ECSE 420 - Parallel Computing
Final Project - Group 10 - Allwolf

T. Jeff Wallace
jeff@tjwallace.ca

February 6, 2010

1 Introduction

Allwolf is a simulation of wolves and sheep on a finite map. Each animal (agent) has a defined sight range and movement speed, and based on what they can see in their vision, will move accordingly. Sheep will run away from wolves, while wolves will run towards sheep. When a wolf is able to move onto the same location as a sheep, the sheep is turned into a wolf, and then the simulation is continued until there are no more sheep left on the map.

Allowing sheep and wolves to move at arbitrary speeds, as opposed to only *one* point, was actually not specified in the original design. I added this to better simulate real life, and to make the simulations complete faster.

From the specifications, the three main design goals will be implementing a *shared memory* system to act as the map, a *barrier* to make sure each agent only moves once per cycle, and an algorithm for calculating an agents next move.

2 Implementation

2.1 Language Choice

For my implementation I used the Java language, as it comes with a *massive* standard library, so I would not have to reinvent the wheel for shared memory and barrier classes.

2.2 Shared Memory

For the map, I chose to use a `ConcurrentMap<Point,Agent>`[1] instead of a two dimensional array. Using a `Map` would give me synchronization, and make moving agents simpler.

To implement the simulation map (shared memory), I used the `ConcurrentHashMap` class[2]. It acts like a standard synchronized `HashTable` in that all operations are synchronized, but it increases concurrency by partitioning the internal table, and only locking the required partition during updates. It also adds some new methods on top of the `Map<K,V>` interface, one being the `remove(Object key, Object value)` method. This only removes the `key` from the map if it is currently mapped to `value`. This is used in the `Board` class to make sure we don't move a sheep that has been replaced by a wolf.

2.3 Thread Synchronization

To ensure that each agent is only allowed to move *once* per cycle, I used the `CyclicBarrier` class[3]. The `CyclicBarrier` is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. Each agent thread calculates its next move (see section 2.4), and then waits for all the other agents to do the same. Once they are all ready, they line up to use the method `synchronized void moveAgent(Agent agent, Point dest)` in the `Board` class. If a conflict is detected, a `MoveException` is thrown, and the `Agent` calculates its next move again.

The `CyclicBarrier` class also allows an object that implements `Runnable` to be run after each cycle. I use this feature to run `EndGameCheck` to count the number of sheep and wolves left, and end the simulation if necessary. It can also slow down the simulation to make viewing easier (`Thread.sleep(long timeout)`).

2.4 Move Selection

All `Agent` moves are calculated by giving the `x` and `y` directions in which the agent wants to move, and optionally a goal `Point` that the `Agent` is trying to reach.

[1] `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentMap.html`
[2] `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html`
[3] `http://java.sun.com/javase/6/docs/api/java/util/concurrent/CyclicBarrier.html`

Once the directions and optional goal have been calculated, the algorithm systematically pushes onto a stack destination `Point`s which are further and further away from the starting position in the given `x` and `y` directions. It stops and returns the final destination point when the agent is not allowed to move any further in a turn or the goal has been reached.

2.4.1 Wolf

A wolf calculates its next move by creating a list of visible sheep, picking the closest one, and then using that sheep's position as a goal for the move algorithm. If it cannot see any sheep, it moves in a random direction.

2.4.2 Sheep

A sheep calculates its next move by creating a list of visible wolves, finding the average location of those wolves, and moving *away* from that point. Similar to a wolf, if a sheep cannot see any wolves, it moves in a random direction.

3 Conclusion

The hardest problem in this project was making sure the simulation board did not become corrupted due to multiple threads updating at the same time. The board was kept stable by using the `ConcurrentHashMap` class, which provides thread-safe operations to use for moving `Agent`ss. Keeping all the `Agent`s in sync would have been difficult as well, but fortunately the `CyclicBarrier` class provides that functionality.

A Source Code

The source code can be found at `http://github.com/tjwallace/allwolf`. If you are using a UNIX like environment, you can check out the source code by running: `git clone git://github.com/tjwallace/allwolf.git`, otherwise you can download a zip archive from `http://github.com/tjwallace/allwolf/zipball/master`.

B Compiling and Running

Load project into Eclipse and run `Main.java`. The map is printed to the console after each cycle.