MASS Java & GraphX Benchmarks for Graph Computing

1. Overview

Distributed graph computing is an evolving field within distributed computing, and many organizations are expanding their systems to address large-scale graph processing. To determine which computing problems various systems' implementations are best suited to solve, we can assess 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 with GraphX, Apache Spark's graph processing library, to evaluate their scalability, efficiency, and ease of use trade-offs.

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

Previous MASS Java benchmarks were primarily developed by James Day, who created key benchmarking programs to evaluate MASS's graph computing performance. Two MASS benchmarks not made by James are weakly connected and strongly connected components, which required a refactor to accommodate recent changes to MASS. By focusing on GraphX, I aim to evaluate its performance, programmability, and scalability compared 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 navigate these places to perform tasks and exchange information. In MASS's graph database implementation, graph nodes are defined as places (GraphPlaces), and agents conduct 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). Optimizations can be made when abstracting Spark RDDs for graph use, such as data locality partitioning. The performance gains from these optimizations are the key concepts we will compare with MASS.

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.

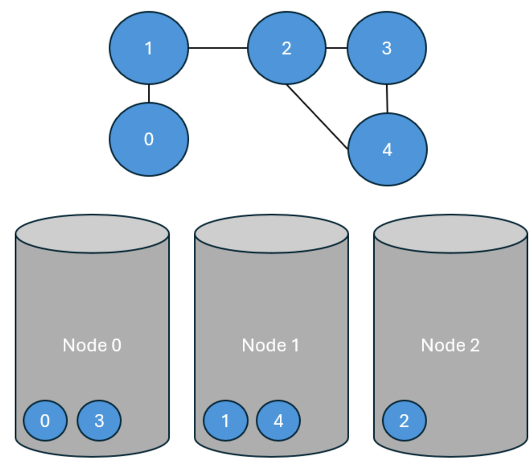


Figure 1

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 2, where V is a vertex in the graph, N_V is the number of links between its neighbors and K_V is its degree.

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

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 areas of a graph, which is helpful in applications such as social networks, where different communities may be loosely connected, or in web graphs, where certain pages are accessible 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 starts 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 method aids in analyzing a directed graph's overall structure and fragmentation, revealing how many different groups exist and how they are internally connected.

2.4. Strongly Connected Components

A Strongly Connected Component (SCC) is a subgraph of a directed graph in which every vertex is reachable from every other vertex in the component following the direction of edges. In other words, for any two vertices u and v in a strongly connected component, there must be a directed path from u to v and from v to u.

SCCs are critical in understanding the internal structure of directed graphs, especially in systems where mutual reachability matters, such as control flow graphs in compilers, software dependency graphs, or web link structures. Identifying SCCs can help detect cycles, uncover tightly-knit modules, or isolate parts of a system that can function independently.

The Strongly Connected Components algorithm decomposes a directed graph into maximal subsets of vertices where each vertex can reach every other vertex through directed paths. In distributed environments, algorithms typically use a vertex messaging paradigm (As well as for weakly connected components).

1. Implementation

1.1. *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.

1.2. *Clustering Coefficient*

1.2.1. 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

```
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));
    }
```

1.2.2. 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()
  }
```

1.3. Weakly Connected Components

1.3.1. MASS Implementation

The previous MASS implementation of Weakly Connected Components (WCC) uses agents to propagate the minimum vertex ID throughout the graph. Initially, an agent is created at each vertex, with its component ID assigned to the vertex's ID.

This previous implementation garnered poor results, as discussed in the results section. Noel Beraki, a member of the DSLab group, used a similar approach to his strongly connected components implementation (discussed in the SCC Implementation section) in a new implementation, where a single pivot vertex is selected and the graph is traversed with agents from there in a BFS fashion.

Beginning with a single agent at the pivot vertex, agents are spawned at neighboring nodes, marked as part of the current component, and visited. The master maintains a set of visited vertices to track which vertices have not been placed as part of a component. Once one BFS search is complete, another occurs, beginning with the following vertex not yet visited. You can find Noel Beraki's term report on the DSLab website to read more about his approach.

Figure 6: GraphX Weakly Connected Components Agent Vertex Arrival Code

```
public Object onArrival() {
        MyVertex place = (MyVertex) getPlace();
        int placeId = place.getIndex()[0];

        if (place.visitedAgents.contains(originalPlaceId)) {
            kill();
            return null;
        }

        visited.add(placeId);
        place.visitedAgents.add(originalPlaceId);

        if (place.componentID >= originalPlaceId) {

            place.componentID = originalPlaceId;
            Object[] placeNeighbors = place.getNeighbors();
            ObjectList<ArgsToAgents> nextVertices = new ObjectArrayList<>();

            // Figure out which neighbors can be visited
            nextVertices = GetGoodNeighbors();

            if (!nextVertices.isEmpty()) {
                ArgsToAgents[] args = nextVertices.toArray(new
ArgsToAgents[0]);spawn(args.length, args);
            }
        }

        kill();

        return null;
    }
```

1.3.2. GraphX Implementation

GraphX weakly connected components can be found in the GraphX library. The algorithm employs an iterative label propagation method to identify subgraphs where all vertices are reachable from each other when edge direction is disregarded. The process begins by assigning each vertex 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 spreading the lowest ID throughout the component. Because edge directions are disregarded, label propagation occurs bidirectionally between vertices. This iterative process continues until convergence, meaning all vertices within the same connected component have the same label. Once there are no further label changes, the algorithm terminates, efficiently grouping weakly connected subgraphs within the graph.

Figure 7: GraphX Weakly Connected Components Implementation (From the GraphX library)

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED],
                                      maxIterations: Int): Graph[VertexId, ED] = {
    require(maxIterations > 0, s"Maximum of iterations must be greater than 0," +
      s" but got ${maxIterations}")

    val ccGraph = graph.mapVertices { case (vid, _) => vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]): Iterator[(VertexId, VertexId)] = {
      if (edge.srcAttr < edge.dstAttr) {
        Iterator((edge.dstId, edge.srcAttr))
      } else if (edge.srcAttr > edge.dstAttr) {
        Iterator((edge.srcId, edge.dstAttr))
      } else {
        Iterator.empty
      }
    }
    val initialMessage = Long.MaxValue
    val pregelGraph = Pregel(ccGraph, initialMessage,
      maxIterations, EdgeDirection.Either)(
      vprog = (id, attr, msg) => math.min(attr, msg),
      sendMsg = sendMessage,
      mergeMsg = (a, b) => math.min(a, b))
    ccGraph.unpersist()
    pregelGraph
  }
```

1.4. Strongly Connected Components

1.4.1. MASS Implementation

For the MASS implementation of SCC (Strongly Connected Components), developed by Noel Beraki, we utilize a parallel-BFS approach instead of the traditional messaging method. First, some preprocessing steps are necessary for agents to follow reverse edges. Then, two agents are spawned at a randomly selected pivot vertex. These agents perform a BFS traversal to the predecessors and successors of the pivot vertex, spawning agents for each neighbor and marking them as part of the component. Once all reachable vertices have been visited, a new pivot vertex is chosen for the remaining components, and the process is repeated.

1.4.2. GraphX Implementation

The GraphX implementation for Strongly Connected Components (SCC) is structurally similar to the WCC implementation but includes additional pruning steps. Vertices with no incoming or outgoing edges (i.e., zero in-degree or out-degree) are initially removed from the graph and directly assigned to their component, reducing the graph size before the main computation.

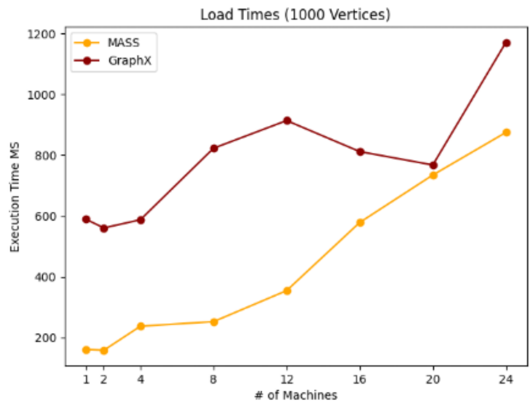
After pruning, the algorithm utilizes the Pregel API to propagate component IDs. Each vertex starts with its ID as its component label and sends messages to its neighbors indicating its current component ID. If a vertex receives a lower component ID than its own, it updates its label and propagates the new value. This message-passing continues iteratively until no further updates occur.

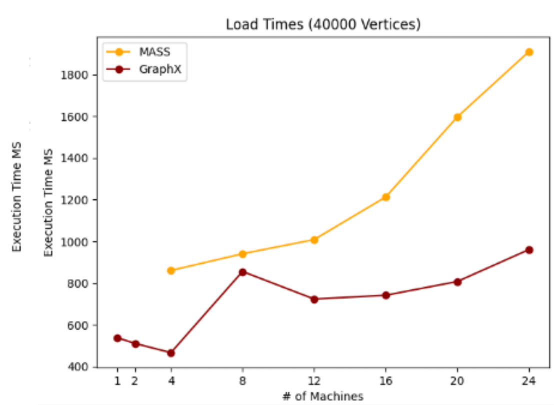
2. Results

All benchmarks have been run on the CSSMPI & Hermes machines using the same graph files.

2.1. 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 larger number of machines more gracefully.





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

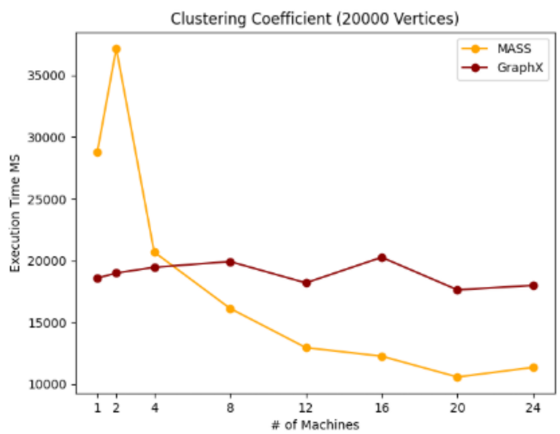
Graph 2: Load Times for 1-24 Computing Nodes w/ 40k vertices Graph

However, MASS processes smaller graphs faster than GraphX because GraphPlaces are distributed in a simple round-robin fashion across computing nodes, while GraphX employs a partitioning algorithm that considers data locality. In contrast, GraphX incurs overhead from shuffling data due to the absence of pre-partitioned vertices. Since GraphX does not pre-partition vertices before processing, it must dynamically reorganize the data, introducing additional computational overhead. In contrast, MASS's round-robin assignment of GraphPlaces enables faster initialization and execution in smaller-scale graphs.

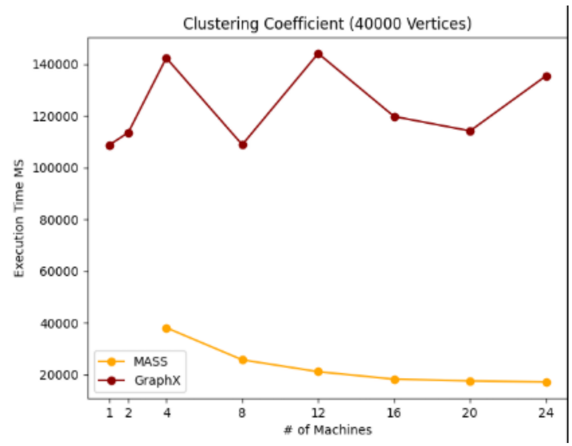
2.2. *Clustering Coefficient Results*

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.



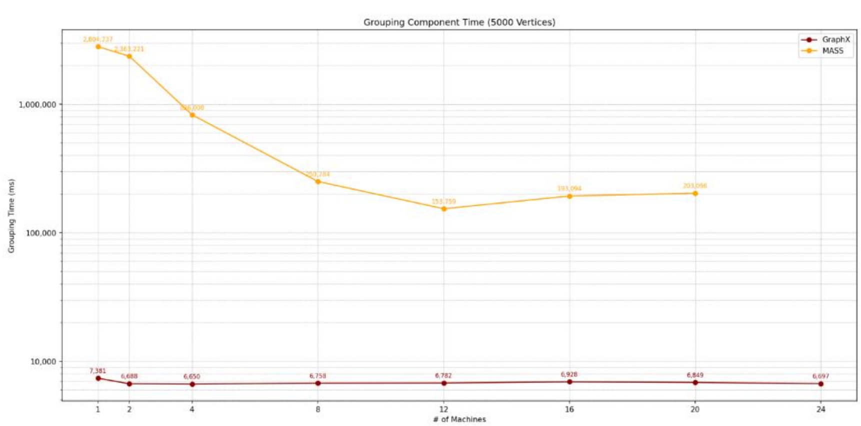
Graph 3: 1-24 Computing Nodes computing Clustering Coefficient on 20k vertices



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

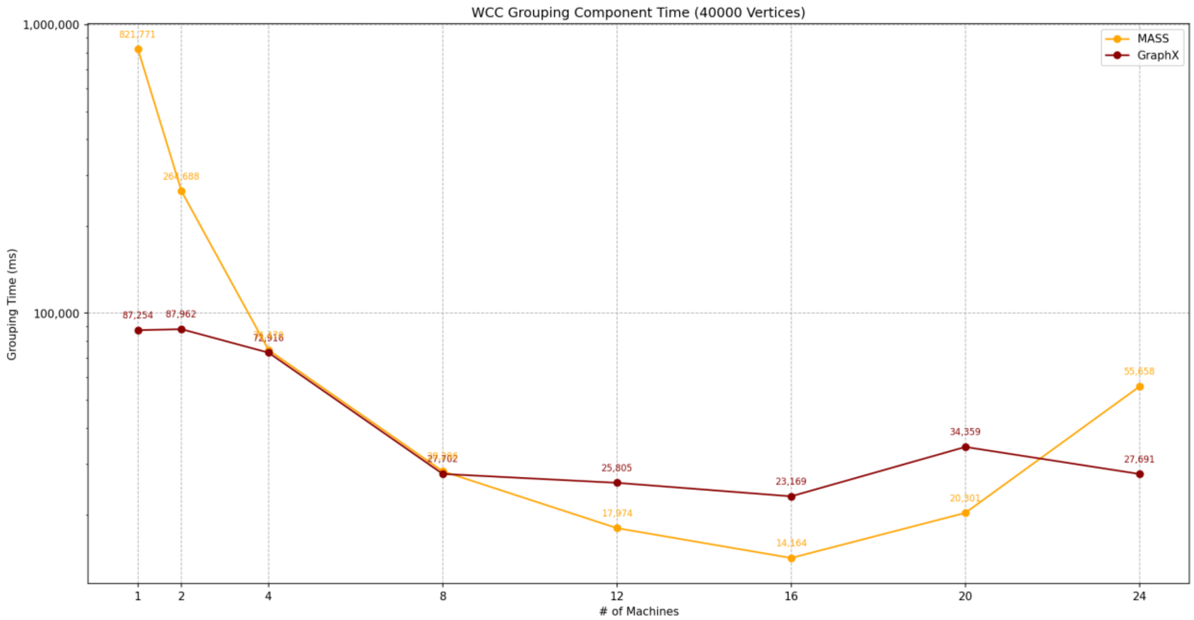
2.3. *Weakly Connected Components Results*

The previous implementation of WCC uses an algorithm similar to GraphX's WCC implementation, which uses agents to send messages to the vertices. However, this approach leads to an explosion of agents spawning. The memory overhead of transmitting agents as messages between vertices is too large and has made it apparent that a traditional parallel approach like this is unfeasible for MASS applications. GraphX also uses a partitioning algorithm that considers data locality, which is a significant boost for algorithms that send messages to neighboring vertices, since it limits inter-cluster communication.



Graph 5: 1-24 Computing Nodes computing the previous WCC implementation on 5k vertices.

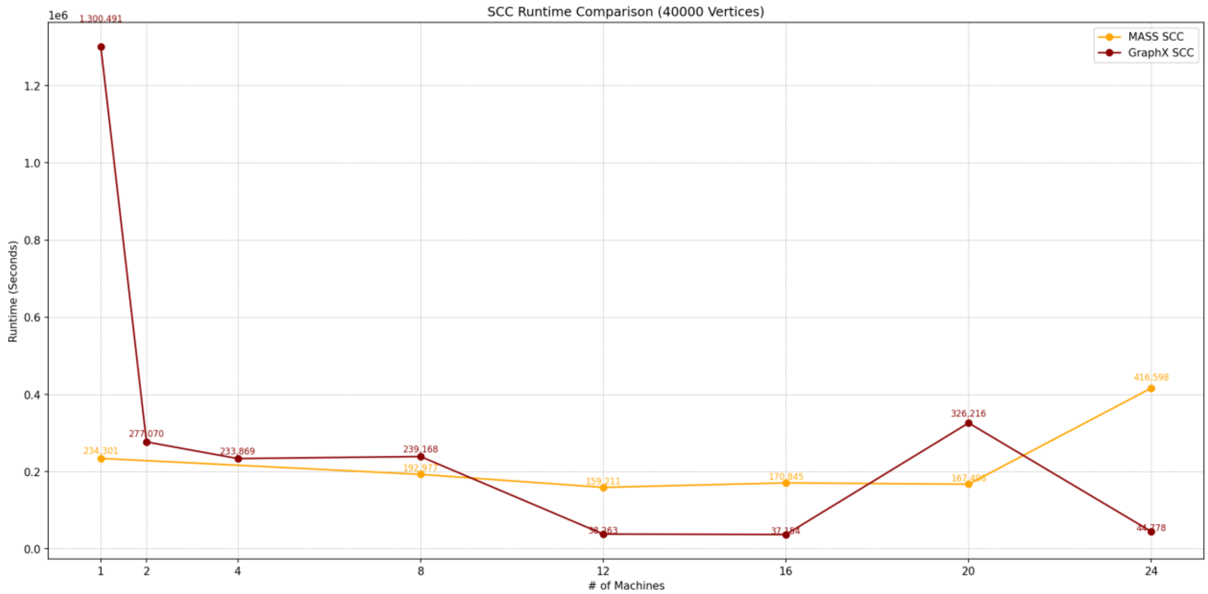
The new implementation uses the parallel-BFS approach that was developed when making the SCC implementation. This approach has led to more competitive results and better scales than GraphX. This is surprising because the version of MASS used for these benchmarks does not utilize a data partitioning algorithm that considers data locality.



Graph 6: 1-24 Computing Nodes computing the previous WCC implementation on 40k vertices.

2.4. Strongly Connected Components

The new SCC implementation in MASS yields competitive results. Reduced agent migration overhead and significantly fewer agents (compared to the WCC implementation) enable feasible computation times comparable to GraphX. While the vertex-messaging approach in GraphX appears to be the best method for a highly parallel implementation, utilizing a BFS that can run in parallel with fewer messages is equally effective in MASS.



Graph 7: 1-24 Computing Nodes computing SCC on 40k vertices.

2.5. Programmability

Spark, which includes the GraphX library, utilizes the MapReduce computing paradigm, making it straightforward and familiar. It is also part of the Apache Software Foundation, allowing numerous open-source contributions. These contributions have simplified Spark programs, making them easier to use than MASS, which requires more setup since it 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 serves as the distributed data structure representing the vertices of a graph; it extends the existing VertexPlace to incorporate custom logic. Another file extends the agent class to implement tailored logic for the individual agents executing clustering coefficient calculations. We created a custom class to pass arguments to agents. Finally, we have the ClusteringCoefficient.java master program, which orchestrates the execution. This collection of files complicates creating the MASS program compared to GraphX's straightforward one-file benchmark. The total number of files and other programming details is displayed in Table 1. All graph computing applications in MASS follow this class structure. Other programmability data can be found in the appendix.

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

3. Conclusion

Evaluating different distributed computing systems and understanding their underlying design choices has helped me develop a deeper understanding of how and why each implementation behaves the way it does.

Three benchmark programs have been implemented in both MASS and GraphX. The benchmark results for weakly connected and strongly connected components illustrate the need for high-messaging algorithms in MASS to be implemented using a parallel-BFS approach rather than vertex messaging. The DSLToGraphX program was also implemented to load DSL graph files into GraphX.

Future work in MASS benchmarking could involve re-implementing weakly connected components using a parallel BFS approach, similar to SCC, to ensure its superiority over the obvious approach. Considering agent migration overhead to reduce communication time could benefit MASS. Implementing an articulation point benchmark in GraphX to evaluate it against the existing MASS BFS approach could provide further insights into BFS techniques in distributed agent computing.

Appendix A: DSL File GraphX Parsing

Figure 7: Parse Edges

```java
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

```java
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: Weakly Connected Components Results

Table 4: MASS Performance

num-members	num-vertices	total-agents-generated	load-time (sec)	runtime (sec)
1	1000	92481	1.025	1.725
2	1000	92481	2.53	1.799
4	1000	92481	4.877	1.556
8	1000	92481	9.821	1.776
12	1000	92481	15.587	2.018
16	1000	92481	31.594	2.419
20	1000	92481	40.641	4.602
24	1000	92481	55.086	20.897
1	3000	290805	1.12	7.464
2	3000	290805	2.642	4.796
4	3000	290805	4.979	3.219
8	3000	290805	9.907	2.853
12	3000	290805	15.502	2.998
16	3000	290805	32.163	3.325
20	3000	290805	44.668	5.957
24	3000	290805	56.991	22.528
1	5000	487891	1.244	17.349
2	5000	487891	2.665	8.469
4	5000	487891	5.065	4.852
8	5000	487891	10.168	3.812
12	5000	487891	15.756	3.896
16	5000	487891	32.166	4.155
20	5000	487891	40.867	6.826
24	5000	487891	55.292	23.319
1	10000	979991	1.4	57.924
2	10000	979991	2.871	21.852
4	10000	979991	5.247	9.441
8	10000	979991	10.16	6.271
12	10000	979991	15.763	5.214
16	10000	979991	31.962	5.549
20	10000	979991	41.044	9.511
24	10000	979991	58.009	30.984
1	20000	1966381	1.898	206.237
2	20000	1966381	3.091	68.935
4	20000	1966381	5.404	24.402
8	20000	1966381	10.344	11.588
12	20000	1966381	16.046	8.259

16	20000	1966381	32.309	8.258
20	20000	1966381	41.529	13.467
24	20000	1966381	56.432	42.204
1	40000	3954425	2.886	821.771
2	40000	3954425	3.79	264.688
4	40000	3954425	5.798	74.47
8	40000	3954425	10.72	28.296
12	40000	3954425	16.543	17.974
16	40000	3954425	32.356	14.164
20	40000	3954425	43.448	20.301
24	40000	3954425	59.305	55.658

Table 5: GraphX Performance

num-members	num-vertices	Time elapsed of reading graph	Time elapsed of grouping components	Time elapsed of collecting components	Total time elapsed
1	1000	488	5511	130	6144
1	3000	486	6086	200	6797
1	5000	425	7381	144	7966
1	10000	453	10024	196	10687
1	20000	439	16158	250	16863
1	40000	512	87254	262	88052
2	1000	556	5046	1794	7413
2	3000	486	5833	169	6511
2	5000	443	6688	1787	8938
2	10000	503	10031	175	10725
2	20000	432	15200	212	15867
2	40000	574	87962	265	88819
4	1000	528	5195	140	5879
4	3000	492	5785	1677	7970
4	5000	416	6650	177	7262
4	10000	538	9285	203	10042
4	20000	453	15378	228	16074
4	40000	496	72916	1678	75113
7	1000	694	5578	1690	7983
8	3000	880	5819	144	6860
8	5000	665	6758	1596	9038
8	10000	680	9950	1831	12478
8	20000	608	15612	1738	17978
8	40000	1311	27702	2883	31968
12	1000	776	25282	2202	28286

12	3000	871	6037	1672	8601
12	5000	663	6782	1545	9011
12	10000	677	9255	1495	11445
12	20000	659	15202	1621	17502
12	40000	1007	25805	2438	29323
16	1000	808	4954	1419	7202
16	3000	748	6819	1639	9228
16	5000	789	6928	1538	9277
16	10000	710	9470	1523	11724
16	20000	767	15359	1567	17708
16	40000	996	23169	2627	26812
20	1000	751	4805	1793	7366
20	3000	734	5810	1681	8242
20	5000	768	6849	1722	9361
20	10000	790	9109	1600	11520
20	20000	713	15306	1723	17759
20	40000	1873	34359	2446	38696
24	1000	851	5093	1671	7637
24	3000	699	5870	2098	8684
24	5000	661	6697	2132	9507
24	10000	786	8310	1497	10614
24	20000	721	15814	1348	17902
24	40000	2188	27691	325	30221

Appendix D: Strongly Connected Components Results

Table 6: MASS Performance

num-members	num-vertices	total-agents-generated	load-time (sec)	runtime (sec)
1	1000	367926	1.092	6.391
2	1000	367926	2.601	7.882
4	1000	367926	5.222	6.37
8	1000	367926	10.295	6.529
12	1000	367926	25.006	14.679
16	1000	367926	24.355	8.429
20	1000	367926	32.504	9.814
24	1000	367926	48.897	31.079
1	3000	1157222	1.099	18.41
2	3000	1157222	2.517	20.679
4	3000	1157222	5.22	18.155
8	3000	1157222	10.206	16.125

12	3000	1157222	23.777	34.037
16	3000	1157222	25.747	16.111
20	3000	1157222	30.478	20.65
24	3000	1157222	46.875	62.471
1	5000	1941566	1.214	32.711
2	5000	1941566	2.62	30.87
4	5000	1941566	5.173	27.232
8	5000	1941566	10.049	24.565
12	5000	1941566	23.958	52.344
16	5000	1941566	22.529	22.971
20	5000	1941566	30.387	33.54
24	5000	1941566	46.823	77.886
1	10000	3899966	1.476	61.159
2	10000	3899966	2.98	60.681
4	10000	3899966	5.255	47.76
8	10000	3899966	10.328	41.672
12	10000	3899966	24.826	76.454
16	10000	3899966	23.72	41.478
20	10000	3899966	30.927	52.704
24	10000	3899966	49.191	133.243
1	20000	7825526	1.92	122.66
2	20000	7825526	3.151	118.97
4	20000	7825526	5.252	98.179
8	20000	7825526	10.404	76.721
12	20000	7825526	26.041	119.219
16	20000	7825526	23.006	71.258
20	20000	7825526	31.704	96.708
24	20000	7825526	48.066	217.065
1	40000	15737702	2.865	234.301
2	40000	NA	NA	NA
4	40000	NA	NA	NA
8	40000	15737702	10.802	192.977
12	40000	15737702	16.901	159.211
16	40000	15737702	23.838	170.845
20	40000	15737702	33.193	167.496
24	40000	15737702	53.377	416.598

Table 6: GraphX Performance

num-members	num-vertices	LoadTime	RunTime
1	1000	614	13136
1	3000	558	19180
1	5000	579	20335

1	10000	543	38710
1	20000	615	107210
1	40000	628	1300491
2	1000	569	4218
2	3000	610	5150
2	5000	569	5433
2	10000	545	8762
2	20000	575	23435
2	40000	575	277070
4	1000	654	3953
4	3000	571	4824
4	5000	568	4816
4	10000	635	8217
4	20000	615	20049
4	40000	586	233869
8	1000	582	3667
8	3000	592	4857
8	5000	633	4555
8	10000	584	7407
8	20000	648	19918
8	40000	506	239168
12	1000	1065	3526
12	3000	1196	4795
12	5000	833	4791
12	10000	909	7857
12	20000	916	8878
12	40000	915	38263
16	1000	909	4280
16	3000	818	4510
16	5000	918	5013
16	10000	1241	6400
16	20000	961	9422
16	40000	1048	37154
20	1000	988	3633
20	3000	1139	5031
20	5000	894	6216
20	10000	1280	6298
20	20000	1072	9688
20	40000	885	326216
24	1000	1062	4804
24	3000	1144	4772
24	5000	910	5278
24	10000	1226	5742
24	20000	920	9674

24	40000	932	44778

Appendix E: More Programmability Data

Table 3: MASS & GraphX Weakly Connected Components Programmability Data

Measurement (MASS)	Count
Number of files	4
Number of methods	20
Number of variables declared	74
Total lines of code	631
Lines of logic	126

Measurement (GraphX)	Count
Number of files	2
Number of methods	2
Number of variables declared	24
Total lines of code	190
Lines of logic	11

Table 4: MASS & GraphX Strongly Connected Components Programmability Data

Measurement (MASS)	Count
Number of files	4
Number of methods	22
Number of variables declared	89
Total lines of code	625
Lines of logic	82

Measurement (GraphX)	Count
Number of files	2
Number of methods	2
Number of variables declared	24
Total lines of code	197
Lines of logic	33

Appendix F: How to Run 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.

   The path for the graph file should be the absolute path.

   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>