReduce.

Parallel Framework	Boilerplate code	Lines of code	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	25	450	6	3
Spark	4	350	3	3
MASS	3	490	6	4

Table 2: *Programmability metrics for Range Search*

The comparison of three different implementations of Range Search shows that MASS requires the fewest number of boilerplate code to set up the environment in contrast to MapReduce and Spark. This boilerplate code consists of initializing MASS, setting up a debugging level, and shutting down MASS when the computation is finished. The total number of lines in MapReduce

and MASS implementations are relatively compatible. Yet, due to the needed custom agent and vertex place classes, the total number of lines is slightly higher than MapReduce implementation. Finally, the fewest number of classes is required by Spark implementation, whereas MapReduce and MASS implementations require more classes.

Cyclomatic complexity is used to measure the complexity of the algorithms based on three parallelization tools, such as MASS, MapReduce, and Spark. Spark and MapReduce use the same algorithms to performs range search, and the cyclomatic complexity is equal to three for both. MASS uses vertex and agents in its algorithms, and the cyclomatic complexity of this algorithm equals four. The cyclomatic complexity measurements show that algorithms designed for MASS provide very similar complexity in comparison to algorithms designed for MapReduce or Spark. The metrics (see *table 2*) prove that MASS is a better programming tool in terms of the required steps needed to set up the environment, and Spark is a better programming tool in terms of the fewer number of classes and lines of code required for the implementation.

3.1.2 Execution Performance

We conducted execution performance tests for range search using 10,000 input points. Figure 1 shows execution performance results of range search.

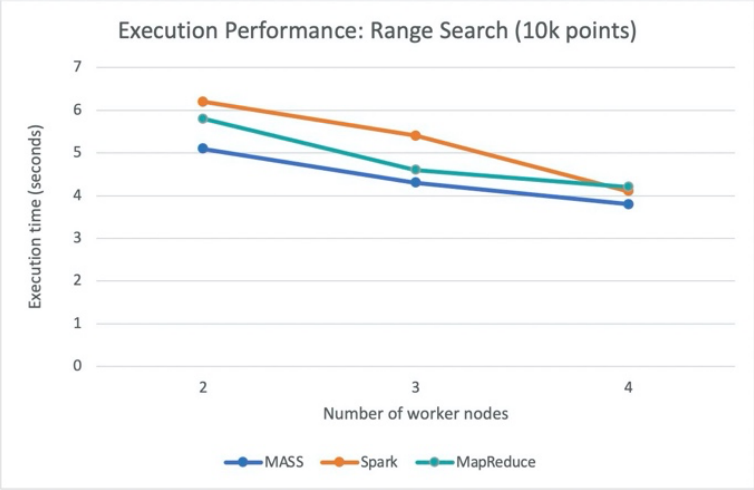

Figure 1: *Execution performance for Range Search*

Figure 1 shows the performance for an input size of 10,000 points. We can see that MASS implementation produces better execution performance results in comparison with MapReduce and Spark applications. According to the performance results gathered during multiple tests, the performance of MASS implementation increases by increasing the number of cluster nodes. Further, MapReduce produced better performance results using two and three worker nodes in comparison with Spark implementation. Yet, in contrast to Spark implementation, the execution performance of MapReduce implementation does not improve when the number of worker nodes is higher than three. MapReduce implementation has the overall best time. However, we can see that adding more computing notes to the cluster do not decrease execution time of the MapReduce program. On the contrary, Spark implementation shows that additional computing nodes decreased execution time.

The Range Search implementation with the MASS library benefits from the fact that MASS allows maintaining the original dataset structure for the duration of computation. The data is not copied continuously or moved as in MapReduce and Spark. The same graph containing points on its vertices is used throughout the entire computation.

3.2 Point Location

The most used point location [9] application is a location query. Given a map and a query point specified by its coordinates find the region of the map containing the query point. A map is nothing more than a subdivision of the plane into regions, a planar subdivision. The trapezoidal map is usually used to create a planar subdivision [2]. To create a trapezoidal map vertical line is drawn from each point going upward and downward. The implemented applications that solve point location problems use a preprocessed trapezoidal map that is presented as an input file contacting trapezoids. The application receives the input file, query point x, and y coordinates. The output is the trapezoid details that contains the query point.

3.2.1 Programmability

Table 3 presents the programmability metrics for the Point Location implementation utilizing MASS, Spark, and MapReduce.

Parallel Framework	Boilerplate code	Lines of code	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	23	283	8	2
Spark	4	236	3	2
MASS	3	322	7	4

Table 3: *Programmability metrics for Point Location*

Spark implementation of the Point Location has the fewest number of lines of code and number of classes in comparison with MASS and MapReduce. Yet, MASS implementation has the best results in the number of boilerplate code lines metric. It also has a better result than MapReduce in terms of the number of classes needed to implement the algorithms. The cyclomatic complexity of MASS algorithms is higher in comparison to Spark and MapReduce algorithms due to using doWhile() loop which is needed to do agent simulations. We do simulations if at least one agent is alive.

MASS implementation has the highest number of lines of code. This is due to the fact that the algorithm implementation needs more code. In order to initialize each Place with trapezoid, we extended Places class and customized to the needs of the algorithms. We also have a class Trapezoid, which describes the Trapezoid object for each Place. In order for the agent to perform point location search we extended Agent class to describe the behavior of each agent. Also, we have one more additional class that is used to pass arguments to each created (spawned) agent.

Even though MASS implementation has the highest number of lines of code in this benchmark we significantly decreased the number of lines of code in comparison to using the traditional programming approach of the MASS library. This improvement is due to utilizing event-oriented programming using annotations [12] feature of the MASS library. MASS library traditionally uses barrier synchronization of agent upon executing callAll(), manageAll(), or doAll(), which users repeatedly invoke from the main() function [12]. The event-programming with annotations automatically invokes such functions when the corresponding event occurs. Thus, we eliminated

the need of writing additional lines of code to manually synchronize agents in Point Location implementation.

3.2.2 Execution Performance

To conduct execution performance tests for all three versions of Point Location applications we used a considerably large dataset of 500k trapezoids. Figure 1 shows the execution performance results of the Point Location application with MASS, MapReduce, and Spark.

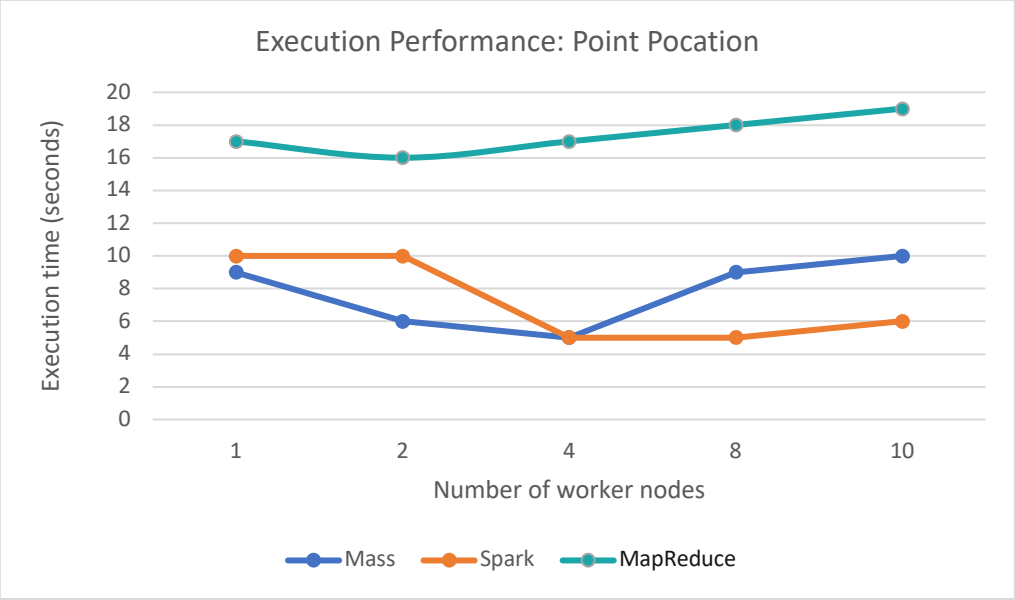

Figure 2: *Execution performance of Point Location*

Overall, the execution performance of the MASS implementation of Point Location is better than the MapReduce implementation. MapReduce takes a longer execution time due to the overhead of intermediate input/output operations between MapReduce jobs. We can see in Figure 1 that when the number of computing nodes in a cluster is less or equal four Point Location MASS performed better than both Spark and MapReduce implementations. After adding more computing nodes to the cluster Point Location MASS execution time starts increasing. The reason for such increase in execution time is due to the fact that we use agent migration and Hazelcast [8]. MASS uses Hazelcast as underlying in-memory data grid (IMDG). MASS uses Hazelcast to distribute data evenly among computing nodes in the cluster. Hazelcast has a disadvantage of using transmission control protocol (TCP), which is known to be slower than user data protocol UDP. MapReduce

implementation has the overall best time. However, we can see that adding more computing notes to the cluster do not decrease execution time of the MapReduce program. On the contrary, Spark implementation shows that additional computing nodes decreased execution time.

3.3 Largest Empty Circle

The largest empty circle problem states, given a set of S site points determine the largest empty circle whose interior does not overlap with any other obstacle [9]. It has been proven that the center of the largest empty circle must lie on the Voronoi vertex. Our Largest Empty Circle (LEC) program receives two input files: one consisting of site points and the other consisting of Voronoi vertices points. The output of the program is the radius of the largest empty circle as well as the center point and the point on the circle.

3.3.1 Programmability

Table 4 presents the programmability metrics for the Largest Empty Circle implementation utilizing MASS, Spark, and MapReduce.

Parallel Framework	Boilerplate code	Lines of code	Number of classes	Cyclomatic complexity (algorithms)
MapReduce	76	549	10	2
Spark	2	320	5	2
MASS	3	211	6	2

Table 4: *Programmability metrics for Largest Empty Circle*

The metrics in Table 3 indicate the results of the programmability analysis for LEC programs in all three platforms MASS, Spark, and MapReduce. MapReduce implementation has the worst programmability results among the three programs. As the metric indicates MapReduce implementation requires the highest number of boilerplate code, lines of code, and classes. These high numbers are due to the need of setting up the environment and multiple MapReduce jobs. On the contrary, LEC Spark implementation requires the fewest number of lines of boilerplate code to set up the environment. Also, Spark implementation has the advantage of having the fewest

number of classes. Spark benefits from its programming model that has resilient distributed datasets (RDDs), which overall shortens the required number of lines and number of classes. The decrease in the number of boilerplate code for LEC Spark implementation in comparison with PointLocation Spark implementation is because we do not broadcast any variables to Spark context as it is the case for PointLocation Spark implementation where we have to broadcast two additional variables x and y coordinates of the query point.

Largest empty circle (LEC) MASS implementation outperforms Spark and MapReduce implementations in terms of a total number of lines of code. It also outperforms MapReduce by having the fewest number of boilerplate code lines and the number of classes. The good programmability performance of LEC MASS is due to embedded SpacePlace and SpaceAgent classes into the MASS library. The algorithm for LEC MASS does not override the SpacePlace and SpaceAgent classes, thus, simplifies the overall implementation and increases programmability performance. However, small changes are made in the SpaceAgent, which locates directly in the MASS library. These changes are needed for computing the furthest pair of points instead of computing the closest pair of points that is coded into the original implementation.

3.3.2 Execution Performance

I conducted execution performance tests for LEC MapReduce and Spark programs with a different number of computing nodes in the cluster. However, the execution performance testing for LEC MASS implementation is not complete. We are having some issues running the program on the cluster while it successfully runs on one node. We are troubleshooting the issue to determine the root of the issue. I will include the execution performance comparison across three platforms for LEC implementations in the next quarter.

3.4 Project Source Code

The implementations of the above applications are located in the *satine_develop* branch in *mass_java_app* bitbucket repository under the *Applications* directory. The link to source code: https://bitbucket.org/mass_application_developers/mass_java_appl/src/satine_develop/Applications/

Chapter 4: CONCLUSION

The results for the point location algorithm showed that MASS implementations have better execution performance results than MapReduce implementation. MASS requires the fewest number of lines of boilerplate code to set up the environment. The MASS program also outperforms the MapReduce program in terms of the number of classes in the implementation. The point location program with MASS utilizes event-driven programming with an annotations feature, which reduced the overall number of lines of code. This feature eliminates the need for manually synchronizing mobile agents migration by replacing callAll() and manageAll() with doWhile and annotated methods @OnCreation, @OnArraival, and @OnMessage.

The execution performance of the MASS program is the best among all three platforms when the number of computing nodes in a cluster is less than five. The need for agents to migrate between computing nodes and the message broadcasting increases the execution performance of the MASS program. MASS implementation of the point location algorithm increases execution time as the number of computing nodes in the cluster increases. This increase in time is due to Hazelcast, which is in-memory data grid used by MASS library. Hazelcast uses TCP that is relatively slow for over the network communications. The potential solution to mitigate this overhead in the future by switching from Hazelcast to our own implementation.

The next for my capstone project is to finish the execution performance tests of the LEC MASS program on a cluster and make the comparison of the execution performance within three platforms. As the final and fourth application, I will design and implement a parallel algorithm for solving the Euclidean Shortest Path problem. Once the fourth algorithm is implemented, I will conduct the programmability and execution performance tests. I plan to finish all implementations in Winter 2021 quarter.

REFERENCES

[1] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction. 1993

[2] M. Berg, O. Cheong, M. Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications. 2008

[3] T. Whire, Hadoop: The Definitive Guide. 2012

[4] M.Fukuda, Parallel-Computing Library for Multi-Agent Spatial Simulation in Java, 2010

[5] H.Karau, A. Konwinski, P.Wendell, M. Zaharia, Learning Spark. 2015

[6] Wikipedia.org. Range searching – Wikipedia.

[7] H. M. Kakde, "Range Searching using Kd Tree." 2005

[8] Hazelcast.org. Hazelcast open-source projects.

[9] Wikipedia.org. Point location – Wikipedia.

[10] Wikipedia.org. Largest empty circle – Wikipedia.

[11] Wikipedia.org. Euclidian shortest path – Wikipedia.

[12] Matthew Sell, Munehiro Fukuda, Agent Programmability Enhancement for Rambling over Scientific Dataset, 2020