Creatures and Spirals
A Data Parallel Object Architecture

I Stephenson R W Taylor

Dept of Electronics
University of York
York, England

Abstract

Creatures*, a new class of cellular machine, retains many of the attractive features of Cellular Automata whilst allowing the user to describe dynamic systems in a more intuitive and efficient manner.*

In order to implement such a machine efficiently a generalisation on the Doubly Twisted Torus has been explored, and routing techniques developed for the surface. Such a surface has improved load balancing properties compared to traditional toroidal surfaces, while retaining the simple embedding of low dimensional structures.

1 Introduction

Cellular Automata (CA)[5] are made up of regular 'surfaces' of locally connected computing units or cells. Each cell examines the state of its neighbors and synchronously modifies its state according to a simple, universal rule. Each cell is identical, both in terms of its neighborhood (connectivity) and the rule or program that it is executing. CA have many attractive features for the programmer attempting to describe highly parallel systems. These include homogeneity, scaleability and simply defined local behaviour. However, attempting to model many physical systems using CA introduces a number of undesirable complications and overheads. These are primarily related to the the spatial nature of CA. Any model must be formulated in terms of the space it occupies. For many real problems the space simply provides a substrate in which active elements of the system may exist. As a result much programming effort goes into providing handshaking between cells to simulate the movement of active elements. Typically *interesting events* take place in relatively few areas of space, leaving many of the processing elements idle.

Presented here is an alternative — "Creatures". This provides a much needed shift of emphasis by modeling the *active* elements of a physical system. A simulation is defined in terms of agents that are able to move in space and interact with other local agents. This shift of focus allows a more intuitive style of programming, and in many cases a more efficient implementation.

2 The Creatures Model

The system is based in an arbitrary, infinite space where a number of *Creatures* may exist. Every creature has the same clearly defined behaviour. This is governed by both its internal state and other creatures it can "see" (on a strictly local basis).

- it may change its state;
- it may "give birth" to other creatures;
- it may move to a new location (adjacent to its starting point) within the space;
- it may "die".

These actions are performed synchronously in the same fashion as CA.

The Creatures model may be described more formally. However this outlined form of creature processing is sufficient to indicate how models may be built. In order to model a system it is only necessary to describe the behaviour of each individual component and provide the initial conditions. The complexity of the resulting system is dependent not on the complexity of the individual elements, but on the interaction between very large numbers of such elements. In developing such models, the system allows the user to experiment with and isolate the *characteristic* properties that determine its behaviour.

The specification of a simulation is written as a number of tests to be performed (for each creature) upon what can be seen, and based upon these, a number of actions which may take place. In addition to the standard language concepts of variable comparison and assignment, a number of special operations to describe the cellular nature of the system are required:

- `NORTH SOUTH EAST WEST CENTER` (or similarly for non von Neumann Neighborhoods) to define movement of the creature.
- `BIRTH(type)` to create a new creature.
- `BECOME(type)` to change type.
- `CANSEE(type)` to find the number of creatures of the given type, at the current location. The current creature will be included in this total if appropriate.
- `DIE` to destroy the Creature.

These form the basis of the "JAM" language, which is used to describe creature behaviour. While JAM lacks many of the features one would expect to find in a fully fledged language, it allows creature behaviour to be described in a concise, understandable manner. Once written, JAM rules are compiled into a machine dependant target language, which itself may then be compiled and linked into a simulator.

3 An Example of Jam Code

In order to illustrate how the system may be used, figure 1 contains the JAM code to solve the firing squad problem.

The Creatures solution takes a very physical approach to the problem: "soldier" creatures produce "bullet" creatures when they see a "shout". A "sargent" creature walks along the line, and counts the number of soldiers, then retreats a suitable distance. It then returns to its starting point, producing a "shout" for each soldier. All these will arrive at the same time.

This very physical approach to programming allows simulations to be constructed in a very intuitive fashion. Simple models have been constructed for road traffic flow, spread of infectious diseases, plant populations, ants, digital filters, ideal gases, nuclear critical mass reactions, and the movement of buoys in water currents. However, these have not been developed beyond the proof of concept stage, as such work must be performed in conjunction with field experts.

```
NEIGHBORHOOD:moore;
TYPES:soldier,bullet,sargent,shout;
VARS:int state,int counter;
INIT:{
     counter=0;
     state=0;
     }

RULE:
{
iam(soldier):
  {
  cansee(shout): birth(bullet);
  true         : CENTER;
  }
iam(bullet)    : NORTH;
iam(shout):
  {
  cansee(soldier): DIE;
  true           : NORTH;
  }
iam(sargent):
  {
  state==0:
    {
    cansee(soldier):
      {counter=counter+1;EAST;}
    !:{state=1;SOUTH;}
    }
  state==1:
    {
    counter>1:
      {counter=counter-1;SOUTH;}
    !:{state=2; NORTHWEST;}
    }
  state==2:
    {
    cansee(soldier)==0:
      {birth(shout);NORTHWEST;}
    !:{birth(shout);state=0;EAST;}
    }
  }
}
```

Figure 1: The Firing Squad Problem

4 Implementation

Creatures may be implemented in a number of ways, as is best suited to the underlying platform. Simulators have been developed for a number of systems, each providing a different cost/performance/ease of use ratio. Efficient implementations may be written for SISD, SIMD and MIMD machines. In all cases, a new back end to the JAM compiler allows existing rules to be ported. Rules may be developed on a relatively slow but flexible simulator where small scale behaviour can be closely examined, then transferred to a "heavyweight" processor to observe large scale behaviour. Only the parallel implementation of creatures on specially designed hardware will be considerd here.

4.1 Bucketing

In order to obtain good performance for large populations, steps must be taken to improve the calculation of which creatures may see others. Naive use of locality in the implementation would introduce undesirable features, similar to those found in cellular automata (poor load balancing of real work for example). It is not adequate to simply divide space into regions, and allocate a region to each processor, as for most simulations a single region (or at least a small number of regions) will hold most of the active processes while large areas of space are empty (particularly as space is required to be infinite).

A system was developed which makes use of a hashing function (of the form $H(x, y) = x + ky$) to split the population into a number of smaller subpopulations, each of which may be mapped far more quickly. Consider dividing the population in two: each half may be mapped four times as quickly (due to the N^2 dependance of the mapping operation). There are of course now two populations to be considered, so performance would be increased by a factor of two.

In dividing the population it is necessary to ensure that all creatures at a single location are allocated to the same processor (or set of processors known as a bucket). However by making use of an effective hashing function, clusters of creatures in neighboring locations may be distributed onto separate elements of the processing array. This method allows several locations to be mapped to the same bucket, reducing the likelihood that a bucket may be idle.

A trade off must be made for the bucket size: too small a bucket will tend towards the problems of a CA (one bucket may be overworked, while others may be empty). Too large a bucket size will result in a slower system. This technique allows the creation of system capable of stepping creatures with a time complexity of N^2/K, where K is the number of buckets. By increasing K in line with population size, then performance inversely proportional to N may be obtained

In a distributed system the number of buckets is fixed. However when implementing the architecture on a serial system it is possible to vary the number of buckets used between each timestep. This allows an optimal bucket size to be chosen for each step (though in such a case the hash function being used may no longer be perfect). This allows a system to maintain good performance over a very wide range of population sizes.

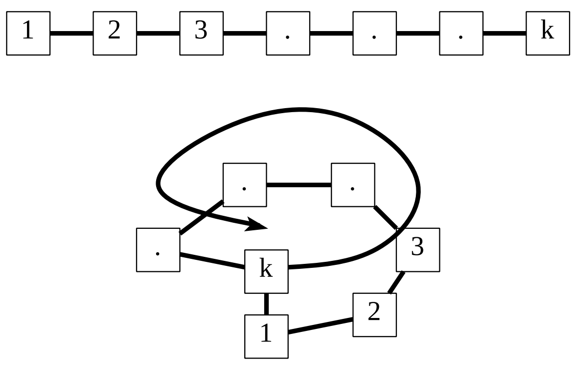

Figure 2: Wrapping buckets into a Spiral

A problem of bucketing concerns the distribution of creatures throughout the space. If a large number of creatures occupy the same location, then they should all be hashed into the same bucket. However the capacity of a bucket is necessarily limited. The simulator must therefore be run with sufficient space to ensure that buckets are unlikely to overflow.

4.2 Spiraling

Bucketing is effective in reducing the mapping overhead, but it introduces the new (significant) overhead of process migration. However by considering the buckets as a long line, the array can be wrapped around into a 3D spiral such that (for the hash function $H(x, y) = x + ky$) the kth bucket is directly above the first, and so on (figure 2). The last bucket is connected back to the first to form a toroidal structure. This ensures that the small changes of x and y involved in the migration of processes result in a short communications distance between adjacent buckets, while still retaining the desirable properties.

The spiral can be viewed as an edge connected mesh, where the edges, rather than being connected back to the same row, are connected back to a row which is offset (where the hashing example is a special case: offset=1, previously discovered for systolic computation[4]). This offset may be applied both to rows and columns. However in doing so special attention must be payed to the connection of nodes at the top right hand corner of the grid, where the two offsets interact. The result of this interaction is not obvious, and is best illustrated by figure 3 (the reader may also like to verify the result by performing the shift operation on four books!). When a shift of distance o is applied

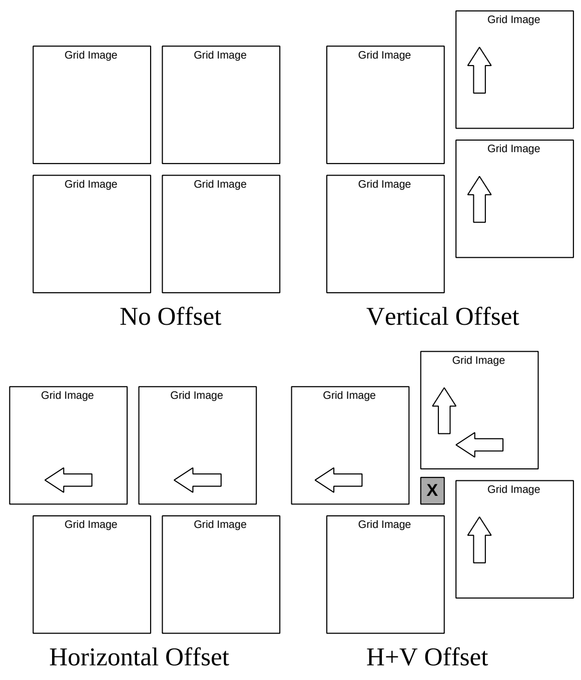

Figure 3: Filling the Gap

in each direction, a gap appears in the grid of size o^2. Once this gap is filled then the surface appears uniform, and may be used as a pseudo infinite surface. Hence effective grid sizes are of the form $n^2 + o^2$.

The next question which should be addressed is what values of n (the grid size), and o (the offset) are optimum — neglecting the obvious more processors is better argument. Without a priori knowledge of the problem to be tackled, few assumptions can be made about the load pattern that will be placed on the grid. It must therefore be assumed that on average loads will be symmetrical, and hence investigation will restricted such that $n_x = n_y$ and $o_x = o_y$.

In the discussion of bucketing a desirable feature of hash functions was that movement in either the x or y direction pass through every node before returning to the start point. This is particularly important, as one-dimensional automata (or near 1D — where one dimension far exceeds the other) are a common special case. This condition is satisfied when o and n are relatively prime: LCM$(o,n) = on$ (or alternatively GCD$(o,n) = 1$: $o = 1$ is always valid). Given a pair of values $< n, o >$ which satisfy this condition $< n, n - o >$ will also be suitable. Offsets may therefore be catagorised as large ($o > n/2$) or small ($o < n/2$).

If the previous argument is generalised to avoid clashes

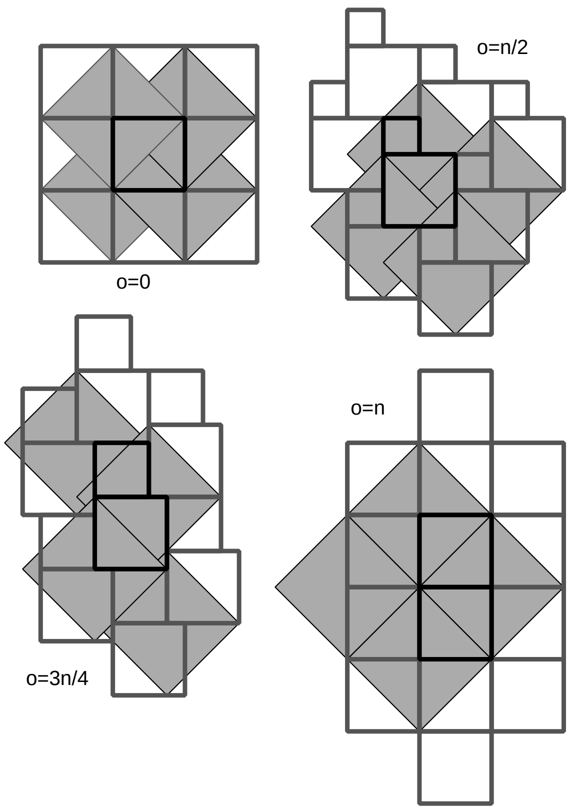

Figure 4: The diameter of Twisted Meshes

when (for example) loads of width 2 are applied, then it can be seen that the hashing model (where $o = 1$) is potentially poor, as the location k,0 is equivalent to 0,1. By increasing the offset, clashes of this form are likely to be reduced. It is therefore advisable to eliminate meshes with $o = 1$ or $o = n - 1$.

Every processor array has a diameter (d) which represents the worst case communications distance between pairs of points[3]. For a traditional n by n toroidal mesh this distance is approximately n. Any twisted mesh $< n, o >$ also has a diameter of n, but by increasing $o \to n$ the number of processors may be doubled without increasing d (figure 4). In addition it can be shown that increasing $o \to n$ increases the mean routing distance from $n/2$ to $2n/3$. This is an improvement over the equivalently sized square mesh which has a routing distance of $n/\sqrt{2}$.

The special case $o = n$ represents the most compact system, though it fails to meet the criteria previously set out for good mesh sizes. Larger values of o which *do* satisfy those guidelines will therefore be chosen.

In order that the wrapping around effect is regularly in-

Grid Size: n	Small Offsets:$o < n/2$	Large Offsets:$o > n/2$	Total number of Nodes
5	2	3	29,34
7	2,3	4,5	53,58,65,74
8	3	5	73,89
9	2,4	5,7	5,97,106,130
10	3,	7	109,149

Table 1: Some Viable Machine sizes

voked, and a good statistical distribution of loads between nodes is achieved it is necessary that the total number of nodes be kept reasonable small (in addition fabrication facilities are limited, so it would not be possible to construct a prototype machine with more than approximately 100 nodes), hence these restrictions are sufficient to indicate which values should be considered. A number of suitable machines are shown table 1.

A transputer system based upon the smallest of these size $< 5, 2 >$ is currently under development. This should allow the Creatures model to be run at high speed, but with relativly low cost.

5 Load balancing on Spirals

By applying the shift found in the twisted torus topology, regular patterns within loads are broken up. This can be seen in even the most basic loads. During initial tests rectangular loads (one load unit in each virtual location within a rectangle of variable size) were applied to twisted grids. Such loads are well balanced for non-twisted grids only when the size of the applied load is an exact multiple of grid size, otherwise the load will have three distinct regions (as show in figure 5). Here a rectangular load (randomly chosen to be 68 by 52 — these values are in no way special) is applied to an 8 by 8 edge connected mesh. The mean load is 55.25, but poor load balancing means that nodes may have as high a load as 63 or as low as 48. The standard deviation is 5.356 (optimum would be 0.433).

By applying a twist to the torus of between 2 and 7 an optimum load balancing is achieved in every case. Figure 5 shows the results for $o = 5$. The increase in the number of nodes results in a reduction of the mean load to 39.7, but now all nodes have a load of either 39 or 40. The deviation in the load is hence reduced to the optimum value of 0.44 — a dramatic improvement. While this is not always the case, the result is typical of many systems (provided certain pathological cases are avoided).

The above results have also been demonstrated with real loads taken from Creatures simulations. Geometric

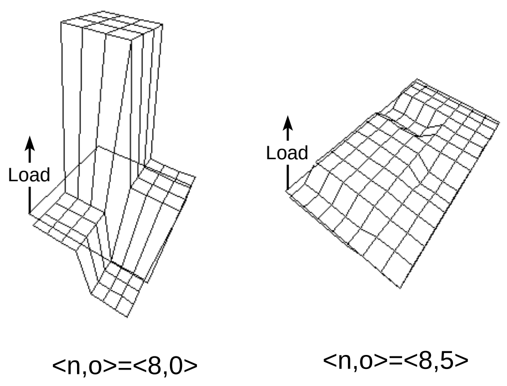

Figure 5: Applying a rectangular load

structures will be poorly mapped onto a traditional mesh, but are (generally) broken up by the spiraling technique.

6 Further Generalisations of the Twisted Torus

A twisted architecture may be fully described in terms of the independent vectors which map a point on the array to its images in virtual space (there will be k of these to span k dimensional space.) While $< n, o >$ is sufficient to describe the properties of a mesh it does not uniquely define its physical construction, and hence there will be several viable layouts, each isomorphic to all others. These vectors may be trivially generated for any dimension. By working back from these vectors a machine may be constructed. The determinant of a matrix composed of these vectors will be equal to the number of processors required to build the machine ($n^k + o^k$). This gives some insight into the negative direction of at least one shift when constructing a machine of even dimension.

The matrix of image vectors may also be used to perform optimal wormhole routing on a twisted array. The matrix and its inverse may be used to map the twisted space onto an orthogonal space where a points images may quickly be found. The shortest path between points may be selected in this simplified space, and then the transformation reversed, indicating how messages should be routed on the twisted surface.

When a machine of high dimensions is constructed all lower dimensional structures are found to embed well within it. However low dimensional structures which require global routing may still make use of the increased connectivity.

7 Conclusion

The *Creatures* Architecture offers a radical alternative to traditional von Neumann architectures whilst retaining an intuitive feel in its programming. Breaking problems down into Creature components is often very simple, making the system potentially more accessible to non-specialists than conventional highly parallel architectures, and perhaps even conventional computing systems. For a broad class of modeling problems this certainly appears to be the case.

In the process of implementing the “Creatures” model a novel and interesting class of processor topologies has been discovered, which appears to have applications in general parallel processing applications. The structure offers increased node density over traditional meshes (reduced diameter), by reducing the redundancy inherent in the communications of simple toriods.

Load balancing is improved for many usage patterns — in particular low dimensional structures are broken up and distributed fairly around a higher dimensional mesh. For many simple structures the load distribution may be near optimal.

These benefits may be obtained transparently to the end user, who may view the topology as a simple pseudo infinite surface. However such a user will benefit from improved performance over a conventional torus.

Acknowledgements

This work is supported by the Defence Research Agency, Malvern, Worcs, UK and the UK Science and Engineering Research Council.

References

[1] S. Abraham and K. Padmanabhan, (1991) *The Twisted Cube Topology for Multiprocessors: A Study in Network Asymmetry.* Journal of Parallel and Distributed Computing, Vol 13, pp 104–110, Academic Press Inc.

[2] P. Hogeweg. (1984) *Simulation Modelling Formalisms: Heterarchical Systems.* Encyclopedia of Systems and Control, Vol.6, pp4350–4353 Oxford: Pergamon Press

[3] D.M.N.Prior *et al* (1990) *What Price Regularity?* Concurrency: Practise and Experience, Vol.2(1), pp55–78, J. Wiley & Sons

[4] C.H.Sequin (1981) *Doubly Twisted Torus Networks for VLSI Processor Arrays.* 8th Annual Symposium on Computer architecture, pp471–480, IEEE Computer Society Press.

[5] Toffoli and Margolus. (1986) *CAM a new environment for modeling.* MIT Press, Cambridge, Mass.