CSS 600 Summary Report

Collin Gordon June 9, 2017

1.Introduction

The goal for my CSS 600 was to program three machine learning algorithms using the MASS library, assess their programmability, and devise new features to add to MASS to improve programmability and assess performance of the machine learning algorithms compared to their implementations using other parallelization techniques such as MapReduce. These additions and their improvements to the machine learning algorithms presented below will be the subject of my CSS 595 Capstone Project.

2.Machine Learning Algorithms

2.1 K Means Clustering

Algorithm Description

K Means Clustering is a method of clustering unlabeled data. It works taking each data point and sorting it into one of k number of clusters. Each cluster has a centroid which is an arbitrary point based on the mean of all the numerical elements of each data point in the cluster. The general outline is as follows:

- Position k number of centroids at arbitrary points among the data points.
- Each data point near the centroid becomes part of that centroid's cluster.
- The centroids are recomputed based on the resulting mean of all points in the cluster.
- Repeat steps 2 and 3 until either the centroids no longer move or the amount of specified iterations has been reached.

Occasionally the clusters made by the algorithm are very close together. In this case, post processing is applied to the clusters to determine whether clusters can be merged or not. MASS has potential to speed up several parts of the algorithm including the movement of the centroids and the post processing. However, the most advantageous part of MASS is the sheer number of data points that can be potentially compared by using places on multiple nodes and numerous agents moving between them.

MASS Implementation

The MASS implementation of K Means Clustering uses Agents as data points and Places as centroids of clusters. Each iteration of the algorithm starts with Agents checking their respective Places and deciding whether they stay with their current cluster or move to a new cluster. Once they decide whether they stay or go, each agent prepares to move to a random centroid. The next step is to move the Agents to their new place. Finally, the Places recalculate their mean based on the Agents that surround them. At the end of the algorithm agents report whether they found a cluster and each place reports its mean. Both elements log their data using the J42Logger.

This algorithm was first implemented in CSS 534 on the MASS C++ version where it was revealed to have several issues with Agents moving smoothly to each place. I ported it to the Java version upon learning

that the goal for MASS Java is to excel at big data analysis which will include machine learning methods eventually. The code for the algorithm is below.

Centroid.java

```
package edu.uw.bothell.css.dsl.MASS.MassKmeans;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;
public class Centroid extends Place {
    public static final int init_    = 0;
    public static final int update_  = 1;
    public static final int findv_   = 2;
    public static final int getNext_ = 3;
    public static final int rept_    = 4;

    private int mean;
    private int csize;
    private int[] nextMove;
    private int arrX, arrY, localX, localY; // array dimensions and local
dimensions
    private Log4J2Logger logger;

    public Centroid() {
        super();
        logger = Log4J2Logger.getInstance();
        nextMove = new int[2];
        Vector<int[]> neighbors = new Vector<>();
        neighbors.add(new int[] {0,-1});
        neighbors.add(new int[] {1,0});
        neighbors.add(new int[] {0,1});
        neighbors.add(new int[] {-1,0});
        setNeighbors(neighbors);
    }

    public Centroid(Object obj) {
        super();
        logger = Log4J2Logger.getInstance();
        nextMove = new int[2];
        Vector<int[]> neighbors = new Vector<>();
        neighbors.add(new int[] {0,-1});
        neighbors.add(new int[] {1,0});
        neighbors.add(new int[] {0,1});
        neighbors.add(new int[] {-1,0});
        setNeighbors(neighbors);
    }

    public Object callMethod(int functionId, Object args) {
        switch(functionId) {
        case init_:
            return init();
        case update_:
            return update();
        case findv_:
```

```
            return findValid();
        case getNext_:
            return sendNextMove();
        }
        return null;
    }

    // initializes essential data for the Centroid
    private Object init() {
        arrX = getSize()[0];
        arrY = getSize()[1];       // places array size
        localX = getIndex()[0];
        localY = getIndex()[1]; // my coordinates
        mean = (int)Math.random() % 20 + 1;
        csize = 0;
        return null;
    }

    //returns mean to point
    public int getMean() {
        return mean;
    }

    // randomly selects a neighbor to move unlocked agents to
    private Object findValid() {
        nextMove[0] = (int)Math.random() % arrX;
        nextMove[1] = (int)Math.random() % arrY;
        return null;
    }
    public Object report() {
        logger.debug("I have " + getAgents().size() + " points around
me");
        logger.debug("My mean is " + mean  + '.');
        return null;
    }
    //sends next move to unlocked points
    public int[] sendNextMove() {
        return nextMove;
    }

    // updates mean of centroids
    private Object update() {
        int sum = 0;
        int total = getAgents().size();

        // loop logic to sum up weights of agents and compute new mean
        if(total != 0) {
            Set<Agent> points = getAgents();
            Iterator iter = points.iterator();
            while(iter.hasNext()) {
                sum += ((Point)iter.next()).getWeight();
            }
            mean = sum / total;
        }
```

```
        return null;
    }

}
```

Point.java

```
package edu.uw.bothell.css.dsl.MASS.MassKmeans;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;
public class Point extends Agent {
    public static final int set_    = 0;
    public static final int move_   = 1;
    public static final int weight_ = 2;
    public static final int dist_   = 3;
    public static final int rept_   = 4;

    private Vector<int[]> neighbors = new Vector<>();
    private int weight;
    private boolean lock;
    private int[] movement;
    private Log4J2Logger logger;

    public Point() {
        super();
        logger = Log4J2Logger.getInstance();
    }

    public Point(Object obj) {
        super();
        logger = Log4J2Logger.getInstance();
        movement = new int[2];
        neighbors.add(new int[] {0, -1});
        neighbors.add(new int[] {1, 0});
        neighbors.add(new int[] {0, 1});
        neighbors.add(new int[] {-1, 0});
    }
    public Object callMethod(int functionId, Object args) {
        switch(functionId) {
        case set_:
            return setWeight();
        case move_:
            return move();
        case weight_:
            return getWeight();
        case dist_:
            return distance();
        case rept_:
            return report();
        }
        return null;
    }
```

```
    private Object setWeight() {
        weight = (int)Math.random() % 20 + 1;
        return null;
    }

    public int getWeight() {
        return weight;
    }

    private Object distance() {
        int m = ((Centroid)getPlace()).getMean();
        int dist = (int)Math.abs(weight - m);
        if(dist <= m) {
            lock = true;
        } else {
            lock = false;
        }
        return null;
    }

    private Object move() {
        if(!lock) {
            movement = ((Centroid)getPlace()).sendNextMove();
            migrate(movement[0], movement[1]);
        }
        return null;
    }

    private Object report() {
        String locked = lock? "locked" : "unlocked";
        logger.debug("I am " + locked);
        return null;
    }
}
```

KMeans.java

```
package edu.uw.bothell.css.dsl.MASS.MassKmeans;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.*;
import java.util.*;
public class KMeans {
    public static void main(String args[]) {
        if(args.length < 3) {
            System.err.println("usage: java -jar <appname> sizeX sizeY
nIter");
            System.exit(-1);
        }

        int sizeX        = Integer.parseInt( args[0] );
        int sizeY        = Integer.parseInt(args[1]);
        int nIter       = Integer.parseInt( args[2] );
```

```
        int nAgents     = 8; // arbitrary number for now



        MASS.init( );
        MASS.getLogger().setLogLevel(LogLevel.DEBUG);
        Places clusters = new Places(1, Centroid.class.getName(), null,
sizeX, sizeY);
        Agents points   = new Agents(2, Point.class.getName(), null,
clusters, nAgents);
        Vector<int[]> neighbors = new Vector<>();
        int[] north = { 0, -1 };
        neighbors.add( north );
        int[] east  = { 1,  0 };
        neighbors.add( east );
        int[] south = { 0,  1 };
        neighbors.add( south );
        int[] west  = { -1, 0 };
        neighbors.add( west );

        points.callAll(Point.set_);
        clusters.callAll(Centroid.init_);
        points.callAll(Point.rept_);
        clusters.callAll(Centroid.rept_);
        for(int i = 0; i < nIter; i++) {
            points.callAll(Point.dist_);
            clusters.callAll(Centroid.update_);
            clusters.exchangeAll(1, Centroid.update_, neighbors);
            clusters.callAll(Centroid.findv_);
            points.callAll(Point.move_);
            points.manageAll();
            points.callAll(Point.rept_);
            clusters.callAll(Centroid.rept_);
        }
        MASS.finish();
    }
}
```

Results

The Java version of this algorithm runs much like the first C++ version. If the number of Agents exceeds the number of Places, then the Agents deadlock. This deadlock occurs because the current implementation of MASS runs under the assumption that each place will only have one agent try to access it at a time. While many of the simulations adapted to MASS have fit this model, K Means Clustering does not. When there are more agents than places, agents attempt to access the same place at the same time. The solution for this problem is to implement Collision Free Migration as it is implemented in MASS C++.

The Collision Free Migration implementation uses each Place's in and out message structure to communicate whether it is available to receive agents or not. This creates pseudo scheduler that helps Agents determine the order in which they can access a Place. Bringing this feature to MASS Java will be one of the focus areas of my CSS 595 work. My work on collision free migration will have to extend

beyond the implementation used in MASS C++. When working with a large number of Agents, this algorithm could become very inefficient. This is because the scheduler still only moves one Agent at a time. Thousands of Agents would greatly decrease the performance of the simulation as well as potentially trigger the Java Garbage Collector.

If a clustering involved several data points, a wise solution would be to take advantage of Utku's Agent Population Control research. His research led to the creation of a limit to how many agents can be active in the Places-space at once. It also serialized and stored Agents that were not being used in order to maximize performance. Agents could be created for each data point and clustered in stages. When an Agent becomes locked it can be serialized and the next Agents could be de-serialized and move as needed until locked. The algorithm would continue alternating the Agents in stages until no Agents moved. Determining the new mean for each cluster would be a simple matter of passing the data points from the Agents to their respective Places. After the last set of Agents is clustered, the Places would read from all the data points that checked in and recalculate their mean. As each set of Agents is de-serialized, they would re-evaluate based on the new mean. Once none of the Agents changed, the algorithm would end.

Overall, this K Means implementation does not take full advantage of the utilities of MASS and therefore will likely not produce a performance improvement over another parallelization method such as MapReduce. A more efficient, less complex algorithm will be developed alongside Agent to Agent direct communication in my CSS 595 work.

2.2 K Nearest Neighbor Classification

Algorithm Description

K Nearest Neighbor is a classifying algorithm where training data is used to classify unclassified testing data by comparing the testing data to a k number of training points. This algorithm normally must be run multiple times to generate an accurate classification. When it comes to running the algorithm multiple times, programmers often employ a majority vote rule or an averaging equation to determine the most accurate classification.

MASS Implementation

Places represent the training data that will be used for classification and the Agents represent data points to be classified. The program performs one pass where Agents are distributed randomly among the Places. Each Place calls exchangeAll twice to store and exchange classifications from far away neighbors. The Agents then request the vector of classifications from its current place and assigns a label to itself. Currently, the labels are Boolean values of true and false. The last step is that each agent logs its classification using the J42Logger.

Training.java

```
package edu.uw.bothell.css.dsl.MASS.MassKnn;
import edu.uw.bothell.css.dsl.MASS.*;
import java.util.*;

public class Training extends Place {
    public static final int gather_    = 0;
    public static final int broadcast_ = 1;
    //public static final int report_    = 2;
```

```
public static final int nLabels_   = 3;

private Vector<Boolean> labels;
private Boolean label;

public Training() {
    super();
    labels = new Vector<>();
    label  = (Math.random() % 2) == 0? true : false;
    Vector<int[]> neighbors = new Vector<>();
    neighbors.add(new int[] {0,-1});
    neighbors.add(new int[] {1,0});
    neighbors.add(new int[] {0,1});
    neighbors.add(new int[] {-1,0});
    setNeighbors(neighbors);
}
public Training(Object args) {
    super();
    labels = new Vector<>();
    label  = (Math.random() % 2) == 0? true : false;
    Vector<int[]> neighbors = new Vector<>();
    neighbors.add(new int[] {0,-1});
    neighbors.add(new int[] {1,0});
    neighbors.add(new int[] {0,1});
    neighbors.add(new int[] {-1,0});
    setNeighbors(neighbors);
}

public Object callMethod(int functionId, Object args) {
    switch(functionId) {
    case gather_ :
        return gather();
    case broadcast_:
        return (Object) broadcast();
    //case report_: return report();
    case nLabels_:
        return (Object) nLabels();
    }
    return null;
}

public Object gather() {
    if(getInMessages() != null) {
        // getting the labels out of inMessages
        for(int i = 0; i < 4; i++) {
            labels.add((Boolean)getInMessages()[i]);
        }
    }
    return null;
}

public Vector<Boolean> nLabels() {
    return labels;
}
```

```
    public Boolean broadcast() {
        return label;
    }
}
```

Test.java

```
package edu.uw.bothell.css.dsl.MASS.MassKnn;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;

public class Test extends Agent {
    public static final int nNeighbors_ = 0;
    public static final int classify_ = 1;
    public static final int report_   = 2;
    public static final int gather_   = 3;
    private Vector<Boolean> nearest;
    private boolean label;
    private int k; // representing the k nearest neighbors we want to
check
    private Log4J2Logger logger;

    public Test() {
        super();
        nearest = new Vector<>();
        logger = Log4J2Logger.getInstance();
        k = 3;
    }

    public Test(Object args) {
        super();
        nearest = new Vector<>();
        logger = Log4J2Logger.getInstance();
        k = 3;
    }

    public Object callMethod(int functionId, Object args) {
        switch(functionId) {
        case nNeighbors_ :
            return kNeighbors(args);
        case classify_:
            return classify();
        case report_:
            return report();
        case gather_:
            return gather();
        }
        return null;
    }

    // overload map method
```

```
    // public int map(int initPopulation, int[] size, int[] index){

    // }
    public int map( int maxAgents, int[] size, int[] coordinates ) {
        int sizeX = size[0], sizeY = size[1];
        int populationPerCell = (int)Math.ceil( maxAgents / ( sizeX *
sizeY * 0.6 ) );
        int currX = coordinates[0], currY = coordinates[1];
        if ( sizeX * 0.4 < currX && currX < sizeX * 0.6 && sizeY * 0.4 <
currY && currY < sizeY * 0.6 ) {
            //System.err.println("mapping max agents " + maxAgents + "
size: " + size[0] + " population per cell: " + populationPerCell);
            return populationPerCell;
        } else
            return 0;
    }

    // This function will allow the user to change the number of training
data compared
    public Object kNeighbors(Object num) {
        k = ((Integer)num).intValue();
        return null;
    }

    public Object classify() {
        int typeA = 0;
        int typeB = 0;
        if(k != 0) {
            for(int i = 0; i < k; i++) {
                boolean l = nearest.elementAt(i).booleanValue();
                if(l) {
                    typeA++;
                } else {
                    typeB++;
                }
            }
        }
        if(typeA == typeB) {
            logger.error("There's a tie between classifications!!");

        } else if(typeA > typeB) {
            label = true;
        } else {
            label = false;
        }
        return null;
    }

    public Object report() {
        logger.debug("I am Testing and I am" + label);
        return null;
    }

    public Object gather() {
```

```
        nearest = ((Training)getPlace()).nLabels();
        return null;
    }
}
```

KNN.java

```
package edu.uw.bothell.css.dsl.MASS.MassKnn;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.*;
import java.util.*;


public class Knn
{
    public static void main( String[] args )
    {
        if(args.length < 2) {
            System.err.println("usage java -jar <appname> size nAgents");
            System.exit(-1);
        }

        int boundaryWidth = 2;
        int size        = Integer.parseInt(args[0]);
        int nAgents     = Integer.parseInt(args[1]);



        MASS.init( );
        MASS.getLogger().setLogLevel(LogLevel.DEBUG);
        Places training = new Places(1, Training.class.getName(),
boundaryWidth, null, size, size);
        Agents testing   = new Agents(2, Test.class.getName(), null,
training, nAgents);
        testing.callAll(Test.nNeighbors_, 4);

        Vector<int[]> neighbors = new Vector<>();
        int[] north = { 0, -1 };
        neighbors.add( north );
        int[] east  = { 1,  0 };
        neighbors.add( east );
        int[] south = { 0,  1 };
        neighbors.add( south );
        int[] west  = { -1, 0 };
        neighbors.add( west );

        // exchanging data between places
        training.exchangeAll(1, Training.broadcast_, neighbors);
        training.exchangeAll(1, Training.broadcast_, neighbors);

        // packaging the labels to send to the testing data
        training.callAll(Training.gather_);
```

```
        // gathering the training labels and classifying
        testing.callAll(Test.gather_);
        testing.callAll(Test.classify_);
        testing.callAll(Test.report_);

        MASS.finish();
    }
}
```

Results

This algorithm works well in MASS as is. Further work that needs to be done on it revolves around brining in real data for training and testing. Using real data may eliminate agents as places could read in both training and testing data to classify and write the results to a file using parallel file I/O. However, reading testing data into the Agents could be as simple as having an outer loop that extends from the Agents initialization to after the callAll for report. Each new instantiation of Agents would receive some data to parsed to the Agents as an argument rather than the null parameter for arguments to Agents. Additionally, splitting the different data between Agents and Places provides several advantages.

First, having Places read, write, and store both sets of data would slow down performance by forcing unnecessary read and write operations. As was demonstrated in CSS 534, performant code typically avoids reads and writes during the computation portion of the program because of the overhead behind the function calls. Reads and writes are most often done before and after computation in order to reduce this overhead. Another benefit is storage. Part of my CSS 595 capstone will be generating a way to take a snapshot of the simulation. Such a snap shot could be used to capture data inside Places with the intent to reuse the data on later testing sets. Agents could then be run over the saved Places using the different testing sets and the data on the Places could be preserved without the concern for data corruption.

Also, as Utku's research pointed out, MASS Java has experienced problems with the sizing of objects. Although he provided future developers with a way to control Agent population size, there is still the unknown of the maximal Place size. If Places were used on their own for both the training and testing sets, then it is possible that this size could be reached and the Java Garbage Collector could destroy objects that are needed for an accurate classification.

2.3 Gradient Descent

Algorithm Description

Gradient Descent is an algorithm that attempts to find the actual minimum of a dataset. This algorithm is important in training neural networks and other machine learning models because it attempts to minimize the error in the calculations the model makes. The algorithm works by calculating the gradient of the training function at each data point in the set and moving through the dataset based on that gradient until it finds the lowest error in the data set. This lowest point is then used as a reference to build the model for the other data points in the set. Gradient Descent can produce a false minimum either by using too large of a gradient or too small of a gradient. If the gradient is too large, then the calculation may miss the actual minimum and if it is too small, the algorithm may take too long to find the actual minimum. Many researchers currently employ Stochastic Gradient Descent to reduce the chance of a false minimum.

Stochastic Gradient Descent takes the gradient descent from several starting points in the data and hopes to find a convergence or agreement between the different starting points. This algorithm relies on using small gradients and limiting the number of iterations in the calculation. Researchers improve the accuracy of this model by adding in the concept of momentum where as a calculation approaches the minimum it gains speed. The calculation will eventually move away from the minimum, but as it does so, it will lose momentum and settle on the lowest point. This addition is necessary because parts of the calculation will settle in false minimums and hinder the chance of converging on the actual minimum. The negative of Stochastic Gradient Descent is that it is slow leading to it commonly being run in parallel.

MASS provides the opportunity to process potentially hundreds or thousands of data points in a variety of ways. This is achieved by the manipulation of the map function in the Agent class. This algorithm run multiple times can accurately and quickly determine the absolute minimum of a dataset.

MASS Implementation

Agents traverse places that are given an arbitrary weight. As the agents encounter places they adjust their weight mimicking the minimum calculation in the actual algorithm. The agents only move to places that have a weight less than the agent and meaning that an agent has completed its run when all weights around the agent are greater than the weight of the agent. Once the agent stops moving, it logs its minimum using the J42Logger.

Grid.java

```
package edu.uw.bothell.css.dsl.mass.apps.gradientDescent;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;

public class Grid extends Place {
    private int minimum;
    private Log4J2Logger logger;
    private Vector<int[]> neighbors;
    public static final int init_ = 0;
    public static final int getM_ = 1;
    public static final int getMove_ = 2;
    public static final int rept_ = 3;
    public Grid() {
        super();
        logger = Log4J2Logger.getInstance();
        neighbors = new Vector<>();
        neighbors.add(new int[] {0,-1});
        neighbors.add(new int[] {1,0});
        neighbors.add(new int[] {0,1});
        neighbors.add(new int[] {-1,0});
        setNeighbors(neighbors);
    }

    public Grid(Object args) {
        super();
        logger = Log4J2Logger.getInstance();
        Vector<int[]> neighbors = new Vector<>();
```

```
    neighbors.add(new int[] {0,-1});
    neighbors.add(new int[] {1,0});
    neighbors.add(new int[] {0,1});
    neighbors.add(new int[] {-1,0});
    setNeighbors(neighbors);
}

public Object callMethod(Object args, int funcId) {
    switch(funcId) {
    case init_:
        return init();
    case getM_:
        return getMinimum();
    case getMove_ :
        return getNextMove();
    case rept_:
        return report();
    }
    return null;
}

public Object init() {
    minimum = (int)(Math.random() % 20) + 1;
    if(Math.random() % 2 == 1) {
        minimum *= -1;
    }
    return null;
}

public int getMinimum() {
    return minimum;
}
public int[] getNextMove() {
  int index = -1;
  int returnVal[] = new int[2];
  returnVal[0] = -999;
  returnVal[1] = -999;
  if(getInMessages() != null){
    for(int i = 0; i < 4; i++){
      int comp = ((Integer)getInMessages()[i]);
      if(comp < minimum){
        index = i;
      }
    }
  }
  if(index >= 0){
    returnVal = neighbors.get(index);
  }
  return returnVal;
}
public Object report() {
    logger.debug("I am a place and my minimum is: " + minimum);
    return null;
}
```

```
}
```

Seeker.java

```
package edu.uw.bothell.css.dsl.mass.apps.gradientDescent;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;


public class Seeker extends Agent {
    public static final int min_  = 0;
    public static final int move_ = 1;
    public static final int rept_ = 2;
    private Log4J2Logger logger;
    private int min; // holds the previous place minimum
    private boolean minSet; // marks the first read of a place

    public Seeker() {
        super();
        logger = Log4J2Logger.getInstance();
        minSet = false;
    }

    public Seeker(Object args) {
        super();
        logger = Log4J2Logger.getInstance();
        minSet = false;
    }

    public Object callMethod(Object args, int funcId) {
        switch(funcId) {
        case min_ :
            return minimum();
        case move_:
            return move();
        case rept_:
            return report();
        }
        return null;
    }
    // overload map method
    // public int map(int initPopulation, int[] size, int[] index){

    // }
    public int map( int maxAgents, int[] size, int[] coordinates ) {
        int sizeX = size[0], sizeY = size[1];
        int populationPerCell = (int)Math.ceil( maxAgents / ( sizeX *
sizeY * 0.6 ) );
        int currX = coordinates[0], currY = coordinates[1];
        if ( sizeX * 0.4 < currX && currX < sizeX * 0.6 && sizeY * 0.4 <
currY && currY < sizeY * 0.6 ) {
```

```
            //System.err.println("mapping max agents " + maxAgents + "
size: " + size[0] + " population per cell: " + populationPerCell);
            return populationPerCell;
        } else
            return 0;
    }

    public Object minimum() {
        int placeMin = ((Grid)getPlace()).getMinimum();
        if(minSet) {
            if(placeMin < min) {
                min = placeMin;
                logger.debug("New minimum: " + min);
            }
        } else {
            min = placeMin;
            logger.debug("Starting minimum" + min);
            minSet = true;
        }
        return null;
    }

    public Object move() {
        int moveSet[] = new int[2];
        moveSet = ((Grid)getPlace()).getNextMove();
        if(moveSet[0] != -999 && moveSet[1] != -999){ //checks if there is
a next move
          migrate(moveSet[0], moveSet[1]); //move the agent
        } else {
          logger.debug("Absolute Minimum Found:" + min); //if no next
move, we found an absolute minimum candidate
        }
        return null;
    }
    public Object report() {
        logger.debug("I am an agent with a min of: " + min);
        return null;
    }
}
```

GradientDescent.java

```
package edu.uw.bothell.css.dsl.mass.apps.gradientDescent;
import edu.uw.bothell.css.dsl.MASS.*;
import edu.uw.bothell.css.dsl.MASS.logging.*;
import java.util.*;
public class GradientDescent
{
    public static void main( String[] args )
    {

        if(args.length < 2) {
```

```
            System.err.println("Wrong number of arguments...will elaborate
later");
            System.exit(-1);
        }


        int size        = Integer.parseInt( args[0] );
        int nIter       = Integer.parseInt( args[1] );
        int nAgents     = 8;
        MASS.init();
        MASS.getLogger().setLogLevel(LogLevel.DEBUG);
        Places grid = new Places(1, "Grid", null, size, size);
        Agents seekers = new Agents(2, "Seeker", null, grid, nAgents);

        Vector<int[]> neighbors = new Vector<>();
        int[] north = { 0, -1 };
        neighbors.add( north );
        int[] east  = { 1,  0 };
        neighbors.add( east );
        int[] south = { 0,  1 };
        neighbors.add( south );
        int[] west  = { -1, 0 };
        neighbors.add( west );

        // Initialize minimums and exchange boundary information
        grid.callAll(Grid.init_, null);
        grid.exchangeAll(1, Grid.getM_, neighbors);

        // move the seekers until a minimum is found
        for(int i = 0; i < nIter; i++) {
            seekers.callAll(Seeker.min_, null);
            seekers.callAll(Seeker.move_, null);
        }

        //final report
        seekers.callAll(Seeker.rept_, null);
        grid.callAll(Grid.rept_, null);

        MASS.finish();
    }
}
```

Results

Currently the difficult programmability of the map function limits Gradient Descent. Currently this program borrows the mapping from the RandomWalk application which distributes agents randomly throughout the data set. This is problematic due to the increased likelihood of two agents trying to access the same place at once resulting in a deadlock. Collision Free Migration will aid in solving this problem.

The programmability problem with the map function will be addressed in my CSS 595 work in the form of overloaded or multiple map functions. The two kinds of functionality that will improve the accuracy of

Gradient Descent are edge mapping and normalized distribution mapping. Edge mapping is closely aligned with the method used to determine the minimum in the original algorithm. Agents would spawn on the edge of the dataset and move inwards toward the minimum. Normalized distribution mapping would provide a boost as agents would be positioned towards the center of the data set. This might result in an agent landing directly on the absolute minimum. In the event of a multiple map function, the MASS algorithm could be run several times testing and recording the results of different maps on each pass. The results could then be aggregated to find the absolute minimum outside of MASS or with the MASS Node Cache.

3.Proposed MASS Additions

3.1 Collision Free Migration

As of January 2016, the C++ version of the MASS library has been outfitted with collision free agent migration thanks to the diligent work of Chris Bowzer. Collision free migration is necessary for many aspects of MASS including programs such as Sugarscape, Bail in/ Bail out, and the K Means and Gradient Descent programs discussed in section 3.

Bowzer's collision free model works by using the existing place to place communication structure to signal to agents when places are ready to receive them. When agents receive the go ahead from the places, they move one at a time to the new place. This algorithm is good for MASS because it uses the existing structure and does not disrupt the mechanics of the multithreaded communication used in the library.

The first aspect of my CSS595 project is to implement the C++ method of collision free migration into the MASS Java version. The benefits of having Bowzer's algorithm in the Java version are the success of the KMeans program and the Gradient Descent program. The successful run of these models will determine the feasibility of future agent intensive simulations. However, the implementation of the Bowzer algorithm is not the stopping point for my planned work on collision free migration.

The second phase of the collision free migration work will involve stress testing both Bowzer's algorithm and Utku Mert's agent recycling work. The object of this research is to understand the efficacy of the current methods of handling agent movement and spawning and identify limitations and areas of improvement for future research. If time permits, I will examine the use of Java 8's concurrent methods and structures to see if any immediate improvements can be made to the collision free migration algorithm.

3.2 Multiple Maps

Agent mapping in the current implementation of MASS comes in two forms: A map where agents are added row by row to places or a user overridden map with different behavior. One of the current pain points in MASS programmability is the overriding of the map method. Such an override requires users to understand how MASS behaves under the hood and complicates programming rather than simplifying it. Additionally, many simulations and data science applications share the same mapping pattern. For instance, K Nearest Neighbor and Random Walk need to use a random distribution of agents to be effective. The need to simplify programming for the user, abstract away the MASS code base, and the sharing of similar mapping patterns are all good reasons to implement a standard set of map functions in the MASS library.

My proposition is to write map functions that provide the following distributions of agents to the user:

- Standard – This is the current default mapping of MASS agents. This mapping is useful for simulations with stationary agents such as Bail in/Bail out and Social Network
- Random – This distribution is currently implemented as an overridden map in the Random Walk application. It distributes agents in random spots throughout the places-space. It is useful for several agent-based applications such as K Means, K Nearest Neighbor, Sugarscape, and Gradient Descent
- Normalized – This will map agents in a normalized distribution with an emphasis on grouping agents towards the center of the places-space with movement outward towards the edge of the space.
- Edge – This distribution is effective for applications such as Gradient Descent where agents need to be distributed around the edge of the places-space and move inward.

Each of these agent distributions will be accessible via an enumerated set of keywords that are passed as an argument to a wrapper map function. The goal of redesigning the map function in this way is to make it easy for users to have a wide variety of common map functions while providing a way for those who wish to hack the library to add a keyword and map function of their own.

3.3 Communication

An ongoing problem in MASS is the need for agent to agent direct communication. This communication can help improve the K Means algorithm and simulations such as Bail in / Bail out. In Professor Fukuda's previous work on UWAgent, agent communication is achieved through a treelike implementation where agents are spawned from parent agents and communicate information back along the tree structure to their parent. This implementation is very complex and would require a rework of the way agents are structured currently within MASS. Additionally, the calls needed to communicate information and track where agents are in relation to their parents would greatly decrease MASS's current performance. An alternate form of communication would be possible using messenger agents. These agents would pass data back and forth between nodes and update the agents and places on that node. Such a behavior can be further augmented using a system such as MASS Node Cache.

A portion of my CSS595 project will include attempts at implementing this communication so that agents can communicate better with themselves as well as places. This feature is of elevated importance in the scope of this project as Bowzer's collision free algorithm relies on using the current communication structure making it so that the user must choose between place and agent communication and collision free migration.

3.4 MASS Node Cache

When I first came to UW Bothell, I observed research work done by Dr. Michael Stiber on the BrainGrid neuron growth simulator. In this project, GPUs are used to simulate the growth of cortical tissue in accordance to several neuroscience-based models for brain development. Some of the observations I made while on that project have influenced my thinking on MASS. The first observation was that obtaining data on the stages of the simulations from beginning to end was very difficult and the other observation was that simulations required considerable planning beforehand due to their long runtime and if anything disrupted the simulation in progress, it would have to be restarted from the beginning. If

MASS is to eventually become a robust, go to framework for data science, it needs to have fault tolerance for large simulations. My solution to this problem is the MASS Node Cache.

3.4.2 Implementation

A version of the MASS Node Cache exists on the collin_develop branch of the mass_java_core repository on bitbucket. Since this initial attempt crossed somewhat into my CSS 595 work I decided to direct my focus to other tasks such as writing my capstone proposal and running my machine learning algorithms.

The MASS Node Cache is implemented using a ConcurrentHashMap internally to store system information on each computing node involved in the MASS application. The decision to base the system off a ConcurrentHashMap comes from the examination of various distributed caching systems for clusters based on a map structure. The keys for information storage into the map are divided in to mode and operation keywords. Mode keywords call the functions to either read from the cache, write to the cache, or take a snapshot with the cache. The operation keywords represent different types of information stored by the cache. Source code for the MASS Node Cache and enumerated keywords is below.

MASSNodeCache.java

```
/* MASSNodeCache CLASS */
// This class caches system information and takes snapshots of simulations
package edu.uw.bothell.css.dsl.MASS.Cache;
import edu.uw.bothell.css.dsl.MASS.logging.Log4J2Logger;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class MASSNodeCache {

  private ConcurrentHashMap<CacheOpKeys, Object> cacheMap;
  private Log4J2Logger logger;
    public MASSNodeCache(){
        cacheMap = new ConcurrentHashMap<>();
        logger = Log4J2Logger.getInstance();
    }

    public Object write(CacheOpKeys key, Object value){
      System.out.println("Operation " + key + " performed in write.");
       cacheMap.put(key, value);
       return null;
    }

    public Object retrieve(CacheOpKeys key){
      System.out.println("Operation " + key + " performed in retrieve.");
      return cacheMap.get(key);
    }
}
```

CacheModeKeys.java

```
package edu.uw.bothell.css.dsl.MASS.Cache;
public enum CacheModeKeys{
  WRITE,
  RETRIEVE,
  SNAPSHOT
}
```

CacheOpKeys.java

```
package edu.uw.bothell.css.dsl.MASS.Cache;

public enum CacheOpKeys {
    PLACES_BOUNDARY,
    AGENTS_BOUNDARY,
    RANK,
    NUM_NODES,
    AGENT_SIZE,
    PLACE_SIZE,
    LOCAL_IP
}
```

CSS 595

My work on the MASS Node Cache for CSS 595 will be to implement a snapshot function that saves simulation data. If time allows, I will write a method to load snapshot information and spin up the simulation where it left off. Over the summer I plan to test the functionality excluding the snapshot mode.

4.Where to find and run programs

The three machine learning algorithms mentioned in this paper can be found in the dslab account under the directory Collin_Workspace. They can be run using the following command:

```
java -jar <app name> arguments
```

Any issues with the programs or questions regarding my work can be directed to colntrev@gmail.com. As of this writing the uw1-320 cluster is experiencing intermittent issues with the UDrive. This has prevented extensive testing of the above code. Over this summer Matthew Sell will be putting together a special cluster exclusively for dslab students. This will hopefully circumvent many of the issues encountered this quarter.