ECSE 420 – Parallel Computing Friday, December 12, 2008

Hunting Game – A parallel implementation in C#

Final Project Report

Presented to: Professor Zeljko Zilic



Presented by:

Ionut Georgian Ciobanu (260242300)

Tudor Cristian Gaulea (260148089)

Table of Contents

[1. Overview 2](#_Toc216877461)

[2. Implementation 2](#_Toc216877462)

[2.1. The Rules 2](#_Toc216877463)

[2.2. The Board 2](#_Toc216877464)

[2.3. Wolves and sheep 2](#_Toc216877465)

[2.4. Parallel programming challenges 3](#_Toc216877466)

[2.4.1 Synchronization 3](#_Toc216877467)

[2.4.2 Deadlock 4](#_Toc216877468)

[2.4.3 Lock-free structures 4](#_Toc216877469)

[2.5. Hunting Algorithm 4](#_Toc216877470)

[2.6. Escaping Algorithm 5](#_Toc216877471)

[3. Code Compilation and Execution Instructions 5](#_Toc216877472)

[3.1 Prerequisites 5](#_Toc216877473)

[3.2 Binary (compiled) format 5](#_Toc216877474)

[3.3 Source code 5](#_Toc216877475)

# 1. Overview

In this project we propose a C# implementation of the wolf and sheep game described in class. We will create an array representing the board, and associate two mutexes with every cell: one for the wolves and the other for the sheep. Thus, any wolf can never be on the same cell as another at the same time, but it can enter cells in which there is a sheep. Each wolf has its own thread, as well as every sheep.

We will use the Strategy design pattern for the all movement algorithms for both the sheep and the wolves. This will allows us to implement and experiment multiple scenarios easily. At the same time, with minimum work, real wolves and sheep will be able to implement algorithms and research the best way to hunt/evade in a pack/herd. The wolves will have a pack behavior implemented – they will cooperate to hunt a sheep in order to minimize the time spent searching/chasing.

# 2. Implementation

## 2.1. The Rules

The board size is configurable, n x n cells. The visual range of wolves and sheep is also configurable, a square of n x n cells. Two sheep or two wolves cannot occupy the same cell; when a wolf eats a sheep, there will be two wolves on the same cell. The wolves cooperate when hunting. Sheep will move together as a herd when escaping from wolves.

## 2.2. The Board

The board is represented by an array of n x n MutexCell objects. A MutexCell has two mutexes: one mutex for the wolves and one for the sheep.

The array of MutexCells is represented by a MutexTable object. This object will provide the functions needed to add and activate the animals on the board. It also provides methods to determine if a given cell is occupied by a wolf or a sheep.

## 2.3. Wolves and sheep

Wolves and sheep are represented by the Wolf and Sheep classes, derived from the general class Animal. Each of these classes provides a method for moving in and out of a cell. When moving from one cell to another, the animal checks the mutex of the cell where it wants to move to. If this mutex is taken the thread is suspended until a timeout occurs or the mutex is released, whichever occurs first. Otherwise, it acquires the mutex and release the old one. For a brief moment, two cells are occupied by one animal at the same time. This is normal behaviour, as it corresponds to the real-life situation, where a transition from A to B does not happen instantly, but is a continuous process.

Important considerations when dealing with mutexes are how synchronization between animals takes place, and how deadlock is avoided.

We represent the sheep with a red circle on the board, and the Wolves with a slightly larger, blue circle.

The wolves have a pack behavior implemented, in order to minimize the time it takes to capture a sheep, or maximize the number of sheep captured. Each of the “wolves” has read access to a lock-free data structure. When a wolf sees a sheep, it updates this array by indicating the position of the sheep in its eye sight. This way, all the wolves share the same view of the board. Anyone can easily change the pack behavior without changing anything else in the code. They just have to use the public interface of the Wolf class.

A similar idea is implemented for sheep. The purpose of the share view changes in this case. Instead of having each sheep run for its life, they all move together like a herd, in order to maximize the time it takes for either any/all sheep to be eaten.

## 2.4. Parallel programming challenges

While developing the application, we encountered several hard parallel-programming related challenges. As our solution is flexible enough to be used for further research without requiring low-level multi-threading code on the user’s part, it is useful to discuss some of these challenges.

### 2.4.1 Synchronization

As each animal moves in its separate thread, there are a couple of problems that can arise. The very first problem is placing an animal on the board. What should we do if there is another animal of the same type at the same location? We choose to wait on the mutex and acquire it as soon as it is released, no matter when this happens. What if the user tries to place a Sheep on a cell where there is already a Wolf? What about the reverse case? In each on these cases we decided to implement the simple, most natural solution: allow the wolf to capture the Sheep. While this might be confusing sometimes, the alternative, namely to wait until the animal is gone, would not help very much. Chances are that as soon as we place a Sheep for example, even if the square is free, the very next instant a Wolf might come in, and the end-effect would be the same. We thus chose the most robust implementation.

The next and most important function of each animal was the movement from a cell to another. How should a wolf make sure that when it decides to move to the next cell, after checking whether it is free, it is still available? What if right after the check, another wolf occupies the square before the first wolf can move it? If a wolf and a sheep occupy a cell simultaneously for some period of time, how do we make sure that they do not “miss” each other? That is, we can visually see they occupy the same cell, but they do not detect this situation due to synchronization issues. For this, we established the following protocol: the movement algorithm of the wolf will check whether there is a sheep on the square it just moved into, and signal the sheep if it is present. But what if just before the wolf gets a chance to check, the sheep leaves? To cover this situation, our protocol requires that before moving out of a cell, a sheep always checks whether any wolf is present, or signaled it (we can easily imagine a situation where a wolf is signaled but is not present anymore). If it detects any of these situations, it then transforms itself into a wolf.

### 2.4.2 Deadlock

Dealing with mutexes requires great care, as they can generate deadlock situations: thread A waiting on a mutex B holds, while B is waiting for a resources A holds. While dependencies between just two animals are not possible in this case, we still have the possibility of deadlock through a circular dependency between four or more animals. To deal with this issue, we chose to design the waitMove function for the animals with a specified timeout. This means that if after a certain amount of time the mutex is not acquired, the function will return and notify the caller. The caller then has the possibility of issuing another call, this time for a different position. This is a good solution in our case. Here is why: if the wolf is in the Wandering state, it is not critical if it deviates a little from the desired path. If it is in the other possible state, chasing a Sheep, it is already delayed by some obstacle, probably another wolf. In that case, if after a certain number of attempts to move in a direction no progress is made (i.e. waitMove returns false), it is better to try another direction (i.e. avoid the obstacle(s)).

The same considerations apply to sheep too.

Thus, although the responsibility to avoid deadlock lies on the user of the primitives, the wolf and sheep classes provide sufficient and flexible support to make it a very easy task.

### 2.4.3 Lock-free structures

In order to enable cooperation, we decided to allow animals to know the position of all the other animals of the same kind. A sheep will thus know the position of all other sheep. In addition to this, it will also know the position of all the wolves currently seen by other sheep, thus the concept of a “shared view”. Wolves are also constantly aware of the position of all the other wolves, and can only see the position of sheep within their own, or other wolves’ visual range.

Our decision was to implement a lock-free data structure for this purpose. We desired to have no other synchronization delays, except for when animals have to move from one cell to another.

Lock-free, efficient data structures are traditionally hard to design and implement.

For the problem at hand we found a robust, efficient way to implement this: we maintain a list of wolves and one of sheep. Anytime we add, remove, or transform an animal, we update this list. For wolves, we present a filtered view of the sheep list, and a full view of the wolves list. For sheep we apply the same logic, reversed.

This allows for very efficient and most importantly consistent information available, in the context of rapidly and independently changing information.

## 2.5. Wolves: Wandering and Hunting Algorithm

A simple wandering algorithm was implemented for wolves. Basically, when the game begins, a pack of wolves starts at top of the board and begins moving downward. The algorithm makes the wolves group up such that each wolf can find another wolf within 3 cells. This applies to the x coordinate and does not reflect the actual distance between two wolves. This algorithm can be observed by starting the application and selecting the “Start: Wolf ONLY” button.

The hunting algorithm kicks in when a wolf detects a sheep in its eyesight. Again, this is a simple algorithm used for demonstration purposes. Each “wolf” has access to the shared view data structure. As mentioned above, when a wolf sees a sheep, each of the wolves sees it. Then each wolf looks into the shared view and picks the target by choosing the sheep that is closest to any of the wolves. Therefore all the wolves end up chasing the same sheep.

## 2.6. Sheep: Wandering and Hunting Algorithm

A simple wandering algorithm was implemented for sheep. As was the case for wolves, each sheep need to stay within 2 cells horizontally from another. This algorithm can be observed by starting the application and selecting the “Start: Sheep ONLY” button. The escaping algorithm was implemented as follows: each sheep “reads” the shared view data structure every second and determines which wolf is closest to it. Then it runs away from it.

# 3. Code Compilation and Execution Instructions

## 3.1 Prerequisites

This code was compiled using the C# 3.5 toolchain using Visual Studio 2008, and it targeted towards the .NET Framework 3.5. Please install the .NET Framework at <http://www.microsoft.com/downloads/details.aspx?FamilyId=333325FD-AE52-4E35-B531-508D977D32A6&displaylang=en> if needed. A free version of Visual C# can be downloaded at <http://www.microsoft.com/express/vcsharp/>.

## 3.2 Binary (compiled) format

The zip file contains a binary version of the application, for ease of use. It consists of the executable named “TheCoolTool.exe” and the DLL file named “HuntingGame.dll”. To run the project, please make sure that the DLL file is in the same folder as the executable, and all the prerequisites are satisfied.

## 3.3 Source code

To run the project in debug mode or compile from source, you can either open the HuntingGame.sln solution file in Visual Studio 2008, or use the command line tools provided with the .NET Framework.