Comparison of Distributed Graph Computing Performance Between Java MASS and Hazelcast

Table of Contents

1. Overview 2
2. MASS Background 3
3. Hazelcast Background 4
3.1 Hazelcast Overview 4
3.2 Hazelcast Features 4
4. Graph Benchmarks Overview 5
4.1 Strongly Connected Components (SCC) 5
4.2 Weakly Connected Components (WCC) 5
4.3 Triangle Counting 6
4.4 PageRank 7
4.5 Label Propagation 7
5. Benchmark Implementations 8
5.1 Strongly Connected Components Implementations 8
5.1.1 MASS Implementation 8
5.1.2 Hazelcast Implementation 11
5.2 Weakly Connected Components Implementations 12
5.2.1 MASS Implementation 12
5.2.2 Hazelcast Implementation 14
5.3 Triangle Counting Implementations 15
5.3.1 MASS Implementation 15
5.3.2 Hazelcast Implementation 15
5.4 PageRank Implementations 17
5.4.1 MASS Implementation 17
5.4.2 Hazelcast Implementation 17
5.5 Label Propagation Implementations 18
5.5.1 MASS Implementation 18
5.5.2 Hazelcast Implementation 19
6. Benchmark results 19
6.1 Strongly Connected Components Results 20
6.2 Weakly Connected Components Results 21
6.3 Triangle Counting Results 22
6.4 PageRank Results 22
6.5 Label Propagation Results 23
7. Programmability 24

8. Conclusion....26
Appendix A: Strongly Connected Components Execution Results....27
MASS SCC Results....27
Hazelcast SCC Using IMap.getAll() Results....28
Appendix B: Weakly Connected Components Execution Results....30
MASS WCC Results....30
Hazelcast WCC Using IMap.getAll() Results....31
Appendix C: Triangle Counting Execution Results....32
MASS Triangle Counting Results....32
Hazelcast Triangle Counting Results....34
Appendix D: PageRank Execution Results....35
MASS PageRank V2 Results....35
Hazelcast PageRank Results....37
Appendix E: Label Propagation Execution Results....38
MASS Label Propagation V2 Results....38
Hazelcast Label Propagation Results....39
Appendix F: Full Strongly Connected Components Execution Diagrams....42
Appendix G: Full Weakly Connected Components Execution Diagrams....43
Appendix H: Full Triangle Counting Execution Diagrams....44
Appendix I: Full PageRank Execution Diagrams....45
Appendix J: Full Label Propagation Execution Diagrams....46
Appendix K: Running Benchmarks....47

1. Overview

Agent-based modeling (ABM) deals with observing the interactions between a large number of agents representing some real-world entity. A focal point of multi-agent simulations is data/pattern discovery, and with a population size potentially ranging in the millions, a single computer isn't capable of handling all responsibilities. As well, a computer handling all of the work may be slow to find meaningful insights, so being able to paralyze the model to have concurrent execution using multiple computers can decrease the time needed to get results. For this reason, the Distributed System Laboratory (DSL) at the CSS division in UWB has developed a Java parallel-computing library named MASS (Multi-Agent Spatial Simulation).

MASS can be applied not only to conventional ABM simulations but also to graph computing, serving as a storage and retrieval system. Graph databases play a major role in modern computing and storage, being applied in many applications such as social networks, recommendation systems, and fraud detection. They represent the relationships between data (nodes/vertices) using edges that connect two nodes. The DSL is currently developing a distributed, agent-based, graph database system with MASS, and would like to evaluate MASS's graph computing performance and programmability against commonly used distributed computation platforms. One such system is Hazelcast.

My project focuses on comparing Java MASS's and Hazelcast's graph computing performance using frequently used graph algorithms. Specifically, my capstone was spent implementing and benchmarking the Strongly Connected Components (SCC), Weakly Connected Components (WCC), Triangle Counting, PageRank, and Label Propagation algorithms. For WCC, I had worked with Aria Naderi to improve the scalability of the previous implementation and MASS's PageRank as well as Label Propagation implementations were completed by Robert Zimmerman.

In terms of benchmarking, I used six different graphs with eight different cluster sizes, evaluating MASS and Hazelcast performances in terms of speed and scalability. As well, to assess their structure and maintainability, programmability metrics recorded were the lines of code (LOC), boilerplate %, cyclomatic complexity, and lack of cohesion in methods (LCOM4).

2. MASS Background

MASS distributes a multi-agent simulation among multiple machine nodes in-memory. Its composition is built mainly on two components: `Places` and `Agents`. A user's application is distributed among `Places`, a matrix/graph where each element is a `Place` object. A `Place` object is capable of storing and exchanging information amongst each other, and is the host location where agents reside. Performing computations on the `Places` are done using `Agents`, execution instances that are able to traverse the matrix. MASS spawns the same number of threads as that of CPU cores per machine node. Whereas places are mapped to threads, agents are mapped to processes, therefore allowing agents to communicate with each other via IPC and spawn child agents to provide parallel processing.

MASS developers have extended MASS to also be used as a graph database by creating the data structures `GraphPlaces`, `VertexPlace`, and `GraphAgent`, which extend from the `Places`, `Place`, and `Agent` classes. Vertices are distributed among the cluster through a round-robin approach as `GraphPlaces` stores a list of `VertexPlace` objects and represents a graph in an adjacency list format, where each `VertexPlace` object has a list of outgoing neighbors. As well, `GraphAgent` objects can override the `map()` function, which specifies how many agents to instantiate at each vertex. Users can create custom classes that extend these three classes for their applications.

MASS applications generally have 3-4 classes: An agent class that extends `GraphAgent`, a vertex class that extends `VertexPlace`, an `ArgsToAgents` class that specifies arguments/instance variables an agent will have, and a class with a `main()` method that runs the simulation. The program starts by creating a `GraphPlaces` object and an `Agents` object to instantiate a set of agents on the `GraphPlaces`. From there, a sequence of `onArrival()` and `migrateTo()` function calls can be made to each agent using `agents.callAll()` until all agents have finished their tasks and terminated or until some other condition ends the simulation.

3. Hazelcast Background

3.1 Hazelcast Overview

Hazelcast is an in-memory distributed computation and storage platform. It can serve as a distributed second level cache for applications, loading data on disk into memory and providing in-memory speeds to users. It stores data as key-value pairs in "shared" RAM spread across a cluster of machines. By default, Hazelcast offers 271 partitions, where a single partition is a memory segment holding a portion of the whole data. Partitions are evenly distributed amongst the cluster, and each partition has a backup copy residing on a different machine to provide fault tolerance. What partition holds a piece of data is determined by hashing the data and modding it to the total partition count. As well, repartitioning of data is automatically done when a machine leaves or joins the cluster.

Machines in the same subnet can form a cluster either through TCP/IP or Multicast discovery. By starting up a cluster instance with the same name, machines are able to discover each other automatically. However, after a cluster is formed, all communication between cluster members is done using TCP/IP.

3.2 Hazelcast Features

All Hazelcast data structures are thread safe, but only the `IMap` data structure (at least of what I used) is partitioned amongst cluster machines. An `IMap` is used to represent a graph, where the key is the vertex id and the value is a `Vertex` object that contains a map of the outgoing neighbors to the vertex.

Hazelcast provides a suite of distributed computing tools, allowing users to run tasks in parallel on different machines. Leveraging the combined processing power of the cluster, machines are able to send data over a network as long as the data can be serialized.

Hazelcast features I used in my implementations include:

- Predicates API: Used to query data from an `IMap`. The query is sent to each member in the cluster and it looks at its local partitions to send any entries that match the query.
- Entry Processor: Used for bulking processing on `IMap` entries. Given a set of keys (or the whole graph), it executes a read and update operation on the partition where the data resides, eliminating costly network hops.
- IExecutorService: Asynchronously executes tasks that don't require modifying `IMap` entries. The user creates a custom class that implements the `Callable` interface to return a value when the task completes (or `Runnable` when a return value is not necessary), and passes it into an `IExecutorService` object along with a specific `Member` or key to execute/submit the task to.
- Aggregators: Executing in parallel across all cluster members, it computes the value of a function over all the entries of an `IMap`. The process consists of three phases: accumulation where each partition runs the `accumulate()` function on its local entries; combination where each

partitions results in the accumulation phase are combined; and aggregation, where the combined result can be further processed before returned.

4. Graph Benchmarks Overview

4.1 Strongly Connected Components (SCC)

Given a directed graph, the strongly connected components algorithm seeks to find all the maximal subgraphs in which there is a directed path from any vertex to another. In other words, the goal is to find all of the connected subgraphs that exist within the whole graph.

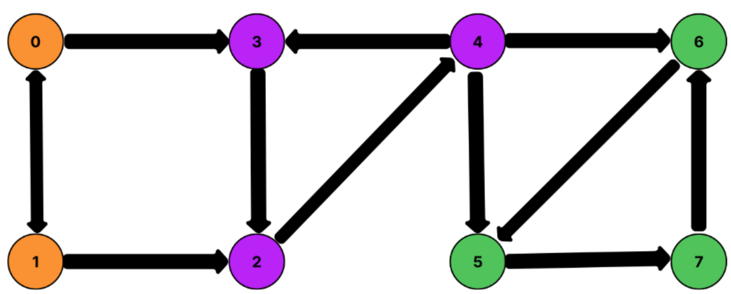


Figure 1: Three strongly connected components in an 8-node graph

The most common approach to this problem is implementing Tarjan's Serial SCC algorithm. It works by finding what vertices share the same low link value: the smallest vertex id reachable from it when performing DFS. Tarjan's algorithm works well on small graphs, but is slow on larger graphs due to the backtracking and stack maintenance steps. As well, its single threaded nature doesn't make it suitable for a distributed environment. For this reason, I used the divide-and-conquer strong components (DCSC) algorithm proposed by Sandia National Laboratories and Texas A&M University. DCSC does not rely on a stack to find SCC's but instead recursively partitions the graph in a way of isolating SCC's within a single subgraph. It works by picking a random pivot vertex and finding its set of predecessors and successor vertices. The intersection of the two sets is the SCC the pivot is located in. From there, the graph can be partitioned into three subgraphs: the set of predecessors not in the SCC, the set of successors not in the SCC, and the remainder vertices. All three subgraphs guarantee not to have overlapping vertices in an SCC, and therefore can be explored concurrently.

A variation of DCSC was used in both the MASS and Hazelcast implementations of SCC.

4.2 Weakly Connected Components (WCC)

Given a directed graph, the weakly connected components algorithm seeks to find all the maximal subgraphs in which there is an undirected path from any vertex to another. The premise to solving this problem is to convert the directed graph into an undirected graph and solve the Connected Components algorithm.

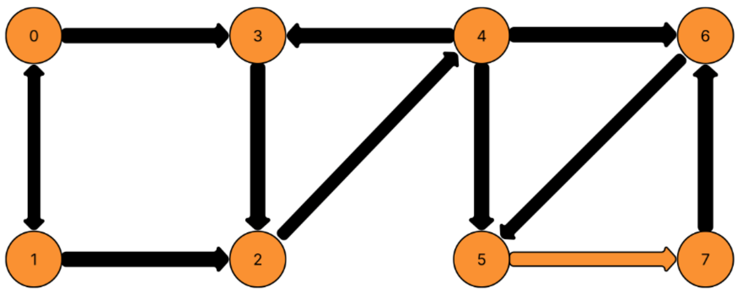


Figure 2: One weakly connected component in an 8-node graph

MASS and Hazelcast adopt different approaches to this problem. MASS uses a low-link concept to represent the smallest vertex id that a vertex can reach. After processing the graph, vertices with the same low-link value are in the same weakly connected component. Hazelcast on the other hand has each vertex examine its outgoing neighbors and makes use of an union-find data structure to combine sets with overlapping vertices.

4.3 Triangle Counting

The triangle counting algorithm seeks to find the number of 3-node cycles that exist in a graph.

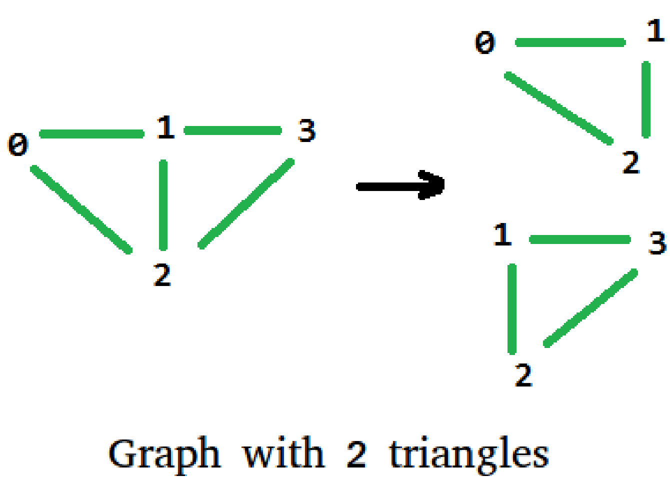


Image source: [geeksforgeeks]

MASS and Hazelcast have different approaches to this problem. MASS follows a three step phase in which agents traverse along their neighbors followed by their second-degree neighbors (neighbor of neighbor). In the last phase, it will attempt to return back to its original vertex. A successful completion of the three phases results in a triangle. Hazelcast on the other hand follows a more iterative approach. Each vertex will examine all of its neighbors and its second-degree neighbors. If there is an edge from the vertex to the second-degree neighbor, the triangle counter is incremented.

4.4 PageRank

PageRank was developed by Google co-founders Larry Page and Sergey Brin to rank web pages in search engine results. Given a directed graph, where nodes represent web pages and edges represent hyperlinks between them, the algorithm assigns a numerical score to each page representing its relative importance. Utilizing the random surfing model which simulates a user randomly following some path of hyperlinks across the internet, the algorithm estimates the likelihood that a random surfer lands on a given page.

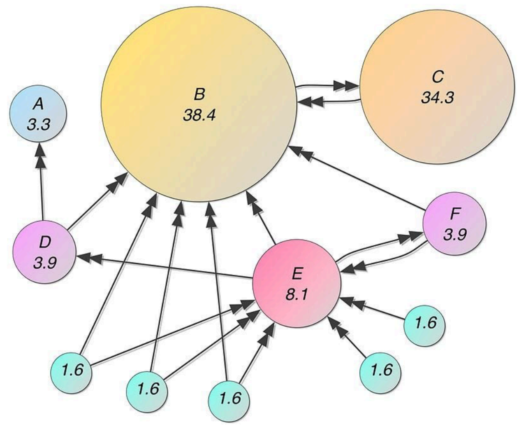


Image source: Medium

Pages linked to by other important pages receive higher scores because they are more likely to be visited in a random walk. Therefore, the probability score of a page is determined not just by the number of incoming links but also the quality of those links. The quality of a link is based on the rank of the linking page and the number of its outgoing links.

Both MASS and Hazelcast follow the same algorithm. Running for a fixed number of iterations, each iteration has two phases. The first phase consists of each vertex evenly distributing its rank to their outgoing neighbors. The second phase has each vertex compute its new rank based on the distributions it received from its incoming neighbors. MASS has four variations of a PageRank implementation worth exploring while Hazelcast has one implementation that somewhat follows the MapReduce programming model.

4.5 Label Propagation

Given an undirected graph, the Label Propagation algorithm finds communities in a graph by having vertices iteratively exchange and adopt labels, with an end result of grouping vertices with a shared label id. As neo4j describes the process: “the intuition behind the algorithm is that a single label can quickly become dominant in a densely connected group of nodes, but will have trouble crossing a sparsely connected region.” As labels propagate, communities form as nodes quickly agree upon a shared label.

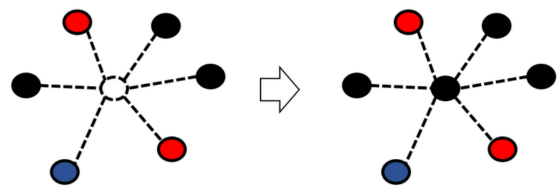


Image source: ResearchGate

The algorithm runs iteratively similar to PageRank, ending after a fixed number of iterations or until convergence where communities have been finalized. Each iteration likewise has two phases, where the first sees vertices sending their labels to neighbors while the second phase has them choosing the label with the highest frequency.

Also similar to PageRank, there are four MASS variations of Label Propagation while Hazelcast has one implementation that also operates akin to the MapReduce model.

5. Benchmark Implementations

5.1 Strongly Connected Components Implementations

To find an SCC of a pivot vertex using the DCSC algorithm, it requires finding the intersection between the set of predecessor vertices and successor vertices. The predecessors are the set of vertices that can reach the pivot while the successors are the set of vertices that the pivot can reach. The successors can be found doing a simple traversal along the outgoing edges starting at the pivot, but the predecessors require traversing along transposed/backward edges. These transposed edges need to be created before the process of finding the SCC's can begin.

5.1.1 MASS Implementation

The MASS implementation of SCC starts with spawning an agent at each vertex. Before the agents can start traversing the graph, transposed edges are created in the `preprocessGraph()` function. In this function, each agent spawns a child agent to migrate to the outgoing neighbors. Then each child agent will tell the neighbor vertex it's on to add a transposed edge by calling `addNeighbor()`, where the neighbor id is the `originalPlaceId` the child agent came from + the number of vertices in the graph, and the weight is -1. The child agents will then terminate and we are back to having one agent per vertex.

After transposed edges are added, a pivot vertex is chosen and sent to all agents. If the agent is not at the pivot vertex, it terminates. Otherwise, the agent will spawn a child agent (which is more like a sibling agent). So before starting the search for an SCC at the pivot vertex, the simulation only has two agents which are on the same vertex: one to traverse along the forward edges and the other to traverse along the transposed edges.

During the simulation, each agent performs a BFS traversal. On each call to `onArrival()`, the agent will pick the first unvisited neighbor as the `nextPlaceId` and spawn child agents at the remaining neighbors. To prevent sibling agents from going along paths another sibling/parent has already done, a guard rail is placed to check if a relative agent has already visited the place the current agent is on. If so, then the agent is done traversing. When an agent finishes its traversal (either through a failure to get past the guard rail or no more unvisited vertices to reach), it returns back to its `originalPlaceId` and reports its visited set as either predecessors or successors. After all agents have completed, the SCC can be computed, returned, and another iteration for an SCC can be found with a new pivot vertex. This process repeats until all vertices have been processed.

Figure 3: MASS SCC onArrival() function

```java
private Object onArrival() {

        nextPlaceId = -1;
        MyVertex place = (MyVertex) getPlace();
        int placeId = place.getIndex()[0];
        Object[] placeNeighbors = place.getNeighbors();
        Object[] neighborWeights = place.getWeights();

        if (doneTraversing && placeId == originalPlaceId) {
            place.onAgentReturn(visited, traversingBackwardEdges);
            kill();
            return null;
        }

        if (place.inComponent) {
            nextPlaceId = originalPlaceId;
            doneTraversing = true;
            return null;
        }

        if (place.visitedAgents.containsKey(originalPlaceId)) {
            int intVal = traversingBackwardEdges ? -1 : 1;
            if (place.visitedAgents.get(originalPlaceId).contains(intVal)) {
                nextPlaceId = originalPlaceId;
                doneTraversing = true;
                return null;
            }
            else {
                place.visitedAgents.get(originalPlaceId).add(intVal);
            }
        }
        else {
            int intVal = traversingBackwardEdges ? -1 : 1;
            IntSet newSet = new IntOpenHashSet();
            newSet.add(intVal);
```

```
        place.visitedAgents.put(originalPlaceId, newSet);
    }

    visited.add(placeId);
    ObjectList<ArgsToAgents> childAgentsToSpawn = new ObjectArrayList<>();
    for (int i = 0; i < placeNeighbors.length; i++) {
        int neighborId = (int) placeNeighbors[i];
        int weight = (int) neighborWeights[i];

        if ((traversingBackwardEdges && weight > 0) || (!traversingBackwardEdges && weight < 0)) {
            continue;
        }
        neighborId = weight < 0 ? neighborId - transposedOffset : neighborId;

        if (!visited.contains(neighborId)) {
            if (nextPlaceId == -1) {
                nextPlaceId = neighborId;
            }
            else {
                // (int transposedOffset, int originalPlaceId, int nextPlaceId
                //  boolean doneTraversing, boolean traversingBackwardEdges,
                //  IntSet visited)
                childAgentsToSpawn.add(new ArgsToAgents(
                    transposedOffset, originalPlaceId, neighborId,
                    doneTraversing, traversingBackwardEdges,
                    new IntOpenHashSet(visited)
                ));
            }
        }
    }

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

    if (nextPlaceId == -1) {
        doneTraversing = true;
        nextPlaceId = originalPlaceId;
    }

    return null;
}
```

5.1.2 Hazelcast Implementation

The Hazelcast implementation of SCC takes more liberty to split the graph into subgraphs when an SCC is found. Each vertex maintains a `tag` variable, which represents what subgraph the vertex is on.

Transposed edges are added with a custom aggregator class. In `accumulate()`, each entry key loops through its neighbors adding an entry to a local map, where the map key is the neighbor id and the value is a list that includes the entry key. The results of each local partition map then gets merged in `combine()`, forming a global map where the value is the list of incoming neighbors for each key. This map then gets sent to every vertex using an `EntryProcessor`, where the vertex will record the incoming neighbors value from its matching key. Finding SCC's can now be had.

The program starts by picking a pivot vertex and submitting an `IExecutorService` task using the `SCCTask` class to find the SCC of that pivot. The task calls `findSCC()` where the set of predecessors and successors are found concurrently by calling `traverseGraph()`. This function submits a separate `IExecutorService` task that uses the `GraphTraversalTask` class to traverse along either the forward or transposed edges according to the boolean value passed in. After both tasks have finished, the intersection can be found and added.

Now with three separate subgraphs, their `tag` variables are updated to reflect the subgraph they belong to. This is done using an `EntryProcessor`, submitting to each set of keys the appropriate `newTag` value to update to. Following this, three new pivot vertices are chosen and explored concurrently. This process repeats until all vertices have been processed.

There are two implementations of Hazelcast SCC. One uses the `IMap.get()` function to retrieve neighbors during graph traversals while the other uses `IMap.getAll()`.

Figure 4: Hazelcast SCC findSCC() function

```
private Map<String,I> findSCC(I vertexId, String tagType, int iteration) {
        try {
            if (vertexId == null) { return null; }

            // current subgraph
            String tag = tagType + iteration;
            iteration++;

            Future<Set<I>> predecessorsFuture = traverseGraph(vertexId, true); // backward edges
            Future<Set<I>> successorsFuture = traverseGraph(vertexId, false);     // forward edges

            Set<I> predecessors = predecessorsFuture.get();
            Set<I> successors = successorsFuture.get();

            // found an scc at the intersection.
            Set<I> scc = new HashSet<>(predecessors);
```

```
            scc.retainAll(successors);
            // Theres a chance the only intersection is the pivot vertex itself. Must be greater than 1.
            if (scc.size() > 1) { addToSCC(scc); }

            predecessors.removeAll(scc);
            successors.removeAll(scc);

            // discard all vertices that formed the scc
            updateTags(scc, "n");
            // update remaining vertices to separate subgraphs
            updateTags(predecessors, ("p" + iteration));
            updateTags(successors, ("s" + iteration));

            // find the vertices in the subgraph that were neither a successor or predecessor
            Set<I> remainderVertices = getKeysByTag(tag);
            updateTags(remainderVertices, ("r" + iteration));

            // get next pivots for each of the three subgraphs
            Map<String,I> nextPivots = new HashMap<>();
            if (!predecessors.isEmpty()) { nextPivots.put("p", getRandomKeyFromSet(predecessors)); }
            if (!successors.isEmpty()) { nextPivots.put("s", getRandomKeyFromSet(successors)); }
            if (!remainderVertices.isEmpty()) {
                nextPivots.put("r", getRandomKeyFromSet(remainderVertices));
            }

            return nextPivots.isEmpty() ? null : nextPivots;
        }
        catch (Exception e) {
            e.printStackTrace();
            return null;
        }
}
```

5.2 Weakly Connected Components Implementations

5.2.1 MASS Implementation

The MASS implementation of WCC starts by spawning an agent at each vertex to collect the set of vertices in the graph. From this set, a pivot vertex is chosen and sent to each agent. If the agent is not at the pivot vertex, it terminates, leaving only one agent before starting the simulation. On each call to `onArrival()`, each agent will check if its `originalPlaceId` is less than or equal to the place's `componentId` (low-link value). If it is, the place's `componentId` is updated to the agent's `originalPlaceId`, and the place's neighbors are examined to see if it needs to be visited. Looping through the list of neighbors, if the neighbor id is less than the agent's `originalPlaceId` and the agent hasn't visited the vertex yet, a child agent is spawned to go to that neighbor. Following this, the current agent just terminates.

After all agents have completed, the program checks the `componentId` of each vertex, grouping vertices with the same `componentId`. This process repeats with a new pivot vertex until all vertices have been processed.

Figure 5: MASS WCC onArrival() function

```
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<>();

            for (Object neighbor : placeNeighbors) {
                int candidate = (int) neighbor;
                if (candidate >= place.componentID && !visited.contains(candidate)) {
                    // (int originalPlaceId, int nextPlaceId, IntSet visited)
                    nextVertices.add(new ArgsToAgents(
                        originalPlaceId, candidate, new IntOpenHashSet(visited)
                    ));
                }
            }

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

        kill();

        return null;
}
```

5.2.2 Hazelcast Implementation

The Hazelcast implementation of WCC makes use of a disjoint set/union-find. A disjoint set is a data structure that stores a set of sets in which no two sets have overlapping elements. It works by assigning a parent to each element which originally, is itself. A call to unionize two elements (put them in the same set) means having them share the same parent. The `merge()` function combines the sets of an input disjoint set and a call to `getComponents()` combines elements with the same parent into a set and returns a list of sets.

The program starts by converting the directed graph into an undirected one and uses Hazelcast aggregation (`WCCAggregator`) to find the weakly connected components. Each partition will have a `DisjointSet` object and in the `accumulate()` function, each entry will loop through its set of neighbors, calling `union()` to give them the same parent. Then the `combine()` function is called where each partition's disjoint set is merged using the `merge()` function. The final step has the `DisjointSet` object now containing the parents for every vertex in the graph call `getComponents()`, which returns the finalized list of weakly connected components in the graph.

There are two implementations of Hazelcast WCC. One uses the `IMap.get()` function to retrieve neighbors during graph traversals while the other uses `IMap.getAll()`.

Figure 6: Hazelcast WCC WCCAggregator class

```
private static final class WCCAggregator<I extends Comparable<I>, V>
      implements Aggregator<Map.Entry<I, Vertex<I, V>>, List<Set<I>>>, HazelcastInstanceAware {

      private transient DistributedSharedGraph<I, V> dsg;
      private final String graphName;
      private DisjointSet<I> disjointSet = new DisjointSet<>();

      public WCCAggregator(final String graphName) {
         this.dsg = null;
         this.graphName = graphName;
      }

      @Override
      public void setHazelcastInstance(final HazelcastInstance instance) {
         this.dsg = new DistributedSharedGraph<>(instance, graphName);
      }

      @Override
      public void accumulate(final Map.Entry<I, Vertex<I, V>> entry) {
         final Vertex<I, V> vertex = entry.getValue();
         final I vertexId = vertex.getVertexId();
         final Set<I> neighbors = vertex.getNeighbors().keySet();

         Map<I, Vertex<I,V>> neighborsMap = this.dsg.graph.getAll(neighbors);
```

```
            disjointSet.makeSet(vertexId);
            for (Vertex<I,V> neighborVertex : neighborsMap.values()) {
                disjointSet.makeSet(neighborVertex.getVertexId());
                disjointSet.union(vertexId, neighborVertex.getVertexId());
            }
        }

        @Override
        public void combine(final Aggregator aggregator) {
            // combine results from each partition
            final WCCAggregator<I,V> other = getClass().cast(aggregator);
            this.disjointSet.merge(other.disjointSet);
        }

        @Override
        public List<Set<I>> aggregate() {
            List<Set<I>> components = disjointSet.getComponents();
            components.removeIf(component -> component.size() == 1);
            return components;
        }
}
```

5.3 Triangle Counting Implementations

5.3.1 MASS Implementation

This implementation of Triangle Counting starts by spawning an agent at each vertex. The simulation only has three steps. In the first step, each agent spawns child agents to migrate to the neighbors. To prevent duplicate triangles from being counted, an agent will only traverse/spawn child agents to neighbors with a lower id. The second step has the agents repeating step 1 (now being the second-degree neighbors the agents are migrating to). In the final step, each agent will try to return back to its original vertex. If it's not able to, then it terminates. The number of remaining alive agents is the total number of triangles that exist in the graph.

Triangle Counting uses the functions `propagateDown()` and `migrateSource()` for agent migration, which is internal to MASS. For more information, I recommend you read Vishnu Mohan, Anirduh Potturi, and Munehiro Fukuda's paper on its implementations: [Automated Agent Migration over Distributed Data Structures]

5.3.2 Hazelcast Implementation

The Hazelcast Triangle Counting implementation uses a custom `TriangleCountingAggregator` class to return an integer of the total number of triangles in the graph. In the `accumulate()` function, each entry loops through its neighbors and second-degree neighbors. If there is an edge from the vertex to the second-degree neighbor, the triangle counter is incremented. To prevent duplicate triangles from being reported, only triangles following this order are recorded: neighbor id > second-degree neighbor id > entry vertex id. In the `combine()` phase, the number of triangles at each partition is combined, and in the `aggregate()` phase is when the total number of triangles is returned.

Figure 7: Hazelcast Triangle Counting TriangleCountingAggregator class

```
private static final class TriangleCountingAggregator<I extends Comparable<I>, V>
        implements Aggregator<Map.Entry<I, Vertex<I, V>>, Integer>, HazelcastInstanceAware {

        private transient DistributedSharedGraph<I, V> dsg;
        private final String graphName;
        private Integer triangles;

        public TriangleCountingAggregator(final String graphName) {
            this.dsg = null;
            this.graphName = graphName;
            this.triangles = 0;
        }

        @Override
        public void setHazelcastInstance(final HazelcastInstance instance) {
            this.dsg = new DistributedSharedGraph<>(instance, graphName);
        }

        @Override
        public void accumulate(final Map.Entry<I, Vertex<I, V>> entry) {
            // each partition runs this for each entry and combines the results (trianglesCount)
            final Vertex<I, V> vertex = entry.getValue();
            final I vertexId = vertex.getVertexId();
            final Set<I> vertexNeighbors = vertex.getNeighbors().keySet();

            // id order: neighborId > sdNeighborId > vertexId
            for (I neighborId : vertexNeighbors) {
                if (neighborId.equals(vertexId)) { continue; }
                if (neighborId.compareTo(vertexId) < 0) { continue; }

                Vertex<I,V> neighborVertex = this.dsg.graph.get(neighborId);
                Set<I> sdNeighbors = neighborVertex.getNeighbors().keySet();

                // check if entry vertex has an edge to the second degree neighbors
                for (I sdNeighborId : sdNeighbors) {
                    if (sdNeighborId.equals(vertexId) || sdNeighborId.equals(neighborId)) { continue; }
                    if (sdNeighborId.compareTo(neighborId) > 0 || sdNeighborId.compareTo(vertexId) < 0) {
                        continue;
                    }

                    // only consider the triangle if the entry vertex has the highest id
                    // prevents duplicate triangles being reported
```

```
                if (vertexNeighbors.contains(sdNeighborId)) {
                    // System.out.println(
                    //     "Found triangle: [" + neighborId + ", " + sdNeighborId + ", " + vertexId + "]"
                    // );
                    this.triangles++;
                }
            }
        }
    }

    @Override
    public void combine(final Aggregator aggregator) {
        // combine results from each partition
        final TriangleCountingAggregator<I,V> other = getClass().cast(aggregator);
        this.triangles += other.triangles;
    }

    @Override
    public Integer aggregate() {
        return triangles;
    }
}
```

5.4 PageRank Implementations

5.4.1 MASS Implementation

There are four implementations of PageRank that were completed in MASS:

- V1 - AgentMigration: Spawns an agent and each vertex and spawns child agents to evenly distribute ranks to neighbors.
- V2 - ExchangeAll: Transposes the graph and uses messaging to send ranks between `VertexPlace`'s.
- V3 - PlacesBased: Each `VertexPlace` returns a map of its local distributions that ends up getting merged amongst each other before being sent out.
- V4 - PseudoAgent: Same implementation as V3 but agent based.

For more information, I recommend you read Robert Zimmerman's report by visiting the DSL website.

5.4.2 Hazelcast Implementation

The Hazelcast implementation runs similarly to versions 3 and 4 of MASS PageRank. Running for a fixed number of iterations, each iteration consists of a 2-step phase for distributing ranks and updating current rank labels. Phase 1 runs the `DistributeRanksAggregator` class to return a map, where the key is the vertex id and the value is a floating point representing the sum of distributions sent by incoming neighbors. In phase 2, the map is passed into an `UpdateRanksProcessor` object which has each entry apply this formula to recompute its new rank: 0.15 + (D * 0.85), where D is the collected distributions received.

Figure 8: Hazelcast PageRank computePageRanks() function

```
public List<MyPair<I,Double>> computePageRanks() {

    for (int i = 0; i < this.numIterations; i++) {
        Map<I, Double> globalDistributions = this.graph.aggregate(
            new DistributeRanksAggregator<>(this.graphName, this.numIterations)
        );

        EntryProcessor<I, Vertex<I,V>, Object> updateRanks = new UpdateRanksProcessor<>(
            globalDistributions
        );
        this.graph.executeOnEntries(updateRanks);
    }

    Map<I, MyPair<I,Double>> results = this.graph.executeOnEntries(
        new CollectRanksProcessor()
    );

    List<MyPair<I,Double>> ranks = new ArrayList<>();
    for (MyPair<I,Double> rank : results.values()) {
        ranks.add(rank);
    }

    ranks.sort(Comparator.<MyPair<I, Double>, Double>comparing(MyPair::getValue).reversed());
    return ranks;
}
```

5.5 Label Propagation Implementations

MASS and Hazelcast's implementations of Label Propagation are similar to their PageRank counterparts.

5.5.1 MASS Implementation

There are four implementations of Label Propagation that were completed in MASS:

- V1 - AgentMigration: Spawns an agent and each vertex and spawns child agents to send its label to neighbors.
- V2 - ExchangeAll: Transposes graph and sends labels between `VertexPlace`'s directly.
- V3 - PlacesBased: Each `VertexPlace` returns a map of the label it's sending to its neighbors, which ends up getting merged amongst each other before being sent out.
- V4 - PseudoAgent: Same implementation as V3 but agent based.

Again, more information can be found on Robert Zimmerman's report.

5.5.2 Hazelcast Implementation

The Hazelcast implementation runs similarly to versions 3 and 4 of MASS Label Propagation. Running for a fixed number of iterations, each iteration consists of a 2-step phase for sending labels and updating the current label. Phase 1 runs the `PropagateLabelsAggregator` class to return a map, where the key is the vertex id and the value is a map of label id to the number of neighbors who sent it. In phase 2, the map is passed into an `UpdateLabelsProcessor` object which has each entry choose the label with the highest frequency for its new label. Tie breakers are broken by choosing the smallest label id.

Figure 9: Hazelcast Label Propagation executeLabelPropagation() function

```
public Map<I, List<I>> executeLabelPropagation() {

    for (int i = 0; i < this.numIterations; i++) {
        Map<I, Map<I,Integer>> globalLabelPropagations = this.graph.aggregate(
            new PropagateLabelsAggregator<>(this.graphName, this.numIterations)
        );

        EntryProcessor<I, Vertex<I,V>, Object> updateLabels = new UpdateLabelsProcessor<>(
            globalLabelPropagations
        );
        this.graph.executeOnEntries(updateLabels);
    }

    Map<I, I> results = this.graph.executeOnEntries(
        new CollectLabelsProcessor()
    );

    Map<I, List<I>> labels = new HashMap<>();
    for (Map.Entry<I,I> entry : results.entrySet()) {
        I label = entry.getValue();
        I vertexId = entry.getKey();

        labels.computeIfAbsent(label, l -> new ArrayList<>()).add(vertexId);
    }

    return labels;
}
```

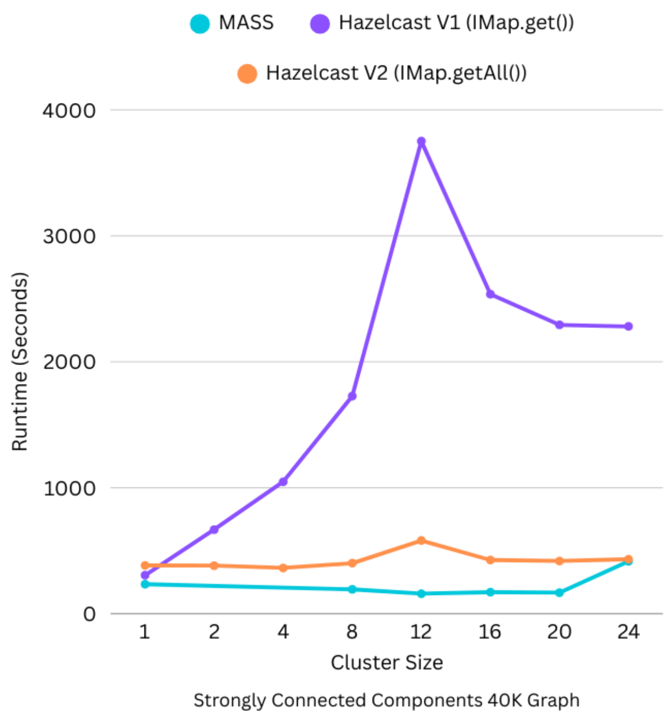
6. Benchmark results

Benchmark results were completed using the UWB hermes1-24 machines. Each benchmark was evaluated using six different graphs (1k, 3k, 5k, 10k, 20k, and 40k vertices) with eight different cluster sizes (1, 2, 4, 8, 12, 16, 20, 24). Each combination of graph and cluster size was ran three times and the average of the three runs were recorded. See Appendixes A-E to view the full execution results and Appendixes F-J to view the line chart comparisons.

6.1 Strongly Connected Components Results

MASS's implementation of SCC generally outperformed Hazelcast's. As viewed on figure 10, all three implementations are comparable on one machine, but as the cluster size increased, Hazelcast V1's runtime spiked while V2 and MASS's stayed somewhat steady.

Figure 10: Strongly Connected Components on 40k graph



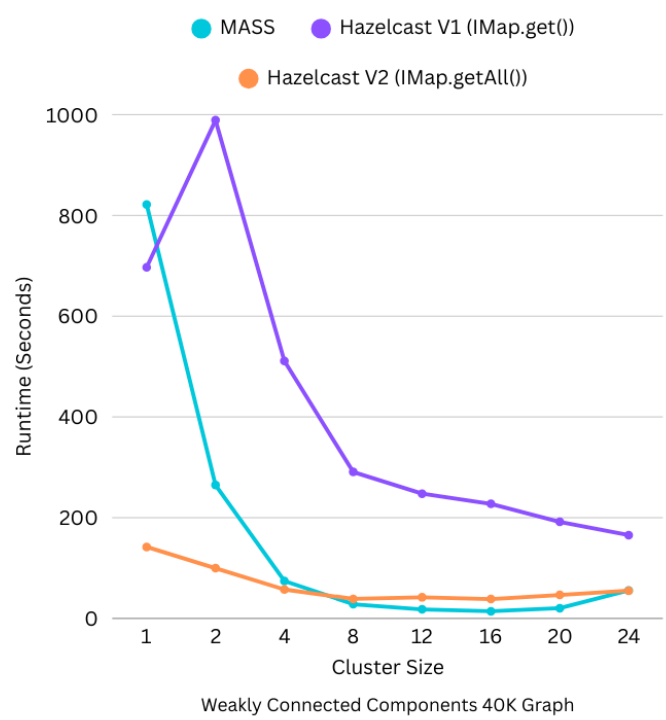
The reason for these differences I believe lie in how the graphs are traversed. Strongly Connected Components require traversing as many edges that can be reached. MASS utilizes an agent-based parallel-BFS, eliminating the need to backtrack to unvisited paths. And because agents operate within their own processes, vertices can be processed concurrently. Increasing the cluster size increases the number of processes available for agents, ultimately minimizing context switching to provide fast traversals. This may explain why MASS isn't affected much by communication overhead until you get to 24 machines. In Hazelcast's case, even though an asynchronous task is being submitted to handle the graph traversal, it is still a single threaded operation. V1 uses IMap's `get()` when traversing the graph, which makes one network call per neighbor. This is why when increasing the cluster size, the number of calls over the network increases, to which Hazelcast V1 gets heavily penalized. Conversely, Hazelcast V2 uses `getAll()` when traversing the graph, which groups a vertex's set of neighbors by the partition that owns it, making a network call per partition. This eliminates repeated network calls to the same machines, which results in a faster execution time. However, V2 still did not perform better than MASS.

Another reason for MASS's good performance also lies in how many agents are being spawned. My earlier point may have alluded to an optimization of spawning an agent at each vertex to speed up SCC detection, but doing so would degrade performance due to such a high number of agents existing at one time. One thing to note about MASS's SCC implementation is that on the 40k graph, it fails when running on 2 and 4 machines. This may be due to there not being enough memory to accommodate the MASS agent workload + communication overhead. As opposed to running it on one machine, there is enough memory to run MASS on the 40k graph since there is no communication overhead. And on 8+ machines, there is now enough memory to also accommodate the storage needed for communication overhead.

6.2 Weakly Connected Components Results

MASS's implementation of WCC generally outperformed Hazelcast on most combinations of graph size to cluster size. For larger graphs (10k+) on smaller cluster sizes, MASS would gradually perform worse than Hazelcast. As the cluster size increased, MASS would be comparable if not better, as can be viewed on figure 11.

Figure 11: Weakly Connected Components on 40k graph



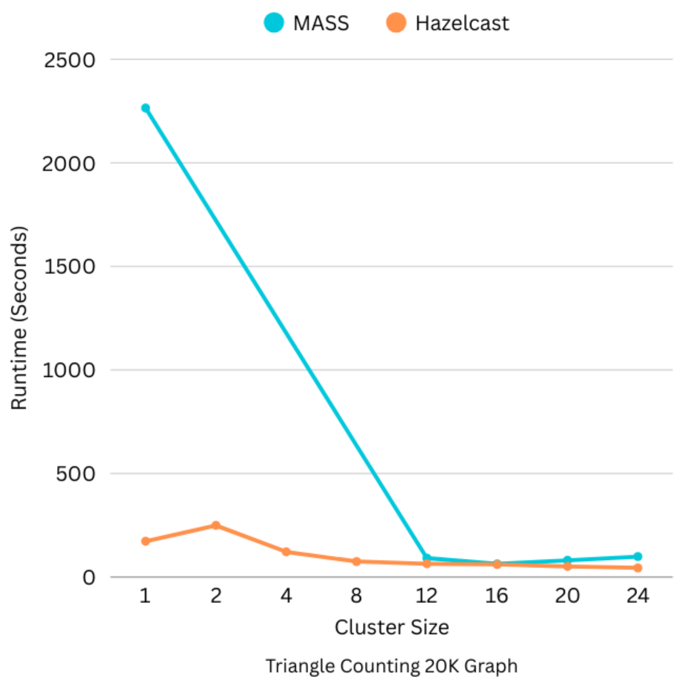
Similar to my conclusion for MASS SCC, MASS WCC utilizing parallel-BFS traversal explains why performance gets better (up until 24 machines). In Hazelcast's case, it uses aggregators to first run at each partition before combining the results. When the cluster size increases, the number of partitions each machine manages decreases, allowing for faster processing of the partitions it does own. And because Hazelcast Aggregation is read-only and runs on the data itself, there is minimal communication overhead.

As well, Hazelcast V2 using `getAll()` provides better performance as it does batch-retrieval of neighbors.

6.3 Triangle Counting Results

Hazelcast Triangle Counting generally outperformed MASS. As viewed on figure 11, they closely matched on larger cluster sizes but Hazelcast had the edge on smaller cluster sizes.

Figure 12: Triangle Counting on 20k graph

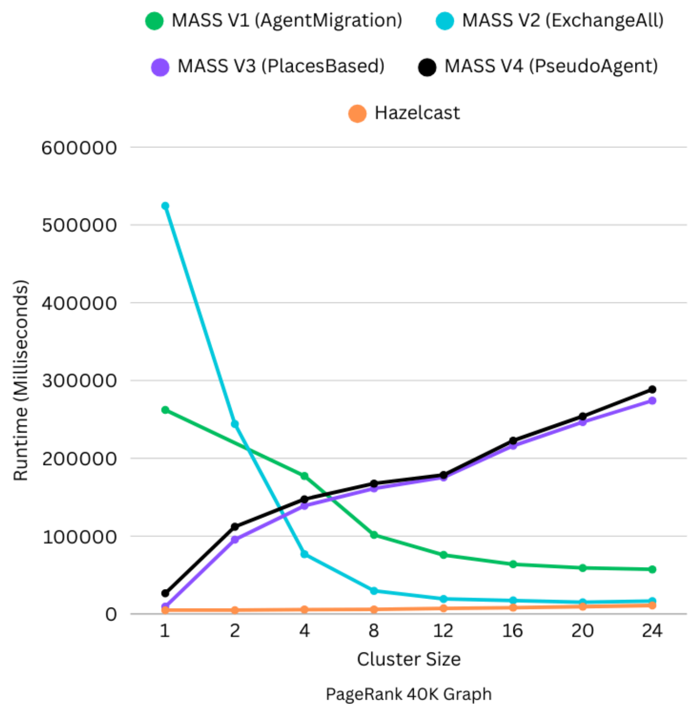


Similar to Hazelcast WCC, Hazelcast Triangle Counting uses aggregators, so increasing the cluster size resulted in better performance. MASS however faced memory issues in its Triangle Counting benchmark. Because so many agents are being created (36 million on the 20k graph), there is too much overhead for the smaller cluster sizes to manage, even crashing on the 2, 4, and 8 cluster sizes, with it also unable to run on the 40k graph. However, as the cluster size increases to 12+ machines, MASS performs as well if not better than Hazelcast.

6.4 PageRank Results

Hazelcast's implementation of PageRank generally outperformed MASS. Although MASS V2 performs the worst on one machine, it becomes the version most competitive with Hazelcast. As the trend we've been seeing, increasing the cluster size closes the gap significantly.

Figure 13: PageRank on 40k graph

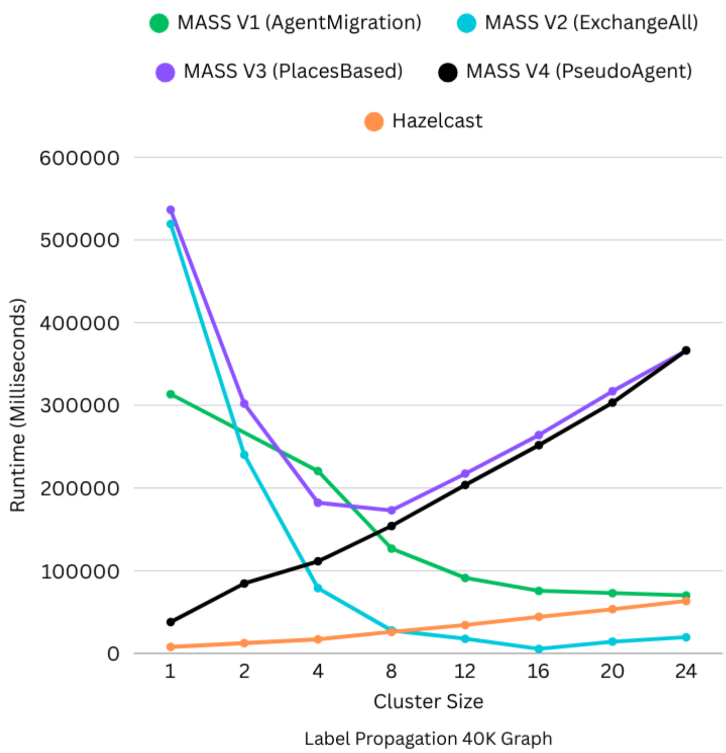


Hazelcast's ability to run computation on the machine where data resides allowed it to perform as well as it did. `DistributeRanksAggregator` does not make any calls over the network when calculating the ranks to distribute, resulting in low network overhead.

6.5 Label Propagation Results

Similar to the PageRank results, Hazelcast for the most part outperforms MASS, with MASS V2 being the most competitive version. Hazelcast's data locality operations using `PropagateLabelsAggregator` and `UpdateLabelsProcessor` allowed it to achieve faster execution results due to the label propagation logic of the computation not making network calls. However, increasing the cluster size led to MASS V2 outperforming Hazelcast on larger graphs.

Figure 14: Label Propagation on 40k graph



7. Programmability

Tables 1-5 show the programmability metrics captured between the MASS and Hazelcast benchmarks. The number of files, number of methods, LOC, lines of logic, and cyclomatic complexity were found using Lizard, LCOM4 was found using ck, and boilerplate % was found using this formula found online: (1 - (lines of logic / LOC)) * 100

Table 1: MASS vs Hazelcast Strongly Connected Components Programmability Metrics

Measurement (MASS)	Value
Number of files	4
Number of methods	23
Total Lines of Code (LOC)	626
Lines of Logic	498
Boilerplate %	20%

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	55
Total Lines of Code (LOC)	1167
Lines of Logic	584
Boilerplate %	49

Cyclomatic Complexity	3.9
Lack of Cohesion in Methods (LCOM4)	0.51

Cyclomatic Complexity	2.3
Lack of Cohesion in Methods (LCOM4)	0.37

Table 2: MASS vs Hazelcast Weakly Connected Components Programmability Metrics

Measurement (MASS)	Value
Number of files	4
Number of methods	20
Total Lines of Code (LOC)	634
Lines of Logic	381
Boilerplate %	39
Cyclomatic Complexity	3.5
Lack of Cohesion in Methods (LCOM4)	0.52

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	45
Total Lines of Code (LOC)	934
Lines of Logic	468
Boilerplate %	49
Cyclomatic Complexity	2.4
Lack of Cohesion in Methods (LCOM4)	0.39

Table 3: MASS vs Hazelcast Triangle Counting Programmability Metrics

Measurement (MASS)	Value
Number of files	3
Number of methods	12
Total Lines of Code (LOC)	653
Lines of Logic	211
Boilerplate %	67
Cyclomatic Complexity	2.7
Lack of Cohesion in Methods (LCOM4)	0.65

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	30
Total Lines of Code (LOC)	708
Lines of Logic	327
Boilerplate %	53
Cyclomatic Complexity	2.5
Lack of Cohesion in Methods (LCOM4)	0.49

Table 4: MASS vs Hazelcast PageRank Programmability Metrics

Refer to Robert Zimman's report for the MASS PageRank Metrics

Measurement (Hazelcast)	Value
Number of files	4
Number of methods	42
Total Lines of Code (LOC)	856
Lines of Logic	435
Boilerplate %	49
Cyclomatic Complexity	2.4
Lack of Cohesion in Methods (LCOM4)	0.37

Table 5: MASS vs Hazelcast Label Propagation Programmability Metrics

Refer to Robert Zimman's report for the MASS Label Propagation Metrics

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	37
Total Lines of Code (LOC)	860
Lines of Logic	433
Boilerplate %	49
Cyclomatic Complexity	2.4
Lack of Cohesion in Methods (LCOM4)	0.38

8. Conclusion

In this capstone, I implemented and benchmarked the Strongly Connected Components, Weakly Connected Components, Triangle Counting, PageRank, and Label Propagation algorithms. From these benchmarks, it can be concluded that MASS's use of agents to perform tasks in parallel provides significant speeds to servicing user requests, especially when traversing a graph. However, data locality sensitive operations present a weakness to MASS in comparison to Hazelcast. As well, due to large agent overhead, MASS is at a slight disadvantage when there is limited memory from smaller cluster sizes attempting to accommodate larger graphs. Future work to reduce agent memory overhead can be achieved, as well as re-testing some benchmarks on different partitioning strategies that considers graph shapes.

Appendix A: Strongly Connected Components Execution Results

MASS SCC Results

num-members	num-vertices	total-agents-generated	load time(ms)	runtime (sec)
1	1000	367926	164	6.391
2	1000	367926	126	7.882
4	1000	367926	161	6.370
8	1000	367926	305	6.529
12	1000	367926	342	14.679
16	1000	367926	603	8.429
20	1000	367926	1378	9.814
24	1000	367926	2372	31.079
1	3000	1157222	274	18.410
2	3000	1157222	239	20.679
4	3000	1157222	235	18.155
8	3000	1157222	386	16.125
12	3000	1157222	441	34.037
16	3000	1157222	664	16.111
20	3000	1157222	1516	20.650
24	3000	1157222	2767	62.471
1	5000	1941566	352	32.711
2	5000	1941566	326	30.870
4	5000	1941566	309	27.232
8	5000	1941566	439	22.565
12	5000	1941566	477	52.344
16	5000	1941566	744	22.971
20	5000	1941566	1702	33.540
24	5000	1941566	2651	77.886
1	10000	3899966	616	61.159
2	10000	3899966	423	60.681
4	10000	3899966	380	47.760

8	10000	3899966	544	41.703
12	10000	3899966	617	76.454
16	10000	3899966	894	41.478
20	10000	3899966	1970	52.710
24	10000	3899966	3102	133.243
1	20000	7825526	1104	122.660
2	20000	7825526	713	118.970
4	20000	7825526	593	98.179
8	20000	7825526	780	76.721
12	20000	7825526	784	119.219
16	20000	7825526	1046	71.258
20	20000	7825526	2142	96.708
24	20000	7825526	3510	217.066
1	40000	15737702	1995	234.301
2	40000	NA	NA	NA
4	40000	NA	NA	NA
8	40000	15737702	1338	192.977
12	40000	15737702	968	159.211
16	40000	15737702	1512	170.845
20	40000	15737702	2695	167.496
24	40000	15737702	4200	416.598

Hazelcast SCC Using IMap.getAll() Results

num-members	num-vertices	num-edges	load time(sec)	runtime (sec)
1	1000	93480	0.467	10.154
2	1000	93480	0.719	10.653
4	1000	93480	0.579	10.176
8	1000	93480	0.526	11.086
12	1000	93480	0.578	10.947
16	1000	93480	0.560	11.743
20	1000	93480	0.604	11.405
24	1000	93480	0.805	12.497
1	3000	293804	0.779	29.862
2	3000	293804	0.838	29.536
4	3000	293804	0.712	28.476

8	3000	293804	0.778	30.408
12	3000	293804	0.863	35.241
16	3000	293804	0.690	34.734
20	3000	293804	0.874	33.343
24	3000	293804	0.986	33.140
1	5000	492890	0.945	47.832
2	5000	492890	1.021	49.993
4	5000	492890	0.894	48.134
8	5000	492890	0.766	50.834
12	5000	492890	0.905	73.129
16	5000	492890	0.815	57.140
20	5000	492890	1.067	52.221
24	5000	492890	1.279	54.766
1	10000	989990	1.308	94.184
2	10000	989990	1.284	98.449
4	10000	989990	1.157	95.092
8	10000	989990	1.049	102.434
12	10000	989990	1.214	115.449
16	10000	989990	1.208	105.711
20	10000	989990	1.408	100.695
24	10000	989990	1.491	101.606
1	20000	1986380	2.053	194.788
2	20000	1986380	1.916	187.717
4	20000	1986380	1.606	188.413
8	20000	1986380	1.393	197.246
12	20000	1986380	1.821	288.561
16	20000	1986380	1.610	225.209
20	20000	1986380	1.871	225.902
24	20000	1986380	1.850	202.941
1	40000	3994424	2.949	382.949
2	40000	3994424	2.790	381.891
4	40000	3994424	2.473	363.753
8	40000	3994424	2.151	400.683
12	40000	3994424	2.472	580.422
16	40000	3994424	2.416	426.136
20	40000	3994424	2.787	418.809
24	40000	3994424	2.913	432.377

Appendix B: Weakly Connected Components Execution Results

MASS WCC Results

num-members	num-vertices	total-agents-generated	load time(ms)	runtime (sec)
1	1000	92481	165	1.725
2	1000	92481	152	1.799
4	1000	92481	161	1.556
8	1000	92481	315	1.776
12	1000	92481	332	2.018
16	1000	92481	575	2.419
20	1000	92481	1242	4.602
24	1000	92481	2345	20.897
1	3000	290805	298	7.464
2	3000	290805	216	4.796
4	3000	290805	274	3.219
8	3000	290805	366	2.853
12	3000	290805	440	2.998
16	3000	290805	732	3.325
20	3000	290805	1606	5.957
24	3000	290805	2402	22.528
1	5000	487891	366	17.349
2	5000	487891	286	8.469
4	5000	487891	313	4.852
8	5000	487891	485	3.812
12	5000	487891	568	3.896
16	5000	487891	760	4.155
20	5000	487891	1542	6.826
24	5000	487891	2619	23.319
1	10000	979991	611	57.912
2	10000	979991	431	22.870
4	10000	979991	383	9.441
8	10000	979991	551	6.271
12	10000	979991	607	5.214
16	10000	979991	898	5.549
20	10000	979991	1792	9.511
24	10000	979991	3168	30.984

1	20000	1966381	1088	206.237
2	20000	1966381	743	68.935
4	20000	1966381	544	24.438
8	20000	1966381	840	11.588
12	20000	1966381	648	8.259
16	20000	1966381	1177	8.258
20	20000	1966381	2305	13.467
24	20000	1966381	3642	42.204
1	40000	3954425	2192	821.771
2	40000	3954425	1382	264.688
4	40000	3954425	919	74.470
8	40000	3954425	1323	28.296
12	40000	3954425	873	17.974
16	40000	3954425	1475	14.164
20	40000	3954425	2655	20.301
24	40000	3954425	4415	55.658

Hazelcast WCC Using IMap.getAll() Results

num-members	num-vertices	num-edges	load time(sec)	runtime (sec)
1	1000	93480	0.491	4.474
2	1000	93480	0.628	3.768
4	1000	93480	0.556	2.790
8	1000	93480	0.522	2.627
12	1000	93480	0.602	2.347
16	1000	93480	0.626	2.375
20	1000	93480	0.692	2.834
24	1000	93480	0.729	3.567
1	3000	293804	0.757	12.404
2	3000	293804	0.718	9.379
4	3000	293804	0.850	6.084
8	3000	293804	0.667	5.437
12	3000	293804	0.781	5.611
16	3000	293804	0.711	4.563
20	3000	293804	0.854	6.592
24	3000	293804	1.052	7.582
1	5000	492890	0.945	18.924

2	5000	492890	0.923	14.012
4	5000	492890	0.834	9.864
8	5000	492890	0.772	7.202
12	5000	492890	0.913	8.029
16	5000	492890	0.858	7.232
20	5000	492890	1.052	8.909
24	5000	492890	1.047	11.129
1	10000	989990	1.201	36.198
2	10000	989990	1.188	26.520
4	10000	989990	1.180	16.922
8	10000	989990	1.073	12.635
12	10000	989990	1.196	13.206
16	10000	989990	1.135	11.178
20	10000	989990	1.225	14.811
24	10000	989990	1.449	17.760
1	20000	1986380	1.715	70.348
2	20000	1986380	1.793	49.044
4	20000	1986380	1.589	31.446
8	20000	1986380	1.503	21.302
12	20000	1986380	1.621	24.466
16	20000	1986380	1.642	20.389
20	20000	1986380	1.876	25.624
24	20000	1986380	1.972	31.843
1	40000	3994424	2.860	141.879
2	40000	3994424	2.514	99.773
4	40000	3994424	2.405	57.478
8	40000	3994424	2.099	38.687
12	40000	3994424	2.452	41.924
16	40000	3994424	2.357	38.488
20	40000	3994424	2.612	46.545
24	40000	3994424	2.775	55.098

Appendix C: Triangle Counting Execution Results

MASS Triangle Counting Results

num-members	num-vertices	total-agents-generated	load time(ms)	runtime (sec)

1	1000	1778723	139	17.048
2	1000	1778723	163	12.535
4	1000	1778723	163	7.069
8	1000	1778723	298	5.694
12	1000	1778723	329	5.330
16	1000	1778723	589	6.018
20	1000	1778723	1290	9.342
23	1000	1778723	2243	12.681
1	3000	5508612	248	80.441
2	3000	5508612	223	48.683
4	3000	5508612	235	22.791
8	3000	5508612	394	14.141
12	3000	5508612	381	10.376
16	3000	5508612	661	11.559
20	3000	5508612	1520	17.197
23	3000	5508612	2367	24.562
1	5000	9192458	333	199.839
2	5000	9192458	283	117.020
4	5000	9192458	303	47.903
8	5000	9192458	448	23.150
12	5000	9192458	461	16.557
16	5000	9192458	770	15.382
20	5000	9192458	1568	24.372
23	5000	9192458	2760	30.521
1	10000	18357946	549	619.898
2	10000	NA	NA	NA
4	10000	NA	NA	NA
8	10000	18357946	494	55.166
12	10000	18357946	559	33.888
16	10000	18357946	858	28.802
20	10000	18357946	1855	40.723
23	10000	18357946	3247	51.215
1	20000	36665733	1150	2264.793
2	20000	NA	NA	NA
4	20000	NA	NA	NA
8	20000	NA	NA	NA
12	20000	36665733	686	91.412

16	20000	36665733	939	63.706
20	20000	36665733	2147	80.698
23	20000	36665733	3561	98.790

Hazelcast Triangle Counting Results

num-members	num-vertices	num-edges	load time(sec)	runtime (sec)
1	1000	93480	1.020	10.095
2	1000	93480	1.034	14.715
4	1000	93480	0.893	7.520
8	1000	93480	0.726	5.319
12	1000	93480	0.984	5.477
16	1000	93480	1.150	6.008
20	1000	93480	1.052	4.995
24	1000	93480	0.842	4.379
1	3000	293804	1.574	28.335
2	3000	293804	1.640	41.173
4	3000	293804	1.326	20.507
8	3000	293804	1.029	12.984
12	3000	293804	1.519	12.381
16	3000	293804	1.468	13.154
20	3000	293804	1.410	11.130
24	3000	293804	1.460	10.093
1	5000	492890	2.117	46.101
2	5000	492890	2.075	64.993
4	5000	492890	1.521	31.553
8	5000	492890	1.360	20.043
12	5000	492890	1.639	18.407
16	5000	492890	1.952	18.692
20	5000	492890	1.612	16.472
24	5000	492890	1.769	14.277
1	10000	989990	2.596	84.124
2	10000	989990	2.592	128.125
4	10000	989990	2.057	63.171
8	10000	989990	1.882	38.063
12	10000	989990	2.196	33.831
16	10000	989990	2.399	33.462

20	10000	989990	2.120	28.141
24	10000	989990	2.168	24.337
1	20000	1986380	4.202	173.062
2	20000	1986380	3.817	249.160
4	20000	1986380	3.032	121.809
8	20000	1986380	2.723	75.091
12	20000	1986380	2.858	64.370
16	20000	1986380	3.306	61.314
20	20000	1986380	2.780	51.202
24	20000	1986380	2.456	44.558
1	40000	3994424	6.440	341.477
2	40000	3994424	6.222	492.082
4	40000	3994424	4.912	241.605
8	40000	3994424	4.149	146.076
12	40000	3994424	4.232	126.036
16	40000	3994424	4.910	116.628
20	40000	3994424	4.095	97.466
24	40000	3994424	3.668	86.392

Appendix D: PageRank Execution Results

MASS PageRank V2 Results

num-members	num-vertices	load time (ms)	runtime (ms)
1	1000	157	2313
2	1000	146	1840
4	1000	165	1487
8	1000	303	1500
12	1000	371	1576
16	1000	512	2192
20	1000	656	2306
24	1000	959	3308
1	3000	309	7260
2	3000	274	5617
4	3000	1964	3242
8	3000	427	2603
12	3000	450	2475

16	3000	608	3172
20	3000	759	3446
24	3000	1069	4254
1	5000	391	14406
2	5000	338	9308
4	5000	402	5650
8	5000	481	3702
12	5000	2198	3003
16	5000	702	3979
20	5000	2526	3649
24	5000	1168	5178
1	10000	585	33573
2	10000	475	22491
4	10000	490	10919
8	10000	649	6723
12	10000	651	5047
16	10000	866	5355
20	10000	990	5269
24	10000	1326	6788
1	20000	1033	140450
2	20000	722	77113
4	20000	600	25358
8	20000	756	12501
12	20000	784	10176
16	20000	900	9513
20	20000	1131	8664
24	20000	1465	9859
1	40000	1800	524454
2	40000	1251	244161
4	40000	897	76922
8	40000	897	29697
12	40000	932	19235
16	40000	1138	17268
20	40000	1249	14936
24	40000	1726	16509

Hazelcast PageRank Results

num-members	num-vertices	num-edges	load time(ms)	runtime (ms)
1	1000	93480	444	350
2	1000	93480	664	429
4	1000	93480	528	480
8	1000	93480	583	518
12	1000	93480	614	591
16	1000	93480	513	710
20	1000	93480	689	771
24	1000	93480	693	830
1	3000	293804	824	694
2	3000	293804	818	872
4	3000	293804	703	849
8	3000	293804	659	932
12	3000	293804	714	1077
16	3000	293804	741	1103
20	3000	293804	944	1466
24	3000	293804	1168	1378
1	5000	492890	974	954
2	5000	492890	1058	1152
4	5000	492890	873	1186
8	5000	492890	834	1228
12	5000	492890	890	1392
16	5000	492890	911	1305
20	5000	492890	1093	1502
24	5000	492890	1257	1814
1	10000	989990	1391	1550
2	10000	989990	1300	1860
4	10000	989990	1082	2006
8	10000	989990	1049	2190
12	10000	989990	1206	2379
16	10000	989990	1144	2395
20	10000	989990	1455	2709
24	10000	989990	1808	2977
1	20000	1986380	1799	2519
2	20000	1986380	1727	2701

4	20000	1986380	1544	3346
8	20000	1986380	1496	3628
12	20000	1986380	1706	4211
16	20000	1986380	1614	4400
20	20000	1986380	2066	5445
24	20000	1986380	2352	5544
1	40000	3994424	2861	4890
2	40000	3994424	2767	4903
4	40000	3994424	2417	5593
8	40000	3994424	2094	5759
12	40000	3994424	2416	7082
16	40000	3994424	2401	8091
20	40000	3994424	2825	9400
24	40000	3994424	3077	10917

Appendix E: Label Propagation Execution Results

MASS Label Propagation V2 Results

num_members	num_vertices	load_time_ms	runtime_ms
1	1000	177	2139
2	1000	149	1759
4	1000	184	1458
8	1000	321	1450
12	1000	338	1544
16	1000	538	2136
20	1000	691	2300
24	1000	1004	3473
1	3000	323	7030
2	3000	261	5162
4	3000	336	3082
8	3000	393	2477
12	3000	467	2348
16	3000	596	3012
20	3000	736	3209
24	3000	1154	5226
1	5000	383	13757

2	5000	359	8984
4	5000	410	5163
8	5000	530	3366
12	5000	511	3029
16	5000	747	4116
20	5000	871	3793
24	5000	1260	6714
1	10000	587	42252
2	10000	454	21216
4	10000	482	10028
8	10000	652	6426
12	10000	681	4852
16	10000	4067	4994
20	10000	948	4488
24	10000	1458	7233
1	20000	1026	122944
2	20000	710	66513
4	20000	591	24602
8	20000	799	11845
12	20000	820	9165
16	20000	893	9571
20	20000	1078	8281
24	20000	1686	12366
1	40000	1795	519370
2	40000	1244	240209
4	40000	924	78938
8	40000	913	27742
12	40000	2558	17724
16	40000	1151	5449
20	40000	1321	14165
24	40000	1923	19542

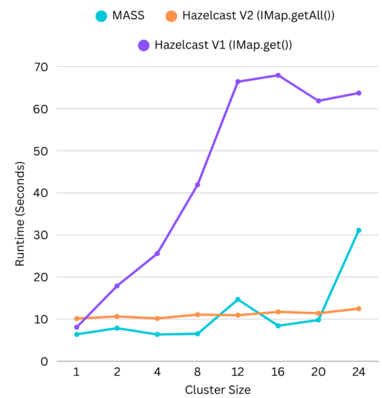
Hazelcast Label Propagation Results

num-members	num-vertices	num-edges	load time(ms)	runtime (ms)
1	1000	93480	513	383
2	1000	93480	481	635

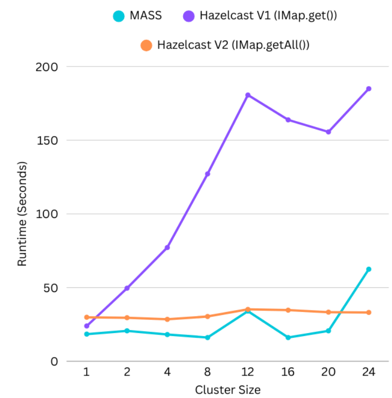
4	1000	93480	546	839
8	1000	93480	572	926
12	1000	93480	555	1063
16	1000	93480	554	1123
20	1000	93480	716	1442
24	1000	93480	854	1415
1	3000	293804	792	777
2	3000	293804	819	1340
4	3000	293804	722	1661
8	3000	293804	702	1856
12	3000	293804	804	2438
16	3000	293804	684	2724
20	3000	293804	922	3163
24	3000	293804	1156	3598
1	5000	492890	879	1167
2	5000	492890	880	1908
4	5000	492890	770	2522
8	5000	492890	735	2822
12	5000	492890	820	3664
16	5000	492890	730	4576
20	5000	492890	1110	5299
24	5000	492890	1139	6144
1	10000	989990	1128	1913
2	10000	989990	1191	3473
4	10000	989990	992	4230
8	10000	989990	1040	5657
12	10000	989990	1086	7231
16	10000	989990	1061	8964
20	10000	989990	1283	10690
24	10000	989990	1614	12747
1	20000	1986380	1817	3659
2	20000	1986380	1867	6048
4	20000	1986380	1556	8569
8	20000	1986380	1382	11754
12	20000	1986380	1569	15832
16	20000	1986380	1537	19273
20	20000	1986380	1983	24715

24	20000	1986380	2143	27660
1	40000	3994424	2796	7802
2	40000	3994424	2536	12460
4	40000	3994424	2503	17046
8	40000	3994424	2121	26040
12	40000	3994424	2493	34250
16	40000	3994424	2431	44211
20	40000	3994424	2905	53387
24	40000	3994424	3221	63392

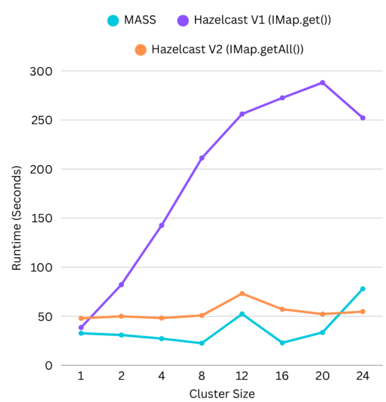
Appendix F: Full Strongly Connected Components Execution Diagrams



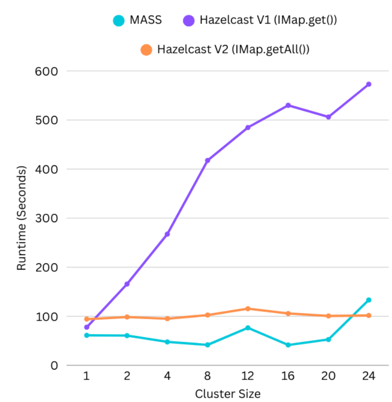
Strongly Connected Components 1K Graph



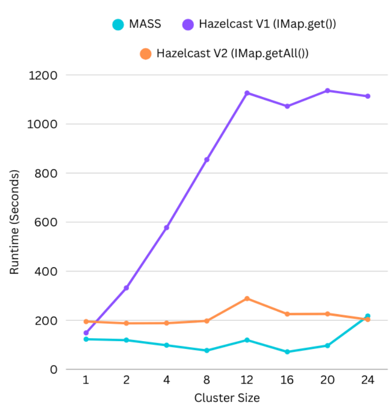
Strongly Connected Components 3K Graph



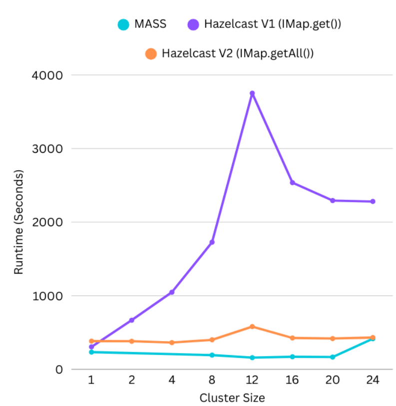
Strongly Connected Components 5K Graph



Strongly Connected Components 10K Graph

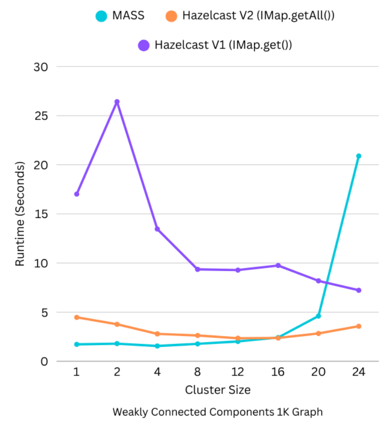


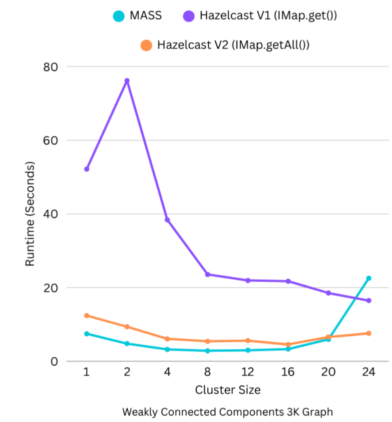
Strongly Connected Components 20K Graph

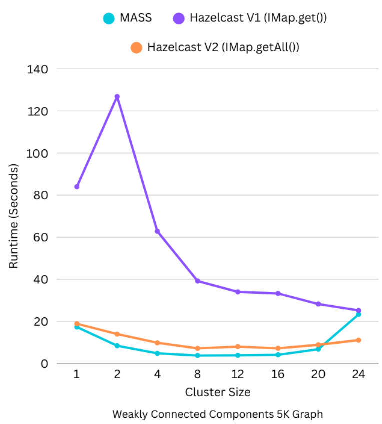


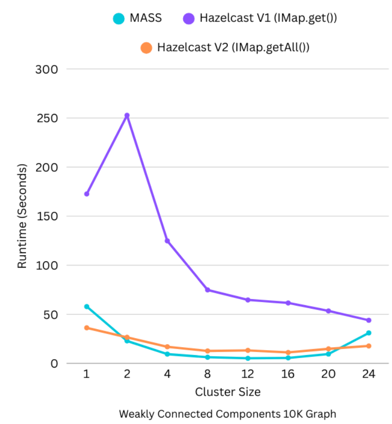
Strongly Connected Components 40K Graph

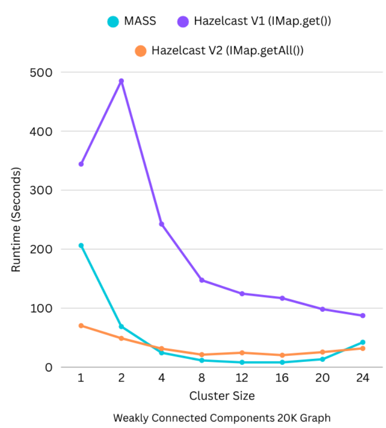
Appendix G: Full Weakly Connected Components Execution Diagrams

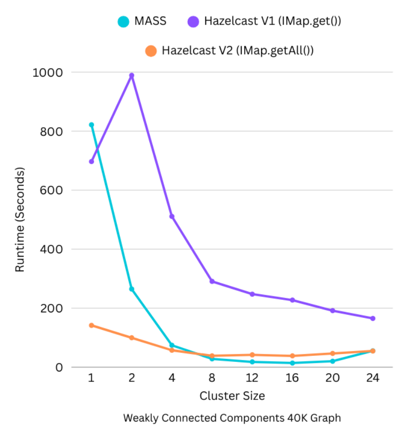




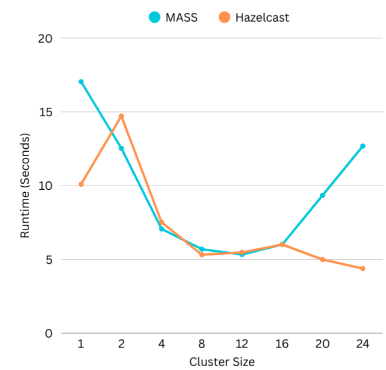




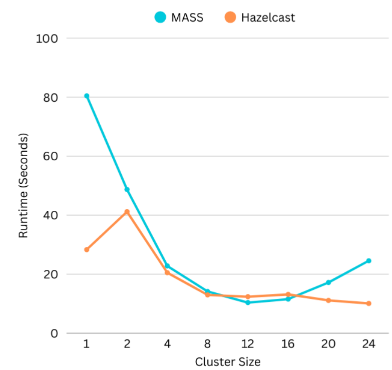




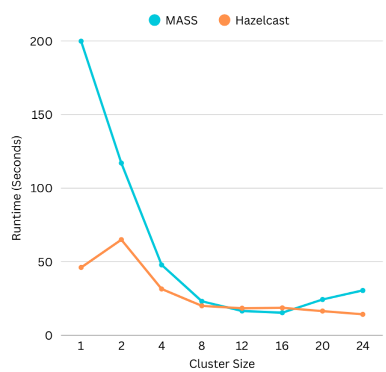
Appendix H: Full Triangle Counting Execution Diagrams



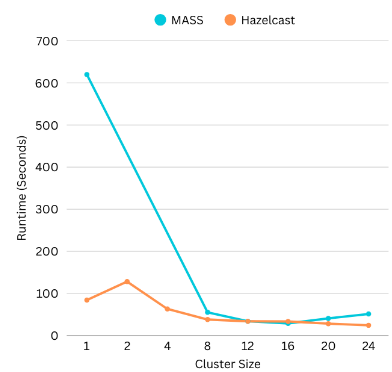
Triangle Counting 1K Graph



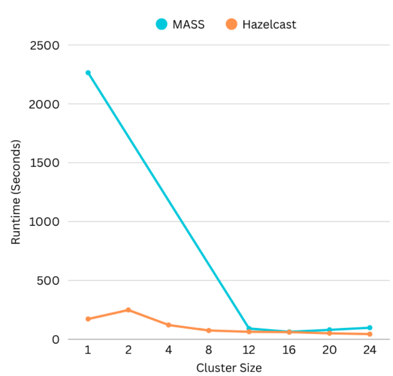
Triangle Counting 3K Graph



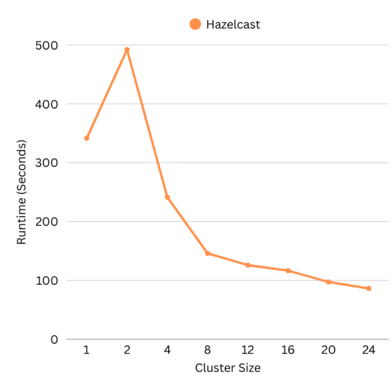
Triangle Counting 5K Graph



Triangle Counting 10K Graph

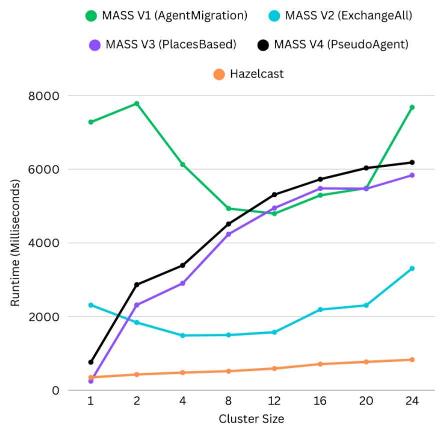


Triangle Counting 20K Graph

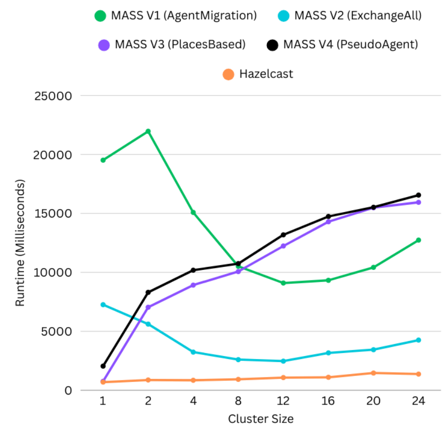


Triangle Counting 40K Graph

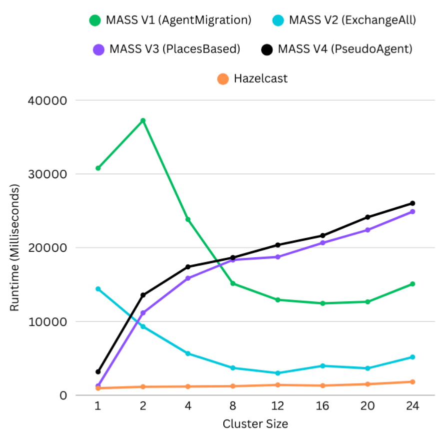
Appendix I: Full PageRank Execution Diagrams



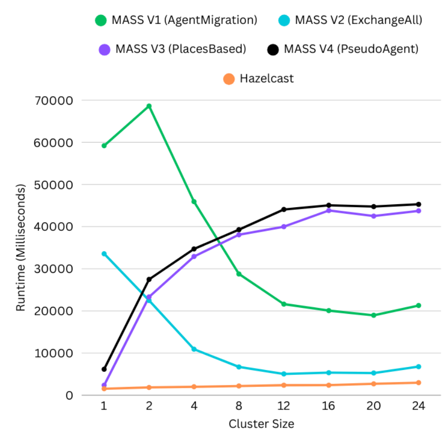
PageRank 1K Graph



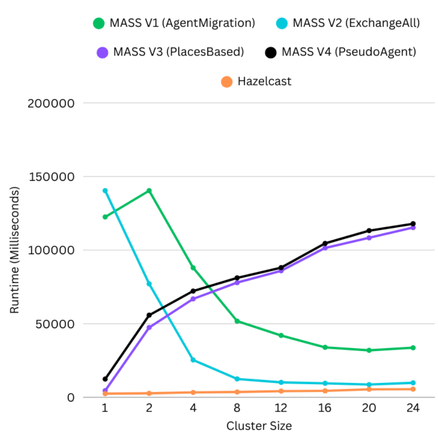
PageRank 3K Graph



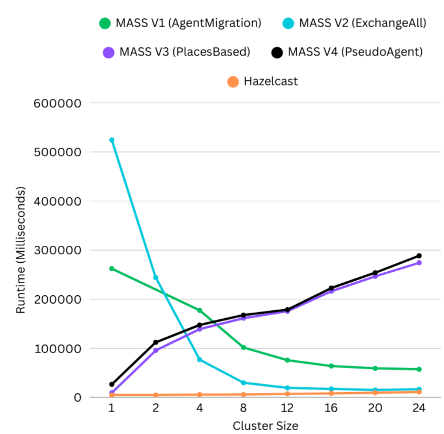
PageRank 5K Graph



PageRank 10K Graph

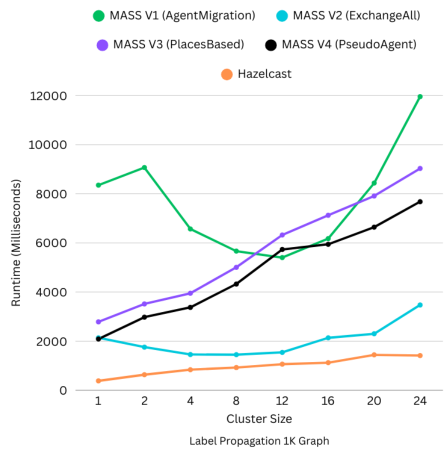


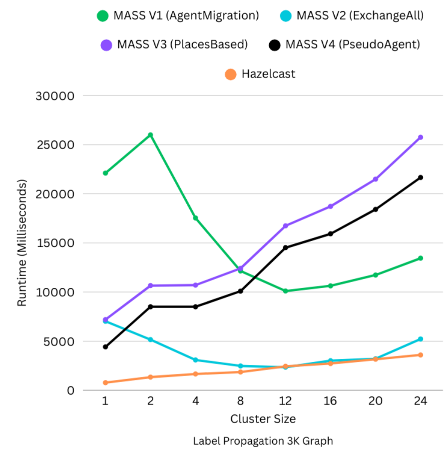
PageRank 20K Graph

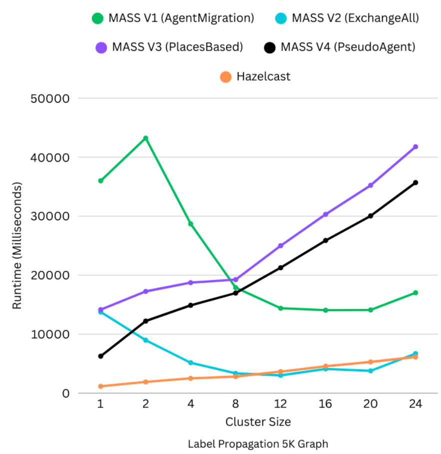


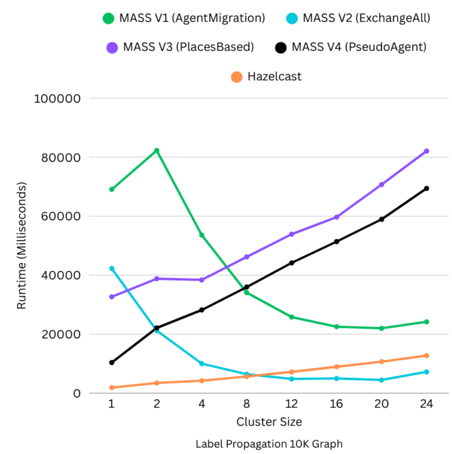
PageRank 40K Graph

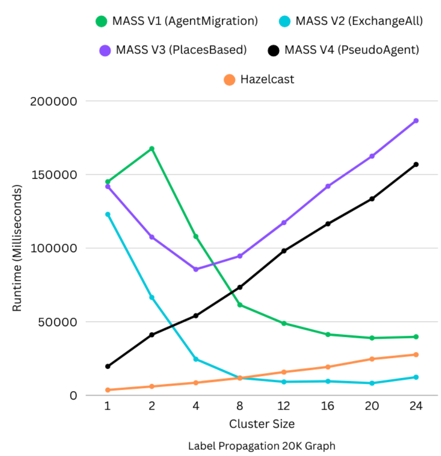
Appendix J: Full Label Propagation Execution Diagrams

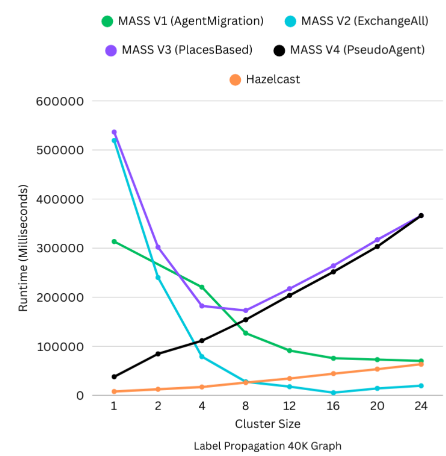












Appendix K: Running Benchmarks

All benchmarks can be found in the develop or noel30/benchmarks branch of the mass_java_appl repository.

For each benchmark:

- Build the jar file using the `make` command.
- For MASS: Configure the `nodes.xml` file to specify what machines you want to run it on and what port to listen in on.
- For Hazelcast: In the `run.sh` file, specify what machines you want the cluster to have.

MASS SCC

1. Location: **Graphs/StronglyConnectedComponents/2025_StronglyConnectedComponents_MASS**
2. Execution: `./run.sh <graph_dsl_file> [boolean_to_print_SCCs]`

Hazelcast SCC

1. Location: **Graphs/Hazelcast_benchmarks/StronglyConnectedComponents**
2. Execution: `./run.sh <graph_dsl_file> <cluster size> [boolean to print components] [boolean to print all components] [component threshold size]`

MASS WCC

1. Location: **Graphs/WeaklyConnectedComponents**
2. Execution: `./run.sh <graph_dsl_file> [boolean_to_print_WCCs]`

Hazelcast WCC

1. Location: **Graphs/Hazelcast_benchmarks/WeaklyConnectedComponents**
2. Execution: `./run.sh <graph_dsl_file> <cluster size> [boolean to print components] [boolean to print all components] [component threshold size]`

MASS Triangle Counting

1. Location: **Graphs/2025_TriangleCounting**
2. Execution: `./run.sh <graph_dsl_file> [boolean_to_print_SCCs]`

Hazelcast Triangle Counting

1. Location: **Graphs/Hazelcast_benchmarks/TriangleCounting**
2. Use the run.sh file to execute the program: `./run.sh <graph_dsl_file> <cluster size>`

MASS PageRank

1. Location/Execution: Refer to Robert Zimmerman's report

Hazlecast PageRank

1. Location: **Graphs/Hazelcast_benchmarks/PageRank**
2. Execution: `./run.sh <dsl graph file> <cluster size> <number of iterations> [boolean to print ranks] [print top x ranks]`

MASS Label Propagation

1. Location/Execution: Refer to Robert Zimmerman's report

Hazlecast Label Propagation

1. Location: **Graphs/Hazelcast_benchmarks/LabelPropagation**
2. Execution: `./run.sh <dsl graph file> <cluster size> <number of iterations> [boolean to print communities]`