Lazy Cellular Automata
with Communicating Processes

Adam Sampson, Peter Welch and Fred Barnes
{atsl,P.H.Welch,F.R.M.Barnes}@kent.ac.uk

University of Kent
http://www.cs.kent.ac.uk/
The Theory Underpinning Nanotech Assemblers project needs to use PC clusters to simulate large numbers of autonomous entities.

... which we’re modelling as CAs for now.

We’re using the occam-π and JCSP languages – based on CSP extended with some ideas from the π-calculus.

Let’s look at Life as an example...
• Infinite grid of cells, each alive or dead
• On each generation step, examine self and 8 adjacent cells
• Alive and 2 or 3 live neighbours $\rightarrow$ alive
• Dead and exactly 3 live neighbours $\rightarrow$ alive
• Otherwise $\rightarrow$ dead
• Interesting emergent behaviour – e.g. the “glider”: (Black cells are alive.)
The Simple Parallel Approach

- One process per cell, connected in a grid
proc cell
while true
  par i = 0 for 8
  ... send state to neighbour[i]
  ... read state from neighbour[i]
  ... compute new state
• Inefficient
• 16 communications per cell per generation
• Most of the time the state hasn’t changed
• . . . so we only want to communicate changes
• We need a new way of synchronising generation steps
Barriers

- Barriers synchronise a set of processes
- Processes `sync` on the barrier, and block until all the enrolled processes are trying to `sync`...
- ...at which point they all proceed happily
- We can use a barrier for our generation tick
proc cell
  ... exchange initial state with neighbours (as before)
while true
  ... compute new state
if my state has changed
  par i = 0 for 8
  ... send state to neighbour[i] down buffered channel
sync barrier
  ... check buffered channels for changes from neighbours
• This is still inefficient
• All cells have to run and synchronise every generation, even if nothing around them has changed
• … so we want them to “sleep” when possible
• Make them resign from the barrier
proc cell
  ... exchange initial state
while true
  ... compute new state
  if my state has changed
    par i = 0 for 8
    ... send state to neighbour[i]
  else
    resign barrier
    ... wait for a change to be received
sync barrier
  ... check for changes
This is still inefficient

Lots of channels – can use one shared channel per cell

Lots of sleeping processes

... so let's only create processes for the active cells

Use `FORKing` and a shared state array

Use phases to control access to the array
The Just-in-Time Approach

```
proc cell
  running := true
while running
  phase 1 -- state array is constant
    ... read neighbour state from array
    ... compute my new state
  if my state has changed
    ... fork new cell processes for
       the affected neighbours
       (if not already running)
  else
    running := false
  phase 2 -- update state array
    ... write my new state to array
```
But Wait A Minute...

- Using shared memory isn’t very occam-ish
- Plus we’ve still got a big array in memory
- ... so use **FORK**ing, but still connect cells with channels
- Use mobile channels to dynamically build the active bits of the network
Clumps of active cells, connected by mobile channels, floating in the ether
• No particular reason why each process should only simulate one cell
• Make each process simulate a block of cells
• Can take advantage of existing fast sequential code
• ... or mix-and-match parallel approaches
• A number of approaches for simulating CAs in CPA environments
• Same approaches could be applied to other simulation tasks (FEA, CFD)
• Applications for new functionality in occam-π
• See the paper for more details – ask us if you’d like a copy of the demo code
Any questions?