Concurrent Programming Practical 2: The Game of Life

It is recommended that you read the chapter of the coursenotes on synchronous data parallel programming before you attempt this practical. The code from the lectures is on the course website.

Introduction

The aim of this practical is to implement the Game of Life, a cellular automaton invented by John Conway in 1970. The Wikipedia entry on Life describes it as follows:[1]

> The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
>
> 1. Any live cell with fewer than two live neighbours dies, as if by needs caused by underpopulation.
> 2. Any live cell with more than three live neighbours dies, as if by overcrowding.
> 3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
> 4. Any tile with exactly three live neighbours cells will be populated with a living cell.
>
> The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed — births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

[1] `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

You might want to read the rest of the Wikipedia article.

We will consider a variant with a finite N by N grid, treated as a toroid, i.e. where the top and bottom edges of the grid are treated as being adjacent, as are the left and right edges.

Your task

Your task is to implement the Game of Life. Your program should use p processes (say 4 or 8) to update the cells on each generation. You will probably want to allocate some region of the grid to each process. There are two essentially different ways to structure the program: (1) as a shared-variable synchronous data parallel program; (2) as a pure message-passing program.[2] In both cases the processes need to be synchronised in some way, so that the rules of the game are followed, so you need to think carefully about how to avoid race conditions.

There is a file `Display.scala` (on the course website), which defines a `Display` class that can be used to build displays that show the state of a grid. A display is created by

```
val display = new Display(N,a)
```

where `N` is an integer, representing the size of the grid, and `a` is the grid, represented by an `N` by `N` array of `Boolean`, e.g. initialised by

```
val a =  Array.ofDim[Boolean](N,N)
```

The display is made consistent with `a` by executing

```
display.draw
```

This shows solid black squares at positions `(i,j)` where `a(i,j)` is true, and grey squares where `a(i,j)` is false.

Test your implementation by considering some interesting seeds (see the Wikipedia page to get some ideas).

[2] In the latter case my experience with a predecessor of thread CSO suggests that allocating a CSO process to each square in the grid means that only a small grid can be dealt with before the operating system/jvm begins to run out of thread resources. Go, or the fibre-based implementation of CSO would be a better bet for a serious attempt: but my advice is not to try at this stage. Other message-passing solutions have workers working on larger rectangular regions of the grid, and exchanging the edges of their regions with neighbour workers every generation.

Optional

Implement one or more of the variants from the Wikipedia page. (You might have to adapt `Display.scala` to do this.) Alternatively, devise a more complicated game that can be modelled using cellular automata. For example, you could model the interactions between foxes and rabbits. Don't make the rules too complicated, though! You could include some randomness in the game.

Reporting

Your report should be in the form of a well commented program, describing any design decisions you have made.

Bernard Sufrin
15th January, 2016(4.1324)