# MASS Java Benchmarks for a Graph Database

## 1. Overview

Distributed graph databases play a critical role in modern computing, they power applications in social networks, transportation systems, and recommendation engines. Evaluating their performance and programmability is crucial to understanding the trade-offs between different approaches. MASS (Multi-Agent Spatial Simulation) is a parallel-computing library being developed by the Distributed Systems Laboratory (DSL). MASS is designed for distributed memory systems to simulate agent-based models for spatial computations. MASS has previously been used to implement a distributed graph database.

The University of Washington Bothell DSL would like to compare the MASS-based graph database to alternative implementations of a distributed graph database such as Hazelcast. Hazelcast is a distributed in-memory computing platform that will be benchmarked against MASS's agent-based approach to graph database computations. The Hazelcast distributed graph database has been implemented using Hazelcast's distributed hashmap (Imap). Hazelcast has a wide range of configurability, including choices of consistency, replication of data, and functionality for continuing computation even if communication with a computing node is lost. The goal is to compare the performance, programmability, and scalability of both approaches for a distributed graph database.

Atul Ahire, another member of the DSL, is implementing improvements to the MASS graph database including a new partitioning algorithm that improves data affinity of the database. Another improvement for MASS that is being made is improvements to the messaging system. Currently most of the messaging is done using TCP but a switch to Aeron is being tested which may reduce latency and increase performance. The first partitioning algorithm that was chosen was Hazelcast which uses hash-based partitioning and doesn't take data locality into account. The second partitioning method that was chosen was METIS which is a partitioning algorithm that divides a graph into partitions while minimizing the number of edges that are cut between the partitions. The vertex partitioning improvements will likely increase load times for the graph but decrease execution time for graph benchmarks.

For this capstone, I will implement two benchmark algorithms in MASS as well as in Hazelcast. The first goal is to evaluate these libraries including programmability and performance of the two libraries for a distributed graph database. The second goal of these benchmarks is to demonstrate the performance improvements made by Atul as mentioned above. The benchmarks are as follows:

1. Clustering coefficient
2. Articulation points

## 1.1 Background

In MASS, places are distributed across the computing nodes and represent the data, while agents are execution entities that traverse and exchange information with these places. A graph database has been implemented with MASS where graph nodes are represented as places (GraphPlaces) and agents can traverse these nodes to perform graph operations such as triangle count, clustering computations (connected components), and connectivity analysis. The following is a breakdown of the clustering coefficient and articulation points benchmarks that I will implement in MASS and Hazelcast.

## 1.2 Clustering Coefficient

Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Calculating the clustering coefficient of a graph can help you understand how tightly knit the groups in your graph are. In social networks, the clustering coefficient can help identify communities, while in transportation networks, it can highlight stations that may become bottlenecks. Clustering coefficients are on a scale from zero to one, where zero is a completely disconnected graph, and a coefficient of one is a graph with connections between every node. There are two types of clustering coefficients. The first is a local clustering coefficient of a given vertex that will tell you the portion of a vertex's neighbors that are neighbors of each other. In other words, the local clustering coefficient tells you how connected a vertex's neighbors are. The other form of clustering coefficient is a global clustering coefficient which is a measure of the clustering tendency of the entire graph. The global or average clustering coefficient of a graph helps bring understanding to the overall connectivity in a network. For example, in energy grids, the average clustering coefficient for a network could help determine the level of redundancy in the grid.

To calculate the local clustering coefficient there are two methods the first is the number of triangles that the vertex is a part of, divided by the number of triplets that are centered at that vertex. Another method for calculating it is to count the number of connections between that vertex's neighbors and divide it by the max possible number of connections between that vertex's neighbor. The maximum number of connections for a given vertex is simply n choose two or $\frac{n(n-1)}{2}$ (for an undirected graph).

To gauge the clustering of the entire graph, there are two methods, first is the average of the local clustering coefficients, and the second is the number of closed triplets in the graph divided by the number of all triplets in the graph. These two approaches give different results with the latter putting more weight on high-degree nodes, and the former putting more weight on low-degree nodes. My implementation will simply take the average of the local clustering coefficients as it is a simpler calculation and still gives us an idea of the graph's clustering.

## 1.3 Articulation Points

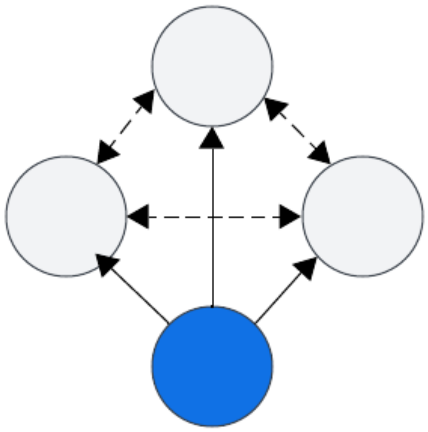
Articulation points (or cut vertices) are vertices in a graph that if removed, would split the network into two or more components. Articulation points are valuable for network design when you are trying to identify single points of failure in a network. In a transportation network, knowing that a station is an articulation point would let you know that its removal would make certain routes impossible. This is very similar to a graph bridge which is an edge in a graph that if removed would disconnect the graph into more components. It is important to note that while all articulation points are connected to a bridge, not all vertices connected to a bridge are articulation points.

The naive approach to finding articulation points would be to remove them from the graph and then traverse the graph (BFS or DFS) to see how many components you have. If the number of components increased, then it was an articulation point. This approach can be improved with Tarjan's Algorithm which is a DFS-based algorithm that also keeps track of each vertex's discovery time and lowest reachable discovery time, which is used to track connectivity and back edges more efficiently.

# 2. Implementation / Progress

## 2.1 Clustering Coefficient

*Figure 1 Local Clustering Coefficient*



### 2.1.1 MASS Implementation

The MASS implementation of clustering coefficient starts by spawning one agent on every vertex. The agents have an originalPlaceId that is stored at this point. Then every agent spawns agents equal to the number of neighbors and they all migrate to their respective neighbors. The agents then use the originalPlaceId to migrate back to the source with the neighbor vertex's neighbors list. This list of neighbor's neighbors can be referred to as second-degree neighbors. The agent then stores the second-degree neighbor list on the source GraphPlace. The GraphPlace calculates the local clustering coefficient of itself by counting how many of the second-degree neighbors it is also neighbors with. To calculate the global clustering coefficient a function has been implemented for the places CallAll which returns all the place's local clustering coefficients to the master node and then averages them for the network average clustering coefficient. More details are available in Appendix A.

**Figure 2: MASS Average Clustering Coefficient Implementation**

```
 1. private Object calculateCC() {
 2.     ClusteringVertex place = (ClusteringVertex) getPlace();
 3.     Object[] placeNeighbors = place.getNeighbors();
 4.
 5.     Set<Object> set1 = new HashSet<>(Arrays.asList(placeNeighbors));
 6.     Set<Object> set2 = new HashSet<>(Arrays.asList(secondDegreeNeighbors));
 7.     // Find intersection
 8.     set2.remove(nextPlaceId); //remove self loops
 9.     set1.retainAll(set2);
10.     place.neighborLinkCount += set1.size();
11.
12.     kill();
13.     return null;
14. }
```

## 2.1.2 Hazelcast Implementation

The Hazelcast implementation of clustering coefficient uses the implementation of Hazelcast's aggregator class which obfuscates all the parallel and distributes complexities into three simple functions. The first of which is the accumulate function which Hazelcast runs on every single vertex on that node. In this accumulate function the vertex's neighbors' vertices are retrieved. In Hazelcast we are able to retrieve the actual vertices that are neighbors which allows for the retrieval of second-degree neighbors easier programmatically. Then the accumulate function finds the size of the intersection of the second-degree neighbors and the neighbors of that vertex, this is the total number of connections between neighbors. To find the local clustering coefficient, the accumulate function calculates the maximum number of connections between neighbors and divides the actual number of connections by the maximum number of connections. The next function, the combine function adds the local clustering coefficients together. The last function, the aggregate function, divides the sum of the local clustering coefficients by the total number of vertices in the graph to give us our graph's average clustering coefficient.

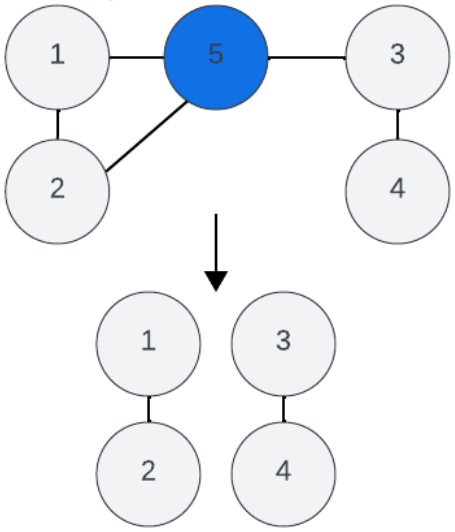
**Figure 3: Hazelcast Average Clustering Coefficient Implementation**

```
1. @Override
 2. public void accumulate(final Entry<I, Vertex<I, V>> entry) {
 3.     final var vertex = entry.getValue();
 4.     final var u      = vertex.getVertexID();
 5.     final Set<I> uNeighbors = vertex.getNeighbors().keySet();
 6.
 7.     Map<I, Vertex<I,V>> uNeighborVertices = dsg.graph.getAll(uNeighbors);
 8.
 9.     uNeighbors.forEach(v -> {
10.         if (Objects.equals(v, u)) return;  // Skip if the neighbor is the vertex itself
11.         Set<I> vNeighbors = uNeighborVertices.get(v).getNeighbors().keySet();
12.
13.         //retain elements in intersection
14.         vNeighbors.retainAll(uNeighbors);
15.         vertex.setLinkCount(vertex.getLinkCount() + vNeighbors.size());
16.     });
17.     vertex.setLocalClusteringCoefficient((uNeighbors.size() != 1 && !uNeighbors.isEmpty())
18.          (double) vertex.getLinkCount() / (uNeighbors.size() * (uNeighbors.size() - 1)) : 0.0);
19.     globalClusteringCoefficient += vertex.getLocalClusteringCoefficient();
20. }
```

Because file sizes are currently a bit smaller, an improvement could be made by using Hazelcast's project method which could return all vertex's neighbor IDs to each node when doing its computations. This method likely would not work as the graph size continues to grow and disregards the purpose of distributed storage (this is also how triangle count was implemented in Hazelcast).

## 2.2 Articulation Points

*Figure 4 Articulation Points*




### 2.2.1 MASS Implementation

The implementation for articulation points for MASS initially built upon Caroline Tsui's prior work for finding graph bridges. Her implementation uses a variation of Tarjan's algorithm where each vertex will store its discovery time and low link value. The low link value is the smallest vertex ID reachable from that vertex including itself. Caroline used an approach where agents propagate and migrate and essentially conduct Tarjan's algorithm but in a BFS manner. The problem with this approach is that it ends up missing certain bridges and articulation points and has very explosive agent growth. This means that this initial approach only could complete with graph files smaller than 30 vertices.

The approach that was taken to address this problem was to make a sequential implementation of this algorithm which essentially works just like the normal Tarjan's algorithm. If the user is attempting to find articulation points in file sizes with one hundred to one million edges, this sequential approach is better. But if the user is simply trying to use a benchmark that better shows the computational performance of MASS (albeit on a very small graph) then the standard articulation points benchmark is better. I will skip over implementation details of the original articulation points / graph bridge approach because as previously mentioned, the performance is non-optimal.

The sequential approach in MASS simply uses one agent that traverses the graph in a DFS manner. In MASS there is no way to simply check the attributes of a vertices neighboring places; to address this issue, child agents are spawned during the backtracking phase that traverse to the neighbors and return with the neighboring low link values. The DFS agent waits at this vertex until the children return to avoid race conditions, then the child agents call kill() on themselves. Initially the determination of a vertex being an articulation point was done during the backtracking phase and then a places call all was used to return all articulation points information. This had to be changed because the METIS and Hazelcast partitioning algorithms did not have the places call all function implemented yet. To address this a change was made to determine if each vertex was an articulation point at the end of the program with a separate set of

agents. This is slightly slower but should be valuable for comparing partitioning strategies for now. In the future this change should be reversed to calculate articulation points in the backtracking. It's also important to mention that this algorithm doesn't work on disconnected graphs currently. This is because MASS does not have the capability to spawn agents on a vertex based on a condition. For example, in DFS, a for loop iterates over all vertices and conducts a DFS starting at that vertex if it is not visited yet. There is currently no way to spawn an agent on a given vertex if it is unvisited. There are talks of changes that would allow for this, and the benchmark would just need to add the standard DFS loop to allow for disconnected graphs.

### 2.2.2 Hazelcast Implementation

The sequential implementation in Hazelcast is a bit simpler and mostly follows Tarjan's algorithm. The only difference is that information about neighbors must be retrieved from other compute nodes dropping the performance of the algorithm. Hazelcast does have a method for returning information about all neighboring vertices, but this data becomes stale as the DFS continues, and ends up with most of the databases data on one machine, defeating the purpose of a distributed database. This means each time the low link value of a neighbor is needed, a request is made to get that value and the process waits for the messaging. Hazelcasts executor service may be able to improve performance for this reason, although my assumption is that the overhead of shifting the computation from one machine to the next is larger than the overhead of retrieving a vertices information. Implementation specifics can be found in appendix A and B.

# 3. Benchmark Results

### 3.1.1 Clustering Coefficient Performance

The Clustering Coefficient benchmark has been run on the CSSMPI cluster, for larger vertex counts MASS generally outperforms the Hazelcast implementation slightly. The execution time for a graph with forty-thousand vertices can be seen in figure 5.

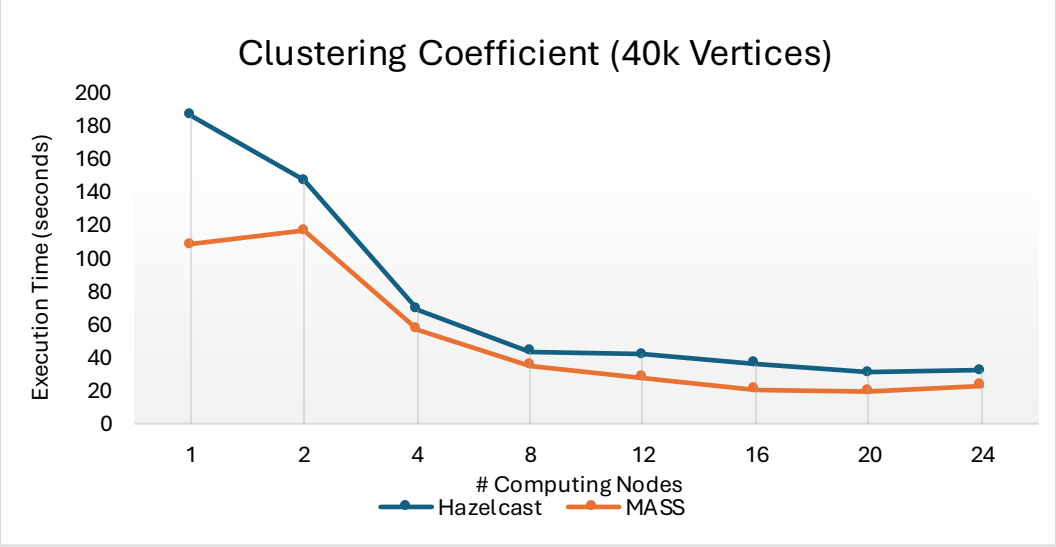



*Figure 5 Clustering Coefficient 40k Vertices*

The MASS implementation is slightly faster regardless of the node count, this is partly due to optimizations in MASS like completing all of a vertex's computation at once on the given place. MASS is especially faster on a single node because MASS takes full advantage of local vertices. This benchmark has been tested on other vertex counts with similar results and the data can be found in appendix C. Interestingly MASS performs a bit slower when increasing from one to two computing nodes. This increase in execution time is due to the overhead of communication between computing nodes including MProcesses. The benefits gained from increasing the number of computing nodes fall off for node counts larger than 16 nodes.

### 3.1.2 Clustering Coefficient Partitioning Algorithm Performance

The difference in runtime performance is negligible for the different partitioning strategies, although this was expected as clustering coefficient was meant to show MASS's performance against Hazelcast, and the articulation points benchmark was meant to show improvements from the partitioning algorithms better.

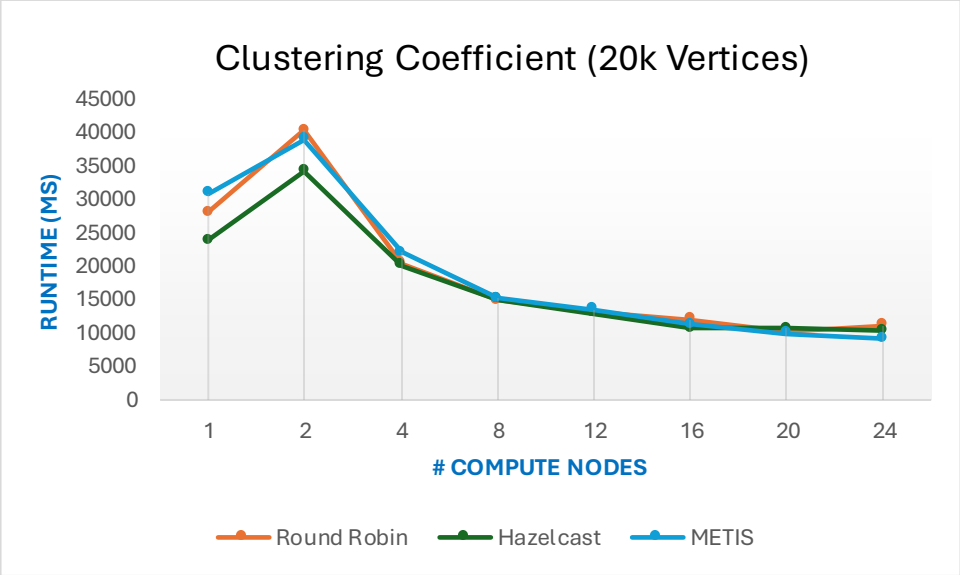



*Figure 6 Clustering Coefficient 20k Vertices Partitioning results*

While the Hazelcast partitioning seems to be a bit better, it's pretty marginal and becomes negligible with four or more compute nodes. All other results seem to show that partitioning strategy does not largely impact the clustering coefficient benchmark.

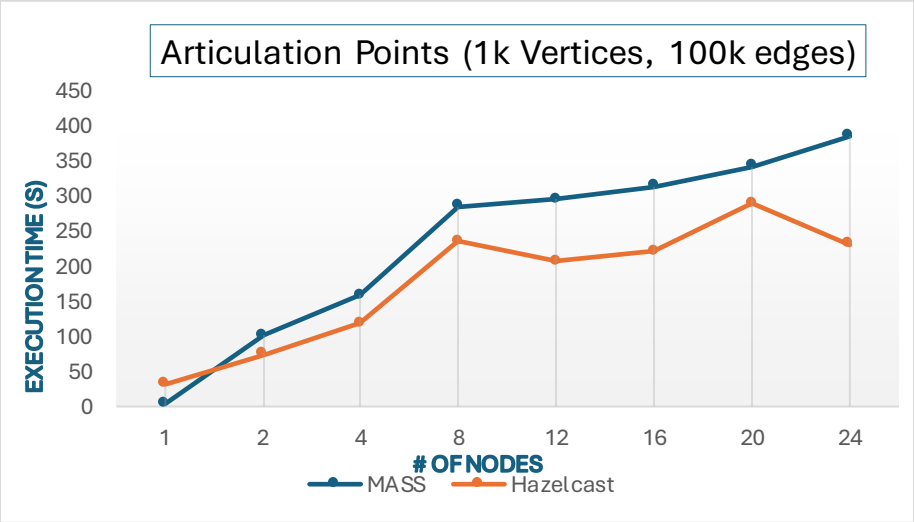



*Figure 7 Articulation Points 1k Vertices*

### 3.2.1 Articulation Points Performance

The Articulation Points benchmark was also run on the CSSMPI cluster and Hazelcast tends to outperform MASS, although there are some instances where MASS is a closer competitor. The execution time of

this benchmark with 1k vertices and 100k edges can be seen in figure 7.

While MASS was a bit faster for clustering coefficient, it seems that for articulation points it doesn't fair as well. With very dense graphs the difference in runtime between Hazelcast and MASS becomes negligible, but for less dense graphs Hazelcast beats MASS in runtime relatively decently. MASS has high overhead for passing an agent, while Hazelcast can simply return the data of a specific vertex which allows for smaller message sizes. As graph sizes get larger and more dense MASS's agent-based computation starts to scale better, while Hazelcast dominates with smaller sparse graphs because of its low overhead.

### 3.2.2 Articulation Points Partitioning Algorithm Performance

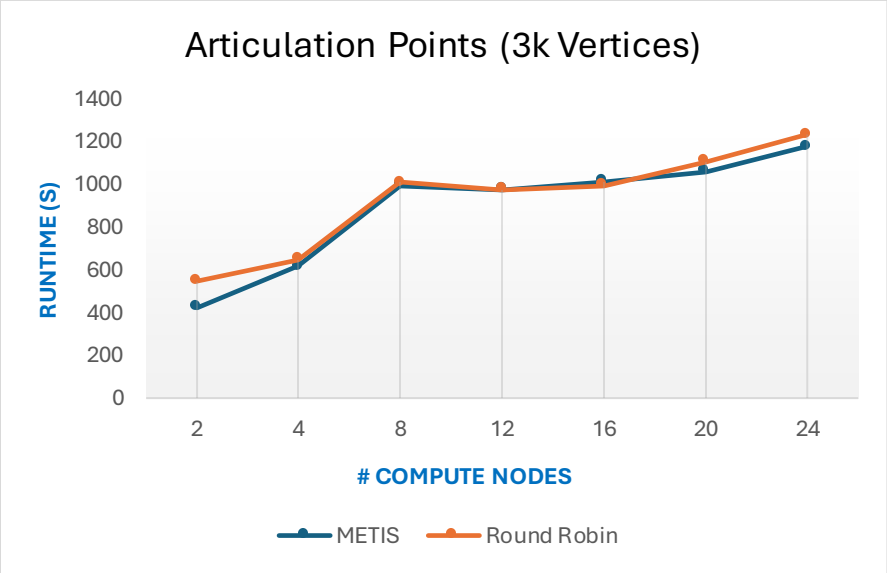



*Figure 8 Articulation Points 3k Vertices*

With smaller amounts of compute nodes, some pretty reasonable improvements could be seen with METIS. Initially benchmarks were showing that the METIS partitioning had negligible benefits to run time on most benchmarks when the number of compute nodes was increased, as can be seen in figure 8.

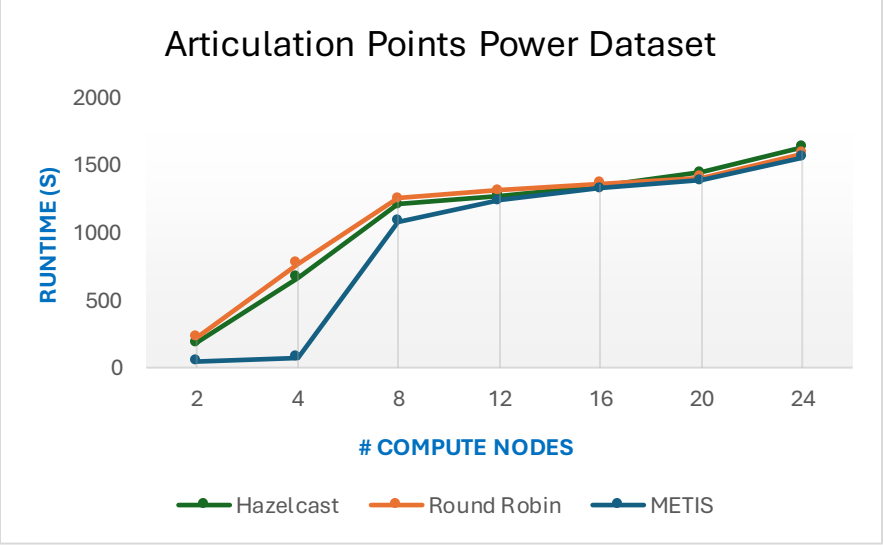



*Figure 9 Articulation Points US Power Grid*

The benchmark in figure 8 was done with a randomized dataset, and after some more benchmarking it was found that METIS's performance benefits can be seen much better with real world datasets that have more logical groupings and clusters within their graphs. More logical groupings means that there are less edges

cut between partitions. Figure 9 shows the performance improvements of the METIS partitioning strategy with the articulation points benchmark run on a dataset based on the US power grid.

With lower numbers of compute nodes, the improvements from METIS can be seen with as much as 80% reductions in run time for four compute nodes, and even with larger compute node counts, there is still 1-5% performance improvements. Future testing is needed for larger real-world datasets for different benchmarks with the Hazelcast and METIS partitioning strategies. The raw data from these benchmarks can be found in Appendix D.

# 4. Programmability

Both MASS and Hazelcast allow for distributed and parallelized computing without much knowledge of those concepts. I found the implementation in Hazelcast to be a bit more straightforward as it feels similar to concepts like MapReduce and thinking about a problem using agent-based modeling is a bit different to anything I've done. To tackle these benchmarks, an Agent and a VertexPlace class had to be designed to act out the computation, while Hazelcast only required one class to be designed. The aggregator for Hazelcast needed to be designed, though the implementation is rather simple since you implement the aggregate class and then override the methods you need. Technically the DistributedSharedGraph class also had to be implemented since there is no built-in distributed graph storage class.

In Table 9 we can see that MASS overall had fewer lines of code and less lines of logic, however, this is because Hazelcast does not have built-in data structures for a graph instead a distributed HashMap must be used to store the graph data. Additionally, Hazelcast logic was necessary for file I/O and graph initialization. If we account for these extra lines of code in Hazelcast, the implementations come out to similar quantitative complexity. I assume in the future that SmartAgents will continue to gain functionality, and if they had a migratePropogate function that did not terminate agents on seen vertices, and a migrateSource function that didn't kill agents when there is no neighboring link, then my implementation in MASS could be quantitatively less complex. As for articulation points, not much could easily be done to lower the quantitative complexity in MASS as agents cause the benchmarks to require slightly more lines of code.

**Table 9: MASS vs Hazelcast 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 (Hazelcast)** | Count |
|---|---|
| Number of files | 3 |
| Number of methods | 30 |
| Number of Variables declared | 54 |
| Total Lines of Code | 707 |
| Lines of logic | 295 |

**Table 10: MASS vs Hazelcast Articulation Points Programmability data**

| **Measurement (MASS)** | Count |
|---|---|
| Number of files | 4 |
| Number of methods | 20 |
| Number of Variables declared | 85 |
| Total Lines of Code | 714 |
| Lines of logic | 421 |

| **Measurement (Hazelcast)** | Count |
|---|---|
| Number of files | 4 |
| Number of methods | 27 |
| Number of Variables declared | 65 |
| Total Lines of Code | 683 |
| Lines of logic | 328 |

# 5. Conclusion

Both the clustering coefficient and articulation point benchmarks have been successfully implemented in both Hazelcast and mass, advancing DSL's capability to compare the MASS-based graph database with a distributed graph database in Hazelcast. This project has shifted my way of thinking greatly, as I had no experience with distributed computing beforehand. I underestimated the different mindset that is required for parallel and distributed computing. Now that the benchmarks have been run for the different partitioning strategies, we can now see the benefits and drawbacks of the different options.

Future work should continue to compare different distributed libraries in their use for a graph database. This will give us more data points to decipher MASS's strong suits. More benchmarks should be made in MASS to test the partitioning strategies, and as new changes are made to MASS, we should continue to benchmark the improvements that partitioning has on performance. As Aeron is being used for more of MASS's messaging, future work should monitor if these partitioning strategies have more of an effect on performance.

# Appendix A: More Implementation Details for MASS

## Clustering Coefficient

Figure 10: Propagation logic

```
1. private Object propagate() {
2.     MASS.getLogger().debug("==================== PROPAGATE ====================");
3.
4.     ClusteringVertex place = (ClusteringVertex) getPlace();
5.     Object[] placeNeighbors = place.getNeighbors();
6.     int placeId = place.getIndex()[0];
7.     int availableEdges = placeNeighbors.length;
8.
9.     // case 1. no neighbors, kill the agent!
10.    // or case 2. only one neighbor and that's itself, kill the agent! (self-loops not counted)
11.    // or case 3. one neighbor and that's the origin, kill the agent!
12.    if (availableEdges == 0 || (availableEdges == 1 && placeId == (int) placeNeighbors[0])
13.            || ((int) placeNeighbors[0] == originalPlaceId && availableEdges == 1)) {
14.        kill();
15.        return null;
16.    }
17.
18.    //it's ok to assume there is at least one neighbor because we've
19.    // already shown that if there's only one neighbor, then it's no the origin
20.    nextPlaceId = ((int) placeNeighbors[0] != originalPlaceId) ?
21.        (int) placeNeighbors[0] : (int) placeNeighbors[1];
22.
23.    int index = 0;
24.    if (placeNeighbors.length > 1 + ((nextPlaceId == (int) placeNeighbors[0]) ? 0 : 1)) {
25.        ArgsToAgent[] argsArr = new ArgsToAgent[placeNeighbors.length - 1];
26.
27.        for (int i = (nextPlaceId == (int) placeNeighbors[0])
28.            ? 1 : 2; i < placeNeighbors.length; i++) {
29.            if ((int) placeNeighbors[i] == originalPlaceId) {
30.                continue;
31.            }
32.            // (originalPlaceId, nextPlaceId)
33.            argsArr[index++] = new ArgsToAgent(originalPlaceId, (int) placeNeighbors[i]);
34.        }
35.
36.        //it's ok if argsArray is not completely full.
37.        spawn(index, argsArr);
38.    }
39.    return null;
40. }
```

Figure 11: MigrateSource Logic

```
1. private Object migrateSource() {
2.     MASS.getLogger().debug("==================== BACKTRACKING ====================");
3.     ClusteringVertex place = (ClusteringVertex) getPlace();
4.     secondDegreeNeighbors = place.getNeighbors();
5.     migrate(originalPlaceId);
6.     return null;
7. }
```

Figure 12: Main Crawler Logic

```
crawlers.callAll(MyAgent.init_);
for (int i = 0; i < 2; i++) {
    if (i < 1) {
        crawlers.callAll(MyAgent.propagate_);
        crawlers.manageAll();
        crawlers.callAll(MyAgent.migrate_);
    }
    else {
        crawlers.callAll(MyAgent.migrateSource_);
    }
    crawlers.manageAll();
}
crawlers.callAll(MyAgent.calculateCC_);
String[][] dummyArgs = new String[graphPlaces.size()][1];
return graphPlaces.callAll(ClusteringVertex.returnLocalCC_, dummyArgs);
```

## Articulation Points

Figure 13: DFS logic logic

```
private Object onArrival() {
    if(wait) {          wait = false;
        return null;
    }
    MyVertex place = (MyVertex) getPlace();
    int placeId = place.getIndex()[0];
    Object[] placeNeis = place.getNeighbors();
    if(neighborLowLinkCheck){
        low = place.low;
        lowLinkNeiCheckId = this.parentID;
         neighborLowLinkCheck = false;
         return null;       }
     if(lowLinkNeiCheckId != -1){
         place.low = Math.min(low, place.low);
         kill();
         return null;
    }
    if(!place.getVisited()) {
        disc++;
        low = disc;
        place.disc = disc;
        place.low = disc;
        place.parentId = (itinerary.isEmpty()) ? -1 :itinerary.getInt(itinerary.size()-1);
    }
    place.setVisited(true);
    for (Object obj : placeNeis) {
        int nei = (int) obj;
        if(!visited.contains(nei)){
            place.children++;
            nextPlaceId = nei;
            if(!itinerary.contains(placeId)) {
                itinerary.add(placeId);
            }
            visited.add(placeId);
            return null;
        }
    }
    int isRoot = (place.parentId == -1) ? 1 : 0;
    int arraySize = (placeNeis.length > 0) ? placeNeis.length - 1 + isRoot : 0;
    ArgsToAgent[] argsArr =  new ArgsToAgent[arraySize];

    int i = 0;
    for (Object obj : placeNeis) {
```

```
        int nei = (int) obj;
        if(place.parentId == nei) continue;
        argsArr[i] = new ArgsToAgent(nei, null, -1, -1, placeId, true);
        i++;
    }

    if(argsArr.length > 0){
       wait = true;
       spawn(argsArr.length, (Object[]) argsArr);
   }
   if(!itinerary.contains(placeId)) itinerary.add(placeId);
   visited.add(placeId);
    itinerary.removeInt(itinerary.size() - 1);
    if(itinerary.isEmpty()){
        kill();
        return null;
    }
    nextPlaceId = itinerary.getInt(itinerary.size() - 1);
    return null;
}
```

# Appendix B: More Implementation Details for Hazelcast

## Articulation Points

Figure 14: Hazelcast Articulation Points DFS implementation.

```
private Vertex<I,V> dfs(int nodeId, int parent, int time) {
Vertex<I,V> v = graph.get(nodeId);
Integer vertexId = (Integer) v.getVertexID();
visited.add(vertexId);

 v.setDisc(time);
 v.setLow(time);
 graph.put((I) vertexId, v);
 time++;
Set<I> neighborsKeySet = v.getNeighbors().keySet();
int children = 0;
for (I neighborKey : neighborsKeySet) {
    int neighborId = (Integer) neighborKey;
    if (neighborId == parent) continue;
    Vertex<I,V> neighbor = graph.get(neighborId);

    if (neighbor.getDisc() == -1) {
        children++;
        neighbor = dfs(neighborId, nodeId, time);
        v.setLow(Math.min(v.getLow(), neighbor.getLow()));
        graph.put((I) vertexId, v);

        if (parent != -1 && neighbor.getLow() >= v.getDisc()) {
            if (neighborsKeySet.size() > 1) {
                ArticulationPoints.add(nodeId);
            }
        }
    }else{
          v.setLow(Math.min(v.getLow(), neighbor.getDisc()));
          graph.put((I) vertexId, v);
     }
 }
   if(parent == - 1 && children > 1) ArticulationPoints.add(nodeId);
   return v;
}
```

# Appendix C: Clustering Coefficient Execution Results

The following are the results for an average of 3 runs, for each node and vertex combination running the Clustering Coefficient benchmark.

## Hazelcast Results

**Table 1: Hazelcast Clustering Coefficient Performance**

| Computing Nodes | # Vertices | # Edges | Load Time | Execution Time |
|---|---|---|---|---|
| 1 | 1000 | 93480 | 528 | 6567 |
| 2 | 1000 | 93480 | 635 | 5012 |
| 4 | 1000 | 93480 | 466 | 3078 |
| 8 | 1000 | 93480 | 530 | 2914 |
| 12 | 1000 | 93480 | 520 | 2491 |
| 16 | 1000 | 93480 | 998 | 2025 |
| 20 | 1000 | 93480 | 523 | 1951 |
| 24 | 1000 | 93480 | 986 | 1853 |
| 1 | 10000 | 989990 | 1738 | 46796 |
| 2 | 10000 | 989990 | 1869 | 37691 |
| 4 | 10000 | 989990 | 1434 | 20012 |
| 8 | 10000 | 989990 | 1236 | 13530 |
| 12 | 10000 | 989990 | 1210 | 12701 |
| 16 | 10000 | 989990 | 1175 | 12650 |
| 20 | 10000 | 989990 | 1698 | 12604 |
| 24 | 10000 | 989990 | 1684 | 12205 |
| 1 | 40000 | 3994424 | 3439 | 185808 |
| 2 | 40000 | 3994424 | 4042 | 146529 |
| 4 | 40000 | 3994424 | 3308 | 68860 |
| 8 | 40000 | 3994424 | 2932 | 43786 |
| 12 | 40000 | 3994424 | 2767 | 41499 |
| 16 | 40000 | 3994424 | 2831 | 36450 |
| 20 | 40000 | 3994424 | 2521 | 31120 |
| 24 | 40000 | 3994424 | 2528 | 32079 |

## MASS Result

**Table 2: MASS Clustering Coefficient Performance**

| Computing Nodes | # Vertices | # Edges | Load Time | Execution Time |
|---|---|---|---|---|
| 1 | 1000 | 93480 | 175 | 3714 |
| 2 | 1000 | 93480 | 145 | 3719 |
| 4 | 1000 | 93480 | 126 | 2711 |
| 8 | 1000 | 93480 | 158 | 2337 |
| 12 | 1000 | 93480 | 120 | 2121 |
| 16 | 1000 | 93480 | 102 | 2197 |
| 20 | 1000 | 93480 | 141 | 2267 |
| 24 | 1000 | 93480 | 141 | 2699 |
| 1 | 10000 | 989990 | 729 | 25863 |
| 2 | 10000 | 989990 | 542 | 25182 |
| 4 | 10000 | 989990 | 773 | 15726 |
| 8 | 10000 | 989990 | 916 | 13374 |
| 12 | 10000 | 989990 | 598 | 10703 |
| 16 | 10000 | 989990 | 477 | 7300 |
| 20 | 10000 | 989990 | 449 | 7461 |
| 24 | 10000 | 989990 | 639 | 8033 |
| 1 | 40000 | 3994424 | 1773 | 108073 |
| 2 | 40000 | 3994424 | 1435 | 116552 |
| 4 | 40000 | 3994424 | 1106 | 56930 |
| 8 | 40000 | 3994424 | 890 | 35081 |
| 12 | 40000 | 3994424 | 960 | 28096 |
| 16 | 40000 | 3994424 | 789 | 20336 |
| 20 | 40000 | 3994424 | 690 | 19352 |
| 24 | 40000 | 3994424 | 1007 | 22982 |

**Table 3: MASS Clustering Coefficient Performance with Partitioning (20k Vertices)**

| Partitioning Strategy | MASS | Hazelcast | METIS |
|---|---|---|---|
| 1 | 28016 | 23873 | 30815 |
| 2 | 40337 | 34148 | 38851 |
| 4 | 20457 | 20036 | 22106 |
| 8 | 14900 | 15087 | 15197 |
| 12 | 13064 | 12896 | 13476 |
| 16 | 11963 | 10644 | 11268 |
| 20 | 10230 | 10688 | 9894 |
| 24 | 11062 | 10360 | 9163 |

# Appendix D: Articulation Points Execution Results

## Hazelcast Results

**Table 4: Hazelcast Articulation Points 1k Vertices**

| # Nodes | # Vertices | Load Time | Run Time |
|---|---|---|---|
| 1 | 1000 | 479 | 31549 |
| 2 | 1000 | 900 | 73550 |
| 4 | 1000 | 522 | 117840 |
| 8 | 1000 | 900 | 235450 |
| 12 | 1000 | 571 | 206888 |
| 16 | 1000 | 1068 | 221153 |
| 20 | 1000 | 632 | 289307 |
| 24 | 1000 | 527 | 229947 |

## MASS Result

**Table 5: MASS Articulation Points (Round Robin)**

| # Nodes | # Vertices | Load Time | Run Time |
|---|---|---|---|
| 1 | 1000 | 1277 | 3447 |
| 2 | 1000 | 2989 | 98132 |
| 4 | 1000 | 5485 | 153165 |
| 8 | 1000 | 10376 | 283612 |
| 12 | 1000 | 15905 | 295073 |
| 16 | 1000 | 21948 | 321770 |
| 20 | 1000 | 27699 | 326961 |
| 24 | 1000 | 32840 | 365805 |
| 1 | 3000 | 1446 | 9750 |
| 2 | 3000 | 3267 | 547057 |
| 4 | 3000 | 5770 | 646419 |
| 8 | 3000 | 10598 | 1009576 |
| 12 | 3000 | 16357 | 976260 |
| 16 | 3000 | 22151 | 995574 |
| 20 | 3000 | 27896 | 1106928 |
| 24 | 3000 | 34281 | 1233555 |

**Table 6: More MASS Articulation Points with Round Robin Partitioning**

| # Nodes | # Vertices | Load Time | Run Time |
|---|---|---|---|
| 1 | 3000 | 1446 | 9750 |
| 1 | 4941 | 1218 | 5274 |
| 2 | 3000 | 3267 | 547057 |
| 2 | 4941 | 2859 | 221600 |
| 4 | 3000 | 5770 | 646419 |
| 4 | 4941 | 5650 | 770713 |
| 8 | 3000 | 10598 | 1009576 |
| 8 | 4941 | 11070 | 1250189 |
| 12 | 3000 | 16357 | 976260 |
| 12 | 4941 | 15966 | 1311934 |
| 16 | 3000 | 22151 | 995574 |
| 16 | 4941 | 21791 | 1359199 |
| 20 | 3000 | 27896 | 1106928 |
| 20 | 4941 | 28461 | 1406847 |
| 24 | 3000 | 34281 | 1233555 |
| 24 | 4941 | 43196 | 1580488 |

**Table 7: MASS Articulation Points with METIS Partitioning**

| # Nodes | # Vertices | Load Time | Run Time |
|---|---|---|---|
| 1 | 3000 | 1528 | 9904 |
| 1 | 4941 | 1163 | 9072 |
| 2 | 3000 | 3507 | 422055 |
| 2 | 4941 | 3100 | 44825 |
| 4 | 3000 | 6575 | 616976 |
| 4 | 4941 | 5782 | 77675 |
| 8 | 3000 | 12828 | 994508 |
| 8 | 4941 | 11230 | 1081067 |
| 12 | 3000 | 19373 | 975802 |
| 12 | 4941 | 16843 | 1238748 |
| 16 | 3000 | 27063 | 1014947 |
| 16 | 4941 | 24833 | 1322420 |
| 20 | 3000 | 34070 | 1060479 |
| 20 | 4941 | 32035 | 1383808 |
| 24 | 3000 | 40861 | 1177949 |
| 24 | 4941 | 35283 | 1552246 |

**Table 8: MASS Articulation Points with Hazelcast Partitioning**

| # Nodes | # Vertices | Load Time | Run Time |
|---|---|---|---|
| 1 | 3000 | 1478 | 7158 |
| 1 | 4941 | 1198 | 1380 |
| 2 | 3000 | 3346 | 180030 |
| 2 | 4941 | 2993 | 184759 |
| 4 | 3000 | 5933 | 342293 |
| 4 | 4941 | 5789 | 666722 |
| 8 | 3000 | 10997 | 789128 |
| 8 | 4941 | 10853 | 1203308 |
| 12 | 3000 | 16550 | 830513 |
| 12 | 4941 | 16358 | 1271779 |
| 16 | 3000 | 22038 | 886839 |
| 16 | 4941 | 21990 | 1345644 |
| 20 | 3000 | 28298 | 926611 |
| 20 | 4941 | 28058 | 1441640 |
| 24 | 3000 | 33873 | 1088204 |
| 24 | 4941 | 35500 | 1628955 |

# Appendix E: How To Run the Benchmark Programs

While I did run these programs with different partitioning strategies, I'm not going to list the instructions for running with these partitioning strategies as it's a very long process and they are still in development so these steps may change. Currently running these benchmarks requires modifications of the core to the agent mapping function.

## Clustering Coefficient

### *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**

*mvn clean package*
*cd target*

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

6. **Running the benchmark**
   *java -jar ArticulationPoints-1.0-SNAPSHOT <Path to input file> <print graph places boolean> <print articulation points boolean>*

### Hazelcast

1. **Clone repo**: (currently under jaday2/graphbenchmarks)
   *git clone -b jaday2/graph-benchmarks --single-branch https://bitbucket.org/mass_application_developers/mass_java_appl.git*

2. **Navigate to project directory**
   *cd mass_java_appl/Graphs/HAZELCASTClusteringCoefficient*

3. **maven package**
   Use included make file: *make*

4. **Update run.sh**
   Currently the run.sh file has been designed for the CSSMPI cluster. You can simply specify the number of compute nodes you would like to use. If you'd like you can switch the addresses in the machines list to any machines you'd like.

5. **Running the benchmark**
   *Bash run.sh <path to input file> <node count>*

   If you'd like to print the local clustering coefficients you can use *java -jar ClusteringCoefficient.jar <input file> <print results> <sort results>*
   Ex: *java -jar ClusteringCoefficient.jar graphInput.dsl true true*

   Extra arguments
   1. Print results: print the local clustering coefficients to the console.
   2. Sort results: because of the way Hazelcast results are returned, they are not in order like they are in MASS, if you'd like to see the local clustering coefficients in order by the vertex id, then type true, otherwise leave this argument blank or type false. Sorting the results has a marginal impact on execution results even for vertex counts up to 40k

## Articulation Points

### *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/ArticulationPointsSequential*

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 to use the correct nodes that 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>*

### *Hazelcast*

1. **Clone repo**: (currently under jaday2/graphbenchmarks)
   *git clone -b jaday2/graph-benchmarks --single-branch https://bitbucket.org/mass_application_developers/mass_java_appl.git*

2. **Navigate to project directory**
   *cd mass_java_appl/Graphs/Hazelcast Graph Benchmarks/ArticulationPointsSequential*

3. **maven package**
   mvn clean package
   cd target
4. **Update run.sh**
   Currently the run.sh file has been designed for the CSSMPI cluster. You can simply specify the number of compute nodes you would like to use. If you'd like you can switch the addresses in the machines list to any machines you'd like.

5. **Running the benchmark**

*Bash run.sh <path to input file> <node count>*

# Appendix F: Example Execution of Benchmarks

## Clustering Coefficient

The following is an example execution of the clustering coefficient benchmark using 1000 vertices and roughly 100 edges per vertex.

Both programs give a clustering coefficient of roughly 0.1082 which makes sense because the odds of an edge existing between 2 randomly selected nodes in this graph would be roughly 100 / 1000 or 0.1 (10%).

### *MASS*

```
java -jar ClusteringCoefficient-1.0.0-RELEASE.jar
"~/GITREPO/mass_java_appl/Graphs/HAZELCASTClusteringCoefficient/graph_1_000.dsl"
#alive-agent = 1000
#alive-agent = 93480
#alive-agent = 93480
Total number of agents used: 187960
Calculation Time: 844ms
Elapsed time = 3559ms
The average local clustering coefficient's of this graph is: 0.1081919281859751
MASS Shutdown Finished
```

### *Hazelcast*

```
bash run.sh "~/GITREPO/mass_java_appl/Graphs/HAZELCASTClusteringCoefficient/graph_1_000.dsl" 7
running test:
test finished after 8.406 seconds
test  result: 0.1082
Load Time:  1.057 seconds
Finished calculating global clustering coefficient...
Gathering benchmark results to clustering-coefficient-result.csv
Saved results to file...
Shutting down dsg cluster...
Finished dsg cluster shutdown...
```

## Articulation Points

The following is an example execution of the articulation points benchmark using different datasets. Both programs save the results to a file including the runtime, load time, articulation points found and more.

### *MASS*

```
java -jar ArticulationPointsSequential-1.0-SNAPSHOT.jar Dolphin.dsl
Start timer!
[Create places time] 1.108s

=============   GO! FIND ARTICULATION POINT(S)!   =============
Start the timer!
#alive-agent = 1
------------- TARJANS FINISHED ------------
------------- BEGIN SEARCH FOR ARTICULATION POINTS ------------
#alive-agent = 62
------------- STEP-1 PROPAGATE ------------
#alive-agent = 318
```

```
------------- STEP-2 MIGRATE ------------
#alive-agent = 318
------------- STEP-3 RETURN TO SOURCE ------------
#alive-agent = 318
------------- STEP-4 DETERMINE ARTICULATION POINTS ------------
#alive-agent = 9

Stop the timer!
====================  STATISTICS ====================
Elapsed time: 0.208 s
Total #place: 62
Total #agents: 1713

================ APPLICATION RESULTS! =================
Gathering benchmark results to articulation-points-benchmark-result.csv
Saved results to file...

---------------- Finishing MASS Java ... ----------------
MASS Shutdown Finished
```

### *Hazelcast*

```
bash run.sh graph_200.dsl 1
finished reading dsl file: 'graph_200.dsl'
Calculating articulation points (this might take some time)...
running test:
test finished after 1.832 seconds
Load Time:  0.215 seconds
Finished finding articulation points
Gathering benchmark results to articulation-points-benchmark-result.csv
Saved results to file...
Shutting down dsg cluster...
Finished dsg cluster shutdown...
```