Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Lazy Cellular Automata with Communicating Processes

Adam SAMPSON, Peter WELCH and Fred BARNES
Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
{ats1, P.H.Welch, F.R.M.Barnes}@kent.ac.uk

Abstract. Cellular automata (CAs) are good examples of systems in which large numbers of autonomous entities exhibit emergent behaviour. Using the occam-pi and JCSP communicating process systems, we show how to construct "lazy" and "just-in-time" models of cellular automata, which permit very efficient parallel simulation of sparse CA populations on shared-memory and distributed systems.

Keywords. CSP, occam-pi, JCSP, parallel, CA, Life, lazy, just-in-time, simulation

Introduction

The TUNA project is investigating ways to model nanite assemblers that allow their safety properties and emergent behaviour to be analysed. We are working with the occam-π language [1] and with the JCSP package for Java [2], both of which provide concurrency facilities based on the CSP process algebra and the π-calculus. The techniques described in this paper may be used in either environment; examples will be given in a pseudocode based on occam-π.

Autonomous devices with emergent behaviour will be familiar to anybody who has experimented with cellular automata; indeed, some of the first models constructed by the TUNA project are in the form of CAs. While CAs are significantly simpler than the sorts of devices we want eventually to model – for example, they have very simple state, usually operate upon a regular grid, and have a common clock – they provide a good starting point for modelling approaches. We examine several sequential and parallel approaches to simulating cellular automata in occam-π and JCSP.

The major desirable feature for a CA simulation is that very large scales can be achieved. This means that it should execute as fast as possible and use as little memory as possible. In particular, we would like to be able to take advantage of both distributed clusters of machines and new multi-core processor chips. We demonstrate approaches to CA modelling that satisfy these goals.

1. The Game of Life

The CA that we will use as an example is John Conway's Game of Life, usually referred to simply as "Life" [3]. First discovered in 1970, Life produces startling emergent behaviour using a simple rule to update the state of a rectangular grid, each cell of which may be either "alive" or "dead". All cells in the grid are updated in a single time step ("generation"). To compute the new state of a cell, its live neighbours are counted, where the cell's neighbours are those cells that are horizontally, vertically or diagonally adjacent to it. If a cell was dead

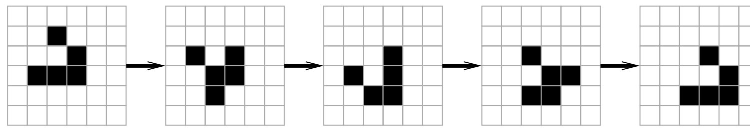

Figure 1. Five generations of a Life glider; black cells are alive.

in the previous generation and has exactly three live neighbours, it will become alive; if it was alive in the previous generation and does not have either exactly two or exactly three live neighbours, it will die. (See Figure 1.)

Thirty-five years of research into Life have produced a vast collection of interesting patterns to try. Simple arrangements of cells may repeat a cyclic pattern ("blinkers"), move across the grid by moving through a cyclic pattern that ends up with the original arrangement in a different location ("gliders"), generate a constant stream of other patterns ("guns" and "puffer trains"), constantly expand to occupy more of the grid ("space-fillers"), or display many other emergent behaviours. Life is Turing-complete; it is possible to create logic gates and Turing machines [4].

Life has some features which allow it to be simulated very efficiently. The most important is that cells only change their state in response to changes in the neighbouring cells; this makes it easy to detect when a cell's state must be recalculated. The new state rule is entirely symmetric; it does not make a difference which of a cell's neighbours are alive, just that a given number of them are, so the state that must be propagated between cells does not need to include cell locations. Finally, the new state rule is based on a simple count of live neighbours, which can be incremented and decremented as state change messages are received without needing to compute it from scratch on each cycle. These features are not common to all CAs – and certainly will not hold for some of the models that TUNA will investigate – but are nonetheless worth investigating from the implementer's point of view; if such a feature makes a system especially easy to simulate or reason about, it may be worth modifying a TUNA design to include it.

Some simple variants on Life exist that can be simulated using near-identical code. The normal Life rule is that a cell must have three neighbours to be born and two or three neighbours to survive; many variations simply change these numbers. (For example, in the HighLife variant, a cell may also survive if it has six neighbours.) Other variations change the topology of the Life grid: HexLife uses a hexagonal grid, and 3D Life uses a three-dimensional grid where cells are cubes and have 26 neighbours. Many other CAs that run on regular grids, such as WireWorld [5], may also be implemented within a Life-simulating framework, although they may require cells to keep or transfer more state.

2. Framework

Input and output for most of these approaches can be handled using common code; during development we constructed an occam-π framework which could support several different simulation approaches.

The input to a CA simulator consists of an initial state for all (or some) of the cells. For testing purposes, simple predictable patterns are the most useful, since correct behaviour may easily be recognised. However, some problems may be difficult to expose except under extreme load, so the ability to generate random patterns, or to load complex predefined patterns from disk, is also desirable. For CAs such as Life, learning to recognise correct and incorrect behaviour by eye is straightforward.

The output clearly must include the state of all of the cells; it is also helpful to display statistics such as the number of active cells. In order to obtain reasonable display performance, it is desirable to only update the screen once per generation (or even less often); this

can be done by having a simulation process send a "tick" to the display once per generation. Depending on how the display is implemented, it may be necessary for it to keep its own state array for all the cells; this can allow more interesting visualisations than simply showing the cells' states. For example, it is useful in Life to show how long each cell has been alive; the present framework uses the occam-π OpenGL bindings [6] to display a 3D projection of the Life grid where cells' ages are represented by their heights and colours.

3. Sequential Approach

The simplest approach to simulating Life is to walk over the entire grid for each generation, computing the new state of each cell (typically writing it into a second copy of the grid, which is exchanged with the first at the end of each step). This algorithm is O(number of cells in the grid).

As the majority of existing Life implementations are sequential, some techniques have been devised to speed up simulation. The most promising is Bill Gosper's HashLife algorithm [7], which computes hash functions over sections of the grid in order to spot repeating patterns; by caching the new state resulting from such patterns the first time they are computed, several generations of the new state for that region may simply be retrieved from the cache rather than computing it again, provided no other patterns interact with it. HashLife is particularly useful for quickly computing the outcome of a long-running Life pattern when there is no need to show the intermediate steps. The performance depends on the type of pattern being simulated; patterns with many repeating elements will perform very well, but the worst-case behaviour (where the pattern progresses without repetition) is worse than the simple approach, since hash values are being computed for no gain.

The sequential algorithms typically have good cache locality, and can thus operate very efficiently on a single processor. (Life has even been implemented using image manipulation operations on a graphics card processor.) However, in order to simulate very large Life grids – those with hundreds of millions of active cells – at an acceptable speed, we need to take advantage of multiple processors and hosts; we must investigate parallel algorithms.

4. Process-per-Cell Approaches

We examine a number of CSP-based parallel approaches to modelling Life in which each Life cell is represented by a process, starting with the simplest approach and demonstrating how incremental changes may be made to the model to improve performance.

4.1. Simple Concurrent Approach

The simplest parallel model of Life using a CSP approach is to have one process for each cell, connected using channels to form a grid (see Figure 2).

Wiring up the channels correctly is the most complex part of this approach – one approach is to say that each cell "owns" its outgoing channels, which are numbered from 0 to 7 clockwise starting from the top; channel N outgoing then connects to channel $(N+4) \bmod 8$ on its destination cell, which can be found by adding an appropriate offset to the current location (see Figure 3). The easiest way to deal with the connections at the edge of the grid is to wrap them around (making the grid topologically equivalent to a torus); alternately, they may connect to "sink cells" which behave like regular cells but act as if they are always dead. None of the cells need to know their absolute locations in the grid.

On each step, each cell must find out the state of those around it. This is done with an I/O-PAR exchange [8] in which each cell, in parallel, outputs its state to its neighbours and

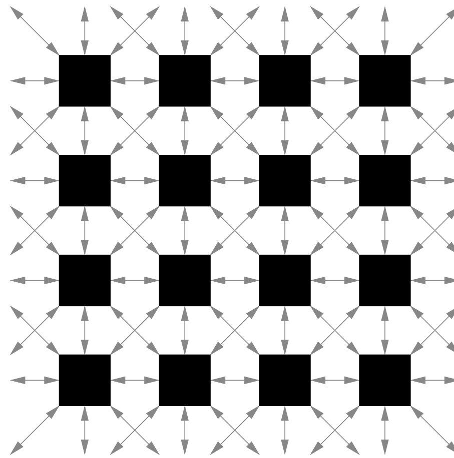

Figure 2. Grid of cell processes with interconnecting channels.

```
INITIAL [Height][Width]BOOL initial.state IS [...]:
[Height][Width]CHAN BOOL changes:
[Height][Width][8]CHAN BOOL links:
VAL [8]INT y.off IS [-1, -1, -1,  0, 0,  1, 1, 1]:
VAL [8]INT x.off IS [-1,  0,  1, -1, 1, -1, 0, 1]:
INT FUNCTION wrap (VAL INT v, max) IS (v + max) \ max:
PAR
  display (changes)
  PAR y = 0 FOR Height
    PAR x = 0 FOR Width
      [8]CHAN BOOL from.others IS
        [i = 0 FOR 8 |
          links[wrap(y + y.off[i], Height)]
               [wrap(x + x.off[i], Width)]
               [(i + 4) \ 8]]:
      cell (from.others, links[y][x], changes[y][x],
            initial.state[y][x])
```

Figure 3. Code to set up Life grid.

to the display, and reads its neighbours' state. Once the cell knows its neighbours' states, it computes its own state for the next generation (see Figure 4). As each cell must do nine outputs and eight inputs for each generation, there is no need for an external clock; the entire grid stays synchronised.

The I/O-PAR design rule guarantees that this implementation is free from deadlock. However, it runs very slowly – particularly when compared to a sequential implementation – because the majority of the time is spent doing communications, many of which are carrying state that has not changed. As we know that a Life cell's state will not change unless its neighbours' states have changed, this is wasteful, particularly for sparse patterns on a large grid.

4.2. Using a Barrier

We thus want to avoid communicating except upon state changes: a cell should only broadcast its state to its surrounding cells when it changes. This implies that we cannot use the I/O-PAR approach any more. Furthermore, it is possible that two groups of cells which are active may not be in contact with each other, so the inter-cell communications cannot provide the "generation tick"; another approach must be found.

```
PROC cell ([8]CHAN BOOL inputs, outputs,
           CHAN BOOL changes!,
           VAL BOOL initial.state)
  INITIAL BOOL my.state IS initial.state:
  [8]BOOL neighbour.states:
  WHILE TRUE
    SEQ
      PAR
        changes ! my.state
        PAR i = 0 FOR 8
          PAR
            outputs[i] ! my.state
            inputs[i] ? neighbour.states[i]
      my.state := compute.new.state (neighbour.states)
:
```

Figure 4. Code for one Life cell using the “simple” approach.

We could synchronise all the cells by having a central “clock” process with a channel leading to each cell, which outputs in parallel to all of them; however, we are trying to *reduce* the number of communications per generation! Fortunately, CSP provides a more efficient alternative in multiway events, which are available in occam-π and JCSP as barrier synchronisations. Barriers maintain an “enrolled” count of processes which may synchronise upon them; a process that attempts to synchronise will not proceed until all processes enrolled with the barrier are attempting to do so. We can provide generation synchronisation by making cell processes synchronise on a barrier shared with all the other cells in the grid.

Cells start by performing a single I/O-PAR exchange, as in the simple approach, in order to obtain the initial state of their neighbours; this could be avoided if all cells had access to a shared initial state array. The state of the cells around them is now held as a simple count of live cells. For each generation, a cell first computes its new state; if it has changed, it broadcasts it to the cells around it and to the display. It then synchronises on the barrier, and finally polls its input channels to collect any changes that have been sent by its neighbours, adjusting the count of live neighbours appropriately (see Figure 5).

This approach would cause instant deadlock if regular unbuffered occam-π channels – which cause writes to block until a matching read comes along, and vice versa – were used to connect the processes, since all writes are done before the barrier synchronisation and reads afterwards. Instead, the channels should be one-place buffered – that is, a process may write one message to the channel without blocking, and the read end may asynchronously collect the message at some point in the future. Unfortunately, while JCSP provides N-buffered channels, occam-π does not; it is, however, possible to simulate them using an “id” buffer process running at high priority [9]. The high priority guarantees that all the buffer processes will run before the barrier synchronisation completes. (This is strictly an abuse of the priority system, which is meant to be used for advisory purposes; however, we have found priorities useful for prototyping new communications mechanisms like this.)

With this approach, we are now only communicating when a state change occurs. However, all the cells on the grid are still taking part in the barrier synchronisation on each cycle; it is faster, but we can do better.

4.3. Resigning from the Barrier – The Lazy Model

A process that is enrolled on a barrier may also resign from it. A resigned process acts as through it were not enrolled; the barrier does not wait for it to synchronise before allowing other processes to run. We can take advantage of this to make cells “sleep” whilst nothing

```
PROC cell ([8]ONE-BUFFERED CHAN BOOL inputs, outputs,
           CHAN CHANGE changes!, BARRIER bar,
           VAL BOOL initial.state)
  INITIAL BOOL my.state IS initial.state:
  INT live.neighbours:
  SEQ
    ...  do one I/O-PAR exchange as before to count
         initially-alive neighbours
    WHILE TRUE
      BOOL new.state:
      SEQ
        ...  compute new.state based on live.neighbours
        IF
          new.state <> my.state
            PAR       -- state changed
              my.state := new.state
              PAR i = 0 FOR 8
                outputs[i] ! new.state
              changes ! new.state
          TRUE
            SKIP      -- no change
        SYNC bar
        SEQ i = 0 FOR 8
          PRI ALT
            BOOL b:
            inputs[i] ? b
              ...  adjust live.neighbours
            SKIP
              SKIP    -- just polling
:
```

Figure 5. Code for one Life cell using the “barrier” approach.

around them is changing. This results in “lazy simulation”, where cells only execute when it is absolutely necessary.

```
...
IF
  new.state <> my.state
    SEQ
      ...  broadcast new state as before
  TRUE
    SEQ       -- no change, so go to sleep
      ...  set priority to high
      RESIGN bar
        ALT i = 0 FOR 8
          BOOL b:
          inputs[i] ? b
            ...  adjust live.neighbours
SYNC bar
...  set priority to normal
...
```

Figure 6. Changes to the “barrier” approach to support resignation.

This requires some simple modifications to the “barrier” approach. The basic idea is that if the state has not changed, then the process resigns from the barrier and performs a regular ALT across its input channels; it will thus not run again until it receives a change message

from a neighbour, at which point it will rejoin the barrier, synchronise on it, and continue as it did with the previous approach (see Figure 6).

However, we have also had to insert some priority changes. If all processes are running at the same priority, then the barrier resignation causes a race condition to be present: between the `ALT` and the end of the `RESIGN` block, it is possible that all the other processes would synchronise on the barrier, meaning that when this process synchronises it must wait for the next generation. The priority changes are the simplest way to accomplish this, but other approaches are arguably more correct [10].

This optimisation causes a significant performance improvement, since only active cells occupy CPU time: a small glider moving across a huge grid will only require the cells that the glider touches to run. For typical patterns, performance is now rather better than a sequential simulation of the same grid, and the performance is much better than the first parallel approach described: after fifty generations on a randomly-initialised large grid, this approach was a factor of 15 faster than the original approach, and the relative performance increases further as the number of active cells decreases. However, it still uses far more memory, as there is a dormant process for each grid cell with a number of channels attached to it.

4.4. Using Shared Channels

Memory usage may be reduced significantly by cutting down on the number of channels. Since Life cells do not care about which neighbouring cell a change message was received from, we can take advantage of another occam-π and JCSP feature: shared channels. The approach is simply to replace the eight channels coming into each cell with a single shared channel; each of the eight neighbouring processes holds a reference to the shared channel.

The code is much the same as the previous approach: the only change is to the polling code, which must poll the shared channel repeatedly until it sees no data. It is also necessary for the one-place buffered channels to become eight-place buffered channels, since it is possible that all eight cells surrounding a cell may have changed. (To simulate this without real buffered channels, the approach is to make the buffers write to the eight neighbouring cells in parallel.)

We have thus reduced the number of channels by a factor of eight. In memory terms, this is not quite as good as it looks, since the buffer size in each channel has been increased by a factor of eight, and some overheads are caused by the channels being shared; nonetheless we have saved memory, and made the code a little more straightforward too.

More importantly, we have freed the code from the constraints of a rectangular grid. It would now be easy to use the same cells for a grid with a different number of neighbours, or even on “grids” with non-regular topologies such as Penrose tiles [11].

While this implementation scales significantly better than the conventional sequential implementation – and even performs better in many cases – its memory usage is still high.

4.5. Using Forking – The Just-In-Time Model

The major problem with the previous approach is that there is still one dormant process per grid cell; while occam-π processes are extremely lightweight compared to OS threads, they still require space to hold their internal state variables. Fortunately, we can avoid dormant processes entirely using occam-π’s “forking” mechanism.

Forking is a safer variant of thread-spawning, in which parameters are passed safely with the semantics of channel communication, and an enclosing `FORKING` block waits for all processes `FORK`ed inside it to finish. It is commonly used to spawn worker processes to handle incoming requests, as a more efficient replacement for the “pool of workers” approach that is often found in classical occam code.

```
REC PROC cell ([Height][Width]PORT BOOL state, running,
               MOBILE BARRIER bar, VAL INT y, x)
  SEQ
    SYNC bar      -- Phase 2 (cells are started from Phase 1)
    INITIAL BOOL me.running IS TRUE:
    WHILE me.running
      BOOL new.state:
      SEQ
        SYNC bar  -- Phase 1: read state, atomic set running
        ...  compute new.state from neighbours
        IF
          new.state <> state[y][x]
            PAR i = 0 FOR 8
              ...  compute neighbour location (n.y, n.x)
              INITIAL BOOL b IS TRUE:
              SEQ
                atomic.swap (running[n.y][n.x], b)
                IF
                  b     -- neighbour already running
                    SKIP
                  TRUE  -- neighbour not running
                    FORK cell (state, running, changes!,
                               bar, n.y, n.x)
          TRUE
            me.running := FALSE
        SYNC bar  -- Phase 2: write state, clear running
        state[y][x] := new.state
    running[y][x] := FALSE
:
```

Figure 7. Code for one Life cell using the "forking" approach.

For this example, we shall do away entirely with channels for inter-cell communication – a very nontraditional approach for occam! Instead, we use shared PORT data with phased access controlled by a barrier [10]. The framework starts the simulation by FORKing off a set of cell processes for the cells that are initially active. Each generation then consists of two phases. In Phase 1, the cell reads the states of the cells around it (directly from the shared state array), computes its new state, and ensures that any cells that need to change are running. In Phase 2, the cell writes its own state back to the shared array (see Figure 7).

The display update can now be done more efficiently: the display process shares the state array and the barrier with the cells, and follows the same phase discipline, reading the state array in Phase 1. It may even be possible to use the computer's display memory directly as the state array, doing away with the separate display process entirely.

The logic to ensure that cells are started correctly requires some explanation. Since a cell may become active for more than one reason – for example, if the cells above and below it both change state – it is necessary to prevent more than one cell process being FORKed for the same cell. A shared "running" array is used for this. In Phase 1, cells atomically swap a variable containing the value TRUE with the element in the array representing the cell they want to start; if the variable contains FALSE after the swap, the cell was not already running and needs to be started. In Phase 2, dying cells reset their slots in the "running" array to FALSE. As new cell processes are FORKed off from Phase 1, they must do an initial barrier synchronisation to get into Phase 2 for the top of the loop. (The only action that would normally be performed in Phase 2 is to write a changed cell's state into the array, and a newly-forked cell will not need to do that.)

The amortised cost of forking off new processes in occam-π is very low (of the order of

70 IA32 instructions), so the sample code will happily consider a cell "dead" if it has been inactive for a single generation. In practice, this is rather pessimistic for most Life patterns; many cells will toggle on and off with a period greater than two generations. If we wished to reduce the rate at which processes are created and destroyed, a simple heuristic could be put into place: count the number of generations that the cell has been inactive, and only cause the cell process to die once it has been inactive for N generations. This may result in better performance with JCSP on a system that uses native threads.

We now have a very efficient parallel Life implementation in which only as many processes as are needed are running at any one time – process creation is done "just in time". However, it relies upon shared memory, and thus cannot be implemented (efficiently) across a cluster of machines. For a cluster solution, our approach needs further modification.

4.6. Dynamic Network Creation

As occam programmers have known since the 1980s, CSP channels provide a convenient way of modelling network connections between discrete processors. We would therefore like to use channels to connect up our cells while keeping as many of the advantages of the "forking" approach as possible – in particular, only having as many processes in memory as are necessary for the level of activity on the grid. To do this, we will need to dynamically build channel connections between cells – which we can do using occam-π's mobile channels [9].

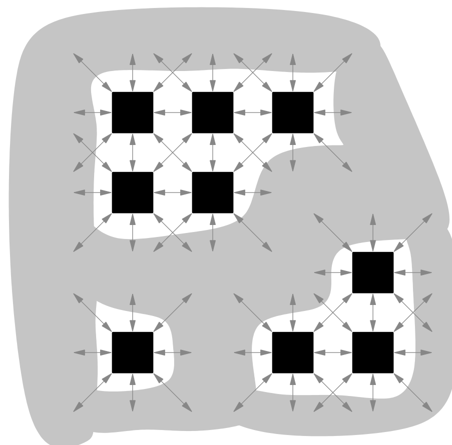

Figure 8. Ether surrounding clumps of active processes.

As with the previous approach, problems are caused when two clusters of cells split apart then rejoin, causing the cells between them to be activated for multiple reasons. In this case, it is necessary to connect up the channels correctly between the groups of rejoining cells. Previously we solved this sort of problem using shared data and atomic operations; now we shall instead use a coordinating process which manages channel ends that are not connected to active processes. As, from the modelling perspective, this process occupies the space around and between the clusters of active cells, it is called the "ether" (see Figure 8).

Cells now need to know their locations relative to an arbitrary reference point, in order that the ether can identify clusters of cells that drift apart and rejoin. For a non-regular topology, it may be possible to use unique identifiers rather than coordinates, and use external data structures to represent the relationships between cells; that scheme is rather less flexible than the shared-channels approach, but may be easier to manage under some circumstances.

Each cell process has channels connecting it to the cells around it (either shared or unshared), much like our previous parallel approaches, except now they are mobile channels, the ends of which may be passed around between processes. Each process also has a connection to the ether (via a channel shared between all cells); when it goes inactive and exits, it

sends a message to the ether returning its channel ends. From the perspective of the cell, all channels are connected to other cells; however, they may actually connect to the ether.

When the ether receives a change notification from a cell, it spawns a new cell in the appropriate location, checking its internal data structures to see whether it should be connected to any other cells in its vicinity using other channel ends that the ether is holding. If the ether can reuse existing channels it will; otherwise it will create new mobile channels, keep one end, and pass the other to the new process. (Since the search for existing channel ends is done purely on the basis of coordinates, it should be possible to do it very efficiently within the ether.)

As well as cluster-friendliness, using this approach also has the advantage that there is no longer a need for a big array of state. Indeed, sections of the grid that are inactive can just disappear, provided their states are known to the coordinating process; if they consist of empty space then this is easy. This approach should therefore work very well for testing gliders, spaceships and other Life patterns that move across the grid leaving little or nothing behind them; a feature that it has in common with HashLife. Visualising the output from a Life simulation implemented this way could be done by automatically zooming the display to encompass the section of the field that is currently being simulated; this could produce a very compelling visualisation for space-fillers and patterns such as the R-pentomino that expand from a simple cluster.

One final problem: the single ether process is a classic bottleneck; not a desirable feature for a parallel system, particularly if we want to make our cluster network topology mimic the connections in our Life grid.

4.7. Removing the Bottleneck

The final change is to parallelise the ether. This may be done straightforwardly by dividing it up into sections by coordinates (wrapping around so that an infinitely large grid may be simulated). Adjacent ether processes would need to communicate in order to create new processes and channels within the appropriate ether; air traffic controllers in the real world provide an appropriate analogy. As processes that need to communicate with each other will most likely be registered with the same ether, this approach offers good locality for cluster implementations of Life. In environments which do not provide transparent network channels, the ether processes can also be made responsible for setting up appropriate adaptors at machine boundaries.

5. Process-per-Block Approaches

While we have described several efficient ways of implementing Life using occam-π's facilities, all of the approaches described use one CSP process per cell, and thus still have significantly higher per-cell overhead than the existing sequential approaches. However, this is relatively easy to fix: all of the above approaches may be applied equally well to situations where each "cell" process is actually simulating a group of cells using a sequential (or even internally parallel) approach. The only change is that the state to be exchanged between processes becomes the set of states of the cells on the adjoining edges or corners.

Existing sequential approaches can be used virtually unmodified to obtain high performance. It may even be possible to switch between several different sequential approaches depending on the contents of the block; for example, the trade-off between HashLife and a "plain" sequential algorithm could be made on the fly depending upon the cache hit rate. To minimise communication costs when two chunks are on the same machine, mobile arrays of data could be swapped back and forth, or shared data could be used, protected by a barrier.

6. Conclusion

We have presented a number of approaches for simulating cellular automata in efficient ways in extended-CSP programming environments. It is to be hoped that some of these ideas could be used to implement highly-parallel CA simulators that can operate efficiently on extremely large grids. It should be possible to extend these ideas beyond CAs and into other cases where many autonomous entities need to be simulated – for example, finite element analysis or computational fluid dynamics.

We have also presented a number of applications for new functionality in the occam-π environment: in particular, some of the first practical uses for barriers and safely-shared data.

7. Acknowledgements

The authors would like to acknowledge EPSRC's support for this work through both a research studentship (EP/P50029X/1) and the TUNA project (EP/C516966/1).

References

[1] F.R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.

[2] P.H. Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[3] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Sci. Amer.*, 223:120–123, October 1970.

[4] A. Adamatzky, editor. *Collision-Based Computing*. Springer Verlag, 2001.

[5] A.K. Dewdney. Computer Recreations. *Sci. Amer.*, 262:146, January 1990.

[6] D.J. Dimmich and C.L. Jacobsen. A foreign function interface generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, Concurrent Systems Engineering, pages 235–248, IOS Press, The Netherlands, September 2005. IOS Press.

[7] R.W. Gosper. Exploiting regularities in large cellular spaces. *Physica D*, 10:75–80, 1984.

[8] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems "93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, The Netherlands. ISBN 90-5199-140-1.

[9] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating processes: Part 1. In J. Pascoe, P.H. Welch, R. Loader, and V. Sunderam, editors, *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, pages 321–352, IOS Press, The Netherlands, September 2002. IOS Press.

[10] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisations for occam-pi. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*. CSREA press, June 2005. to appear.

[11] R. Penrose. U.S. Patent #4,133,152: Set of tiles for covering a surface, 1979.