MASS Java & GraphX Benchmarks for Graph Computing

1. Overview

Distributed graph computing is an evolving field within distributed computing, with many organizations extending their systems to tackle large-scale graph processing. To determine which computing problems different graph computing implementations are best suited for, we can evaluate their performance and programmability.

At the University of Washington Bothell, the Distributed Systems Laboratory (DSL) has developed MASS (Multi-Agent Spatial Simulation), a distributed memory system designed for agent-based spatial computations, including its graph computing system for distributed graph databases. This study compares MASS's agent-based approach to GraphX, Apache Spark's graph processing library, to assess their scalability, efficiency, and ease of use trade-offs.

GraphX, built on Apache Spark, integrates graph computation with data-parallel processing, utilizing Resilient Distributed Datasets (RDDs) and a Pregel-inspired API to optimize iterative graph algorithms. In contrast, MASS distributes computations across agents operating independently over partitions of the graph. Understanding the strengths and weaknesses of each model will provide insight into their suitability for different workloads.

Previous MASS Java benchmarks were primarily developed by James Day, who implemented key benchmarking programs to evaluate MASS's graph computing performance. Rather than reimplementing MASS benchmarks, my role is developing GraphX benchmarks to compare the two systems directly. By focusing solely on GraphX, I aim to assess its performance, programmability, and scalability relative to MASS, providing insights into the trade-offs between these two approaches to distributed graph computing.

2. Background

MASS (Multi-Agent Spatial Simulation) is a parallel computing library for distributed memory systems. It models computations using places, representing data distributed across computing nodes and agents. Agents traverse these places to execute tasks and exchange information. Within MASS's graph database implementation, graph nodes are defined as places (GraphPlaces), and agents perform operations such as triangle counting, clustering computations (e.g., connected components), and connectivity analysis.

GraphX, a graph processing framework built on Apache Spark, takes a different approach by abstracting graph computation with Spark's data-parallel model. GraphX represents graphs using Resilient Distributed Datasets (RDDs). Unlike MASS, which distributes execution through mobile agents, GraphX leverages Spark's underlying dataflow model and lazy evaluation to efficiently execute graph operations at scale.

2.1. *Graph Loading*

The version of MASS used for comparison with GraphX distributes GraphPlaces in a round-robin fashion across the machine cluster, with each node being assigned sequentially to a different machine, as seen in Figure 1. Graphs are loaded using a proprietary DSL file, which defines the nodes and their edges in a text-based format. These DSL files are also used to generate graphs in GraphX for a direct comparison.

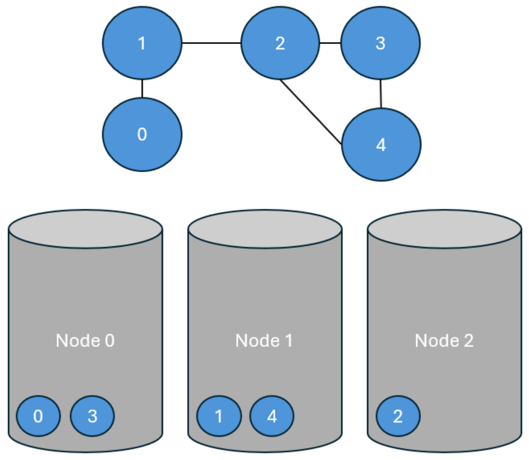


Figure 1

In GraphX, graphs are constructed by creating an RDD of vertices and edges, which is then used to generate the graph structure. In our comparison, we developed a program that parses MASS's DSL files and converts them into a format suitable for loading graphs in GraphX.

2.2. Clustering Coefficient

Clustering Coefficients show how tightly knit each vertex in a graph is with its neighbors. They measure the degree to which vertices cluster together. Many real-world graphs, such as social network graphs, have a high degree of clustering due to cliques of people (small and dense groups of people who know each other). Calculating the clustering coefficient can help recognize these groups of people. Clustering coefficients are computed by examining whether a node's neighbors are also connected, forming triangles of edges. The coefficient is determined by comparing the number of connections between a node's neighbors to their total possible connections. The result ranges from 0 to 1, where 1 indicates that all neighbors of a node are fully connected (forming a clique), and 0 means none of its neighbors are connected. This provides the local clustering coefficient, which measures how tightly a single vertex's neighbors are connected. To understand the overall connectivity of the graph, we compute the global clustering coefficient by averaging the local clustering coefficients across all vertices. A high global clustering coefficient indicates that, on average, most nodes exist within densely interconnected communities, suggesting strong network cohesion. In contrast, a low global clustering coefficient implies a sparser, more fragmented network structure.

The formal calculation of local clustering coefficients can be seen in Figure 1, where V is a vertex in the graph, N_V is the number of links between its neighbors and K_V is its degree.

$$\frac{2 \cdot N_V}{K_V\,(K_V - 1)}$$

Figure 2

2.3. *Weakly Connected Components*

A Weakly Connected Component (WCC) is a subgraph in which all vertices are connected by some path, regardless of edge direction. In directed graphs, this means that even if some edges are one-way, there is still a way to traverse between any two nodes in the component when ignoring edge direction.

WCCs help identify disconnected regions of a graph, which is helpful in applications such as social networks, where different communities may be loosely linked, or in web graphs, where certain pages are reachable only when considering undirected paths.

The Weakly Connected Components (WCC) algorithm identifies subgraphs in which all vertices are reachable from one another when edge direction is ignored. It begins by treating the directed graph as undirected, ensuring all connections are bidirectional. The algorithm then groups nodes into connected subgraphs, where each node has at least one path to any other node within the same component. Once these subgraphs are identified, each is assigned a unique identifier, effectively labeling distinct weakly connected components. This approach helps analyze a directed graph's overall structure and fragmentation, revealing how many different groups exist and how they are internally connected.

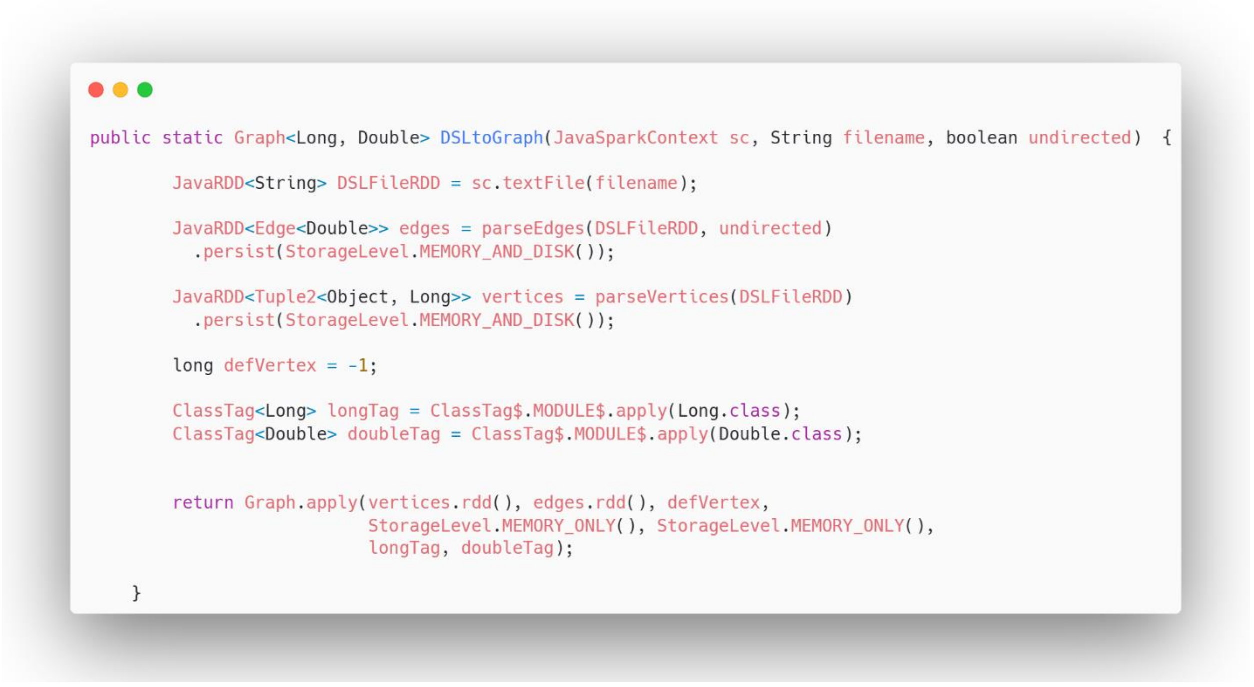
3. *Implementation/Progress*

3.1. *Graph Loading*

MASS loads graphs by placing a GraphPlace at each node for each vertex seen in the DSL file, going round-robin across the machines, as explained in the overview. We created a

Java GraphX program to parse the DSL file and create a GraphX graph by generating the RDDs for the vertices and edges.

Figure 3: GraphX DSL Graph Generation



```java
public static Graph<Long, Double> DSLtoGraph(JavaSparkContext sc, String filename, boolean undirected)  {

        JavaRDD<String> DSLFileRDD = sc.textFile(filename);

        JavaRDD<Edge<Double>> edges = parseEdges(DSLFileRDD, undirected)
          .persist(StorageLevel.MEMORY_AND_DISK());

        JavaRDD<Tuple2<Object, Long>> vertices = parseVertices(DSLFileRDD)
          .persist(StorageLevel.MEMORY_AND_DISK());

        long defVertex = -1;

        ClassTag<Long> longTag = ClassTag$.MODULE$.apply(Long.class);
        ClassTag<Double> doubleTag = ClassTag$.MODULE$.apply(Double.class);


        return Graph.apply(vertices.rdd(), edges.rdd(), defVertex,
                           StorageLevel.MEMORY_ONLY(), StorageLevel.MEMORY_ONLY(),
                           longTag, doubleTag);

    }
```

Figure 3 illustrates in GraphX how the graph's vertices and edges persist in memory, leveraging Spark's in-memory processing capabilities. This enables graph transformations to execute significantly faster than retrieving data from disks. More details about DSL graph loading with GraphX can be found in Appendix A.

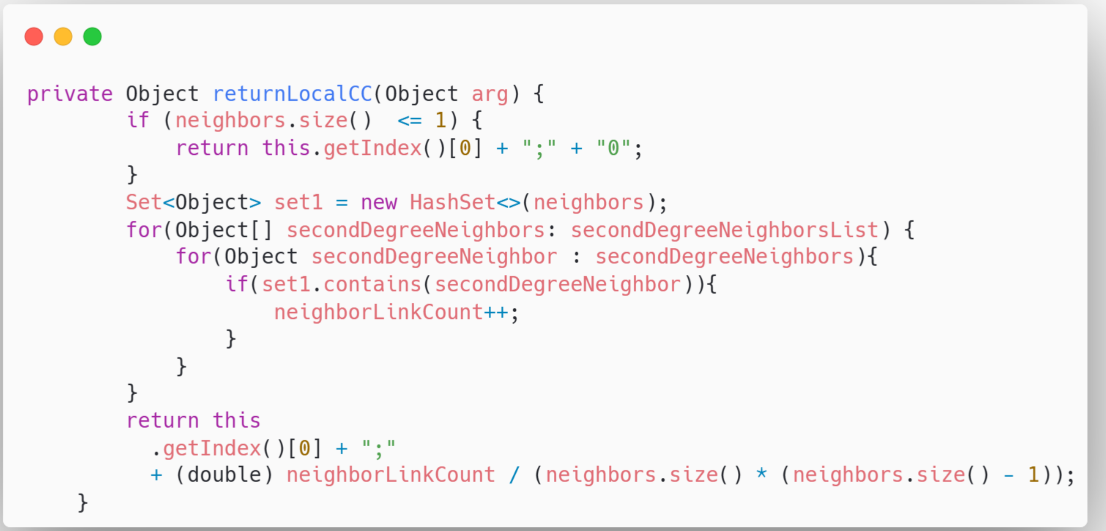
3.2. Clustering Coefficient

3.2.1. MASS Implementation

The MASS implementation of the clustering coefficient begins by assigning an agent to each vertex in the graph. Each agent stores its original place ID before proceeding. The agent then creates additional agents equal to the number of its neighboring vertices, and these agents migrate to their respective neighbors. Once there, they collect a list of that vertex's second-degree neighbors and then return to their original vertex with this information.

At this point, the GraphPlace stores the gathered second-degree neighbor lists. It then computes the local clustering coefficient by determining how many of its second-degree neighbors are directly connected. A function called CallAll retrieves the local clustering coefficients from all GraphPlaces to obtain the global clustering coefficient. The results are sent to the master node and averaged to compute the overall graph clustering coefficient.

Figure 4: MASS Local Vertex Cluster Coefficient Computation

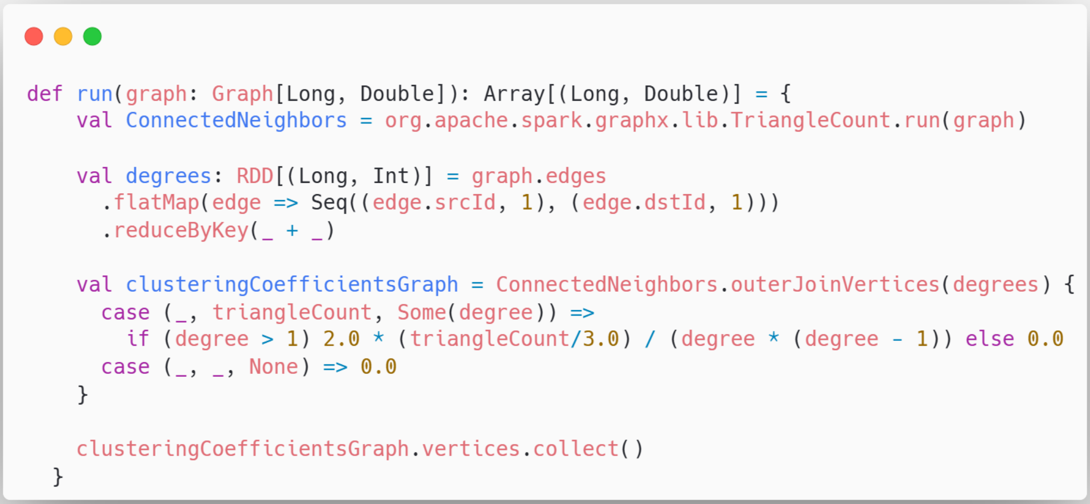


```java
private Object returnLocalCC(Object arg) {
        if (neighbors.size()  <= 1) {
            return this.getIndex()[0] + ";" + "0";
        }
        Set<Object> set1 = new HashSet<>(neighbors);
        for(Object[] secondDegreeNeighbors: secondDegreeNeighborsList) {
            for(Object secondDegreeNeighbor : secondDegreeNeighbors){
                if(set1.contains(secondDegreeNeighbor)){
                    neighborLinkCount++;
                }
            }
        }
        return this
          .getIndex()[0] + ";"
          + (double) neighborLinkCount / (neighbors.size() * (neighbors.size() - 1));
    }
```

3.2.2. GraphX Implementation

The GraphX implementation first gathers each vertex's degree and triangle count. These values are then joined to construct a new graph, where each vertex is represented as <VertexID, Degree, Num. of Triangles>. Using the formula shown in Figure 1, the local clustering coefficient is computed for each vertex. The results are then collected and sent back to the master node as an array, where each entry contains the vertex ID and its corresponding local clustering coefficient. Finally, the master node calculates the average clustering coefficient for the entire graph.

Figure 5: GraphX Local Vertex Cluster Coefficient Computation



```scala
def run(graph: Graph[Long, Double]): Array[(Long, Double)] = {
    val ConnectedNeighbors = org.apache.spark.graphx.lib.TriangleCount.run(graph)

    val degrees: RDD[(Long, Int)] = graph.edges
      .flatMap(edge => Seq((edge.srcId, 1), (edge.dstId, 1)))
      .reduceByKey(_ + _)

    val clusteringCoefficientsGraph = ConnectedNeighbors.outerJoinVertices(degrees) {
      case (_, triangleCount, Some(degree)) =>
        if (degree > 1) 2.0 * (triangleCount/3.0) / (degree * (degree - 1)) else 0.0
      case (_, _, None) => 0.0
    }

    clusteringCoefficientsGraph.vertices.collect()
  }
```

3.3. *Weakly Connected Components*

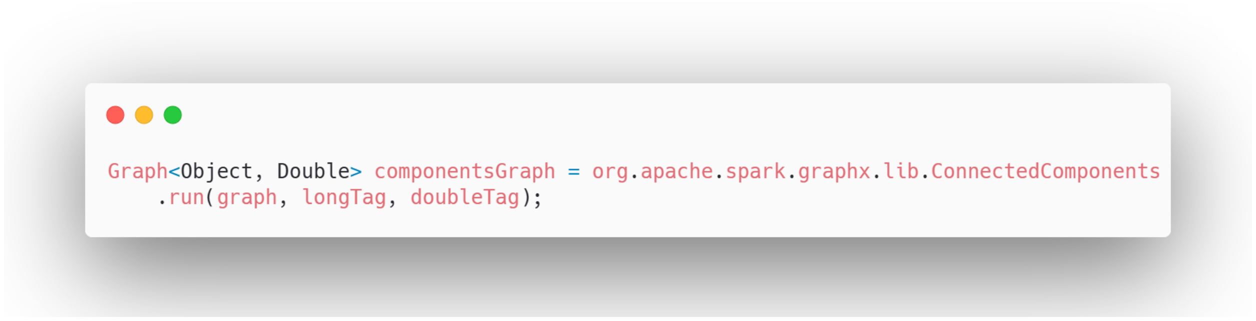
3.3.1. MASS Implementation

The current MASS implementation of weakly connected components had been completed previously; however, it does not load a graph but generates a random graph on each run. To keep benchmarks fair, we must modify this program to load graphs.

3.3.2. GraphX Implementation

GraphX weakly connected components can be found in the GraphX library. The algorithm follows an iterative label propagation approach to identify subgraphs in which all vertices are reachable from one another when edge direction is ignored. The process begins by assigning each vertex to a unique label, initially set to its vertex ID. Every vertex updates its label to the smallest ID among its connected neighbors in each iteration, effectively propagating the lowest ID throughout the component. Since edge directions are ignored, label propagation occurs bidirectionally between vertices. This iterative process continues until convergence, meaning all vertices within the same connected component share the same label. Once no further label changes occur, the algorithm terminates, efficiently grouping weakly connected subgraphs within the graph.

Figure 6: GraphX Weakly Connected Components Function



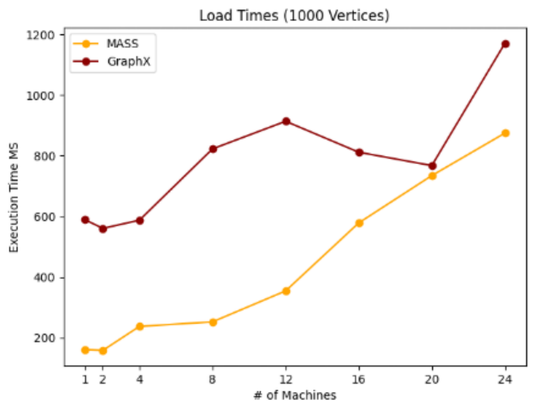
```
Graph<Object, Double> componentsGraph = org.apache.spark.graphx.lib.ConnectedComponents
    .run(graph, longTag, doubleTag);
```

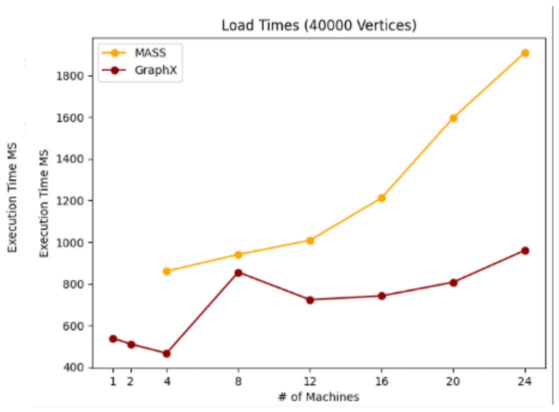
4. Results

All benchmarks have been run on the CSSMPI machines.

4.1. Load Time Performance

MASS and GraphX have run times that are generally similar, although GraphX typically performs better on larger graphs and handles loading with a more significant number of machines more gracefully.





Graph 1: Load Times for 1-24 Computing Nodes w/ 1k vertices Graph

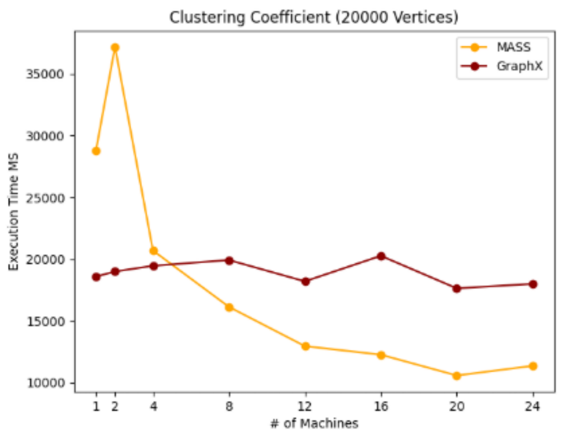
Graph 2: Load Times for 1-24 Computing Nodes w/ 40k vertices Graph

However, MASS processes smaller graphs faster than GraphX, likely because GraphPlaces are distributed more efficiently across computing nodes. In contrast, GraphX incurs overhead from shuffling data due to the lack of pre-partitioned vertices. Since GraphX does not pre-partition vertices before processing, it must reorganize the data dynamically, which introduces additional computational overhead. In contrast, MASS's round-robin assignment of GraphPlaces allows for faster initialization and execution in smaller-scale graphs.

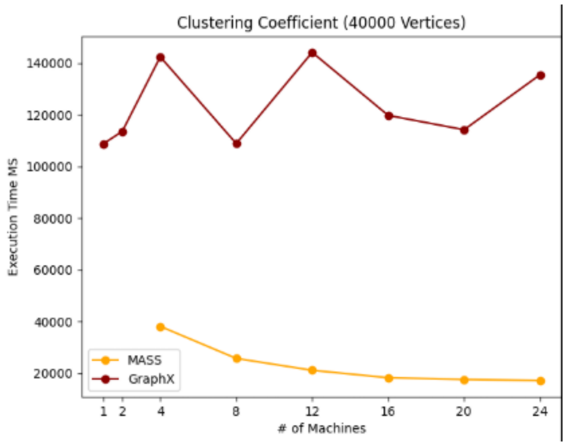
4.2. Clustering Coefficient Results

The GraphX performance curve remains relatively constant due to shuffling overhead when calculating the degree of each vertex. This overhead limits GraphX's ability to scale performance efficiently as more machines are added, reducing the benefits of increased parallelism. In contrast, MASS consistently outperforms GraphX as the number of machines increases and shows significant performance advantages on large graphs, such as those with 40K vertices.

The MASS implementation is faster because agents autonomously compute the local clustering coefficients with much less communication than needed, as the GraphX implementation does with shuffling.



Graph 3: 1-24 Computing Nodes computing Clustering Coefficient on 20k vertices



Graph 4: 1-24 Computing Nodes computing Clustering Coefficient on 40k vertices

4.3. Programmability

Spark, the library GraphX is a part of, utilizes the MapReduce computing paradigm, which makes it straightforward and familiar. It is also part of the Apache Software Foundation, which gives it many open-source contributions. These contributions have made Spark programs simpler and easier to use than MASS, which requires more setup since MASS is still in development.

In the clustering coefficient program for MASS, we require multiple files to create our program. One file is ClusteringVertex.java, which is the distributed data structure that represents the vertices of a graph; this extends the existing VertexPlace to create custom logic. We also have a file that extends the agent class to implement custom logic for the individual agents executing clustering coefficient calculations. We made a custom class to be able to pass arguments to agents. We finally have the ClusteringCoefficient.java master program, which orchestrates the execution. This file collection complicates MASS program creation compared to GraphX's simple one-file benchmark. The total number of files and other programmability data is in Table 1.

Table 1: MASS & GraphX Clustering Coefficient Programmability Data

Measurement (MASS)	Count
Number of files	4
Number of methods	14
Number of variables declared	51
Total lines of code	528
Lines of logic	224

Measurement (GraphX)	Count
Number of files	1
Number of methods	3
Number of variables declared	28
Total lines of code	96
Lines of logic	20

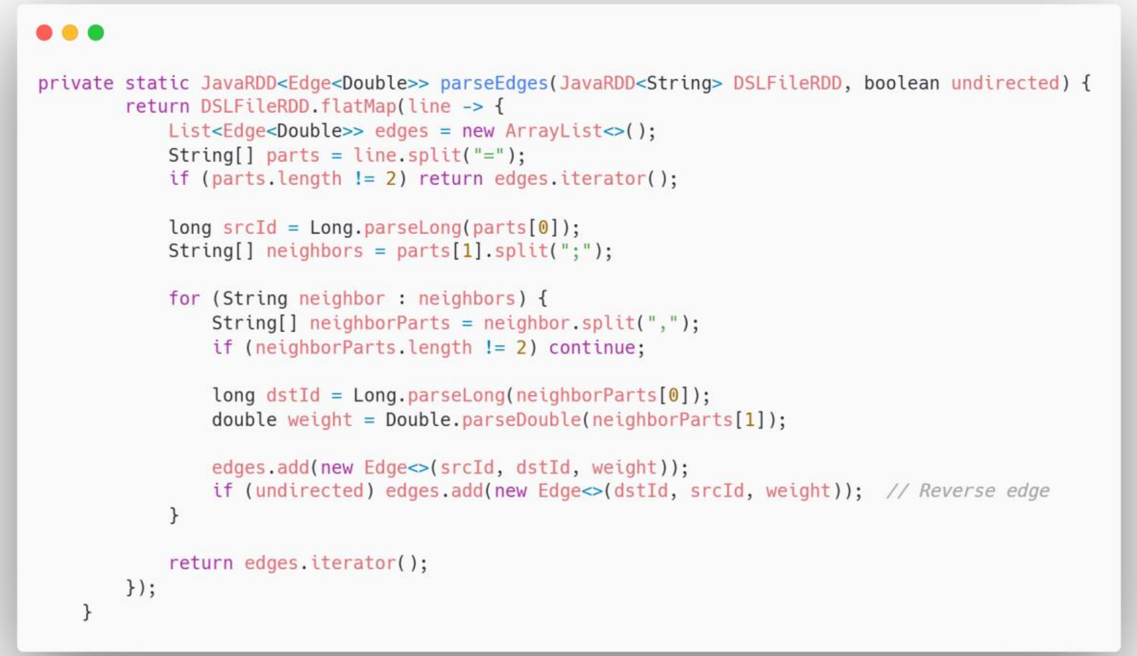
5. Conclusion

Programming with GraphX has allowed me to gain great insight into the world of distributed computing programming and has helped me understand how to approach new computing systems. So far, two benchmarks have been written, and one MASS benchmark needs to be modified to compare with GraphX appropriately, which is where my new knowledge of distributed systems programming will be handy. Graph loading with DSL files in GraphX has been implemented, allowing for fair benchmarks between the two systems using the same graph files.

The following term's steps are to tweak the MASS weakly connected components benchmark for an accurate comparison to GraphX. We also want to compare MASS's articulation points and strongly connected components to GraphX implementation.

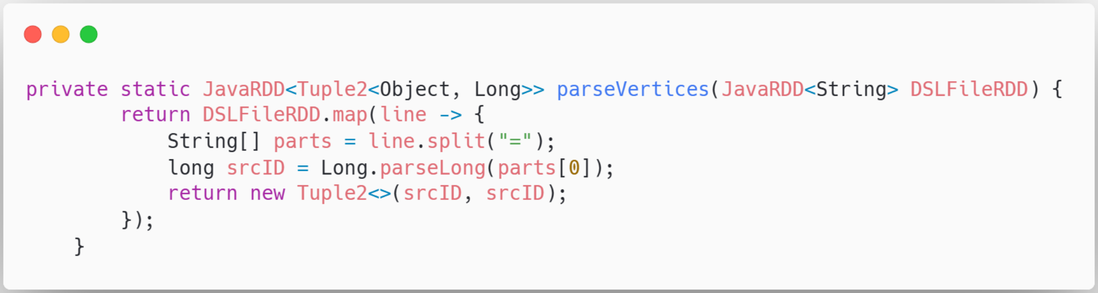
Appendix A: DSL File GraphX Parsing

Figure 7: Parse Edges



```
private static JavaRDD<Edge<Double>> parseEdges(JavaRDD<String> DSLFileRDD, boolean undirected) {
        return DSLFileRDD.flatMap(line -> {
            List<Edge<Double>> edges = new ArrayList<>();
            String[] parts = line.split("=");
            if (parts.length != 2) return edges.iterator();

            long srcId = Long.parseLong(parts[0]);
            String[] neighbors = parts[1].split(";");

            for (String neighbor : neighbors) {
                String[] neighborParts = neighbor.split(",");
                if (neighborParts.length != 2) continue;

                long dstId = Long.parseLong(neighborParts[0]);
                double weight = Double.parseDouble(neighborParts[1]);

                edges.add(new Edge<>(srcId, dstId, weight));
                if (undirected) edges.add(new Edge<>(dstId, srcId, weight));  // Reverse edge
            }

            return edges.iterator();
        });
    }
```

Figure 8: Parse Vertices



```
private static JavaRDD<Tuple2<Object, Long>> parseVertices(JavaRDD<String> DSLFileRDD) {
        return DSLFileRDD.map(line -> {
            String[] parts = line.split("=");
            long srcID = Long.parseLong(parts[0]);
            return new Tuple2<>(srcID, srcID);
        });
    }
```

Appendix B: Cluster Coefficient & Graph Load Results

Table 2: MASS Performance

num-members	num-vertices	LoadTime	agentsUsed	avgCC	RunTime

1	1000	162	187960	0.108192	2320
1	3000	302	590608	0.03807	5686
1	5000	456	990780	0.023167	8173
1	10000	571	1989980	0.011619	14452
1	20000	1104	3992760	0.005841	28810
1	4941	85	31317	0.080104	652
2	1000	159	187960	0.108192	2861
2	3000	292	590608	0.03807	7403
2	5000	400	990780	0.023167	10781
2	10000	544	1989980	0.011619	19788
2	20000	710	3992760	0.005841	37185
2	4941	112	31317	0.080104	788
4	1000	238	187960	0.108192	2125
4	3000	412	590608	0.03807	4860
4	5000	438	990780	0.023167	7701
4	10000	572	1989980	0.011619	12193
4	20000	657	3992760	0.005841	20696
4	40000	861	8028848	0.002922	38079
4	4941	164	31317	0.080104	744
8	1000	253	187960	0.108192	1921
8	3000	438	590608	0.03807	3505
8	5000	495	990780	0.023167	5190
8	10000	648	1989980	0.011619	9172
8	20000	825	3992760	0.005841	16136
8	40000	941	8028848	0.002922	25680
8	4941	235	31317	0.080104	1039
12	1000	355	187960	0.108192	1988
12	3000	526	590608	0.03807	3051
12	5000	674	990780	0.023167	4512
12	10000	721	1989980	0.011619	7584
12	20000	822	3992760	0.005841	12959
12	40000	1009	8028848	0.002922	21087
12	4941	337	31317	0.080104	1076
16	1000	579	187960	0.108192	2100
16	3000	829	590608	0.03807	5269
16	5000	822	990780	0.023167	4541
16	10000	957	1989980	0.011619	6480
16	20000	1175	3992760	0.005841	12264
16	40000	1213	8028848	0.002922	18201
16	4941	485	31317	0.080104	1227
20	1000	735	187960	0.108192	1944

20	3000	1122	590608	0.03807	3056
20	5000	1136	990780	0.023167	3958
20	10000	1314	1989980	0.011619	6114
20	20000	1342	3992760	0.005841	10580
20	40000	1597	8028848	0.002922	17545
20	4941	782	31317	0.080104	1312
24	1000	875	187960	0.108192	2290
24	3000	1262	590608	0.03807	3133
24	5000	1173	990780	0.023167	4166
24	10000	1399	1989980	0.011619	6050
24	20000	1684	3992760	0.005841	11361
24	40000	1910	8028848	0.002922	17106
24	4941	902	31317	0.080104	1413

Table 3: GraphX Performance

num-members	num-vertices	LoadTime	avgCC	RunTime
1	1000	590	0.002234	2533
1	3000	477	0.000786	4366
1	5000	495	0.000478	5363
1	10000	574	0.00024	8432
1	20000	467	0.000121	18589
1	40000	539	0.00006	108709
2	1000	561	0.002234	2485
2	3000	546	0.000786	4447
2	5000	554	0.000478	5185
2	10000	503	0.00024	9133
2	20000	505	0.000121	18998
2	40000	511	0.00006	113540
4	1000	588	0.002234	3036
4	3000	495	0.000786	4203
4	5000	576	0.000478	5995
4	10000	485	0.00024	8858
4	20000	596	0.000121	19461
4	40000	467	0.00006	142373
8	1000	823	0.002234	2764
8	3000	617	0.000786	3901
8	5000	691	0.000478	5490
8	10000	783	0.00024	8561
8	20000	633	0.000121	19928

8	40000	856	0.00006	108826
12	1000	914	0.002234	2310
12	3000	662	0.000786	4052
12	5000	818	0.000478	4770
12	10000	780	0.00024	8992
12	20000	655	0.000121	18201
12	40000	724	0.00006	144105
16	1000	812	0.002234	2323
16	3000	792	0.000786	4299
16	5000	809	0.000478	5178
16	10000	884	0.00024	10196
16	20000	672	0.000121	20271
16	40000	742	0.00006	119723
20	1000	768	0.002234	2700
20	3000	705	0.000786	4539
20	5000	778	0.000478	5590
20	10000	797	0.00024	8563
20	20000	792	0.000121	17635
20	40000	808	0.00006	114159
24	1000	1171	0.002234	2337
24	3000	635	0.000786	4178
24	5000	611	0.000478	5354
24	10000	768	0.00024	8648
24	20000	866	0.000121	18000
24	40000	961	0.00006	135436

Appendix C: How to Run the Benchmark Programs

MASS

1. Install the latest version of MASS core
   A. git clone -b develop https://bitbucket.org/mass_library_developers/mass_java_core.git
   B. cd mass_java_core
   C. mvn clean package install
   D. Return to the previous directory
2. Clone repo: (currently under jaday2/graphbenchmarks)
   git clone -b jaday2/graph-benchmarks --single-branch https://bitbucket.org/mass_application_developers/mass_java_appl.git

3. Navigate to the project directory
   cd mass_java_appl/Graphs/ClusteringCoefficient
4. maven package
   Use included make file: make
5. Update node file

Change nodes.xml's mass home tag to point to the jar file, and use the correct nodes you'd like to be in the compute cluster. Add the username tag with your username as well.

6. Running the benchmark
   java -jar ClusteringCoefficient-1.0.0-RELEASE.jar <Path to input file> <print boolean>

GraphX

1. Clon repo: (currently under arian23/GraphXBenchmarks) or switch branch if already cloned MASS application repo.
   Git clone -b arian23/GraphXBenchmarks –-single-branch
   Or
   Git checkout arian23/GraphXBenchmarks
2. Navigate to the project directory

   Cd GraphXBenchmarks/GraphXBenchmarks

3. Install dependencies

   Mvn clean install

4. Run benchmark
   spark-submit --master spark://<Spark-Master> --total-executor-cores <#> --executor-cores <#> --class edu.uw.bothell.css.dsl.mass.ClusterCoefficient --jars ClusteringCoefficients/target/ClusteringCoefficients-1.0-SNAPSHOT-jar-with-dependencies.jar target/classes <DSL Graph File Name>