Benchmarking Graph Computing Performance Between Java MASS and Hazelcast

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 can't handle the whole simulation. 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 execution time. 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 pieces of data. The DSL is currently developing a distributed, agent-based, graph database system with MASS, and would like to evaluate MASSs graph computing performance and programmability to commonly-used distributed graph computation platforms. One such system is Hazelcast.

My project focuses on comparing Java MASSs and Hazelcast's graph computing performance using graph algorithms frequently used in graph databases. Specifically, I spent the first quarter of my capstone implementing and benchmarking the Strongly Connected Components and Weakly Connected Components algorithms in both Hazelcast and MASS, and the Triangle Counting algorithm in Hazelcast (as the MASS version already existed). I used six different graphs with eight different cluster sizes to benchmark the algorithms, evaluating MASS vs Hazelcast performance in terms of speed and scalability. As well, the programmability metrics assessed 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. Agents can also communicate with each other and spawn child agents, allowing for parallel computation of data.

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 respectively. `GraphPlaces` stores a list of `VertexPlace` objects and represents graphs 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.

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.

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.
- 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 to its entries; combination where the results of each partition 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

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.

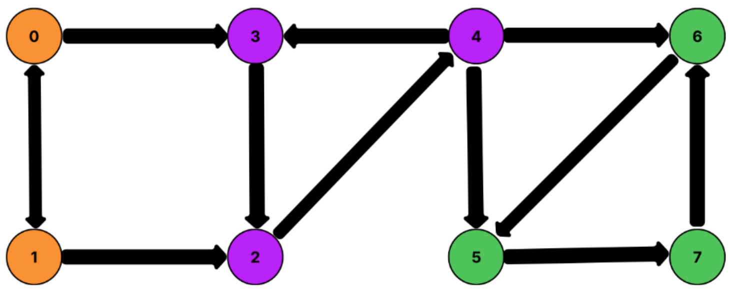


Figure 1: Three strongly connected components in an 8-node graph

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. Initially, the low-link value of each vertex is its own id. Then as you traverse the graph, you maintain a visited set and a stack of valid vertices from which to update the low-link values of when it comes time to. Vertices are added to the stack as they are first explored. As you traverse the graph, when you reach a vertex that has already been visited and is currently on the stack, you start backtracking, updating the low-link values and exploring further paths. When all paths for a vertex have been explored and the vertex's low-link value is equal to its own id, you can start popping from the stack. You pop from the stack until you've reached the vertex you're currently on, which makes up an SCC. This process continues until all vertices have been visited and the stack is empty.

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 SCCs but instead recursively partitions the graph in a way of isolating SCCs 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

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. For each edge in a directed graph, adding an edge going in the opposite direction can be used to convert it into an undirected graph.

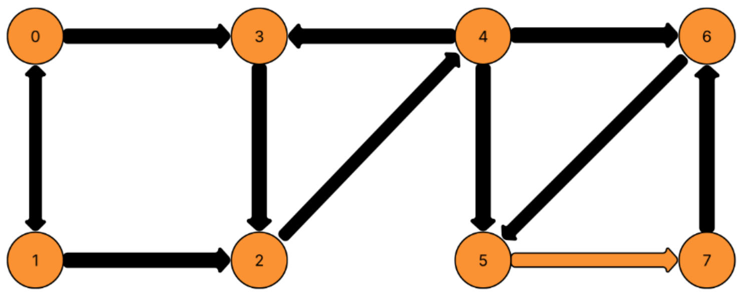


Figure 2: One weakly connected component in an 8-node graph

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. A 3-node cycle is defined as a set of vertices where each vertex is connected to the other two while ignoring edge direction.

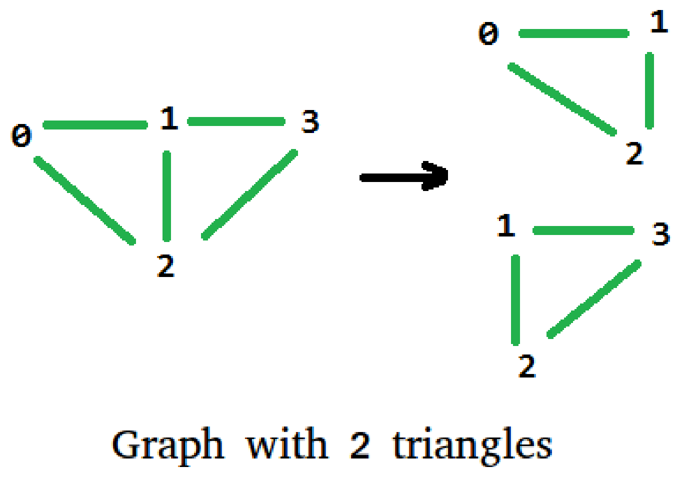


Image source: geeksforgeeks

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.

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. Given a forward edge from vertex v to u, a transposed edge is a forward edge from vertex u to v. These transposed edges need to be created before the process of finding the SCCs 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 (which is only called once). In `preprocessGraph()`, 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

```
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. As well, transposed edges are represented by having a separate `IMap` to store those edges.

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.

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. Before starting the simulation, there is only one agent. 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 (originally, each element is its own parent). 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 containing the parents for every vertex in the graph calling `getComponents()` to return the finalized list of weakly connected components in the graph.

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();

            disjointSet.makeSet(vertexId);
            for (I neighbor : neighbors) {
                Vertex<I,V> neighborVertex = this.dsg.graph.get(neighbor);
                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

There are two MASS implementations of the Triangle Counting algorithm, with them differing in graph setup. The first version uses a random graph generator to create neighbors for each vertex. The user passes in the number of vertices the graph should have, and neighbors of each vertex are dependent on the number of machines in the cluster. The second version uses `GraphPlaces` to read the graph from a DSL file and uses a `GraphModel` object to get a list of vertex ids.

After the graph setup, the two implementations are identical. It starts by spawning an agent at each vertex, and then starting the simulation. 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.

5.3.2 Hazelcast Implementation

The Hazelcast Triangle Counting implementation uses aggregation. A custom `TriangleCountingAggregator` class returns an integer representing 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;
        }
}
```

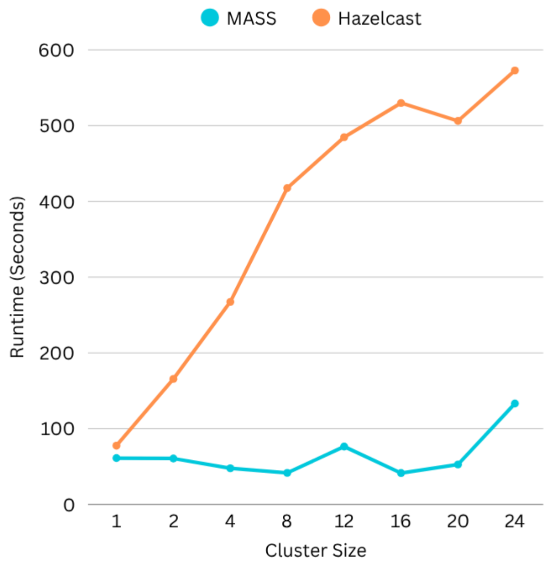
6. Benchmark results

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 run three times and the average of three runs were recorded. See Appendixes A-C to view the full execution results and Appendixes D-F to view the line chart comparisons of the benchmarks.

6.1 Strongly Connected Components Results

MASSs implementation of SCC outperformed Hazelcast's on every combination of graph size to cluster size. As viewed on figure 8, MASS and Hazelcast's performance are somewhat similar on one machine, but as you increase the cluster size, the gap intensifies, with MASSs runtime staying somewhat steady while Hazelcast's would see spikes.

Figure 8: Strongly Connected Components on 10k graph



Strongly Connected Components 10K Graph

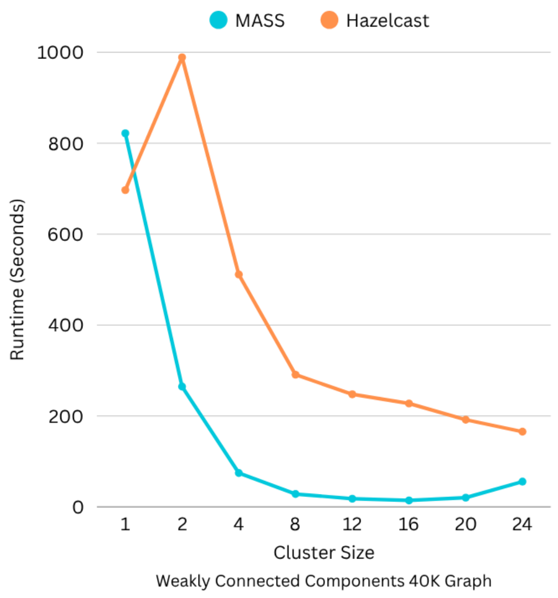
The reason for these differences I believe lie in how the graphs are traversed. Strongly Connected Components require traversing as many edges that can be reached. MASS utilizes a parallel BFS traversal where for every unvisited neighbor, an agent will pick one to traverse along and spawn child agents at the remaining. Because agents operate within their own processes, vertices can be processed concurrently, eliminating the need to backtrack to unvisited paths. Increasing the cluster size increases the number of processes available for agents, ultimately minimizing context switching. 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. Increasing the cluster size spreads the data out, ultimately increasing the number of calls over the network, to which Hazelcast gets heavily penalized.

Another reason for MASSs 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 MASSs 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 enough memory to also accommodate the storage needed for communication overhead.

6.2 Weakly Connected Components Results

MASSs implementation of WCC slightly outperformed Hazelcast's on most combinations of graph size to cluster size. As viewed on figure 9, both platforms are on par with each other, increasing in speed as the cluster size grows.

Figure 9: Weakly Connected Components on 40k graph

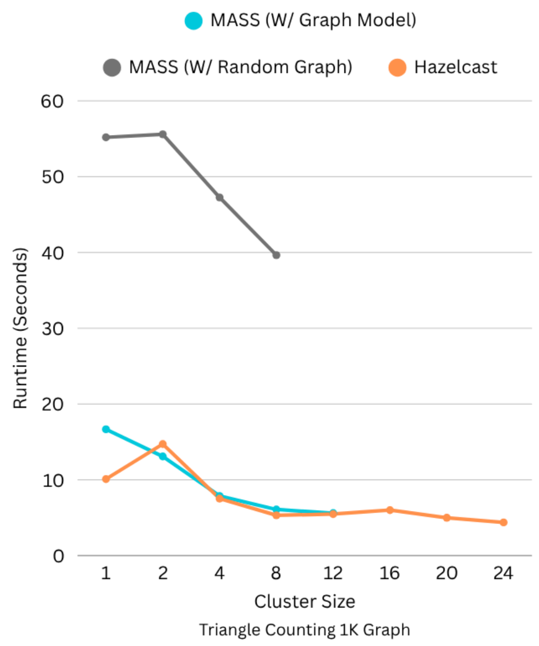


Similar to my conclusion for why MASS SCC performs well, MASS WCC also utilizes a parallel BFS traversal, which can explain why performance gets better (up until 24 machines). In Hazelcast's case, it uses aggregators to first run at each partition before combining the results. When the cluster size increases,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

6.3 Triangle Counting Results

Hazelcast Triangle Counting outperformed MASSs two implementations, with MASSs implementation of Triangle Counting using `GraphPlaces/GraphModel` also outperforming the MASS implementation using a random graph generator. As viewed on figure 10, Hazelcast and MASSs use of `GraphPlaces` are closely matched with Hazelcast having a slight edge regardless of cluster size.

Figure 10: Triangle Counting on 1k graph



Similar to Hazelcast WCC, Hazelcast Triangle Counting uses aggregators, so increasing the cluster size resulted in better performance. For MASS, unlike SCC and WCC, a traversal through the graph is not necessary, so fewer agents are generated requiring less computation. This is why spawning an agent at each vertex works for triangle counting but not SCC/WCC. However, the Hazelcast implementation proved to be more scalable as MASS had a lower ceiling before it crashed/failed. Some observations of the two MASS approaches I noticed include:

- For triangle counting with `GraphModel`, I get a `nullPointerException` on `getGraph()` when I run with a cluster size of 16 or more. As well, only graphs 1k, 3k, and 5k using 1, 2, 4, 8, and 12 machines worked without any errors. Beyond this, the program would crash with either a `nullPointerException` or take too long to finish.
- For triangle counting without using `GraphModel` (the random graph generator), only 1k worked using 1, 2, 4, and 8 machines. After that, I get an `ArrayIndexOutOfBounds` error. And if I tried a 3k graph or higher, I would get a timeout error.

7. Programmability

In terms of programmability, both Java MASS and Hazelcast provided intuitive APIs that made implementing graph computing algorithms flexible and straightforward. Hazelcast's key-value representation of graphs is more prevalent than MASSs agents and places, so I had an easier time onboarding/understanding Hazelcast than MASS. As well, Hazelcast's rich suite of distributed computing features provided me with a wide range of possible implementations. But MASSs ability to have agents not only traverse places but also store information and communicate amongst each other provides a level of performance enhancements I don't think Hazelcast offers. As well, being able to spawn child agents to contribute to traversals is why I believe MASS performed better than Hazelcast in the SCC and WCC benchmarks.

Tables 1, 2, and 3 show the programmability metrics captured between MASS and Hazelcast's benchmarks.

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)	624
Lines of Logic	498
Boilerplate %	35.54%
Cyclomatic Complexity	3.9
Lack of Cohesion in Methods (LCOM4)	1.25

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	43
Total Lines of Code (LOC)	1157
Line of Logic	529
Boilerplate %	30.81%
Cyclomatic Complexity	2.6
Lack of Cohesion in Methods (LCOM4)	2.17

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)	631
Line of Logic	380
Boilerplate %	39.47%
Cyclomatic Complexity	3.5
Lack of Cohesion in Methods (LCOM4)	1.25

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	38
Total Lines of Code (LOC)	899
Line of Logic	420
Boilerplate %	38.1%
Cyclomatic Complexity	2.6
Lack of Cohesion in Methods (LCOM4)	2

Table 3: MASS (using GraphModel) vs Hazelcast Triangle Counting Programmability Metrics

Measurement (MASS)	Value
Number of files	3
Number of methods	12
Total Lines of Code (LOC)	564
Line of Logic	181
Boilerplate %	36.96%
Cyclomatic Complexity	3.7
Lack of Cohesion in Methods (LCOM4)	2.3

Measurement (Hazelcast)	Value
Number of files	3
Number of methods	30
Total Lines of Code (LOC)	717
Line of Logic	325
Boilerplate %	37.28%
Cyclomatic Complexity	2.5
Lack of Cohesion in Methods (LCOM4)	1.75

8. Conclusion

This quarter, I implemented and benchmarked the Strongly Connected Components and Weakly Connected Components algorithms in both Hazelcast and MASS, and the Triangle Counting algorithm in Hazelcast. MASS outperformed Hazelcast in the SCC and WCC algorithms while Hazelcast not only outperformed but scaled better in the Triangle Counting algorithm.

Future work can include improving MASSs `GraphModel` performance. Since `GraphModel` is attempting to load the whole graph onto one machine, a graph too large (or a large cluster size attempting a high usage of message transfers) will result in runtime failures.

Appendix A: Full Strongly Connected Components Execution Results

MASS SCC Results

num-members	num-vertices	total-agents-generated	load-time (sec)	runtime (sec)
1	1000	367926	1.092	6.391
2	1000	367926	2.601	7.882
4	1000	367926	5.222	6.37
8	1000	367926	10.295	6.529
12	1000	367926	25.006	14.679
16	1000	367926	24.355	8.429
20	1000	367926	32.504	9.814

24	1000	367926	48.897	31.079
1	3000	1157222	1.099	18.41
2	3000	1157222	2.517	20.679
4	3000	1157222	5.22	18.155
8	3000	1157222	10.206	16.125
12	3000	1157222	23.777	34.037
16	3000	1157222	25.747	16.111
20	3000	1157222	30.478	20.65
24	3000	1157222	46.875	62.471
1	5000	1941566	1.214	32.711
2	5000	1941566	2.62	30.87
4	5000	1941566	5.173	27.232
8	5000	1941566	10.049	24.565
12	5000	1941566	23.958	52.344
16	5000	1941566	22.529	22.971
20	5000	1941566	30.387	33.54
24	5000	1941566	46.823	77.886
1	10000	3899966	1.476	61.159
2	10000	3899966	2.98	60.681
4	10000	3899966	5.255	47.76
8	10000	3899966	10.328	41.672
12	10000	3899966	24.826	76.454
16	10000	3899966	23.72	41.478
20	10000	3899966	30.927	52.704
24	10000	3899966	49.191	133.243
1	20000	7825526	1.92	122.66
2	20000	7825526	3.151	118.97
4	20000	7825526	5.252	98.179
8	20000	7825526	10.404	76.721
12	20000	7825526	26.041	119.219
16	20000	7825526	23.006	71.258
20	20000	7825526	31.704	96.708
24	20000	7825526	48.066	217.065
1	40000	15737702	2.865	234.301
2	40000	NA	NA	NA
4	40000	NA	NA	NA
8	40000	15737702	10.802	192.977

12	40000	15737702	16.901	159.211
16	40000	15737702	23.838	170.845
20	40000	15737702	33.193	167.496
24	40000	15737702	53.377	416.598

Hazelcast SCC Results

num-members	num-vertices	num-edges	load time (sec)	runtime (sec)
1	1000	93480	0.543	7.914
2	1000	93480	0.652	16.911
4	1000	93480	0.635	25.575
8	1000	93480	0.612	41.917
12	1000	93480	0.699	66.407
16	1000	93480	1.163	67.929
20	1000	93480	0.812	61.841
24	1000	93480	1.031	55.176
1	3000	293804	0.958	23.919
2	3000	293804	0.922	49.59
4	3000	293804	0.87	77.206
8	3000	293804	0.824	127.151
12	3000	293804	0.861	180.68
16	3000	293804	0.905	163.825
20	3000	293804	1.041	155.639
24	3000	293804	1.13	184.951
1	5000	492890	1.161	38.521
2	5000	492890	1.128	82.249
4	5000	492890	1.06	142.589
8	5000	492890	1.037	211.193
12	5000	492890	1.186	256.158
16	5000	492890	1.151	272.53
20	5000	492890	1.3	288.086
24	5000	492890	1.398	252.14
1	10000	989990	1.663	77.599
2	10000	989990	1.687	165.679
4	10000	989990	1.528	267.375
8	10000	989990	1.359	417.562
12	10000	989990	1.597	484.762

16	10000	989990	1.583	529.904
20	10000	989990	1.85	506.216
24	10000	989990	2.126	572.812
1	20000	1986380	2.533	148.734
2	20000	1986380	2.634	331.664
4	20000	1986380	2.131	577.592
8	20000	1986380	2.045	854.984
12	20000	1986380	2.351	1126.491
16	20000	1986380	2.586	1070.5
20	20000	1986380	2.771	1135.935
24	20000	1986380	4.836	2280.741
1	40000	3994424	4.057	305.06
2	40000	3994424	4.402	668.154
4	40000	3994424	3.958	1048.195
8	40000	3994424	3.621	1726.983
12	40000	3994424	6.762	3751.369
16	40000	3994424	4.175	2536.314
20	40000	3994424	4.358	2292.878
24	40000	3994424	4.836	2280.741

Appendix B: Full Weakly Connected Components Execution Results

MASS WCC Results

num-members	num-vertices	total-agents-generated	load-time (sec)	runtime (sec)
1	1000	92481	1.025	1.725
2	1000	92481	2.53	1.799
4	1000	92481	4.877	1.556
8	1000	92481	9.821	1.776
12	1000	92481	15.587	2.018
16	1000	92481	31.594	2.419
20	1000	92481	40.641	4.602
24	1000	92481	55.086	20.897
1	3000	290805	1.12	7.464
2	3000	290805	2.642	4.796
4	3000	290805	4.979	3.219
8	3000	290805	9.907	2.853

12	3000	290805	15.502	2.998
16	3000	290805	32.163	3.325
20	3000	290805	44.668	5.957
24	3000	290805	56.991	22.528
1	5000	487891	1.244	17.349
2	5000	487891	2.665	8.469
4	5000	487891	5.065	4.852
8	5000	487891	10.168	3.812
12	5000	487891	15.756	3.896
16	5000	487891	32.166	4.155
20	5000	487891	40.867	6.826
24	5000	487891	55.292	23.319
1	10000	979991	1.4	57.924
2	10000	979991	2.871	21.852
4	10000	979991	5.247	9.441
8	10000	979991	10.16	6.271
12	10000	979991	15.763	5.214
16	10000	979991	31.962	5.549
20	10000	979991	41.044	9.511
24	10000	979991	58.009	30.984
1	20000	1966381	1.898	206.237
2	20000	1966381	3.091	68.935
4	20000	1966381	5.404	24.402
8	20000	1966381	10.344	11.588
12	20000	1966381	16.046	8.259
16	20000	1966381	32.309	8.258
20	20000	1966381	41.529	13.467
24	20000	1966381	56.432	42.204
1	40000	3954425	2.886	821.771
2	40000	3954425	3.79	264.688
4	40000	3954425	5.798	74.47
8	40000	3954425	10.72	28.296
12	40000	3954425	16.543	17.974
16	40000	3954425	32.356	14.164
20	40000	3954425	43.448	20.301
24	40000	3954425	59.305	55.658

Hazelcast WCC Results

num-members	num-vertices	num-edges	load time (sec)	runtime (sec)
1	1000	93480	1.095	17.017
2	1000	93480	1.033	26.421
4	1000	93480	0.847	13.461
8	1000	93480	0.873	9.354
12	1000	93480	1.037	9.284
16	1000	93480	1.188	9.748
20	1000	93480	1.14	8.186
24	1000	93480	1.072	7.23
1	3000	293804	1.527	52.164
2	3000	293804	1.496	76.198
4	3000	293804	1.364	39.4
8	3000	293804	1.169	23.578
12	3000	293804	1.333	21.942
16	3000	293804	1.445	65.211
20	3000	293804	1.274	18.531
24	3000	293804	1.427	16.477
1	5000	492890	2.074	84.015
2	5000	492890	1.764	126.815
4	5000	492890	1.414	62.825
8	5000	492890	1.298	39.209
12	5000	492890	1.513	34.039
16	5000	492890	1.764	33.298
20	5000	492890	1.586	28.276
24	5000	492890	1.545	25.244
1	10000	989990	2.575	172.581
2	10000	989990	2.232	252.7
4	10000	989990	1.905	124.799
8	10000	989990	1.86	74.847
12	10000	989990	2.057	64.633
16	10000	989990	2.21	61.553
20	10000	989990	2.046	53.437
24	10000	989990	1.695	43.905
1	20000	1986380	4.391	344.179
2	20000	1986380	4.004	485.215

4	20000	1986380	2.906	242.39
8	20000	1986380	2.621	147.382
12	20000	1986380	3.014	124.564
16	20000	1986380	2.912	116.918
20	20000	1986380	2.71	98.284
24	20000	1986380	2.445	87.432
1	40000	3994424	6.269	696.875
2	40000	3994424	6.805	988.923
4	40000	3994424	5.323	511.219
8	40000	3994424	4.292	290.735
12	40000	3994424	3.888	247.65
16	40000	3994424	4.401	227.437
20	40000	3994424	4.26	191.692
24	40000	3994424	4.006	165.389

Appendix C: Full Triangle Execution Results

MASS Triangle Counting Results Using GraphModel

num-members	num-vertices	total-agents-generated	load-time (sec)	runtime (sec)
1	1000	1778723	0.151	16.668
2	1000	1778723	0.133	13.09
4	1000	1778723	0.159	7.87
8	1000	1778723	0.297	6.086
12	1000	1778723	0.325	5.607
1	3000	5508612	0.238	79.753
2	3000	5508612	0.215	50.743
4	3000	5508612	0.26	24.202
8	3000	5508612	0.356	15.302
12	3000	5508612	0.45	12.1
1	5000	9192458	0.335	203.333
2	5000	9192458	0.315	122.801
4	5000	9192458	0.31	49.977
8	5000	9192458	0.451	26.094
12	5000	9192458	0.511	19.184

MASS Triangle Counting Results Using Random Graph

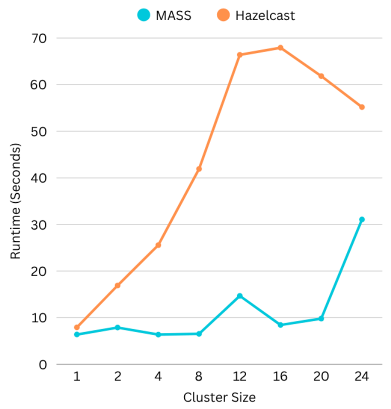
num-members	num-vertices	total-agents-generated	num-triangles-found	load-time (sec)	runtime (sec)
1	1000	2125319	1000000	56.09	55.19
2	1000	4125319	2000000	52.21	55.594
4	1000	7273102	3147783	64.82	47.264
8	1000	11396769	4309409	105.57	39.647

Hazelcast Triangle Counting Results

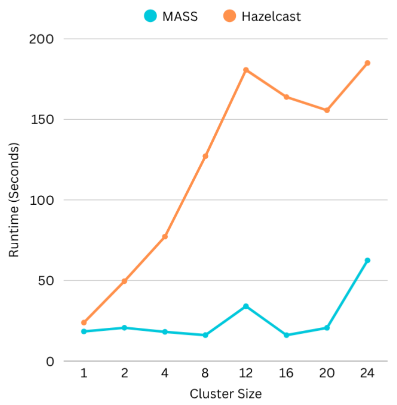
num-members	num-vertices	num-edges	load time (sec)	runtime (sec)
1	1000	93480	1.02	10.094
2	1000	93480	1.172	14.715
4	1000	93480	0.893	7.519
8	1000	93480	0.726	5.319
12	1000	93480	0.984	5.477
16	1000	93480	1.15	6.005
20	1000	93480	1.052	4.995
24	1000	93480	0.842	4.379
1	3000	293804	1.574	28.335
2	3000	293804	1.64	41.173
4	3000	293804	1.326	20.507
8	3000	293804	1.029	12.985
12	3000	293804	1.519	12.381
16	3000	293804	1.468	13.154
20	3000	293804	1.41	11.13
24	3000	293804	1.46	10.093
1	5000	492890	2.117	46.1
2	5000	492890	2.075	64.993
4	5000	492890	1.512	31.554
8	5000	492890	1.36	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	84.295
2	10000	989990	2.592	128.125

4	10000	989990	2.063	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	2.12	28.141
24	10000	989990	2.168	24.337
1	20000	1986380	4.202	173.062
2	20000	1986380	3.817	249.16
4	20000	1986380	3.032	121.809
8	20000	1986380	2.723	75.061
12	20000	1986380	2.858	64.37
16	20000	1986380	3.306	61.314
20	20000	1986380	2.78	51.202
24	20000	1986380	2.462	44.564
1	40000	3994424	6.44	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	4.91	116.628
20	40000	3994424	4.095	97.466
24	40000	3994424	3.668	86.392

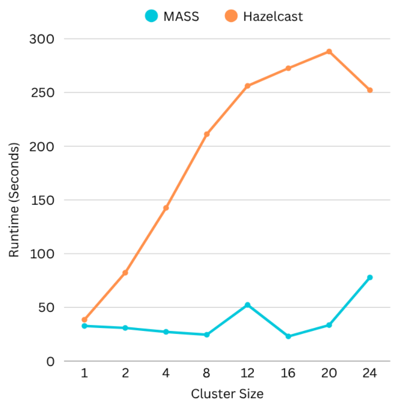
Appendix D: Full Strongly Connected Components Execution Diagrams



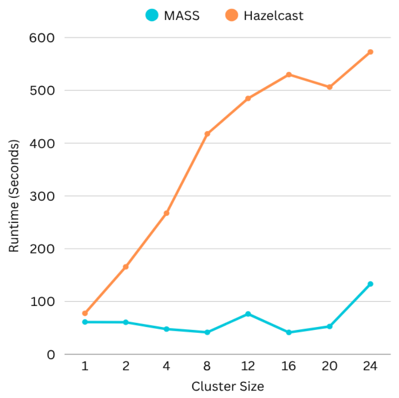
Strongly Connected Components 1K Graph



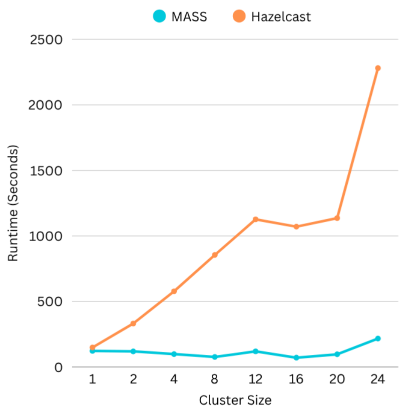
Strongly Connected Components 3K Graph



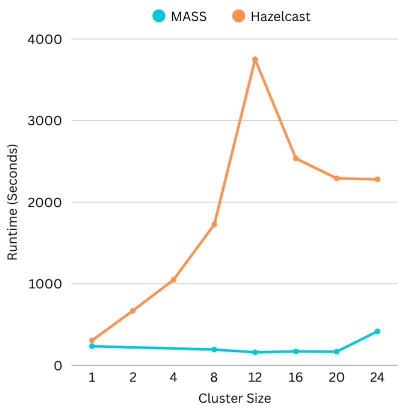
Strongly Connected Components 5K Graph



Strongly Connected Components 10K Graph

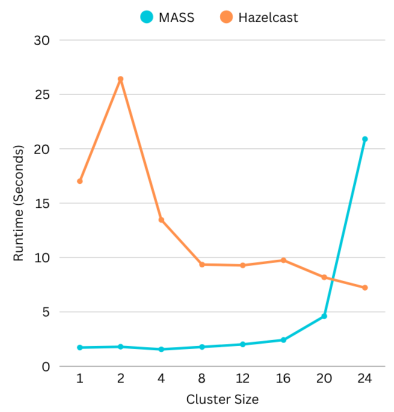


Strongly Connected Components 20K Graph

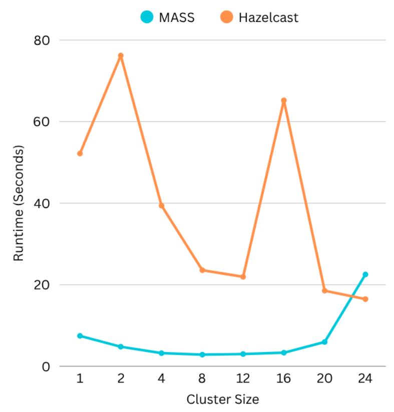


Strongly Connected Components 40K Graph

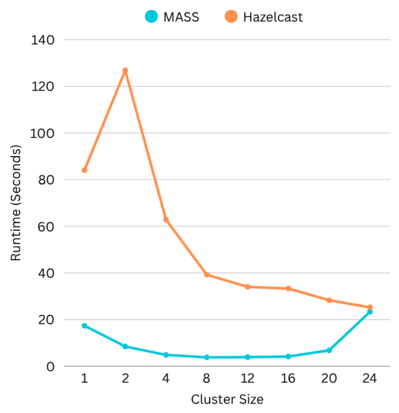
Appendix E: Full Weakly Connected Components Execution Diagrams



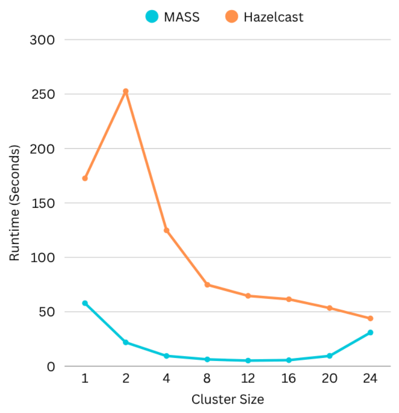
Weakly Connected Components 1K Graph



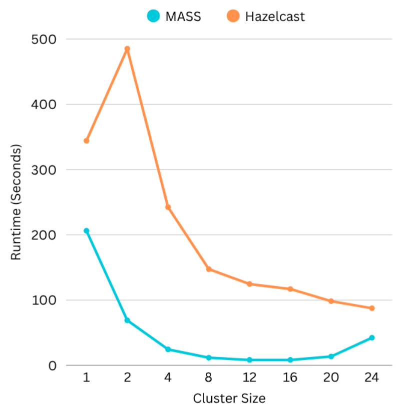
Weakly Connected Components 3K Graph



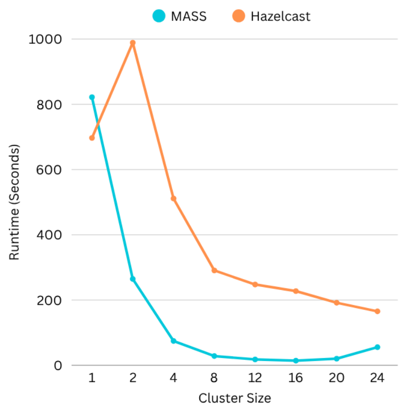
Weakly Connected Components 5K Graph



Weakly Connected Components 10K Graph

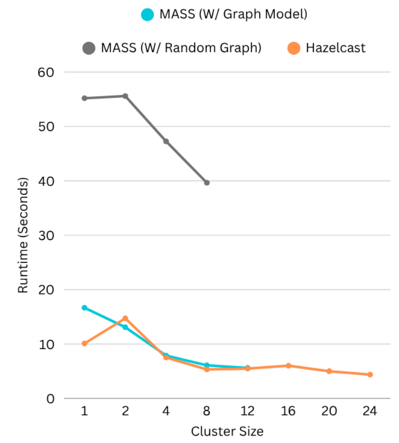


Weakly Connected Components 20K Graph

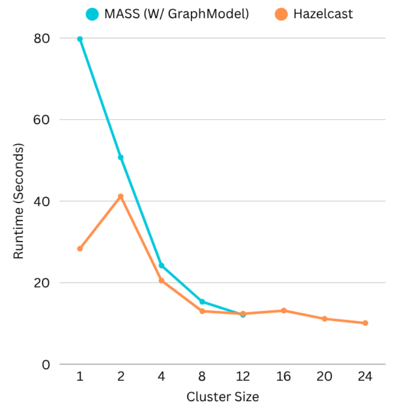


Weakly Connected Components 40K Graph

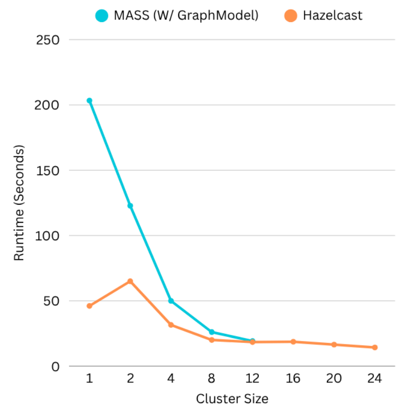
Appendix F: Full Triangle Counting Execution Diagrams



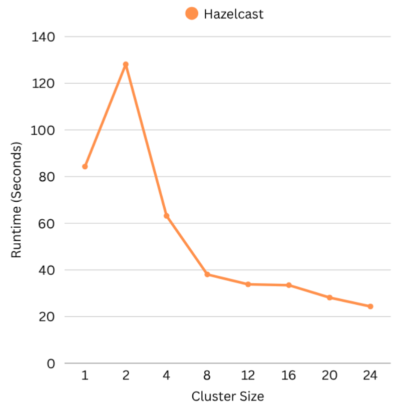
Triangle Counting 1K Graph



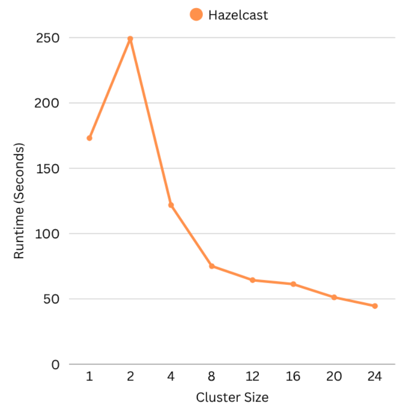
Triangle Counting 3K Graph



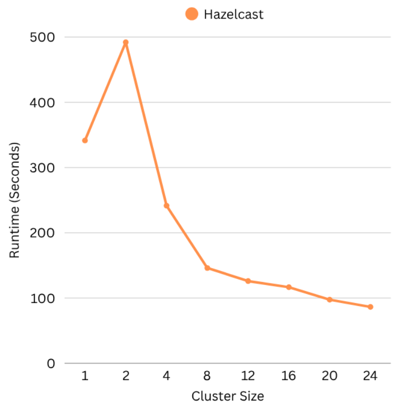
Triangle Counting 5K Graph



Triangle Counting 10K Graph



Triangle Counting 20K Graph



Triangle Counting 40K Graph

Appendix G: Running Benchmarks

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

If you don't have the latest version of mass_java_core, you can download following these steps:

1. git clone https://<ACCOUNT>@bitbucket.org/mass_library_developers/mass_java_core.git
2. `cd mass_java_core`
3. Run: `mvn -DskipTests clean package install`

MASS SCC

1. Location: **Graphs/StronglyConnectedComponents/MASS_StronglyConnectedComponents**
2. Build the jar file using the `make` command.
3. Configure the `nodes.xml` file to specify what machines you want to run it on and what port to listen in on.
4. Use the `run.sh` file to execute the program: `./run.sh <graph_dsl_file> [boolean_to_print_SCCs]`

Hazelcast SCC

1. Location: **Graphs/Hazelcast_benchmarks/StronglyConnectedComponents**
2. Build the jar file using the `make` command.
3. In the `run.sh` file, specify what machines you want to run the Hazelcast cluster on.
4. Use the `run.sh` file to execute the program: `./run.sh <graph_dsl_file> <cluster size> [boolean to print components] [boolean to print all components] [component threshold size]`

MASS WCC

1. Location: **Graphs/StronglyConnectedComponents/MASS_StronglyConnectedComponents**
2. Build the jar file using the `make` command.
3. Configure the `nodes.xml` file to specify what machines you want to run it on and what port to listen in on.
4. Use the `run.sh` file to execute the program: `./run.sh <graph_dsl_file> [boolean_to_print_SCCs]`

Hazelcast WCC

1. Location: **Graphs/Hazelcast_benchmarks/WeaklyConnectedComponents**
2. Build the jar file using the `make` command.
3. In the `run.sh` file, specify what machines you want to run the Hazelcast cluster on.
4. Use the `run.sh` file to execute the program: `./run.sh <graph_dsl_file> <cluster size> [boolean to print components] [boolean to print all components] [component threshold size]`

Hazelcast Triangle Counting

1. Location: **Graphs/Hazelcast_benchmarks/Triangle Counting**
2. Build the jar file using the `make` command.
3. In the `run.sh` file, specify what machines you want to run the Hazelcast cluster on.
4. Use the run.sh file to execute the program: `./run.sh <graph_dsl_file> <cluster size>`