Parallelization of the Naïve Articulation Points Algorithm with Four Frameworks

Logan Choi, Ryan Decker, Akbarbek Rakhmatullaev

University of Washington Bothell

CSS 534: Parallel Programming

December 11, 2023

Introduction

An articulation point in a connected graph is a vertex whose removal disconnects the graph. Identifying articulation points is important as it pinpoints weaknesses within any network. For instance, if an articulation node fails, the whole network can crash because of the disconnection.

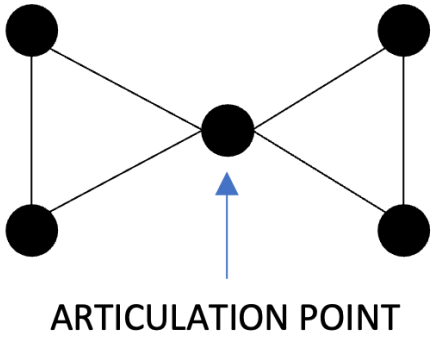


Figure 1: Articulation Point in a Connected Graph

Some modern use cases for identifying articulation points include power grids, computer networks, and epidemiology. For power grids and computer networks, locating the weaknesses enhances the resilience and reliability of the network. For epidemiology, these points help in understanding key locations or individuals that can prevent the rapid spread of a disease through a population. However, identifying these articulation points can be tedious and time consuming, thus this project outlines and measures parallelization frameworks to find the most optimal solution. In this project, we parallelized a brute force algorithm for finding articulation points using mpiJava, MapReduce, Spark, and MASS. The mpiJava implementation was developed by Logan Choi, MapReduce by Ryan Decker, Spark by Akbarbek Rakhmatullaev, and the MASS implementation was developed together.

Brute Force Algorithm

In order to facilitate the parallelization of the identification process for articulation points, a brute force algorithm has been devised. The brute force algorithm is outlined as follows:

1. For each node, systematically remove the node and all associated connections, retaining only one connection
2. Instantiate a queue with the remaining connection
3. Utilize a Breadth-First-Search (BFS) traversal through the modified graph
4. Keep a count of the visited nodes during the traversal
5. Evaluate whether the total number of visited nodes is equal to the total number of nodes minus one. If the condition is met, the node is identified as an articulation point.

Parallelization with mpiJava

The Message Passing Interface (MPI) is a standard communication protocol used for parallel programming, and mpiJava is the Java implementation of the framework. The protocol utilizes communication by sending and receiving messages, allowing multiple processes to work concurrently. In the context of identifying articulation points, mpiJava functions were utilized to analyze scalability and enhance efficiency of the brute force algorithm. The methodology is as follows:

1. Broadcast the adjacency matrix to all ranks, ensuring a smooth traversal.
2. Partition the adjacency matrix by rows, assigning each rank a specific range.
3. Execute the brute force algorithm independently for each rank.
4. Utilize `MPI_Reduce` to aggregate and sum the total number of identified articulation points across all ranks.

Parallelization with MapReduce

MapReduce is a parallelization framework designed for processing big data. It is built on top of the virtual Hadoop filesystem, which distributes data in blocks across a cluster of computing nodes. MapReduce programs using a sequence of mappings and reductions, between which the data is written to disk, which makes the program tolerant to faults but decreases its efficiency. In the initial implementation, we attempted to parallelize each of the breadth-first searches in concert across a single distributed graph using an iterated reduction task, where the key was the vertexID and the values were search tickets, Text objects which tracked the progress of each breadth-first search. While this implementation gave the correct result, it required 38 minutes to process a graph with only 10 vertices, and it ran out of space given a graph with 100 vertices.

In the second implementation, we simplified the program to one map method and one reduce method. Given an input file where each line specified a vertex and its neighbors, the map method collected <vertex ID, line> for each vertex from 0 to the number of vertices. This propagated each line of the input file to each vertex key. In the reduce method, each vertex key reconstructed the whole graph, performed a sequential BFS, and then incremented a global counter if that vertex key was articulating. The main method retrieved the global counter to find the number of articulation points. This implementation obtained correct results with much greater efficiency.

Parallelization with Apache Spark

Apache Spark is a parallelization framework designed to process big data and analytics. It builds upon MapReduce and its limitations. The methodology implemented was given a file where each line specifies a vertex and its neighbors. Map and index the input to the corresponding format and create the graph from the obtained result as JavaPairRDD. Afterwards, create an array representation of the graph from the JavaPairRDD graph. While mapping to a new

JavaPairRDD, for each tuple, perform the sequential BFS algorithm and return true if the number of visited vertices is equal to the total number of vertices minus one. Finally, iterate over each element in a new JavaPairRDD, and filter the keys where the value is true.

Parallelization with MASS

For the MASS parallelization framework, the implementation was straightforward. By spawning a number of agents equal to the number of nodes in the graph, each agent has a unique ID number starting from 0. By utilizing that characteristic, the graph was created, and the agent's ID correlated with each node within the graph. Afterwards, each agent traversed through the graph utilizing the brute force algorithm and returned their ID to know which node was an articulation point. Throughout that process, the total number of articulation points was calculated and shown. However, all results processed were within one node, as there were numerous complications when implementing more nodes.

Performance Analysis

Number of Vertices	Execution Time (s)				
	Sequential	mpiJava (4 Ranks)	MapReduce (4 Nodes)	Spark (4 Nodes)	MASS (1 Node)
1000	1.819	1.000	26.981	1.106	1.680
2000	20.684	6.273	30.082	7.566	1.924
4000	162.568	47.076	42.661	22.637	2.188

Table 1: Performance of Parallelization Frameworks Over Different Graph Sizes

Table 1 shows that there were significant improvements from all parallelization frameworks compared to the sequential implementation. Although MapReduce was slower when the graph size was the smallest, as the graph size scaled upwards, MapReduce resulted in a better execution time when measured against the sequential implementation. mpiJava and Spark also

showed significant improvement compared to the sequential. However, mpiJava scaled the worst out of the four parallel implementations as the graph grew. This is due to the nature of broadcasting the entire adjacency matrix instead of sending and receiving only necessary information. MASS showed the best results with an astonishing 2.188 seconds runtime for a graph size of 4000. However, MASS only utilized one node as there were numerous complications when executing the program with multiple nodes.

Programmability Analysis

Table 2 shows the number of lines of code and the cyclomatic complexity for each implementation of the algorithm. The cyclomatic complexity was determined by counting the number of branch and loop statements in the program. mpiJava was relatively simple to utilize as the adjacency matrix was broadcasted easily. However, it was not easily scalable with bigger graphs due to the broadcasting element. We felt that MapReduce was not well-suited to iterative tasks or graph processing. We could only get good performance by using as few maps and reductions as possible. It also required a lot of boilerplate code. Spark works well with any type of datasets, but it requires considerable memory resources. Even the finest optimization might not help if the dataset is too big. For MASS, it took us some time to get our heads around the key ideas, and we found it difficult to set up multiple computing nodes, but we felt that it was a versatile framework that could be applied naturally to a wide range of problems, and we were impressed by its performance.

	Sequential	mpiJava	MapReduce	Spark	MASS
Lines of Code	47	67	90	72	107
Cyclomatic Complexity	7	9	9	6	8

Table 2: Programmability Measures for Articulation Points Implementations

Appendix A: Programs

The implementation for each program is listed below.

Listing 1: Sequential Implementation

```
package artpoints;

import java.io.File;
import java.io.IOException;
import java.util.*;

/**
 * Implements the brute-force algorithm for finding articulation
points
 * in a graph as a sequential program.
 */
public class SequentialArticulation {
    public static final int GRAPH_SIZE = 4000;

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        try (Scanner fileReader = new Scanner(new File("data/graph" +
GRAPH_SIZE + ".txt"))) {
            int[][] edges = new int[GRAPH_SIZE][GRAPH_SIZE];
            for (int i = 0; i < GRAPH_SIZE; i++) {
                String[] neighbors =
fileReader.nextLine().split("\\s+");
                for (int j = 0; j < GRAPH_SIZE; j++) {
                    edges[i][j] = Integer.parseInt(neighbors[j]);
                }
            }
            System.out.println(findArticulationPoints(edges));
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

        long endTime = System.currentTimeMillis();
        long elapsedTime = endTime - startTime;
        System.out.println("Elapsed time: " + elapsedTime);
    }

    public static int findArticulationPoints(int[][] edges) {
```

```
        int articulationPoints = 0;
        // Try removing each vertex i
        for (int target = 0; target < edges.length; target++) {
            Deque<Integer> queue = new ArrayDeque<>();
            Set<Integer> visited = new HashSet<>();
            queue.addLast((target + 1) % edges.length);
            visited.add((target + 1) % edges.length);

            while (!queue.isEmpty()) {
                int vertex = queue.removeFirst();
                for (int neighbor = 0; neighbor < edges.length;
neighbor++) {
                    if (neighbor != target && edges[vertex][neighbor]
> 0 && !visited.contains(neighbor)) {
                        visited.add(neighbor);
                        queue.addLast(neighbor);
                    }
                }
            }

            if (visited.size() != edges.length - 1) {
                articulationPoints++;
            }
        }
        return articulationPoints;
    }
}
```

Listing 2: mpiJava Implementation

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashSet;
import java.util.Set;
import mpi.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class MPIArticulation {
    public static void main(String[] args) throws MPIException
,FileNotFoundException {
        String filePath = "./output_matrix.txt";
        File file = new File(filePath);
```

```
        Scanner scanner = new Scanner(file);
        int graphSize = scanner.nextInt();
        int[][] edges = new int[graphSize][graphSize];
        for(int i =0; i< graphSize;i++){
            for(int j =0; j<graphSize;j++){
                edges[i][j] = scanner.nextInt();
            }
        }
        scanner.close();

        long startTime = System.currentTimeMillis();
        MPI.Init(args);
        int my_rank = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();

        int rows = edges.length;
        MPI.COMM_WORLD.Bcast(new int[]{rows}, 0, 1, MPI.INT, 0);

        for (int i = 0; i< rows; i++){
            MPI.COMM_WORLD.Bcast(edges[i], 0, edges[i].length,
MPI.INT, 0);
        }

        int vertices = edges.length / size;
        int start = my_rank * vertices;
        int end = start + vertices;

        int articulationPoints = 0;
        // Try removing each vertex
        for (int target = start; target < end; target++) {
            Deque<Integer> queue = new ArrayDeque<>();
            Set<Integer> visited = new HashSet<>();
            queue.addLast((target + 1) % edges.length);
            visited.add((target + 1) % edges.length);

            while (!queue.isEmpty()) {
                int vertex = queue.removeFirst();
                for (int neighbor = 0; neighbor < edges.length;
neighbor++) {
                    if (neighbor != target && edges[vertex][neighbor]
> 0 && !visited.contains(neighbor)) {
                        visited.add(neighbor);
                        queue.addLast(neighbor);
                    }
```

```
                }
            }

            if (visited.size() != edges.length - 1) {
                articulationPoints++;
                System.out.println("Rank " + my_rank + " -
Articulation Point: " + target);
            }
        }

        int[] sendPts = new int[]{articulationPoints};
        int[] recvPts = new int[1];
        MPI.COMM_WORLD.Reduce(sendPts,0,recvPts, 0, 1, MPI.INT,
MPI.SUM, 0);

        if (my_rank == 0){
            System.out.println();
            System.out.println("Total Articulation Points: " +
recvPts[0]);
            long endTime = System.currentTimeMillis() - startTime;
            System.out.println("Total Time: " + endTime + "
milliseconds");
        }
        MPI.Finalize();
    }
}
```

Listing 3: MapReduce Implementation

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

/**
 * Parallelizes the naive articulation points algorithm using
MapReduce.
 *
 * @author Ryan Decker
 */
```

```
public class MapReduceArticulation {
    public enum Counter {
        COUNT
    }

    /**
     * Configures and executes the MapReduce job.
     *
     * @param args the input directory, output directory, and keywords
     */
    public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(ArticulationPoints.class);
        conf.setJobName("Articulation Points");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);

        conf.setMapperClass(Map.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        // n maintains number of vertices
        conf.set("n", args[2]);

        RunningJob job = JobClient.runJob(conf);
        long artPoints =
job.getCounters().findCounter(Counter.COUNT).getValue();
        System.out.println(artPoints);
    }

    /**
     * Implements the mapping operation for this program.
     */
    public static class Map
            extends MapReduceBase
            implements Mapper<LongWritable, Text, Text, Text> {
        private JobConf conf;
        private int numVertices;
```

```
        /**
         * Configures this mapping object by initializing the job
         * configuration and the keyword hash set.
         *
         * @param conf the job configuration
         */
        public void configure(JobConf conf) {
            this.conf = conf;
            this.numVertices =
Integer.parseInt(conf.get("n").toString());
        }

        /**
         * Maps the inverted indexing operation to this file split.
         *
         * @param docId the document ID, not used
         * @param value the text of the document
         * @param output the output collector
         * @param r the reporter
         * @throws IOException
         */
        public void map(LongWritable docId, Text value,
OutputCollector<Text, Text> output, Reporter reporter
        ) throws IOException {
            String[] lines = value.toString().strip().split("\n");
            for (String line : lines) {
                if (!line.isBlank()) {
                    for (int i = 0; i < numVertices; i++) {
                        output.collect(new Text(String.valueOf(i)),
new Text(line));
                    }
                }
            }
        }
    }

    /**
     * Implements the reduction operation for this program.
     */
    public static class Reduce
            extends MapReduceBase
            implements Reducer<Text, Text, Text, Text> {
        private JobConf conf;
        private int numVertices;
```

```
        /**
         * Configures this mapping object by initializing the job
         * configuration and the keyword hash set.
         *
         * @param conf the job configuration
         */
        public void configure(JobConf conf) {
            this.conf = conf;
            this.numVertices =
Integer.parseInt(conf.get("n").toString());
        }

        /**
         * Collects each filename and count for key, sums the counts,
         * and collects the key with a list of the filenames and
counts.
         *
         * @param key
         * @param values
         * @param output
         * @param r
         * @throws IOException
         */
        public void reduce(Text key, Iterator<Text> values,
OutputCollector<Text, Text> output, Reporter reporter
        ) throws IOException {
            int candidate = Integer.parseInt(key.toString());
            // Create adjacency list
            int[][] edges = new int[numVertices][];
            while (values.hasNext()) {
                String value[] =
values.next().toString().strip().split(":");
                int vertex = Integer.parseInt(value[0]);
                String[] neighbors = value[1].split(",");
                edges[vertex] = new int[neighbors.length];
                for (int i = 0; i < neighbors.length; i++) {
                    edges[vertex][i] = Integer.parseInt(neighbors[i]);
                }
            }

            // Do breadth-first-search
            Deque<Integer> queue = new ArrayDeque<>();
            Set<Integer> visited = new HashSet<>();
```

```
            int start = (candidate + 1) % numVertices;
            queue.addLast(start);
            visited.add(start);
            while (!queue.isEmpty()) {
                int vertex = queue.removeFirst();
                for (int neighbor : edges[vertex]) {
                    if (neighbor != candidate &&
!visited.contains(neighbor)) {
                        visited.add(neighbor);
                        queue.addLast(neighbor);
                    }
                }
            }

            if (visited.size() != numVertices - 1) {
                reporter.incrCounter(Counter.COUNT, 1);
            }
        }
    }
}
```

Listing 4: Spark Implementation

```
// Spark Dependencies
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;

import scala.Tuple2;

// General Dependencies
import java.util.Arrays;
import java.util.Iterator;
import java.util.Map;

import java.util.Queue;
import java.util.Deque;
import java.util.ArrayDeque;

import java.util.Set;
import java.util.HashSet;
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class SparkArticulation {

    public static void main(String[] args) {

        // Get input file
        String inputFile = args[0];

        // Configure Spark
        SparkConf sparkConf = new SparkConf().setAppName("AR -
Articulation Points");
        // Start a Java Spark Context (JSC)
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // Obtain information from input file, and put it as a String
type value to Java Resilient Distributed Dataset (RDD)
        JavaRDD <String> lines = jsc.textFile(inputFile).cache();
        lines.count();

        JavaPairRDD <Integer, List <Integer> > graph =
lines.zipWithIndex().mapToPair(tuple -> {
            // where 1st element in tuple is line of type String
            // 2nd element is its index

            // split string to integers
            // put each integer into the list inside of new Tuple2,
and tuple._2() as index
            // return new Tuple2
            List <Integer> edges = stringToIntList(tuple._1());
            return new Tuple2 <> (tuple._2().intValue(), edges);
        }).repartition(10);

        int graphLength = graph.countByKey().size();

        int [][] graphAsArray = new int [graphLength][graphLength];

        for(Tuple2 <Integer, List <Integer> > tuple : graph.collect())
{
            for(int i = 0; i < graphLength; i++) {
                graphAsArray[tuple._1()][i] = tuple._2().get(i);
            }
        }
```

```
        // Start a timer
        long startTime = System.currentTimeMillis();

        // Find articulation points using BFS
        List <Integer> articulationPoints = articulationPoints(graph,
graphAsArray, graphLength);

        System.out.println("++++++++++++++++++++++++++++++++++++
Articulation Points: " + articulationPoints);


        // End a timer
        long endTime = System.currentTimeMillis();
        // Overall elapsed time
        long elapsedTime = endTime - startTime;

        System.out.println("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Elapsed Time: " +  elapsedTime);

        jsc.stop();
    }

    // Return articulation points as a list
    public static List <Integer> articulationPoints(JavaPairRDD
<Integer, List <Integer> > graph, int [][] graphAsArray, int
graphLength) {
        // Call bfs function for each target vertex we are calling
while mapping
        // Here, visitedNetwork is represented by a pair, where the
key is vertexID, and values are IDs of all visited nodes when this
vertex was removed
        //JavaPairRDD <Integer, Boolean> visitedNetwork =
        return graph.mapToPair(targetVertex -> {
            return new Tuple2 <> (targetVertex._1(), bfs(graphAsArray,
graphLength, targetVertex._1()));
        }).filter(vertexAndVisited ->
vertexAndVisited._2()).keys().collect();
    }

    public static boolean bfs(int [][] graphAsArray, int graphLength,
Integer startVertex) {
        Deque <Integer> queue = new ArrayDeque <> ();
        Set <Integer> visited = new HashSet <> ();
```

```
        queue.addLast((startVertex + 1) % graphLength);
        visited.add((startVertex + 1) % graphLength);

        while (!queue.isEmpty()) {
            int vertex = queue.removeFirst();
            for (int neighbor = 0; neighbor < graphLength; neighbor++)
{
                if (neighbor != startVertex &&
graphAsArray[vertex][neighbor] > 0 && !visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.addLast(neighbor);
                }
            }
        }

        return visited.size() != graphLength - 1 ? true : false;
    }

    public static List <Integer> stringToIntList(String input) {
        String[] strArray = input.split(" ");
        List <Integer> intList = new ArrayList <Integer> ();
        for(int i = 0; i < strArray.length; i++) {
            intList.add(Integer.parseInt(strArray[i]));
        }
        return intList;
    }

}
```

Listing 5: MASS Implementation: MassArticulation

```
import java.util.Date;
import java.util.*;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

import edu.uw.bothell.css.dsl.MASS.Agents;
import edu.uw.bothell.css.dsl.MASS.Agent;
import edu.uw.bothell.css.dsl.MASS.MASS;
import edu.uw.bothell.css.dsl.MASS.Places;
import edu.uw.bothell.css.dsl.MASS.logging.LogLevel;
```

```
/**
 * @author Akbarbek Rakhmatullaev
 * @author Logan Choi
 * @author Ryan Decker
 */
public class MassArticulation {
    private static final String NODE_FILE =
"/home/NETID/rjdecker/mass_articulation/MassArticulation/nodes.xml";

    private static int GRAPH_SIZE = 4000;
    public static final String INPUT_DIR =
"/home/NETID/rjdecker/mass_articulation/MassArticulation/src/main/java
/edu/uwb/rjdecker/";

    @SuppressWarnings("unused")       // some unused variables left
behind for easy debugging
    public static void main(String[] args) throws IOException {
        long startTime = new Date().getTime();
        System.out.println("Before init");

        // Populate the global graph
        GraphInjector.populateEdges(GRAPH_SIZE, INPUT_DIR + "graph" +
GRAPH_SIZE + ".txt");

        // Initalize the MASS library
        MASS.setNodeFilePath(NODE_FILE);
        MASS.setLoggingLevel(LogLevel.DEBUG);

        // Start MASS
        MASS.getLogger().debug("MassArticulation initializing MASS
library...");
        MASS.init();

        MASS.getLogger().debug("MassArticulation creating Places...");
        Places places = new Places(1, SimplePlace.class.getName(),
null, 1);
        MASS.getLogger().debug("Places created");

        // create Agents (number of Agents = x * y in this case), in
Places
        MASS.getLogger().debug("Quickstart creating Agents...");
        Agents agents = new Agents(1,
ArticulationAgent.class.getName(), null, places, GRAPH_SIZE);
```

```
        MASS.getLogger().debug("Agents created");

        // Call all
        int articulationPoints = 0;
        Object[] results = (Object[]) agents.callAll(0, null);
        System.out.print("Articulation points: ");
        for (int i = 0; i < GRAPH_SIZE; i++) {
            if ((int) results[i] >= 0) {
                articulationPoints++;
                System.out.print(results[i] + " ");
            }
        }
        System.out.println();
        System.out.println("Total articulation points: " +
articulationPoints);

        MASS.getLogger().debug("MassArticulation is instruction MASS
to finish operations...");
        MASS.finish();
        MASS.getLogger().debug("MASS has finished.");

        long execTime = new Date().getTime() - startTime;
        System.out.println("Execution time = " + execTime + "
milliseconds");
    }
}
```

Listing 6: MASS Implementation: ArticulationAgent

```
import java.util.Deque;
import java.util.ArrayDeque;
import java.util.Set;
import java.util.HashSet;
import edu.uw.bothell.css.dsl.MASS.Agent;

@SuppressWarnings("serial")
public class ArticulationAgent extends Agent {
    /**
     * This constructor will be called upon instantiation by MASS
     * The Object supplied MAY be the same object supplied when Places
was created
     * @param obj
     */
    public ArticulationAgent(Object obj) {}
```

```
    /**
     * This method is called when "callAll" is invoked from the master
node
     */
    public Object callMethod(int method, Object object) {
        return bfs();
    }

    /**
     *
     * @return
     */
    public Object bfs() {
        int target = getAgentId();
        Deque<Integer> queue = new ArrayDeque<>();
        Set<Integer> visited = new HashSet<>();
        queue.addLast((target + 1) % GraphInjector.edges.length);
        visited.add((target + 1) % GraphInjector.edges.length);

        while (!queue.isEmpty()) {
            int vertex = queue.removeFirst();
            for (int neighbor : GraphInjector.edges[vertex]) {
                if (neighbor != target && !visited.contains(neighbor))
{
                    visited.add(neighbor);
                    queue.addLast(neighbor);
                }
            }
        }
        if (visited.size() != GraphInjector.edges.length - 1) {
            return getAgentId();
        }
        return -1;
    }
}
```

Listing 6: MASS Implementation: GraphInjector

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class GraphInjector {
```

```
    public static int[][] edges;

    public static void populateEdges(int graphSize, String inputPath)
throws IOException {
        Scanner fileReader = new Scanner(new File(inputPath));
        edges = new int[graphSize][];
        for (int i = 0; i < graphSize; i++) {
            String[] tokens = fileReader.nextLine().split(":");
            int vertex = Integer.parseInt(tokens[0]);
            String[] neighbors = tokens[1].split(",");
            edges[vertex] = new int[neighbors.length];
            for (int j = 0; j < neighbors.length; j++) {
                edges[vertex][j] = Integer.parseInt(neighbors[j]);
            }
        }
    }
}
```

Listing 6: MASS Implementation: SimplePlace

```
import edu.uw.bothell.css.dsl.MASS.Place;

public class SimplePlace extends Place {
    /**
     * This constructor will be called upon instantiation by MASS
     * The Object supplied MAY be the same object supplied when Places
was created
     * @param obj
     */
    public SimplePlace(Object object) {}
}
```

Appendix B: Execution Outputs

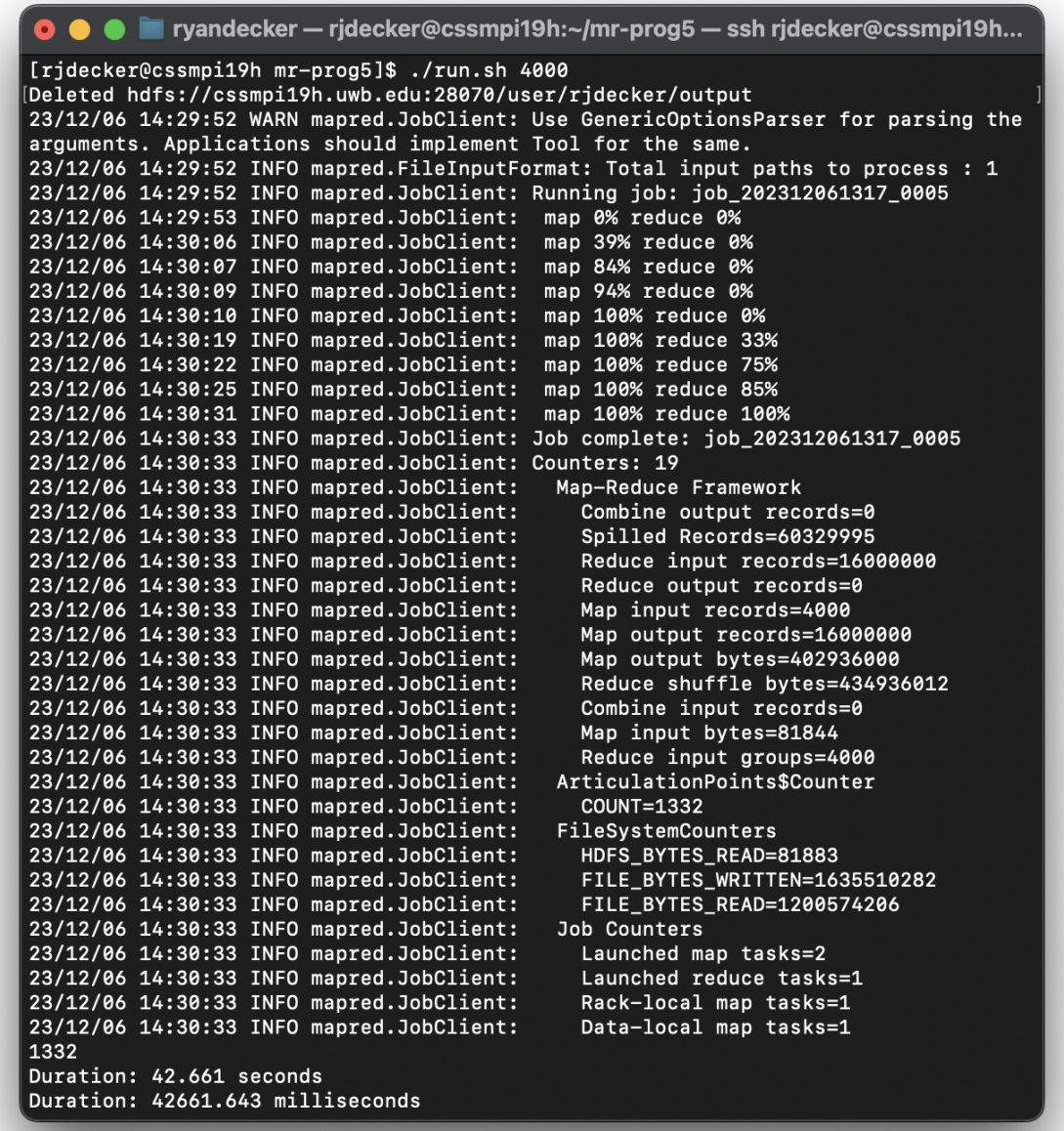
The listings below show an execution snapshot for each parallel implementation with 4000 vertices.

Listing 7: mpiJava Execution Snapshot

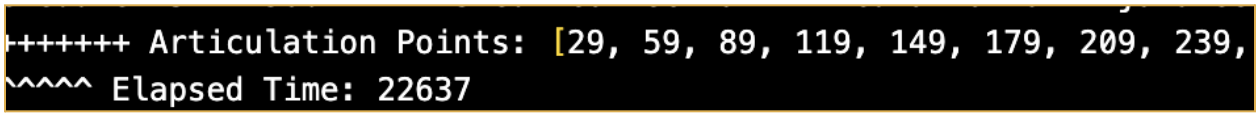
```
Rank 2 – Articulation Point: 2554
Rank 3 – Articulation Point: 3569
Rank 0 – Articulation Point: 664
Rank 1 – Articulation Point: 1679
Rank 2 – Articulation Point: 2589
Rank 3 – Articulation Point: 3604
Rank 0 – Articulation Point: 699
Rank 1 – Articulation Point: 1714
Rank 2 – Articulation Point: 2624
Rank 3 – Articulation Point: 3639
Rank 0 – Articulation Point: 734
Rank 1 – Articulation Point: 1749
Rank 2 – Articulation Point: 2659
Rank 3 – Articulation Point: 3674
Rank 0 – Articulation Point: 769
Rank 1 – Articulation Point: 1784
Rank 2 – Articulation Point: 2694
Rank 0 – Articulation Point: 804
Rank 2 – Articulation Point: 2729
Rank 3 – Articulation Point: 3709
Rank 0 – Articulation Point: 839
Rank 1 – Articulation Point: 1819
Rank 0 – Articulation Point: 874
Rank 2 – Articulation Point: 2764
Rank 3 – Articulation Point: 3744
Rank 1 – Articulation Point: 1854
Rank 1 – Articulation Point: 1889
Rank 3 – Articulation Point: 3779
Rank 0 – Articulation Point: 909
Rank 2 – Articulation Point: 2799
Rank 1 – Articulation Point: 1924
Rank 3 – Articulation Point: 3814
Rank 0 – Articulation Point: 944
Rank 2 – Articulation Point: 2834
Rank 1 – Articulation Point: 1959
Rank 0 – Articulation Point: 979
Rank 3 – Articulation Point: 3849
Rank 2 – Articulation Point: 2869
Rank 1 – Articulation Point: 1994
Rank 3 – Articulation Point: 3884
Rank 2 – Articulation Point: 2904
Rank 3 – Articulation Point: 3919
Rank 2 – Articulation Point: 2939
Rank 3 – Articulation Point: 3954
Rank 2 – Articulation Point: 2974
Rank 3 – Articulation Point: 3989
Rank 3 – Articulation Point: 3997

Total Articulation Points: 115
Total Time: 48207 milliseconds
```

Listing 8: MapReduce Execution Snapshot



Listing 9: Spark Execution Snapshot



Listing 10: MASS Execution Snapshot

```
[rjdecker@cssmpi19h MassArticulation]$ java -jar
target/mass-articulation-1.0.0-SNAPSHOT.jar
Before init
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller
(file:/home/NETID/rjdecker/mass_quickstart/Quickstart/target/mass-arti
culation-1.0.0-SNAPSHOT.jar) to field
sun.nio.ch.SelectorImpl.selectedKeys
WARNING: Please consider reporting this to the maintainers of
org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further
illegal reflective access operations
WARNING: All illegal access operations will be denied in a future
release
MASS.init: done
...
Total points: 1332
MASS Shutting Down...
MASS Shutdown Finished
Execution time = 2188 milliseconds
```