Erlang: concurrent programming

Johan Montelius

October 2, 2016

Introduction

In this assignment you should implement the game of life, a very simple simulation with surprising results. You will implement it in a quite unusual way, instead of iterate over a data structure you will implement the state as a set of communicating processes.

1 The game of life

The game of life, defined by John Horton Conway in 1970, shows that one can generate very complex pattern, even self replicating, from a set of very simple rules. You will find tons of information on what the game looks like if you surf the web so this is a very short description.

The state of the game is a two dimensional grid, in our example we will connect the upper side to the lower side and the right edge to the left edge, resulting in a toroid. Each cell in the grid has eight neighbors; we will denote these south, south-east, east, north-east etc. Each cell in the grid can be either *alive* or *dead*.

Life evolves in a sequence of generations. The state of a cell in one generation is determined by the state of the cells in the previous generation. So given a pattern of living and dead cells in one generation, we can determine the state of each cell in the next generation. We have three simple rules that determines the state of a cell in the next generation.

- A cell with less than two living neighbors will be dead.
- A cell with exactly two living neighbors will be keep its state, dead or alive.
- A cell with exactly three living neighbors will be alive.
- A cell with more than three living neighbors will be dead.

Living cells thus require two or three neighbours to stay alive, if they have more they will die. A cell that is dead will come to life if it has exectly three living cells as neighbours.

The first generation is determined by you, or by random, and the rules then determine all future generations.

2 The obvious solution

The obvious way of representing the state is of course as a two dimensional array. In an imperative language this would be straight forward but in functional language it's quite tricky. We will call our first solution `life1` and it will be a solution only for the special case when the the size of the grid is four times four.

Assume that we represent the state of the game as a tuple of tuples. A four by four grid could then look like this:

```
 {{dead,  dead,  dead,  alive},
  {alive, alive, dead,  dead},
  {dead,  alive, alive, dead},
  {dead,  alive, dead,  dead}}
```

2.1 accessing the state

We have a built-in function `element/2` that can select a particular element in a tuple and using it we can implement an access function for our two dimensional grid. For reasons that will soon be clear we will use a zero indexed array so will add one to the arguments before applying the elements function.

```
element(Row, Col, Grid} ->
   element(Col+1, element(Row+1, ...)).
```

Since we need to know the state of a cells neighbors we define a set of functions to access these. We name these functions after the directions: south, south-west, south-east, west etc. The south neighbor of a cell R, C is thus the element $R+1$, C. However, we have to consider the case where the grid wraps around so the south neighbor is $R+1 mod 4$. The north neighbor is in the same manner $R+3 mod 4$ or $R-1 mod 4$ but we see why this is not so good.

There is no builtin *mod* operator in Erlang but a related *rem* operator that implements the reminder function. For positive integers the *rem* function is the same as the modular function but that is not the case for negative numbers; $-1 mod 4 = 3$ but $-1 rem 4 = -1$. This is why we define the northen neigbour as $R+3 rem 4$.

Define access functions for all the neighbors of a cell.

```
s(R,C, Grid) ->
    element((R+1) rem 4, C, Grid).
sw(R,C,Grid) ->
    element((R+1) rem 4, (C+3) rem 4, Grid).
se(R,C,Grid) ->
```

```
    element((R+1) rem 4, (C+1) rem 4, Grid).
   :
   :

ne(R,C,Grid) ->
    element((R+3) rem 4, (C+1) rem 4, Grid).
```

Also define a function `this/3` that returns the state of the cell it self.

```
this(R, C, Grid) ->
    element(R, C, Grid).
```

2.2 creating the next state

A function can then be defined that takes a grid and generates a new generation.

```
next_gen(Grid) ->
    R1 = next_row(0, Grid),
    R2 = next_row(1, Grid),
    R3 = next_row(2, Grid),
    R4 = next_row(3, Grid),
    {R1, R2, R3, R4}.
```

As you see this becomes a very special solution for the four times four grid but it serves our purpose for now.

```
next_row(R, Grid) ->
    C1 = next_cell(R, 0, Grid),
    C2 = next_cell(R, 1, Grid),
    C3 = next_cell(R, 2, Grid),
    C4 = next_cell(R, 3, Grid),
    {C1, C2, C3, C4}.
```

Almost done, now for you to implement the last step.

```
next_cell(R, C, Grid) ->
    S = s(R, C, Grid),
    SW = sw(R, C, Grid),
      :
      :
    NE = ne(R, C, Grid),
    This = this(R,C, Grid)
    rule([S, SW, SE ...], This).
```

The function `rule/2` is the function that implements the rules of the game of life. In order to know the new state of a cell you need to count the number of alive neighbors and then

```
rule(Neighbours, State) ->
    Alive = alive(Neighbours),
    if
       Alive < 2 ->
                 ....;
       Alive == 2 ->
                 ....;
       Alive == 3 ->
                 ....;
       Alive > 3 ->
                 ....
    end.
```

The function `alive/1` should return the number of alive states. Try to implement it first as a regular recursive function, then using an accumulator then using the library function `lists:foldl/3`.

Implement `alive/1` and you're done. Test your implementation and make sure that you understand how the different functions work.

3 A more general solution

The above solution hopefully works but it would be more fun to have a general solution for arbitrary big grid. It is fairly simple to do this but we have to solve a problem; we need to construct a tuple with a size that is unknown at compile time.

3.1 list to tuple

We can not construct a tuple incrementally as we do with a list but there is one useful built-in function that will save us; `list_to_tuple/1` will take a list and construct a tuple with the elements of the list as the elements of the constructed tuple.

Creating a new row of size M can now be done by first creating a list of M cells and then turn this list into tuple. Create a new module `life2` and implement the more general solution.

```
next_row(M, R, Grid) ->
    Cells = next_cells(M, R, 0, Grid),
    list_to_tuple(Cells).
```

```
next_cells(M, _, M, _) ->
    [];
next_cells(M, R, C, Grid) ->
    [next_cell(M, R, C, Grid)|next_cells(M, R, C+1, Grid)].
```

A new grid is constructed in a similar way. Fill in the blanks and complete the implementation.

```
next_gen(M, Grid) ->
    Rows = ...,
    list_to_tuple(Rows).

next_rows(M, M, _) ->
    ....;
next_rows(M, R, Grid) ->
    [... | ...].
```

Run some experiments to see that you can generate new grids of various size.

3.2 why use a tuple?

Now let's stop and think for a while; why do we represent the grid as a tuple of tuples? Why not represent the grid as a list of lists?

If we represent the grid as a list of lists it will of course be more expensive to lookup a particular cell. In the case of tuples it's a constant time operation (two calls to `element/2`) while in the case of lists it depends on the size of the grid (the length of the lists). So represent the grid as a list of list certainly has its disadvantages. The advantage would of course be that we do not have to turn a list into a tuple every time we construct a new row.

What is the access pattern when constructing a new grid? Does it look like random access? No, to construct a new row, one will access the cells of three rows only and the cells of these rows a access in order.

In order to construct the cells of row R, one need the rows $R-1$, R and $R+1$ ... unless it's the first or last row. If we construct the first row we need to look at the last row etc. This leads to an idea:

- let's represent a grid of R rows, as a list of $R+2$ lists.

We will duplicate the first and the last row, so a grid of rows $R1 \ldots Rn$ is represented as the list:

```
[Rm, R1, R2, ..., Rm, R1]
```

The third module will be called `life3` and is at first sight much more complicated but once you see the pattern it all becomes clear. Now what will the `next_gen` function look like? How about this:

```
next_gen([Rm, R1, R2 | Rest]) ->
     First = next_row(Rm, R1, R2),
     {Rows, Last} = next_rows([R1, R2 | Rest], First),
     [Last, First | Rows].
```

The function `next_rows/2` will consume the the rows and when it has produced the `Last` row it will add the `First` row to the list and return the tuple {`Rows, Last`}. We then add the `Last` row to the front of the list and we're done. Note that we assume that the grid is larger than a single cell. We could add a clause to handle this special case but we will simply ignore the problem.

Now define the function `next_rows/2`. You will find that the pattern is repeating:

```
next_rows([R1, Rm, R1], First) ->
     Last = ... ,
     {[..., ...], ...};
next_rows([R1, R2, R3 | Rest], First) ->
     Next = ... ,
     {Rows, Last} = ....,
     {[...|...], ...},
```

How are the rows represented? Why not repeat the structure and let each row be represented by a list where the first and last cell has been repeated: `[Cm, C1, C2, ... Cm, C1]`.

Implementing `next_row/3` now becomes a quite easy, we have all information we need and we can repeat the pattern above. We name the cells of the rows by the points of the compass.

```
next_row([NE, N, NW | Nr], [.., .., ..|Tr], [.., .., ..|Sr]) ->
    First = rule(..., This),
    {Row, Last} = next_row([.., ..|Nr], [.., ..|Tr], [.., ..|Sr], First),
    [.., ..| ..].
```

As before we assume that the grid is lager than a single cell but you can easily add a clause to take care of the special case where there is only one cell.

```
next_row(..., ..., ..., First) ->
    Last = rule(..., This),
    {[..., ...], ...};
next_row(..., ..., ..., First) ->
    Next = rule(..., This),
    {Row, Last} = next_row(..., ..., ..., First),
    {[Next|Row], Last}.
```

You're done, now let's do some benchmarking.

4 Benchmarks

You should now have three modules: life1, life2 and life3, time to do some benchmarking. The first thing we need is for each of the modules a function that returns an initial grid. We also need to run multiple transformations and to measure the execution time.

4.1 the simple case

For the first module this is trivial since we only can handle grids of size $4x4$.

```
state() ->
    {{dead,  dead,  dead,  alive},
     {alive, alive, dead,  dead},
     {dead,  alive, alive, dead},
     {dead,  alive, dead,  dead}}.
```

We then need a function that can run several generations since measuring only one transformation will not be enough to get a precise estimate on the execution time. The function `erlang:system_time(micro_secons)` will give the time in micro seconds.

```
bench(N) ->
    State = state(),
    Start = erlang:system_time(micro_seconds),
    Final = generations(N, State),
    Stop = erlang:system_time(micro_seconds),
    Time = Stop - Start,
    io:format("~w generations computed in ~w us~n", [N, Time]),
    io:format("final state: ~w~n", [Final]),
    Time.
```

The function `generations/2` will take a state a generate a number of generations returning the last. Include these function in the `life1` module and run some test.

4.2 the general case

In `life2` and `life3` things are either almost identical, if we only want to measure the execution time for $4x4$ grids. We only need to change the function `state/0` and pass the size of the grid as a parameter to `generations/3`.

If we also want to do measurements on larger grids we will have to implement a function `state/1` that generates a grid of arbitrary size. This requires some thinking but if you give it a try you will see that the state function looks very similar to the `next_gen/2` function.

The `state/1` function for `life2` should not be a a problem but for `life3` it's a bit tricky. The skeleton for this function could be something like this. Assume that we have a function `row/1` that returns a correct row (with the first element replicated last and the last element replicated in the beginning), we can then write something like this:

```
state(M) ->
    First = row(M),
    {Rows, Last} = state(M, 2, First),
    [Last, First | Rows].
```

The function `state(M, 2, First)` should generate rows $2 \ldots M$ and place them all in a list called `Rows` with `First` added to the end. It should also tell us which one was the last row generated i.e. the row M. If we have this function the final result is the list `[Last, First | Rows]`.

The first clause of `state/3` is simple, generate the last row and return the result `[Last, First], Last`. The recursive case is also quite simple (once you see it), generate a new row and use `state/3` to generate the rest of rows. Then add the generated row to the front of the list.

```
state(M, M, First) ->
    Last = row(M),
    {[Last, First], Last};
state(M, R, First) ->
    Row = row(M),
    {Rows, Last} = state(M, R+1, First),
    {[Row|Rows], Last}.
```

The only thing that is now missing is the function `row/1` but this will have the same structure as `state/1`. Give it a try, you can do it.

If you choose to generate larger grids it could be fun to give them a random initial state. The following function uses a built-in function to generate a random dead or alive state.

```
flip() ->
    Flip = random:uniform(4),
    if
Flip == 1 ->
    alive;
true ->
    dead
    end.
```

5 A concurrent implementation

The next solution is not something that one would come up with as a sensible solution in any language but we give it a try.

The idea is to implement each cell as an independent process. The cell would be connected to its neighbors and in each iteration:

- send its current state to all its neighbors,
- collect the states from all neighbors and,
- update its state.

We begin by defining the cell process and then work on how to connect the cells in the grid.

5.1 a living cell

A cell will have a state that consist of:

- N: the number of generations to compute.
- Ctrl: a process identifier of a process that wants to know when we're done.
- State: the current state, alive or dead.
- Neighbors: a list with the process identifiers of all eight neighbors.

The state of the process will be the arguments of a recursive function. In our case we will have two clauses that defines the function: one for the case where N is zero and a second for the general case.

Create a new module called `cell` in a file called `cell.erl`. The recursive function that defines the behavior of the process is also called `cell`. This is not necessary but good programing practice.

```
cell(0, Ctrl, _State, _Neighbors) ->
    Ctrl ! {done, self()};
cell(N, Ctrl, State, Neigbors) ->
    multicast(State, Neigbors),
    All = collect(Neigbors),
    Next = rule(All, State),
    cell(N-1, Ctrl, Next, Neigbors).
```

The multicast function will take a State (alive or dead) and send a message to each Neighbor. We need to tell the neighbor who we are and we can do so by sending a tuple {`state, Self, State`} where `Self` is our process

identifier. Here we use the library function `foreach/2` that applies the function to all of the elements in a list much in the same way as `map/2` would do.

```
multicast(State, Neighbors) ->
    Self = self(),
    lists:foreach(fun(Pid) -> Pid ! {state, Self, State} end, Neighbors).
```

Collecting the messages sent to us by our neighbors is a simple task. We know exactly form which cells the messages should arrive so we simply pick them up one by one. This

```
collect(Neighbors) ->
    lists:map(fun(Pid) ->
      receive
  {state, Pid, State} ->
      State
      end
      end,
      Neighbors).
```

As an exercise its worth the trouble to rewrite the two functions above without using the library functions.

The `rule/2` function is the same as used in the previous modules. If you want to make it more efficient you can change the `collect/1` function so that it returns the number of alive states; why first construct a list of states and then count the number of alive states.

5.2 starting the cell

When a cell is started it does not know any of its neighbors (since we have probably not created then yet). The cell is thus started knowing only its state.

```
start(State) ->
    spawn_link(fun() -> init(State) end).
```

We us the `spawn_link/1` function to start the process instead of the `spawn/1` function. This will ensure that if the creator of the process dies the cell will also die.

The first thing a cell does is to wait for a message that will give it access to all its neighbors. We will send this message to the cell as soon as we have created all the cells. It then waits for a message that tells it to start executing N generations and to whom it should report when it is done.

```
init(State) ->
    receive
{init, Neighbors} ->
    receive
{go, N, Ctrl} ->
    cell(N, Ctrl, State, Neighbors)
    end
    end.
```

This completes the `cell` module and we can now work on how to create a grid of cells.

5.3 creating the grid

This is the most complicated part of the implementation. We want to create MxM cells and have them connected so that they can communicate with its neighbors. We also want to produce a list of all the cells that we have created so that we can tell them to start working.

Let us write a function `state/1` that returns two things:

- A grid represented as a list if lists in the same way as we represented the grid in the previous solution.
- A list of all cells created.

Let us adapt the state function used in `life3` so that it also returns the list of all cells. This is what it would look like.

```
state(M) ->
    {First, S0} = row(M, []),
    {Rows, Last, S1} = state(M, 2, First, S0),
    {[Last, First | Rows], S1}.
```

The function `row/2` now takes a second argument, all the cells created so far, and generates not only a row but a list, `S1`, of all newly created cells added to the list of cells created so far.

In the same way we add an extra argument and have `state/4` that takes a list of all cells created so far, producing not only th rows and the last row but also the list `S1`.

Adapt the other functions in the same way and you soon have your function. The inly question is how to create a cell but this is simply done by calling the exported `start/1` function from the `cell` module.

5.4 connecting the cells

Having the state function we could write a function `all/1` that generates the grid, connect all cells and returns the list of all cells.

```
all(M) ->
    {Grid, All} = state(M),
    connect(Grid),
    All.
```

Connecting cells becomes trivial since we have the representation of the grid in a very handy format.

```
connect([Rl, Rm, R1]) ->
    connect(Rl, Rm, R1);
connect([R1, R2, R3 | Rest]) ->
    connect(R1, R2, R3),
    connect([R2, R3 | Rest]).
```

Connecting the cells of a row is simply done by running through the lists in the same way as you traversed the rows in the `next_row/4` function in `life3`. What message should you send to each cell?

```
connect([NE, N, NW], [E, This, W], [SE, S, SW]) ->
    This ! ...;
connect([NE, N, NW | Nr], [E, This, W | Tr], [SE, S, SW | Sr]) ->
    This ! ...,
    connect([N, NW | Nr], [This, W | Tr], [S, SW | Sr]).
```

That was not that problematic was it? Now for some benchmarking.

5.5 benchmarking

The only thing left is to adapt our benchmarking function so that it sends out a {`go, N, Ctrl`} message to all cells and collect the {`done, Pid`} replies.

```
bench(N, M) ->
    All = all(M),
    Start = erlang:system_time(micro_seconds),
    init(N, self(), All),
    collect(All),
    Stop = erlang:system_time(micro_seconds),
    Time = Stop - Start,
    io:format("~w generations of size ~w computed in ~w us~n", [N, M, Time]).
```

Sending out the messages and collecting the replies is implemented very similarly to the `multicast/2` and `collect/1` functions in the `cell` module.

Run some benchmarks and see how well the concurrent version compares to `life3`.

If you have access to a multicore machine you could try to run some experiments using one or more cores. If you start the Erlang shell with the `+S` flag you can control how many operating threads that should be used. The command `erl +S1` will start Erlang with only one thread. If nothing is specified the system is started with as many threads as you have cores. Can the Erlang virtual machine make use of the additional computations power of a multicore?