Creatures

A Fine Grained Parallel Computer Architecture

David Ian Stephenson

D Phil Thesis

University of York

Department of Electronics

1995

Abstract

The Creatures model of parallel processing offers an alternative to conventional Cellular Automata based SIMD (Single Instruction Multiple Data) systems. This thesis investigates the Creatures model, and shows it to have a place alongside traditional data parallel techniques. The model shifts focus from the space in which simulations take place, to the active agents existing within that space. This change of emphasis allows more intuitive reasoning about models, as the agents will frequently have a very physical significance. This physical style of programming makes the architecture suitable for use by those who are inexperienced in parallel computing, while retaining the attractive SIMD features of scalability and homogeneity. The system is better suited to the modeling of dynamic systems than traditional cellular systems and may be more efficient when dealing with sparse data. These results may be more generally applied to a broader class of agent based computational models.

This thesis defines the Creatures model both informally and in a more rigorous mathematical notation, and shows it to be computationally complete. The model's implementation on both serial and parallel machines is demonstrated, leading to the development of a novel topology with attractive load balancing properties. A number of simulations are considered which demonstrate how Creatures may be applied in a number of fields.

Contents

1 Introduction **13**

1.1 The Problems of Parallel Processing 13

1.2 The Creatures Solution 16

1.3 Thesis 16

1.3.1 Chapter 2 — The Creatures Model 17

1.3.2 Chapter 3 — Implementing the Creatures Model 17

1.3.3 Chapter 4 — Applications 17

1.3.4 Chapter 5 — Discussion 17

1.3.5 Chapter 6 — Conclusion 17

2 The Creatures Model **18**

2.1 Background 18

2.1.1 Cellular Automata 18

2.1.2 Discrete Event Simulation 20

2.1.3 Artificial Life Systems 22

2.1.4 Mirror 23

2.1.5 *Logo 25

2.2 Defining Creatures 34

2.2.1 An Informal Description of the Creatures Model 34

2.2.2 A Formal Definition of the Creatures Model 38

2.2.3 Equivalences 45
2.2.4 Complexity 48
2.3 Conclusions 49

3 Implementing the Creatures Model 50

3.1 The JAM Language 50
3.1.1 Jam Grammar 52
3.2 A Sequential Implementation 55
3.2.1 A Naive Implementation 57
3.2.2 A Scalable Implementation Strategy 58
3.2.3 A Statistical Analysis of Bucketing 61
3.3 Creatures on the MasPar MP1 64
3.3.1 A Naive Implementation 66
3.3.2 Bucketing on the MasPar 68
3.3.3 Conclusions about the MasPar System 70
3.4 Creatures on the Thinking Machines CM2 71
3.5 Spiralling — Reducing the Movement Problem 72
3.5.1 Designing A Double Twisted Torus 75
3.5.2 Load Balancing 79
3.5.3 Construction and Routing 85
3.5.4 Higher Dimensional Spirals 88
3.5.5 A Proof of the Negative Offset Effect 92
3.6 A Transputer Implementation 93
3.6.1 Hardware 93
3.6.2 Software 96
3.6.3 Performance 99

3.7 Comparison of Performance between Implementations 101

4 Applications 104

4.1 Classic CA Problems 105

4.1.1 An Ideal Gas 105

4.1.2 A Model of Digital Logic 106

4.1.3 The French Flag Problem 108

4.1.4 The Firing Squad Problem 110

4.1.5 The Game of Life 110

4.1.6 Langton's Ant 112

4.2 More Complex Models 116

4.2.1 Simulating Road Traffic Flow 116

4.2.2 Water Current Analysis 118

4.2.3 A Model for the Spread of Sexually Transmitted Disease 121

4.2.4 Taxis as a Goal Orientated Navigation Strategy 123

4.3 Conclusions on the Application of Creatures 134

5 Discussion 135

5.1 Review 135

5.2 Further Work 137

5.2.1 Simulator Development 137

5.2.2 Extending the Model 138

5.2.3 Simulation Techniques 139

5.2.4 Applications 139

6 Conclusions 141

6.1 General Aims 141

6.2 Creatures 141

List of Figures

2.1 Emergent Insect Behaviour . . . 23

2.2 Ant foraging Turtle Procedures . . . 27

2.3 Ant foraging PATCH Procedures . . . 28

2.4 Ant foraging OBSERVER Procedures . . . 29

2.5 Fire PATCH Procedures . . . 30

2.6 Fire OBSERVER Procedures . . . 31

2.7 Rope TURTLE Procedures . . . 32

2.8 Rope OBSERVER Procedures . . . 33

3.1 Compiling a Rule for Multiple Platforms . . . 51

3.2 The NeXTStep Front End . . . 56

3.3 NeXT Performance . . . 59

3.4 Probability of success for $K = 64$ and $B = 64$ . . . 65

3.5 Probability of success for $K = 64$ and $B = 128$ . . . 65

3.6 Probability of success for $K = 64$ and $B = 256$ . . . 65

3.7 Broadcasting the Creature Map . . . 66

3.8 MasPar Mp-1 Performance . . . 67

3.9 Generating the Creature Map using buckets . . . 69

3.10 Bucketed Performance . . . 70

3.11 CM2 Performance . . . 72

3.12 CM2 Stationary Performance . . . 73

3.13 Wrapping buckets into a Spiral 74
3.14 Filling the Gap 76
3.15 The diameter of Twisted Meshes 78
3.16 Applying a Rectangular Load to a Traditional Grid 80
3.17 Applying a rectangular load 81
3.18 Applying a Rectangular Load to a Twisted Grid 82
3.19 Loads applied to an 8 grid 83
3.20 Loads applied to an $8 \times 8 + 5 \times 5$ grid 84
3.21 Two Similar Networks 85
3.22 Origin Vectors Spanning Space 87
3.23 A Three Dimensional Twisted Torus 89
3.24 The INMOS B042 Transputer Board 94
3.25 A Spiral Embedded within a B042 Board 95
3.26 Routing In The B042's Spare Coloumn 98
3.27 Performance of the Transputer Implementaion 99
3.28 Speedup using the B042 100
3.29 Performance of the Implementations 102

4.1 An Ideal Gas Simulation 105
4.2 A Digital Logic Simulation 107
4.3 The French Flag Problem 109
4.4 The Firing Squad Problem 111
4.5 The Game of Life 113
4.6 Langtons Ant 114
4.7 Langton's Ant Code 115
4.8 Road Traffic Flow 117

4.9 Currents on the river Oler 119
4.10 Simulation Water Currents 120
4.11 A Sexual behaviour Model 122
4.12 Diffraction Round an Obstacle 124
4.13 The Movement of Scent particles 125
4.14 The Cheese Creature 126
4.15 Rat and Random Rat 127
4.16 Persistent Rat and Frantic Rat 128
4.17 The Maze 129
4.18 Rat Performance 131

5.1 Improving I/O Performance 138

List of Tables

3.1 Some Viable Machine sizes 79

4.1 Rat Performance: Timesteps taken to reach the cheese 130

4.2 Statistical Results 131

Acknowledgments

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

Thanks to MasPar Computer Corporation for the use of an MP-1104 machine as part of the MasPar Challenge programme.

Access to the Thinking Machines CM-2 used during the course of this project was provided by the Edinburgh University Parallel Computing Center.

Thanks to Andy Tyrrell for proof reading — a task that no one deserves, and for his comments during the final stages of this research.

Special mention should also go to Richard Taylor for his unique role in this project.

Finally thanks to Paula, the cats (Gem and Biscuit), and every one in the Adaptive Systems Lab for putting up with me for the last three years.

Declaration

I declare that this thesis has been written by myself and the work reported within it is my own with the following exceptions:

1. The *Logo code found in section 2.1.5 is directly taken from the *Logo manual[54], and is the copyright of the original author.
2. The original development of the river mouth simulation was done by Dominique Snyers at Laboratoire I.A et Systemes Cognitifs, ENST de Bretagne, BP 832, 29285 BREST CEDEX, France.

and any other works indicted in the text.

The work has not hitherto been published with the exception of:

1. Parts of Chapters 2, 3 and 4 are based upon work discussed in *Creatures and Spirals: A data parallel object architecture*, by I. Stephenson and R. W. Taylor. Proceedings of the Euromicro workshop on Parallel and Distributed Processing 1994, IEEE Press.

1

Introduction

The demand for faster computers shows little signs of abating. However the traditional methods of improving performance are reaching their limits. It is no longer possible to simply increase component densities (the scales of which are approaching atomic dimensions) or to increase clock rates (where wavelengths are becoming comparable to the physical dimensions of the machine). Designing and fabricating such machines is increasingly difficult, and prohibitively expensive. Though hardware manufacturers may be able to offer diminishing rewards for the immediate future it is unlikely that the rapid expansion of compute power previously available to users at negligible cost can continue indefinitely.

Parallel computing systems[31][19] offer an effective, and virtually unlimited opportunity to increase performance at near linear cost. Regardless of the performance of a single processor, two such processors could *potentially* do twice as much work. In addition n processors will cost n times as much as a single processor. If n is large enough it is likely that many very cheap processors may offer better performance than a single large processor at lower cost.

Unfortunately performing tasks with many small processors is not as simple as it may be with a single processor. Despite this the promise of better performance requires that parallel computing be investigated and techniques developed to overcome the limitations of existing parallel systems.

1.1 The Problems of Parallel Processing

A large task may be broken down into a number of smaller sub-tasks. If this decomposition is done correctly each of the sub-tasks may be allocated to one processor and performed in parallel (perhaps one processor reads in data, another processes it, and another outputs the

results). Each processor performs a different task (or number of tasks) on its own set of data, and calls on the other processors to perform other parts of the global task as they are required. This form of parallel computing known as MIMD (Multiple Instruction, Multiple Data)[18] can work particularly well in distributed control systems where one processor can be made responsible for part of the system, communicating with other processors only when necessary. Each processor is independent of all others, and capable of performing useful work in its own right[41][42].

Unfortunately MIMD computing fails to deliver the extreme high performance required for theoretical and simulation work. MIMD requires that a problem be broken down into small functional units, each unit being specific to one virtual processor. However most problems are limited as to how far they can be broken down — the size of the units into which a problem may be split is known as the problem's grain size. Given a large number of processors it may not be possible to break a problem down such that one unit of useful work can be done on each processor. Further, with current technology the task of breaking a problem down requires skilled human intervention. Partitioning a task by hand is practical for perhaps up to ten processors. However it becomes manually intractable when tens of thousands of processors are considered — MIMD techniques are inherently difficult to scale.

Even when considering only a small number of processors the interactions between components can rapidly become exceedingly complex. Even trivially simple code may reach a state (known as deadlock) where each task is waiting for another task to complete before it may complete itself. Such a system will never complete any task, and hence wait forever. Such states are likely to occur unless the system is very carefully designed. The allocation of tasks to specific processors is also an issue which may drastically influence performance, again requiring expert intervention. The entire structure of the software is highly dependent upon the hardware and communications structures available.

The alternative to partitioning the task into sub-tasks is to allocate one element of computing resources to each data element in the problem, and perform identical operations on each one. This approach, known as Single Instruction Multiple Data (SIMD)[18] is not as generally applicable as the MIMD approach. However it is particularly well suited to the analysis of mathematical problems and high speed simulations[65] where large numbers of homogeneous local programs may be applied to the problem space[29][21][45]. Consider for example the problem of finite element analysis where identical "physical laws" are applied repeatedly across (say) an aircraft wing. Each task does little useful work by itself, but collectively the system may produce meaningful results.

SIMD systems are typified by Cellular Automata (CA)[66][69][63][23][17][14] where the programs are known as rules. The application of these very simple local rules produces complex global behaviour, suitable for use in many different fields. The advantages of cellular architectures over more conventional heterogeneous MIMD multi-processor organisation schemes may be summarised as:

- simple local behaviour; all cell behaviour is defined in terms of local properties, requiring minimal communications (it is almost impossible to deadlock SIMD systems, as all nodes in the system execute the same rule at the same time, with no provision for handshaking. Any form of deadlock which does occur will be at a higher level, preventing the "program" progressing, though the basic "rule" still executes successfully), and permitting simple, proven structures to be implemented on each node in either hardware or software.
- complex global properties; the behaviour of groups of these simple node processors may be forced to approximate to very much more complex global programming schemes.
- homogeneity; all nodes are identical both in hardware and software, and hence large systems may be built and programmed without reference to their size. Scaling of the system is through node duplication rather than a new, position dependent synthesis.
- locality; communications are local in nature (as they are in most problems), hence the performance of the system (per processor) need not degrade as the machine size is increased.

Despite these advantages, SIMD programming is still a specialised activity. Because the complexity arises out of simple rules in often unexpected ways it is generally difficult to understand why a system behaves as it does, or to modify the behaviour of a system to suit a particular application.

It is proposed that the main disadvantages of conventional data parallel models are a result of the models' static nature, forcing problems to be expressed in terms of space rather than the agents within that space — a road traffic flow problem would be expressed in terms of roads, rather than cars. The "location" of a cell is defined by its neighbors, and is fixed for all time — it is therefore difficult to express movement within the model. This thesis explores this area through a novel SIMD architecture known as "Creatures" which attempts to address this by describing systems in terms of their active elements. While still retaining

the attractive features of traditional SIMD/Cellular paradigms, Creatures allows problems to be described in a fashion which is far more intuitive than more traditional SIMD paradigms.

1.2 The Creatures Solution

The system is made up a number of *creatures* which exist in an otherwise empty, infinite space. Each creature has a clearly defined behaviour. A creature's behaviour is governed by its interaction with other creatures that it can "see". As a result of observing other creatures, a creature may:

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

This generalised form of creature processing describes many complex systems in a natural and understandable way. In order to model a system it is only necessary to describe the behaviour of the individual components and provide the initial conditions. Each element performs the same sequence operations, but its resultant behaviour is differentiated from its peers by the observations it makes. The complexity of the resulting system is dependent not on the complexity of the individual elements, but on the 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.

1.3 Thesis

The brief description of Creatures in the previous section outlines a novel approach to tackling the problem of parallel computing. In the following chapters the Creatures model will be refined and implemented. A number of simulations are used to illustrate its features, and comparisons will be made with other SIMD models.

1.3.1 Chapter 2 — The Creatures Model

A number of models of SIMD computation and simulation techniques are considered. Their strengths and weaknesses are discussed. The Creatures model is then presented: first in an informal fashion then in a more mathematical style. The model is demonstrated to be complete by the implementation of a Turing machine, and a simple equivalence to CA is demonstrated.

1.3.2 Chapter 3 — Implementing the Creatures Model

This chapter considers the implementation of Creatures on a range of platforms. The system is first developed in a naive serial form, and techniques for improving performance of this implementation are discussed. Implementation on two commercial parallel machines is considered, and the programming techniques are further refined. Finally the development of a semi-custom machine based on transputers is discussed, and the performance of all the implementations compared.

1.3.3 Chapter 4 — Applications

Having established a stable implementation of the Creatures model, the system is tested by the development of a number of simulations. These are either classic CA problems or demonstrations drawn from fields where the Creatures solution may be appropriate. In particular the final example (Taxis as a Goal Orientated Navigation Strategy) demonstrates the complete development of a simulation from concept, through implementation to the collection and statistical analysis of results.

1.3.4 Chapter 5 — Discussion

The strengths and weaknesses of the Creatures model are considered, taking into account the experience gained in implementing the model and developing simulations using it. A number of proposals are made as to how the model could be further explored and developed.

1.3.5 Chapter 6 — Conclusion

The major results of this work, and the conclusions drawn are reiterated.

2

The Creatures Model

2.1 Background

A number of systems attempt to address problems similar to those that the Creatures model deals with. By examining these systems their strengths and deficiencies may be identified, enabling the Creatures model to provide a more useful tool to developers of simulations. In addition to Cellular Automata the programming paradigms and simulation techniques of Discrete Event Simulation, Mirror modeling, and *Logo must be considered. The field of artificial life also touches upon the problems of massively parallel agent based systems, though often in an informal fashion. As such it provides a set of problems which a successful simulation tool should be able to handle in an efficient manner.

2.1.1 Cellular Automata

Cellular Automata (CA)[63][69][66][23][17] 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. The system is defined by a triplet: $<$S(tate) ,N(eighborhood),T(ransition function)$>$ for each cell. Normally N and T will be the same for every cell in the system — should this not be the case a more general form of N, T and S may be derived such that N and T are uniform throughout the system, hence non-uniform cases need not be considered as interesting.

CA have many attractive features for the engineer and programmer attempting to build a highly parallel system and describe its behaviour. Attempting to specify the hardware and software of every node in a system becomes increasingly difficult as the number of nodes increases. If each node requires special attention by the programmer then a limit on the

number of processors is quickly reached. The homogeneity of CA enables any number of processors to be controlled with a single set of instructions. This in turn leads to the property of scaleability: the number of processors is no longer specified as part of the design of hardware or software (at least at the logical level), and hence the size of a machine, or simulation may be simply increased as required.

The behaviour of a CA type system is specified at a very primitive level compared to the type of code typically found in traditional simulation software. This simply defined local behaviour may be closely tied to the physical systems being modeled. A system is described by defining the simplest properties of the elements of which it is composed and observing the consequences of such rules[65]. It is therefore no longer necessary to make global assumptions about the behaviour of a system. By establishing a structural isomorphism[72] between a simulation and a (hopefully) equivalent system the predictive strength of the simulation is greatly increased. Should the global simulation not behave as expected then the error relates to the description of the basic elements. Using this technique the properties of the elements relevant to the systems behaviour may be identified.

Despite these strengths, attempting to describe many physical systems using CA introduces a number of undesirable complications and overheads. These are primarily related to the the spatial nature of CA. The cells of a CA system have a (typically) small set of neighbours which is fixed for all time. Such locality is well suited to the description of points in space, but is difficult to reconcile with objects within that space which may be mobile, and hence have a constantly varying set of neighbours of unpredicatable size. A CA based simulation must therefore be formulated in terms of the space it occupies. Unfortunately for many real problems space simply provides a substrate in which active elements of the system may exist. As a result the basic elements which must be described as part of the CA rule are not the elements one would naturally use to describe the system. This is conceptually difficult for those not experienced in such programming tasks. For example consider the implementation of a road traffic simulation: a CA description of such a system must consist of cells which represent roads. A road may hold a car or not hold a car, and at appropriate times pass the car to an adjacent section of road. While clearly such a system is workable, the solution will be less than intuitive and somewhat convoluted.

In practise this problem is compounded by CA being essentially shared memory systems. Communication between adjacent nodes is by setting flags in a node's state, in the hope that an adjacent node will observe the flag and act appropriately. As a result much programming effort goes into providing handshaking between cells to simulate the movement of interesting data elements (these being cars in the above example). It is necessary for the

programmer to introduce synchronization mechanisms to ensure data integrity (for example to ensure that two adjacent nodes do not both believe the car has moved into their location) much like those found in course grained systems (atomic locks, semaphores, conversations etc[9]). Techniques such as the Margolis neighborhood[63] have been developed to address this problem, but at the cost of further removing the physical system from the implementation. This complication is often neglected by novice programmers leading to incorrect simulations, and the increased complexity introduced into the system by the additional code (which may perhaps engulf completely the original physical model) is likely to induce mistakes from all but the most experienced users.

It may appear to the naive observer that CA offer good load balancing, as each cell performs identical operations, and hence requires the same compute time as all other cells. However *interesting events* typically take place in only few areas of space, leaving many of the processing elements performing useless operations. The spatial homogeneity of CA forces work to be done even when the operations are clearly redundant. To refer back to the road traffic example: accidents will only occur when cars meet. If one area of the simulation contains no cars then there is no useful work to be done. If a cell contains one car then there is a little work to be done in moving the car to an adjacent location. Should a cell contain many cars then there is much work to be done detecting collisions, and processing each of the cars' individual movements. In a CA system all nodes would be forced to perform all the collision detection routines even though the operation is pointless for (perhaps) most cells. At an instruction level, the system is balanced, but this is far from being the case when a metric of "useful work" is used.

2.1.2 Discrete Event Simulation

DEVS[72] and Next Event Simulation[73] attempt to shift the emphasis of cellular systems towards *interesting events* — those actions and interactions which drive the behaviour of the system rather than the continuous steady state that many cells of a CA typically may find themselves in. In particular DEVS models are significantly better than CA for simulating the movement and collision of particles within space.

The description of a CA is extended, where instead of being defined by the standard <S(tate), N(eighborhood), T(ransition)> triplet, S includes a value which is the time at which the cell will next update (a further parameter SELECT acts as a tie breaker should two cells wish to update simultaneously — the system operates in continuous time, so theoretically no two events may occur simultaneously). Cells only update when the time held in S is

reached, or they are updated by a neighbor. In addition, T (the transition function) can *set* the value of cells in the neighborhood; it transforms the states of the cells in N. This greatly simplifies movement, as a "particle" may almost directly propel itself across space.

For example consider the simplest case of a particle moving across space from left to right. One cell at the far left may initially contain a particle, and be scheduled to update imminently, while all other cells are essentially idle, being empty with no updates scheduled. Upon updating a cell can mark the cell to its right as containing the particle, schedule it to be updated, and then return itself to the idle state. By such a mechanism the particle will move across space with a minimum of computational effort.

This is indeed much simpler than an equivalent model implemented in a CA style system, and the introduction of time into the model may be valuable in relating a simulations results to the real world. Because no two events of the simulation occur at the same time many of the difficulties of maintaining data integrity are resolved. The ability of a cell to change the state of a neighbour rather than just observe it allows cells to "drive" data across the space.

However the encapsulation of data that is present in the CA model has been lost. The ability to change data in another cell, though a practical simplification for the programming of some simulations, may limit the implementation on non-shared memory hardware. A more significant limitation from a parallel processing point of view is the model's use of coroutines to ensure data integrity — no two events occur at the same time, so in a naive implementation there is no parallelism, though extensions to the model, and more complex data flow analysis of the system make parallel DEVS possible[3].

DEVS simplifies the programming of agent based models by simplifying the communication between adjacent nodes so data can securely be passed between them. However a discrete event simulation is still fundamentally spatially based: considering the previous example, it is now easy to move cars around, but the roads must be programmed to do it. Though the practicalities of building such a system are much simpler than with CA the programming must still be twisted to fit the DEVS model.

"Space" in DEVS models is frequently used simply to represent part of the system, without any reference to physical space. A location may represent (for example) a garage, which would hold a number of cars requiring repair. From this location cars may be moved to a scrap location or to a road location depending on the operation that the garage chooses to perform on them. Such systems are far from homogeneous, each cell potentially having its own unique rules and connectivity. However such systems are merely a practicality, and could be implemented as a more complex rule applied equally to all cells. Therefore

simulations of this type are computationally of little theoretical interest, and have limited relevance to Creatures.

2.1.3 Artificial Life Systems

A large subset of research in the field of artificial life tries to model systems where the collective behaviour of many individual elements is more complex than would be apparent from examining a single element. Cellular Automata are often used, but systems developed are often based on the collective behaviour of mobile agents analogous to that often found in insect colonies[13][20][55][5] — CA (for reasons discussed previously) are less than ideal for this type of simulation. The term swarm behaviour is often used in ALife work to describe systems where the collection of agents displays collective intelligence beyond that found in any individual.

Systems developed by ALife researchers show the power of large numbers of simple elements cooperating according to very simple rules to produce complex behaviour. The Creatures model should be able to simply describe the kinds of problems encountered in developing such simulations, allowing systems to be rapidly developed by specifying the required local behaviour without recourse to low level programming. Unfortunately ALife is focused upon specific tasks, and few systems make distinction between the simulation being run, and the model being used — programs are typically written in an ad hoc fashion to describe a particular physical system without reference to simulation techniques that could (or should) be applied. The exception to this being Mirror (section 2.1.4). Though the power of agent based simulation may be observed through ALife work, little is being learnt about the simulation techniques involved.

To illustrate the kind of problems being tackled in ALife research consider the action of tunneling conducted by ants during nest building. Ants may dig with a certain probability, and upon doing so deposit pheromone. This increases the likelihood that an ant will dig there in the future. As a result they dig branching tunnels, rather than unstructured holes. In addition they will dig more where digging is successful (say as in softer soil), rather than where it is fruitless (up against rock). One may expect that this positive feedback mechanism would result in an explosion of digging activity. However as the nest size is increased, as a result of excavation, the density of pheromone decreases and digging activity is reduced, producing a correctly sized nest for a given population. Such a simulation has been implemented using the Creatures model.

Creatures

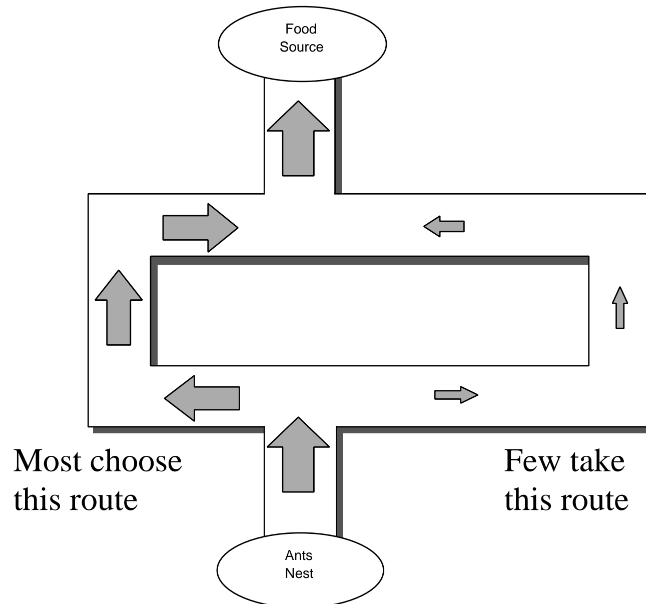


Figure 2.1: Emergent Insect Behaviour

A second form of collective behaviour may be observed by considering the situation where a path which leads to food branches (figure 2.1). At first the ants take random paths, but as they go they lay a trail behind them. On the return route there is more pheromone on the short branch, as more ants will have already passed that way, so more ants go that way. This in turn leads to more pheromone on the short path, and the resultant positive feedback ensures that almost all the ants take the optimum route. Initially few ants will go along the path at all. However if food is found, more and more ants will follow the path as the chemical trail builds up.

Closely related to artificial life is the subject of anti-chaos where simple, but seemingly disordered local behaviours with arbitrary initial state evolve into highly ordered structures. A simple example of this known as Langton's ant[60] is considered in section 4.1.6.

2.1.4 Mirror

Probably the most effective simulation of socioinformatic processes[1] may be seen in the Mirror system[32][33][34][35]. This simulation environment provides a powerful platform,

[1] "Socioinformatic processes are defined here as informatic processes which cause behavioural differentiation among individuals who are basically the same, thus generating a social structure in groups of individuals"— Hogeweg (1983)

upon which a number of simulations have been built.

The most successful (and well documented) Mirror simulation is probably the study of bumble bee colonies[36]. This produced theoretical results which were then verified against real colonies, providing new understanding of the bees behaviour. Certain species of bee drive the queen out of the nest at a particular time of year. However the mechanisms by which this occurs were not clear. A Mirror simulation was developed, the prime component of which was the dominance relations between members of the nest. Whenever two bees met a confrontation was simulated, taking into account the importance of each participant within the nest. Awards of rank were made depending on the outcome of the challenge. It was found that bees in the center of the nest rose in rank, not by winning many challenges — they frequently lost challenges (for example to the queen), but on the occasions that they did win the rewards were greater. Conversely other bees could win many challenges against lowly colony members, but would gain little in reward. After a certain period of time (dependent upon simulated conditions) the dominance relations within the nest would evolve such that the queen could no longer retain control of the nest, and would be forced out. Having demonstrated the concept in a "Mirror world" it was possible to observe identical actions taking place in real colonies (the simulations having told observers what to look for).

Mirror users advocate a TODO model of animal behaviour: individuals simply do what there is "to do". However the implementation of these simple behaviours is somewhat more complex. The system is written in LISP, and models both continuous time and space. DWELLERs exist in a SPACE, and are further defined by a private skinSPACE (the MIRROR terminology for a DWELLERS internal state), which contains each DWELLER's state. The system also contains DEMONs. Each DEMON may have a number of TARGET conditions. When these are met, the DEMON can force a TIE (a DWELLER) to perform an action. Otherwise DWELLERs cannot observe DEMONS. A SENTINEL is a particular form of DEMON which is bound to a single PATCH (part of SPACE), and hence can observe DWELLERs within that patch. This allows configurations of DWELLERs to be examined, and meta-level structures to be defined.

Agents are revived periodically, by other agents which hold pointers to them (usually a DEMON). DEMONS are always revived when variables they attach to are accessed. They may then decide whether their TARGET is met, and if so activate/revive/influence their TIE. Upon being revived, an agent will know what action caused the revival and may modify its behaviour accordingly.

The system is somewhat complex (only the most basic form has been described here), but it has been clearly shown that by defining local behaviour of individuals in this fashion, complex behaviour can occur on a global level and that knowledge so derived may be successfully applied to real systems. Unfortunately the complexity of Mirror modeling has limited the application of the system by other researchers to fields where it may have proved useful.

The successful division and computational model from biological simulation, and the experimental power afforded to users of the Mirror model must be aspired to by the Creatures model. However it also demonstrates the danger of complexity, as this has prevented the exploitation of Mirror where it should have proved invaluable.

2.1.5 *Logo

This variant of the LOGO language[54] developed for the Connection Machine[29] provides multiple TURTLEs which may be instructed to move in an SIMD (Single Instruction Multiple Data) fashion. They move over an array of fixed elements, called PATCHes which may be used as a traditional CA. Finally there is provision for serial code (the OBSERVER) which runs in parallel with the TURTLEs and PATCHes.

The system is an experimental programming environment, which allows naive users to easily operate a connection machine, and to some extent exploit the parallelism afforded there. In that sense it may be successful, but as a basis for serious simulation it has a number of shortcomings.

- The model is overly rich in providing TURTLEs, PATCHes and OBSERVERs. These structures do not operate together in a consistent fashion — most obviously TURTLEs operate in a continuous, infinite space, while PATCHes form a discrete, finite space (*Logo PATCHes being only a pale imitation of the general entities which drive Mirror models).
- Some of the “Primitive” operations are unduly heavyweight, allowing the real issues of a simulation to be glossed over. The “Sniff” operator for example considers the value of a variable over a number of PATCHes and calculates the local gradient of the variable (in two dimensions). While this may be a useful thing to do, it is difficult to consider it as a “primitive” given the amount of sampling and computation involved.
- Some of the parallel operations are ambiguously defined. By allowing an object to

perform arbitrary operations upon another, the order of evaluation becomes important. This may potentially reduce parallelism.

- There is little enforcement of locality in the model — any object may communicate with any other provided it has a method of referencing it. In particular the OBSERVER exists in a totally global fashion, making it difficult to ensure the integrity of simulations developed.

Of the problems in the *LOGO system, the first is the most interesting. Examining the code provided with the system fails to provide any illustration of PATCHes and TURTLEs operating together in an effective manner. Either PATCHes are used as CA or TURTLEs are used to provide an agent based modeling environment, with little useful interaction between the two modelling techniques. A number of examples taken from the *LOGO programming manual are considered here.

The "Ants" example (figures 2.2, 2.3 and 2.4) describes a system of ants represented by TURTLEs which move around searching for food. Trails are marked by forcing PATCHes to carry pheromone. The patches reduce their pheromone level by a factor at each time step. There's also a nest, which operates in a similar fashion. The PATCHes are a somewhat inefficient way to hold pheromone, as they exist at all space but carry useful information only over a very small subset of that space. It would be equally practical to use TURTLEs to hold this data by creating a class of TURTLE which remains stationary signifying the presence of pheremone at a particular location. The Observer initializes the system, and then adds new Ants after a while. The existence of an observer is useful but the work it performs in this case could just as easily have been implemented by a TURTLE.

By contrast the "Fire" example (figures 2.5 and 2.6) uses only PATCHes, and operates entirely as CA. PATCHes read their neighbors, then set their own state. The final example "Rope"(figures 2.7 and 2.8) represents a rope with one end driven sinusoidaly, the other end being fixed. The mid points look to the TURTLE on their left and right, and calculate their velocity accordingly (PATCHes play no part in this model). Though it appears at first sight the model relies on only local communication (each TURTLE observes its immediate neighbors) the simulation relies on any agent being able to talk to agents that it can identify (by somewhat unqualified means), even when the two agents have moved apart.

The available documentation fails to provide a single example where TURTLEs, PATCHes and the OBSERVER are all used together in an effective fashion. Any single one of these mechanism is computationaly complete, and hence the inclusion of all three is an

```
to turtle-demon
ifelse :my-food > 0 [look-for-nest][look-for-food]
end

to look-for-nest
ifelse ask patch-here [:nest?]
    [make "my-food 0 rt 180 fd 1 setc red]
    [demand patch-here
        [make "pheromone :pheromone
                       + ask turtle-here [:pheromone-drop-size]]
    if :pheromone-drop-size > 0
        [make "pheromone-drop-size :pheromone-drop-size - 0.6]
    seth uphill "nest-scent right random 40 left random 40 fd 1]
end

to look-for-food
ifelse ask patch-here [:food > 0]
        [make "my-food 1
         demand patch-here [make "food :food - 1]
         make "pheromone-drop-size 35 setc yellow
         rt 180 fd 1]
        [make "pheromone-here ask patch-here [:pheromone]
         if :pheromone-here < 3.0
                 [ifelse :pheromone-here < 0.2
                            [rt random 40 lt random 40]
                            [seth uphill "pheromone]]
         fd 1]
end
```

Figure 2.2: Ant foraging Turtle Procedures

```
to setup
make "food 0
make "pheromone 0
ifelse (dist 0 0) < 5 [make "nest? true make "nest-scent 1000]
          [make "nest? false make "nest-scent 1000 / dist 0 0]
if (dist 20 0) < 4 [make "food 1]
if (dist -24 -36) < 5 [make "food 1]
if (dist -44 44) < 6 [make "food 1]
set-diffusion-rate 0.15
update-colors
end

to patch-demon
diffuse "pheromone
make "pheromone :pheromone * 0.95
update-colors
end

to update-colors
ifelse :nest? [setc purple]
              [ifelse food > 0 [setc blue]
                      [scale-color green :pheromone 0 2]]
end
```

Figure 2.3: Ant foraging PATCH Procedures

```
This program simulates the foraging behavior of ants.

Ants search for food (shown in blue), then leave a pheromone
trail as they return to the nest (shown in purple). Other ants
follow the trail to the food, then reinforce the trail on
their way back to the nest.

Instructions for use:
 * Type SETUP to setup the ants.
 * Then type STARTD (or GOFOR <number>) to start the demons.

Notice how the colony as a whole seems to exploit the food sources
systematically, starting with the closest food source then working
outward.

to setup
clear-all
reset-clock
fep [setup]
make "total-ants 100
end

to observer-demon
if clock < :total-ants
     [create-custom-turtle 1 [setxy 0 0
      set-sniff-distance 3.0
                              make "my-food 0]]
end
```

Figure 2.4: Ant foraging OBSERVER Procedures

```
to patch-demon
if red? [burn-a-bit
 demand patch 0 [if color = green [setc red]]
 demand patch 90 [if color = green [setc red]]
 demand patch 180 [if color = green [setc red]]
 demand patch 270 [if color = green [setc red]]]
end

to red?
(color >= 4) and (color <= 10)
end


BURN-A-BIT makes the trees become darker as they burn

to burn-a-bit
if color > 4 [setc color - 1]
end

to border-cell?
(xpos = left-edge) or
    (xpos = right-edge) or
    (ypos = top-edge) or
    (ypos = bottom-edge)
end
```

Figure 2.5: Fire PATCH Procedures

```
This program simulates the spread of a forest fire.
The spread of the fire depends critically on the density of trees.

Instructions for use:
 * Type SETUP <number> to setup up the forest.
 * Start the demons with STARTD to watch the fire spread.

Things to try:
 * Try different densities of trees (different inputs to SETUP).
 * Try different resolutions (use SET-SCALE).

to setup :percentage
fet [die]
clear-patches
fep [if :percentage > (random 100) [setc green]
         if xpos = (left-edge + 1) [setc red]
         if border-cell? [setc blue]]
end

As an alternative to starting the demons,
you can use the BURN procedure

to burn
fep [patch-demon]
if (patch-subtotal [color = red]) > 0 [burn]
end
```

Figure 2.6: Fire OBSERVER Procedures

```
to setup
seth 0
sety 0
setx who + left-edge
if xpos = left-edge  [setc green
                      deactivate-demon "rope-demon]
if xpos > left-edge  [setc red
                      deactivate-demon "input-force-demon]
if xpos = right-edge [setc blue
                      deactivate-all-demons]
make "yvelocity 0
make "yaccel 0
make "spring-constant 0.3
make "friction 0
end

to input-force-demon
sety :amplitude * sin ask observer [:frequency * clock]
end

to rope-demon
make "yaccel :spring-constant * (((ask who - 1 [ypos]) - ypos)
                               + ((ask who + 1 [ypos]) - ypos))
make "yvelocity (:yvelocity + :yaccel) * (1 - :friction)
fd :yvelocity
end
```

Figure 2.7: Rope TURTLE Procedures

```
This program simulates waves on a rope.

The rope is composed of turtles. Each turtle acts as if
it is connected to its neighbors by imaginary springs.

The left end of the rope moves up and down sinusoidally.
The right end of the rope is fixed.

Instructions for use:
 * Type SETUP to setup the turtles.
 * Then type STARTD (or GOFOR <number>) to start the demons.

Things to try:
 * Vary the frequency of the input force. Try: MAKE "FREQUENCY 2
 * Vary the friction. Try: FET [MAKE "FRICTION .01]

to setup
clear-all
create-turtle 2 * right-edge
reset-clock
fet [setup]
make "frequency 4
make "amplitude 32
nowrap
end
```

Figure 2.8: Rope OBSERVER Procedures

unnecessary complication. *LOGO's aim is to provide a programming environment in which inexperienced programmers may exploit the power of parallel computation. From such a perspective the overly rich set set primitives may be considered an advantage allowing the programmer to select the techniques which best suit the problem at hand. From a theoretical viewpoint however, as a tool for understanding the nature of computational systems, *LOGO is unlikly to be of benefit. PATCHes, TURTLEs and OBSERVERs each individually provide a model of computation which is sufficiently complex that it defies all but the simplest analysis. Only by stripping such models to their simplest components can there be any hope of understanding the nature of systems they may describe.

2.2 Defining Creatures

2.2.1 An Informal Description of the Creatures Model

A Creatures simulation consists of a number of active elements (creatures) which exist in a discrete space, and update synchronously at discrete time intervals. A simulation is defined by specifying the behaviour and initial location of each creature. The complexity of the simulation is dependent not on the complexity of the individual elements, but on the very large numbers of such elements, and the interactions between them. In developing such simulations, the model allows the user to experiment with and isolate the *characteristic* properties that determine the system's behaviour[2].

Each creature has its own state which only it may modify, though some of this state may be made visible to other creatures. The behaviour of a creature may depend on its own state, and the external state of any other creatures in the same location. This behaviour may include changing its own state, creating new creatures, and moving to an adjacent location.

The model may be summarised as follow:

1. A simulation shall be composed of agents which shall update their state synchronously.

[2]During this section the following definitions shall be used:

- "system": the real world structure being simulated.
- "model": the simulation techniques being applied ie Creatures.
- "simulation": the rules and resultant behaviour which attempt to approximate the "system".

2. An agents state shall be represented as a finite number of parameters. These may be used to influence any future behaviour.

3. An agent's state shall be private to that agent with the exception of one parameter, known as that agent's type.

4. An agent's location in space shall be defined by a subset of the agents state, which the agent may not directly operate upon.

5. The space in which agents will exist shall be discrete, uniform, and infinite.

6. An agent may indirectly change its location by moving to an "adjacent" location, *relative* to its current location.

7. Interactions between agents shall be limited to the detection of other agents, by observing the number of agents of a given type.

8. Agents may only observe other agents whose location is identical to their own.

9. An agent may create any number of new agents in its current location. These may be of any type, as specified by the parent. This is the only data the parent may pass on.

These rules describe a simple model which can be implemented efficiently, while providing a sufficiently rich environment to develop simulations, of a similar level of complexity to those found in cellular automata.

The parameters of the model may justified as follows:

A simulation shall be composed of agents which shall update their state synchronously. The emphasis of a model should be on the active agents within a system rather than the space which they occupy (as illustrated by the limitations of Cellular Automata). The use of discrete time may be regarded as a global exchange of information. However discrete time systems are generally easier to program and to implement. The use of a programming model ensures that the synchronous nature of simulations is explicit. Without predefined model of computation, the nature of time within a simulation may be unclear, increasing the likelihood that global synchronization may be accidentally misused.

An agents state shall be represented as a finite number of parameters. These may be used to influence any future behaviour. Each creature essentially carries its own data space with it. This encapsulation provides the necessary separation between agents to allow

the parallel implementation of a system. It also simplifies the development of a creatures behaviour specification, as a behaviour may be developed for one or a few creatures, then scaled to operate on many creatures.

An agent's state shall be private to that agent with the exception of one parameter, known as that agent's type. By separating the private and public state of a creature the variables that are used in interactions are made explicit. A creature may only interact based upon the public state of other creatures, without concern that information may accidentally be distributed due to programming errors. Consider for example ants searching for food — an ant may remember the location of a piece of food, but has no explicit method of communicating this information to another ant. By holding the location in its private space any collective behaviour is proven not to be a result of direct observation.

Typically "type" will be a simple scalar quantity (if only from the practical perspective of implementing non-scaler types). However there is no theoretical reason why type should not be a vector of arbitary complexity. The distinction of the type paramenter is that it is observable, and hence is imprtant in the communitation of information between agents.

An agent's location in space shall be defined by a subset of the agents state, which the agent may not directly operate upon. By hiding a creature's absolute location the existence of special locations is prohibited. This prevents creatures from migrating to certain hard coded locations. In a system being simulated it is unlikely that the real world elements would be able to identify their location without reference to external stimuli. This does not prevent creatures counting their own movements to create their own personal coordinate system.

The space in which agents will exist shall be discrete, uniform, and infinite. Discrete space greatly simplifies the interaction of creatures, and the implementation of the model. A creature is either in the same location as another or in a different location — the issue is always clear cut. The uniformity of space provides the simplest of environments in which creatures may interact. Should it be necessary to indicate spacial features, creatures may be placed in locations to act as signposts. The provision of explicit space based operations would therefore be redundant (as it is in *Logo). By making space infinite there is no need to consider boundary conditions unless explicitly required by a specific simulation (which may define boundaries by means of a specific creature type).

Creatures

An agent may indirectly change its location by moving to an "adjacent" location, relative to its current location. By limiting the movements of a creature locality is encouraged, and the required connectivity of space is reduced (which may be important for an efficient implementation). Movement must be specified relative to a creature's current location as there is no method of specifying a specific location. This ensures that part of a simulation which operates correctly will continue to function in a new location, facilitating easier development and debugging.

Interactions between agents shall be limited to the detection of other agents, by observing the number of agents of a given type. Specification of the interaction of creatures is a potential source of complexity in a model such as this. Limiting the interaction to this simplified form may make certain kinds of simulation more complex. However an alternative more elegant solution has not been found.

It is essential that interactions be limited to observation due to the parallel nature of the system (the order of evaluation of creatures should be hidden from the user). Such a model ensures that each data element within the system is writable by only one creature — the creature that holds that data as a parameter within its state. If a pair of creatures were allowed to perform a write operation (theoretically at the same time) on the same data it is unlikely that both would be able to succeed in any predicatble fashion. The limited form of interaction available allows a snap shot of a spacial location to be produced, and used as the input for each creature's transition function. The "real" agents in the location may then be processed entirely independently, their instantaneous external state having being captured.

Agents may only observe other agents if their locations are identical. This greatly simplifies the interaction of creatures, as the observation of other creatures is now associative and distributive. If A can see B then B can see A, and if A can see B and B can see C then A can see C. Any operation performed by a set of creatures is self contained, and is (in a weak sense) participated in by all creatures in the location. This constraint also prevents the hidden sharing of data in a simulation — communication at a distance does not occur without a carrier of that information moving between the two locations. By explicitly coding this into a simulation the true nature of the interaction is made clear.

The alternative to a strict locality would lead to vastly increased complexity in the specification of behaviour — it must be considered whether the distance at which interactions may take place is a function of the observer, the observed, both or neither. Because space is discrete some form of neighborhood function based upon a number of these parameters

would need to be specified for each simulation. The added complexities do not justify the extension of the model.

An agent may create any number of new agents in its current location. These may be of any type, as specified by the parent. This is the only data the parent may pass on. During the development of the Creatures model a number of systems were developed which allowed more complex information to be passed on to offspring. However as the system developed the mechanisms necessary to perform this operation became increasingly complex, from both a developer and user's perspective. In reaction to this, the current form of agent creation was developed. Although this greatly restricted the model it has proved possible with experience to convert all previous models to this form despite their apparent complexity. The restricted form of producing offspring generally produces simpler simulations.

A number of these points would appear to overly restrict the model and perhaps prevent the development of certain types of simulation (for example: that creatures may only observe creatures with identical locations might appear to limit the transfer of information around a system), in practice, it has proved possible to overcome these difficulties without compromising the integrity of the model. Techniques applicable in one simulation are often useful in others, and the examples found later in this report should illustrate some of the common program structures (in addition to clarifying the details of the model).

2.2.2 A Formal Definition of the Creatures Model

The Creatures model may be defined in a more rigorous fashion by the use of set theory, and finite state machine techniques[30][46][37]. In doing so an accurate definition of the model will be produced. In addition to clarifying the previous informal description, a mathematical form of a simulation may potentially be transformed to a more desirable description.

A single creature at a single instant in time (referred to as a creature instance) is a state machine

$$c = [s, f_s, f_o] \tag{2.1}$$

Where s ranges over S, the set of states attainable by a creature. States S are in fact the triple $[\lambda, \tau, \sigma]$ where λ is the location, τ the creature's *type* and σ the internal state of the creature.

The output of a creature (c) represents the creature's children which will be placed into

the system at the next time step. This set of creature instances is formed by applying the output function f_o to $s(c)$ and an input to the creature from the rest of the system:i. This input (which is defined in a following section) is based on the set of creatures which have a location $\lambda = \lambda(c)$. All creatures in the set $f_o(s, i)$ have the internal state σ_0 (a state globally defined by rules of the system) and location $\lambda(c)$ (the location of the parent creature). τ may be range freely for each child, specified by the output function of the parent.

Similarly the next state function f_s is applied to $s(c)$ and i to give a new creature instance. This new instance is used to represent the creature at the next time step — the old creature instance only exists at a single point in time (the current timestep), and is therefore discarded once its output and next state have been calculated. This new instance is distinct from the child creatures of the output function, as both τ and σ elements of its state may be freely specified. In addition λ may be indirectly manipulated.

Space

The location of a creature λ is an n-tuple of integers (the examples described elsewhere in this thesis are typically of order two).

$$\lambda(c) \stackrel{def}{=} \text{The location of creature instance } c \tag{2.2}$$

A creatures location defines the set of creatures it may interact with (which are used to produce the input to the transformation functions f_o and f_s). This is expressed in terms of the "Can See" relation $\odot$ defined such that:

$$c \odot d \stackrel{def}{=} \lambda(c) = \lambda(d) \tag{2.3}$$

where c and d are creature instances. A creature may see another if it occupies the same location in discrete space.

The concept of location is also used to restrict the movement of creatures. Each location l has a neighborhood

$$\lambda^*(l) \stackrel{def}{=} \text{The set of locations a creature at location } l \text{ may move to in a single timestep} \tag{2.4}$$

The neighborhood operator is associative with translation such that:

$$\{n + v | n \in \lambda^*(l)\} = \lambda^*(l + v) \tag{2.5}$$

where n, and l are locations, and v is any vector of appropriate dimmension.

Creatures

Next state functions always produce creature instances such that:

$$\lambda(f_s(s(c),i)) \in \lambda^*(\lambda(c)) \tag{2.6}$$

It is required that the definition of f_s be free from references to absolute locations — if the entire system is translated its behaviour should remain unchanged. This is achieved by defining the f_s in terms of a function f'_s with the property that:

$$\lambda(f'_s(\tau(c), \sigma(c), i)) \in \lambda^*(\vec{0}) \tag{2.7}$$

where $\vec{0}$ represents an origin location such at $\forall l : l + \vec{0} = \vec{0} + l = l$. The operation of addition of vectors and creature instances is defined such that:

$$[[\lambda, \tau, \sigma], f_s, f_o] + l = [[\lambda + l, \tau, \sigma], f_s, f_o] \tag{2.8}$$

The new creauture instance is a translation by l, all other parameters being unchanged. f_s may therefore be ensured to be free from absolute refrences to location by defining it as:

$$f_s(s(c),i) = f'_s(\tau(c), \sigma(c), i) + \lambda(c) \tag{2.9}$$

The notation $\lambda^*(c)$ may be used as shorthand for $\lambda^*(\lambda(c))$, representing the locations c's next state instance could occupy. The abbreviated form is more readable, and indicates better the intention of the statement.

The output function always produces creatures at the location of the parent:

$$\forall d.d \in f_o(s(c),i) : \lambda(d) = \lambda(c) \tag{2.10}$$

Once again it is necessary that the function is free from absolute references to location and hence a function f'_o is created:

$$\forall d.d \in f'_o(\tau(c), \sigma(c), i) : \lambda(d) = \vec{0} \tag{2.11}$$

f_o may then be defined as

$$f_o(s(c), i) = \{d + \lambda(c) | d \in f'_o(\tau(c), \sigma(c), i)\} \tag{2.12}$$

A neighborhood $\lambda^*(l)$ may contain a location DIE. This single location will generally be adjacent to all other locations and has the additional special properties that $DIE + l = DIE$, and $\lambda^*(DIE) = \{DIE\}$ (a creature at this location may never move to any other location). This is used in the following sections to remove creatures from the system.

Observations

At an instant in time a creatures system is a set of creature instances u. The input i of a creature c instance is based upon the subset of creatures in u that may be observed by c. Only a subset of a creatures process space may be observed — the creatures type: $\tau(d)$, as this ensures that information which is private to a creature remains so.

A creature c may observe a subset of the system u in which is exists:

$$u(c) = \{d : d \in u \wedge c \odot d\} \tag{2.13}$$

$u(c)$ represents the system u as perceived by creature c.

The information available to a creature is therefore the collection (*not* a set, as duplicates are allowed) of creature types:

$$i(c) = \{\tau(d) | d \in u(c)\} \tag{2.14}$$

This collection is used as part of the input to the transition functions to influence a creatures next state and output.

Time

The Creatures model has so far been considered to exists at a single point in time. Each creature within that set produces an output which may be collected to produce the output of a complete system.

$$f(u) = \{c : (\exists d.d \in u \wedge c \in f_o : d) \vee (\exists d.d \in u \wedge c = f_s : d \wedge \lambda(c) \neq DIE)\} \tag{2.15}$$

The output of a set of creatures is the union of all creatures that are born, and all creatures that do not die.

A simulation U represents a complete creatures system existing over time. Having been initialised to the state U_0 it exists at discrete time steps $t > 0$. U_t is the set of creature instances representing the system at time t.

$$U_t = f(U_{t-1}) \tag{2.16}$$

Information preserving creature definitions may be prepared - in these special cases "negative" time may be introduced into the state history. This is most easily observable when f_o and f_s are restricted such that

$$\forall c.(f_o : c = \emptyset \wedge \lambda(f_s : c) \neq DIE) \tag{2.17}$$

That is no creatures may be created or destroyed. Consider a sytem where all creatures move east at each time step — given the state u of such a system a unquie state u' may trivially be constructed such that $f(u') = u$. Therefore given a state U_0 it is equally possible to contruct a unique state U_{-1} such that $(f(U_{-1}) = U_0$, even though the system was only considered to be created at time $t = 0$.

Creature Traces

In addition to slices through the time axis, we may also consider the continuous existence of a single creature as it exists through time as:

$$C_n = \begin{cases} \text{undefined} & n < 0 \\ [\![\sigma_0, \tau, \lambda]\!], f_s, f_o] & n = 0 \\ f_s : C_{n-1} & n > 0 \wedge \neg\exists m.(m < n \wedge \lambda(f_s : C_m) = DIE) \\ \text{undefined} & \exists m.(m < n \wedge f_s : C_m = DIE) \end{cases} \tag{2.18}$$

Where $n = 0$ is the time at which creature is first created, either by the initialisation of the system ($C_0 \in U_0$) or by being given birth to by another creature ($C_0 \in f_o(d) \wedge d \in U_{t-1}$). At this time ($n = 0$) the creature is created in an initial state. At each subsequent time step the creature calculates its own next state, until such time it decides to die. Outside of this time interval the creature trace is not defined.

A creature (C) exists at time t if

$$C \cap U_t \neq \emptyset \tag{2.19}$$

It may also be noted that these definitions of U and C guarantee that a creature trace will be continuous:

$$(C \cap U_t \neq \emptyset) \wedge (C \cap U_{t+\delta t} \neq \emptyset) \Rightarrow \forall n.0 < n < \delta t : (C \cap U_{t+n} \neq \emptyset) \tag{2.20}$$

that is if a creature exists at time t, and it exists and time $t + \delta t$ then it must exist at all times between.

Stability and local equivalence

Two creature instances c, d are weakly equivalent if for some specific function F, $F(c) = F(d)$. They are strongly equivalent if $F(c) = F(d)$ for *any* function F[3].

[3] When considering equivalence it should be noted that in practise the state transition "functions" may not be true functions — many simulations employ non-determinism

Similarly the stability of complete systems may be considered. A creature system is said to be *weakly stable* when for a specific function F:

$$F(U_n) = F(U_{n+1}) = F(U_{n+2}) = \ldots F(U_\infty) \tag{2.21}$$

and *stable* if for *any* function:

$$F(U_n) = F(U_{n+1}) \tag{2.22}$$

In such a case, for true transition "functions":

$$F(U_n) = F(U_{n+1}) \Rightarrow F(U_n) = \ldots F(U_\infty) \tag{2.23}$$

Guards

The part of a creatures next state, and output functions which may be defined by an end user (f'_s and f'_o) are specified as a set of conditions, known as guards that $\sigma(C_n)$, $\tau(C_n)$ and $i(c)$ must satisfy before an operation is performed. As the transition function is applied to a single creature these paramenters may be abbreviated to σ, τ and i — the state, type, and input of the creature being evaluted. These guards and actions define C_{n+1} and $f_o(C_n)$. For example

$$\#i > 1 \rightarrow \text{DIE} \tag{2.24}$$

Where i is the collection of types of creatures that may be observed, as defined previously. $\#i$ is the number of elements in the collection. The creature will perform the action DIE if it can see more then one creature (the creature itself is included in i, hence $\#i$ will always be at least 1). For convenience guards are evaluated in order. This removes the need to test for all previous cases.

$$\begin{array}{lcl} \#i > 1 & \rightarrow & \text{DIE} \\ \text{TRUE} & \rightarrow & \text{CENTER} \end{array} \tag{2.25}$$

The operation DIE is a "terminal" operation, and therefore does not return. Therefore operation CENTER will only be performed when the first operation's guard has failed.

Guard conditions commonly are based around the number of creatures of a given type which are visible. The cardinality operation is therefore extended to take a parameter:

$$i\#\tau = \#\{t : t \in i \wedge t = \tau\} \tag{2.26}$$

The operation $i\#3$ would therefore return the number of creatures of type 3 that can be seen.

$$\begin{array}{lcl} i\#1 = 1 & \rightarrow & \left\{ \begin{array}{lcl} \tau = 1 & \rightarrow & \text{CENTER} \\ \text{TRUE} & \rightarrow & \text{NORTH} \end{array} \right. \\ \text{TRUE} & \rightarrow & \text{DIE} \end{array} \tag{2.27}$$

Creatures

This example also shows how guard structures may be used as part of actions. If a creature can see a single creature of type 1 it moves north, unless it is itself that type 1 creature in which case it remains in its current location. If a type 1 creature cannot be seen then the current creature dies.

Actions

The actions available to a creature are

- Changes of internal State σ
- Changes of Type τ
- Changes of Location λ
- Birth of new creatures $f_o(c)$

The final item — Birth is dealt with independently since it relates to $f_o(c)$, rather than $f_s(c)$.

A creature may perform arbitrary operations upon its internal state $\sigma(C_n)$. A creature may also act upon its external status (or type) $\tau(C_n)$. The results of these will be passed on to $f_s : C_n = C_{n+1}$.

The creatures location λ is not passed into the transition function as defined by the user (f'_s), and hence may not be used as part of the behaviour definition. Movement is restricted to relative operations. The result of $f_s(c)$ will be a creature at a location in $\lambda^*(c)$, but the result of $f'_s(c)$ will be at a location within within $\lambda^*(\vec{0})$. The graveyard location is also included in this domain. The DIE operator sets $\lambda(f_s(c)) = \lambda(f'_s(c)) = DIE$. This location is used to indicate that $f_s(c)$ should not be included in the set U_{n+1}. All movement operators are *terminal* operators. That is, following a movement operator the values of τ, σ, and λ are formed into a triple, and returned as $f'_s(c)$. No further operations may be performed. However if no movement operator is present within an action, then other guards within the rule may be tested, and additional actions performed.

For example if σ is defined as an integer:

$$\begin{array}{lcl} \#i > 1 & \rightarrow & \sigma = \sigma + \#i - 1 \\ \sigma > 10 & \rightarrow & \sigma = 0; LEFT \\ \text{TRUE} & \rightarrow & \text{FORWARD} \end{array} \quad (2.28)$$

This rule moves a creature FORWARD. As it moves it counts the number of creatures it sees. When it has found 10 or more it turns LEFT (a terminal operator, as it it is assumed to

include the change of location), and resets the count. It should be noted that upon observing a creature, the action taken is does not include a terminal operator. The following rules are therefore tested, and the creature moves either FORWARDs or LEFT.

Birth

A creature may give birth to any number of other creatures of any type. These will begin their ”life” at the location of their parent.

$$d \in f_o : C_n \Rightarrow \lambda(d) = \lambda(C_n) \tag{2.29}$$

However in terms of the definable part of the transition function births take place at the origin:

$$d \in f'_o : C_n \Rightarrow \lambda(d) = \lambda(\vec{0}) \tag{2.30}$$

Offspring perform no operations at time n. Their state is set to σ_0, and they are inserted into U_{n+1} in preparation for $t = n + 1$, when they behave according to the standard rule set.

For Example:

$$\begin{array}{lll} (\tau = FEMALE) & & \\ \wedge(i\#MALE = 1) & & \\ \wedge(i\#FEMALE = 1) & \rightarrow & \begin{cases} \sigma = 0 & \rightarrow \quad \sigma = 1; BIRTH(MALE) \\ \sigma = 1 & \rightarrow \quad \sigma = 0; BIRTH(FEMALE) \end{cases} \\ \text{TRUE} & \rightarrow & \text{WANDER} \end{array} \tag{2.31}$$

All creatures perform the operation WANDER. However, should a FEMALE and a MALE find themselves alone together, a single offspring will be produced. The offspring of a specific female will alternate between male and female.

In addition to the transition rule it may also be necessary to specify the initial state of new creatures σ_o. In the above example $\sigma_o = 0$ would suffice.

2.2.3 Equivalences

For the Creatures model to be generally useful it must be computationally complete[68]. The simplest method of establishing this is to demonstrate that the Creatures model is equivalent to another system which has been established as being complete. The most

basic computationally complete system is a Turing machine, and hence the implementation of a Turing machine in CA will be considered. In addition the equivalence of Creatures and CA may be simply demonstrated. This suggests a method by which CA rules may be mechanically transformed into Creatures, though in an inefficient manner, and shows how basic computing models may be implemented within Creatures.

The Turing Machine

A Turing Machine[64] consists of a finite state machine and a tape divided into cells, each cell containing at most, one symbol from an allowable finite alphabet (without loss of generality we may consider only a binary alphabet of symbol or no-symbol). The finite state "unit" may

1. read the contents of a cell;
2. print a symbol on the cell read;
3. move to the next state;
4. move the read/write head one cell left or right;

The equivalence of the *Creature* system to a Turing machine may be shown by implementing a Turing machine as follows:

1. the set of creatures types is defined as one type representing the symbol and one type representing the finite state unit (FSU).
2. the "tape" is replaced by the presence or otherwise of symbol creatures within a one dimmensional space, each symbol creature remaining in a single specific location from brith to death. When a symbol creature observes the FSU creature it *always* dies. However in such an event the information carried by the symbol has been observed by the FSU and my be reprinted if necessary.
3. the "finite state unit" is represented a single creature in the system which can
   (a) detect the presence of a symbol creature;
   (b) change its state;
   (c) give birth to a new symbol creature
   (d) move to either the left or the right;

Any system which can implement a Turing machine is by definition computationally complete, and can be shown to be equivalent to any other complete system. Creatures is therefore computationally complete.

Cellular Automata

CA are also computationally complete, and hence the completeness of Creatures could alternatively have been shown by implementing a CA in Creatures. Implementing CA in Creatures also reveals the key differences between the two systems. Using a transformation based on this equivalence it should be possible to mechanically implement any CA system in Creatures. However such an implementation would require many creatures for each CA cell, and would therefore be inefficient.

The converse operation (to convert Creatures to CA) is not always possible, as (in the general case) in a Creatures system there is no limit to the number of creatures which may occupy a single location. Any CA cell will have a finite limit on the number of states it may be in (both by definition of being a finite state machine, and by practicalliy of implementation), and hence a limit to the number of creatures it could simulate holding. This however does not invalidate that CA are complete — it is simply that there is no simple mapping.

To implement CA within the Creatures model the array that is the cellular automaton can be represented with a regular mesh of identical creatures, each of which remains in the same location from step to step. The array will then be discrete, it will compute at regular time intervals, each cell will have a fixed number of states, be identical, and update synchronously based upon previous states.

The primary difference between the "creature" array and the cellular array is in the handling of neighborhood. The Creatures "cansee" operator is not capable of "looking across" to neighboring locations. Instead, the neighborhood must be implemented by forcing the cell creature to give birth to multiple "communications creatures" that traverse the space and on travelling a set distance (defined by the neighborhood) give birth to an "information creature" and then die.

Each cell updates when it has a full complement of "information" creatures. This ensures that all updating is synchronous (since each cell creature will be both broadcasting and receiving the same number of neighbors). The cellular automaton will therefore "compute" at time intervals proportional to the longest path within the neighborhood.

Using this framework it is possible to mechanically transform any CA simulation into

an equivalent Creatures simulation (though the practicality of such a transformation is limited). CA and Creatures therefore can be seen to be computationally equivalent, and hence Creatures is shown to be complete.

2.2.4 Complexity

In general, the task of computing the "next state" of a creature system is a many body problem — a large number of independent agents each potentially influencing any and all of the others. It may be considered as having two parts

- determining the input of each creature (mapping)
- performing the transformation function on each creature (stepping).

The complexity of the transformation function may be assumed to be independent of its inputs, and hence stepping has a complexity of N (N creatures would require the function to be applied N times, taking N times as long for N creatures as would be required for one).

Without the application of a priori knowledge each creature must be compared to each other creature to establish whether they can "see" each other. In the general case the mapping operation has a complexity of N^2 — each creature must be compared to every other.

The total time to naively step a system is therefore $mN^2 + sN$. However the time taken to calculate the input to each creature rapidly comes to dominate the performance of the system. The system's step time will be approximately proportional to N^2. A system of this type will rapidly become impractical to implement as for large populations even a small increase in the number of creatures will have a detrimental effect on performance. However locality may be exploited to improve this situation.

If the system were implemented with one processor per location (as in a CA), then the complexity of mapping would be totally removed — any creature on the processor would be able to see all other creatures on the processor so no calculation of inputs would be required other than to compile a list of all creatures on the node. However this would potentially reduce stepping performance, as processors representing empty locations would be idle, while processors representing locations which contain many creatures would be heavily loaded.

The worst case occurs given N creatures and N processors. In such a situation it would be possible to step all creatures in a single time unit by placing one creature on each node. However if spatial locality is used to place one location on each processor and all N creatures should end up in a single location — $N-1$ of processors must stand idle while one processor steps N creatures. The stepping operation would therefore be N times slower than it ideally should be.

As will be seen in section 3.2.2, by making a trade off between these two extreme implementation strategies: no locality vs complete locality, an effective large scale system may be developed.

2.3 Conclusions

The Creatures model attempts to draw on the best features of current computer architectures and simulation techniques:

- the simplicity and parallelism of CA.
- the focus on *interesting events* found in DEVS.
- the ease of application of *LOGO.
- the simulation power of Mirror.

Ideas from these systems are drawn together to produce a logically coherent modeling paradigm and parallel computing architecture. The architecture is computationally complete, and has been shown capable of simply implementing basic computing models.

The model has so far been considered in the abstract with only occasional reference to implementation and application. In following chapters these issues will be considered in greater detail to show that Creatures may be easily implemented and applied to a range of problems.

3

Implementing the Creatures Model

In order for Creatures to be a practical programming environment and a useful simulation tool, it must be possible to implement it effectively on at least one platform. Creatures has been implemented on a number of systems, each with its own strengths and weaknesses reflected in a unique set of performance characteristics. The implementation strategies used on each type of platform and the techniques developed to improve performance are described in this chapter.

The serial implementation on a NeXT workstation provides a complete, flexible, interactive work environment on which simulations may easily be developed. Implementations on parallel machines are currently more difficult to work with, but offer potentially better performance when large populations are being considered.

In order that different implementations of the system may fully complement each other, a language for specifying Creature simulations (known as JAM) was developed. For each platform, a pre-compiler produces machine dependent code from a machine independent JAM specification. This allows simulations to be developed in the well supported NeXT environment, then transferred to a high speed system for large scale simulations.

3.1 The JAM Language

During early development of the Creatures system, rules were written in the native language of the simulator (C or objective C), using a macro package to deal with the creature specific concepts such as movement and births. While this was effective, and simple to use, it precluded any possibility of transferring simulations between simulators written in

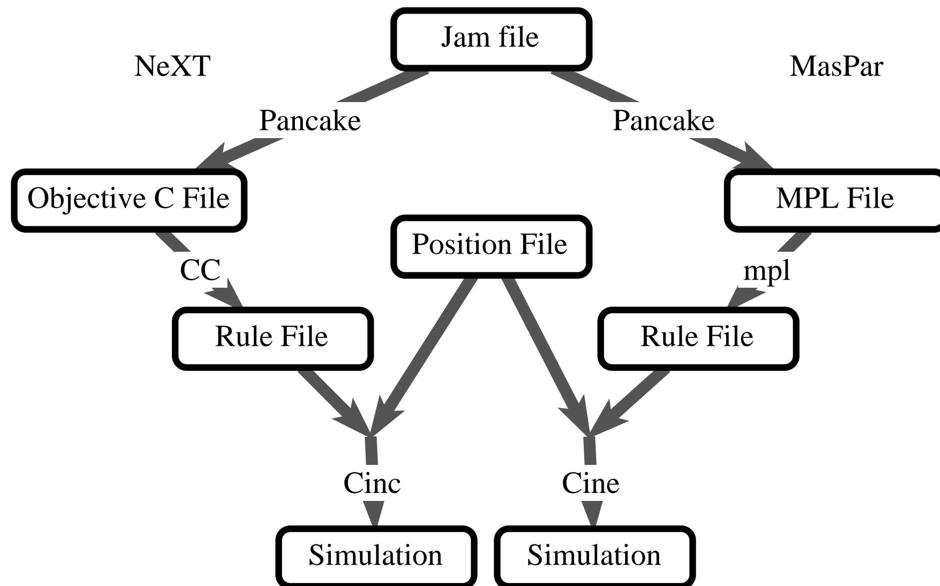


Figure 3.1: Compiling a Rule for Multiple Platforms

different languages, on different platforms. The creatures specification language "JAM" was developed, allowing a simulation to be described independently of the final platform.

A precompiler — "pancake" takes a JAM specification and produces a source file for a specific simulator. This may then be compiled using the appropriate language compiler (figure 3.1). Pancake was written using Yacc and Lex, and has been targeted to C, Objective C, MPL, C* and Occam. The code produced makes use of system dependent header files to define neighborhood and interaction macros, simplifying the operation of the precompiler, and reducing the time required to retarget it.

A simulation is specified in JAM by a number of declarations (some of which are optional):

- `NEIGHBORHOOD:` The directions creatures may move in. Typically this may be VonNeuman (four neighbors), Moore (eight neighbors) or Hex (six neighbors)[63], though more can be added. Certain simulators may not support all types of neighborhood due to limitations in the underlying hardware. Transputers implementations for example are likely to be restricted to VonNeuman neighborhoods due to their limited connectivity.
- `TYPES:` Gives names to each of the external states a creature may be in.
- `VARS:` Defines the names and types of the state variables within each creature. The

available types may be restricted depending on the simulator being used.

- `INIT:` Defines the initial state of all creatures.
- `USE:` allows system specific code to be included into the rule, for example to redefine the display operation.
- `RULE:` Defines the state transition function for creatures in the system.

The final declaration (`RULE:`) is based on the guards notation described in section 2.2.2. A rule consists of a list of test-action pairs (evaluated in the order in which they occur), where the action is performed only when the test is true. An action may itself be a test-action pair list. Alternatively it may be statement or a compound statement. At each time step the transition function is run until a terminal operation (movement) is found, for each creature in the system.

The input to a creature is represented by the "cansee" function, which returns the number of creatures in the current location with the specified type. As an extension to this, a range of creature types may be specified, and a total of all creatures within that range is given. The birth operation also supports ranging, allowing one creature of each type to be created in a single action. In addition the birth operation also supports a multiplier, allowing multiple creatures to be produced.

A formal description of the JAM syntax is given the following section (3.1.1), and examples of code may be found in chapter 4.

JAM has proved simple to use, and many rules have been described using it. These have generally been more concise then equivalent high level descriptions. JAM provides no looping constructs, functions, or other operators that would be found in a traditional programming language, as these are not required when describing Creatures simulations. The reduced number of concepts ensures that JAM source files are particularly readable to non programmers, allowing field experts access to SIMD parallel simulation without having to learn traditional programming methods. Code has been successfully moved between platforms with only minor modifications.

3.1.1 Jam Grammar

⟨input⟩ :

⟨neighbordeclaration⟩ ⟨typedeclaration⟩ ⟨vardeclaration⟩
⟨usedeclaration⟩ ⟨initdeclaration⟩ ⟨ruledeclaration⟩ ■

Creatures

neighbordeclaration :

NEIGHBORHOOD: ⟨variable⟩ **;** ■

typedeclaration :

TYPES: ⟨typelist⟩ **;** ■

⟨typelist⟩ :

⟨typelist⟩ **,** ⟨variable⟩
| ⟨variable⟩ ■

⟨vardeclaration⟩ :

VARS: ⟨varlist⟩ **;**
| ■

⟨varlist⟩ :

⟨varlist⟩ **,** ⟨vartype⟩ ⟨variable⟩
| ⟨vartype⟩ ⟨variable⟩ ■

⟨usedeclaration⟩ :

USE: ⟨uselist⟩ **;**
| ■

⟨uselist⟩ :

⟨uselist⟩ **,** ⟨variable⟩
| ⟨variable⟩ ■

⟨initdeclaration⟩ :

INIT: ⟨statement⟩
| ■

⟨ruledeclaration⟩ :

RULE: action ■

⟨action⟩ :

{ ⟨tplist⟩ }
| ⟨statement⟩ ■

⟨tplist⟩ :

⟨expression⟩ **:** ⟨action⟩ ⟨tplist⟩

| ⟨expression⟩ **:** ⟨action⟩ **!:** ⟨action⟩ ⟨tplist⟩ ■

⟨statement⟩ :

⟨expression⟩ **;**

| { ⟨statementlist⟩ }

| **;** ■

⟨statementlist⟩ :

⟨statement⟩ ⟨statementlist⟩

| ⟨statement⟩ ■

⟨expression⟩ :

⟨variable⟩

| ⟨constant⟩

| ⟨function⟩ **(** ⟨expression⟩ **)**

| ⟨function⟩ **(** ⟨variable⟩ **...** ⟨variable⟩ **)**

| ⟨function⟩ **(** ⟨expression⟩ **)** **(** ⟨number⟩ **)**

| ⟨movement⟩

| ⟨fnumber⟩

| ⟨number⟩

| **(** ⟨expression⟩ **)**

| ⟨expression⟩ ***** ⟨expression⟩

| ⟨expression⟩ **/** ⟨expression⟩

| ⟨expression⟩ **%** ⟨expression⟩

| ⟨expression⟩ + ⟨expression⟩

| ⟨expression⟩ **-** ⟨expression⟩

| ⟨expression⟩ **&** ⟨expression⟩

| ⟨expression⟩ **|** ⟨expression⟩

| ⟨expression⟩ == ⟨expression⟩

| ⟨expression⟩ $>$ ⟨expression⟩

| ⟨expression⟩ $<$ ⟨expression⟩

| ⟨variable⟩ = ⟨expression⟩ ■

⟨fnumber⟩ :

⟨number⟩ **.** ⟨number⟩ ■

⟨number⟩ :
  ⟨digit⟩
  | ⟨number⟩ ⟨digit⟩ ■

⟨digit⟩ :
  0 . . . **9** ■

⟨constant⟩ :
  random
  | **type**
  | **true** ■

⟨function⟩ :
  cansee
  | **alert**
  | **birth**
  | **become**
  | **iam** ■

⟨movement⟩ :
  A . . . **Z** ⟨movement⟩
  | ⟨movement⟩ ■

⟨variable⟩ :
  a . . . **z** ⟨variable⟩
  | ⟨variable⟩ ■

⟨vartype⟩ :
  int
  | **float** ■

3.2 A Sequential Implementation

The system has been implemented on a NeXT workstation[49] in Objective C[11]. The interactive simulator (CINC) uses a NeXTStep interface[24] to take input from the user, and display the results in a flexible fashion. This is shown in figure 3.2. A command line based

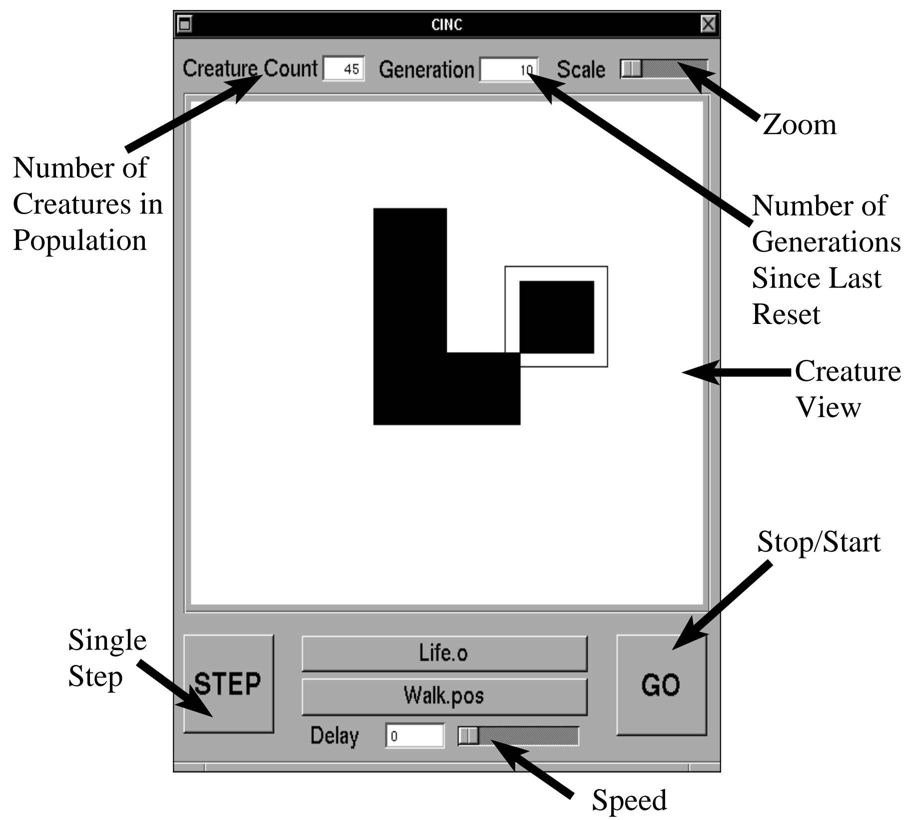


Figure 3.2: The NeXTStep Front End

simulator (cinb) uses the same core code, but does not implement the interactive facilities. Its performance is significantly better.

The NeXTStep implementation is the most complete, and the most flexible of all the current Creatures systems. Rules may be loaded into the simulator without recompiling, the movement of creatures is displayed graphically, and simulation may be interactively controlled by the user. This makes it an ideal platform for refining simulations with small numbers of creatures (< 100). The rapid feedback provided may be used to refine the interactions between types of creatures. Once the rule is suitably developed the number of creatures may be increased. At a certain population size (~ 10000 depending upon the application and the users expectations) the performance of this simulator becomes impractically slow, and the rule may be transferred to a larger machine.

This section describes the development of the serial simulator with particular emphasis on the techniques which allow it to provide such flexibility, and the scalability issues which will become more important as parallel implementations are considered.

3.2.1 A Naive Implementation

Cinc and cinb operate by having a central controller object which holds a "List" of creatures representing the current generation. When this receives a "step" message it takes the first creature from the list, and compiles a new list, by transferring all creatures which are in that creatures location. From this the "cansee" array for that location is evaluated, and passed to each creature in the new list as part of their step method. In addition two further lists are passed to the creature — one represents the next generation, and all new creatures are placed into this list. The other list is the graveyard list, and is used to collect all creatures that are no longer required so they may be disposed of and their memory reclaimed.

This is repeated until the generation list is empty. Garbage collection is then performed, and the controller's step method terminates. In Cinc, the creatures are also sent a drawSelf message, allowing them to draw themselves on the screen.

Objective C proved to be a particularly effective tool for the implementation of Creatures. By representing each creature as an object[26], the creature state, and rule definition may easily be incorporated into the Class system. A generic Creature class provides a basic drawing method, the essential creature parameters (type and position), and a default step action of doing nothing. Subclassing of the Creature class may be used to redefine the step action to introduce the behaviour required for a specific simulation. The subclass of

creature may also easily incorporate extra instance variables. The initial state of a creature (σ_0) is coded into the "init" method, which is executed whenever an object is created. For more complex simulations other actions may be overwritten, redefining the default concepts of space, the graphical appearance of a creature, and replacing the default file handing code to load and save creatures in various formats.

In addition to the basic objective C functionality, NeXTStep provides functions to dynamically load Classes in a simple fashion. This allows new rules to be loaded into an already running simulator.

3.2.2 A Scalable Implementation Strategy

The performance (measured in creature steps per second — CPS) of the non-interactive system[1] for a range of population sizes is shown in figure 3.3. The interactive system performs at approximately one fifth of this speed. While the "naive" implementation proved useful for developing rules by allowing small scale behaviour to be examined, the performance degrades rapidly as the population increases. During the step operation the location of every creature is compared with every other creature[12] to produce the "cansee" arrays. The time taken to perform this action (referred to as mapping) is approximately proportional to the square of the population size. Though as few as $n-1$ comparisons may be required to map the system, the worst case of $\sum_{i=1}^{n} i \approx n^2/2$ comparisons is more likely to apply.

As the population size increases mapping will inevitably come to dominate the performance of the system, dwarfing the time required to evaluate the transition function. The system must be mapped once at each time step. The time taken to perform this mapping is $O(n^2)$, and hence performance measured in generations per second falls away as $1/n^2$. However at each time step n creatures are processed. The mapping time *per creature* is therefore $O(n)$. The measure of system performance that has been used in this thesis is "Creatures Steps Per Second" (CPS), which for the naive implementation is proportional to $1/n$ as shown in figure 3.3.

In order to avoid this loss in performance as population increases, steps must be taken to improve the efficiency of the mapping operation[6][44]. Direct use of locality in the implementation would introduce undesirable features, similar to those found in cellular automata: empty locations with spare resources while other locations contain many creatures;

[1] The NeXT used for these performance figures contained a 25MHz 68040, benchmark performance around 15 Mips

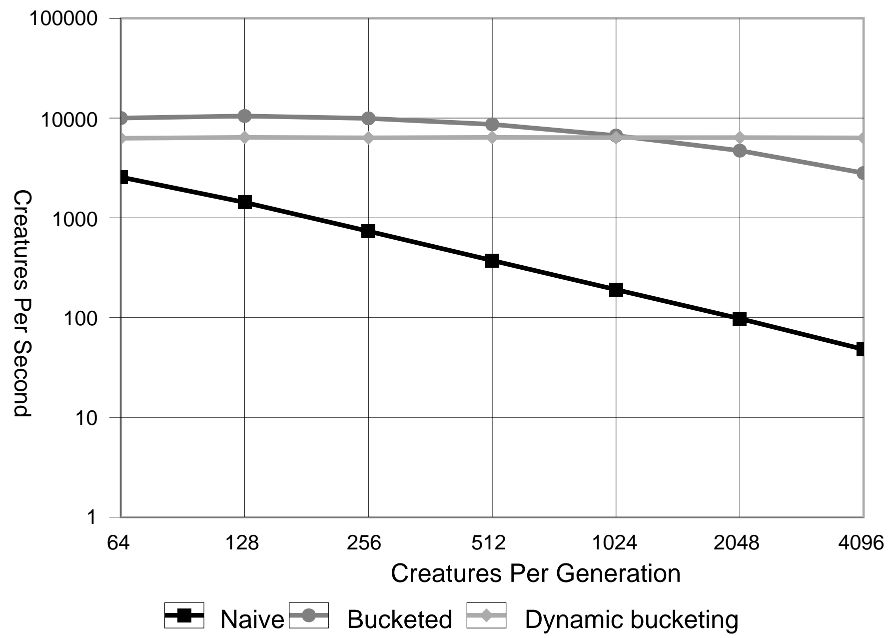


Figure 3.3: NeXT Performance

the increased calculation required when creatures meet leading to poor load balancing of real work. Instead a system was developed which makes use of a hashing function to split the population into a number of smaller sub-populations, each of which may be mapped far more quickly. If the population is divided in two, then 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 collected into the same partition (which shall be referred to as a bucket, due to similarity with hashed files in database systems). However by making use of an effective hashing function, clusters of creatures in neighboring locations may be distributed to different buckets. By this method several locations (or in fact infinite locations for an infinite space) may be mapped to the same bucket, reducing the likelihood that a bucket may be empty.

A trade off must be made as to 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. The use of hashing and buckets allows a system to be created 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, a mapping performance inversely proportional to n may be obtained (and hence a constant performance when measured in CPS).

The major disadvantage of this approach is that when a creature moves in space it must be moved from one bucket to the bucket appropriate to its new location. In a simple serial implementation this may not be a problem (in fact the simple approach of rehashing all creatures at every time step proves to be effective — as all creatures are in the same address space the cost of hashing is insignificant). However this will be inefficient in a parallel implementation, as buckets are likely to be held on separate processors, and hence large amounts of data must be copied between nodes. Despite this apparent problem, the new workload is proportional to the number of creatures moving (as each task is independent), and hence performance is guaranteed to be better than the naive implementation for large populations ($an < bn^2$ for large n even if $a \gg b$).

An additional problem 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 must be necessarily limited. This particularly applies in parallel implementations where resources such as memory are local to single node and hence must be tied to a single (or small number of) buckets. Though a serial

implementation also has limited resources, these resources may be dynamically allocated to buckets as they are needed. A parallel implementation is likely to require a static allocation and hence, though there may be sufficient resources on the machine as a whole to perform an operation, a particular bucket may overflow. The simulator must be run with sufficient space to ensure that buckets are unlikely to overflow. This limitation is analysed in greater depth in the following section.

The above problems do not apply when using a single processor workstation architecture (particularly if virtual memory is available), and dramatic improvements in performance may be obtained. The step method of the controller object was modified to hash the population into a number of buckets prior to the stepping operation as described for the naive system. Using a fixed number of buckets allows good performance to be maintained as the population size increases (see figure 3.3) until the computation time required to map each bucket becomes greater that the time required to step the bucket. As the population increases above this level (around 1000 for the current implementation) the mapping operation again dominates and an $1/n$ CPS performance curve is evident, though with better performance than the “naive” implementation.

In a simple serial implementation the number of buckets used may be varied dynamically to ensure that the number of creatures on each bucket is slightly smaller than the size at which mapping becomes a significant overhead. The computational overhead in this is small, and allows performance to be maintained for arbitrarily large populations (provided hardware limitations are not exceeded).

3.2.3 A Statistical Analysis of Bucketing

Despite the performance benefits that bucketing offers, when implemented on a platform where the resources of each bucket are limited, the problem of bucket overflow reduces the applicability of the approach. The system was therefore analysed, to provide a theoretical insight into the overflow problem. Much of this analysis is common to the theory of hash files used in database systems[7]. Similar results also exist in the field of queueing theory[27][10], which could be applied to yield a more complete analysis.

Consider a simulator containing K buckets, each capable of holding B creatures. The maximum population size is therefore $N = KB$. However in practise, there will be a smaller number of creatures n.

It must be assumed that the distribution of creatures through space is uniform and indepen-

dent of other creatures and the hash function used to assign locations to bucket is fair. While this assumption is unlikely to be reliable for real simulations, it is necessary for virtually any analysis. In practise the distribution may be far better or far worse than a random distribution depending upon the application. However such distributions are specific to a given simulation. No matter how complex the analysis it would still be possible to produce a "badly behaved" simulation which exceeds the capacity of a well designed simulator. All that can practically be derived is the typical behaviour of the system. Though random distribution is an unreliable assumption to make, its general applicability, and mathematical simplicity make it the only viable option.

Individual Buckets

A creature has a probability of being in any given bucket $p = 1/K$, independent of other creatures, and buckets. When many creatures are placed in the system, the number of creatures in a particular bucket is therefore described by a binomial distribution[47]. The mean number of creatures in a given bucket

$$\mu = np = n/K \tag{3.1}$$

The variance of the number of creatures in a given bucket is

$$\sigma^2 = np(1-p) = n(1/K - 1/K^2) \tag{3.2}$$

These may be used in a normal approximation to the binomial distribution. Such an approximation would become necessary if calculations were to be performed on very large systems. However an accurate binomial description has so far proved tractable.

The probability of there being exactly i creatures in a given bucket is

$$P(n(k) = i) = \left(\begin{array}{c} n \\ i \end{array} \right) (1/K)^i (1 - 1/K)^{n-i} \tag{3.3}$$

The probability of the bucket *not* overflowing $P(k)$ is therefore

$$P(k) = \sum_{i=0}^{B} \left(\begin{array}{c} n \\ i \end{array} \right) (1/K)^i (1 - 1/K)^{n-i} \tag{3.4}$$

Multiple Buckets, and multiple steps

In order for the simulation to succeed (for a single step) no single bucket may overflow. The probability of this is

$$P(s) = P(k)^K \tag{3.5}$$

Given the probability of the system succeeding at a given step, the probability of reaching time t is

$$P(t) = P(s)^t = P(k)^{Kt} \tag{3.6}$$

It should be noted that this assumes complete independence between consecutive steps. This is blatantly untrue, as the distribution of creatures in one step is strongly correlated to the previous step — particularly if the amount of movement is small. If no creatures move then the probability of overflow is zero. Only if a large number of creatures move using a large neighborhood will the steps be truly uncorrelated. Such analysis may only be made on a case by case basis, as it requires detailed consideration of the hashing function used, the initial population distribution, and the transition function. Even in such specific cases the problem may be intractable due to the computationally complete nature of the system being considered. Assuming independence between timesteps gives the most pessimistic approximation to the value of P(t). Using such a value ensures that the simulation is unlikely to fail if the statistics indicate it should succeed.

Births, Deaths and Movements

The derivation so far has assumed that the number of creatures is constant, and that the movement of creatures between buckets has little effect on the simulator (each step being independent). The inclusion of Births and Deaths to the system means that the number of creatures is no longer constant. It also allows large numbers of creatures to be suddenly placed in a single location. This clearly invalidates the model previously developed. Some form of statistical independence must again be assumed: Provided that births are distributed through space in a sufficiently random way, an upper limit for the population may be estimated, and the probability of success may be derived based on this population size. A more detailed analysis is not possible without considering a specific application, and hence a pessimistic estimate is again adopted.

When a creature is moved it must temporarily exist in one bucket, while at the same time finding an empty location in its destination bucket. It therefore effectively occupies two locations. The current simulators only move one bucket of creatures at a time, freeing the source locations for use by creatures moving from other buckets. Making assumptions of statistical independence as has been done previously, movement may therefore be incorporated into the model of simulator behaviour by increasing the population size n to $n' = n(1 + P(m)/K)$ where $P(m)$ is the probability that a creature will move at any given step (which may be pessimistically be set to 1).

Numerical Examples

Each of the implementations of Creatures was benchmarked over 500 time steps, with every creature moving at every step. This was also used for the theoretical results derived below.

For the basic case of $K = 64$ and $B = 64$ the graph shown in figure 3.4 was produced. Increasing K, the number of buckets, has little effect on the shape of the graph — the breakpoint is always a little over 50% full. In database terms it is recognised that it is the ratio of the n and K parameters rather than their absolute values which determine overflow performance.

If B, the bucket size, is increased then the breakpoint also increases (relative to the absolute maximum population size N), as shown in figures 3.5 and 3.6. From an overflow point of view it is clearly desirable that buckets are as large as possible. However large buckets will give poor performance for the mapping operation, and may not be practical on a parallel architecture where resources on a particular node (or collection of nodes) may be limited.

Considering the special case of a single bucket (as in the naive serial implementation) the probability of finding i creatures in a bucket is zero for all values of i except $i = n$. For values of B less than n the probability of succeeding for a single timestep $P(k) = 0$. For $B > n$ the probability of succeeding is 1. Although bucket overflow still applies in the trivial case the system must be completely full before an error occurs.

For a bucket size of 64, the statistics show that it should be possible to operate a simulator that is approximately half full (figure 3.4). Testing a real system of this size showed that (in the idealised benchmark situation) the simulation was always successful at 25% capacity, but would always fail when the population was increased to 50% (for those implementations where bucket size has an absolute fixed limit). Given the nature of the assumptions made regarding independence of the elements, this may be viewed as a good correlation.

3.3 Creatures on the MasPar MP1

A creatures simulator was developed in MPL[45] to run on a MasPar MP1104 machine. This has 4096 nodes arranged in a square grid with local connections, plus additional global routing hardware. Two implementations were developed — one which placed one creature on each node, a second which used the bucketing techniques described previously. The implementation demonstrates the strengths and weaknesses of the bucketing approach when

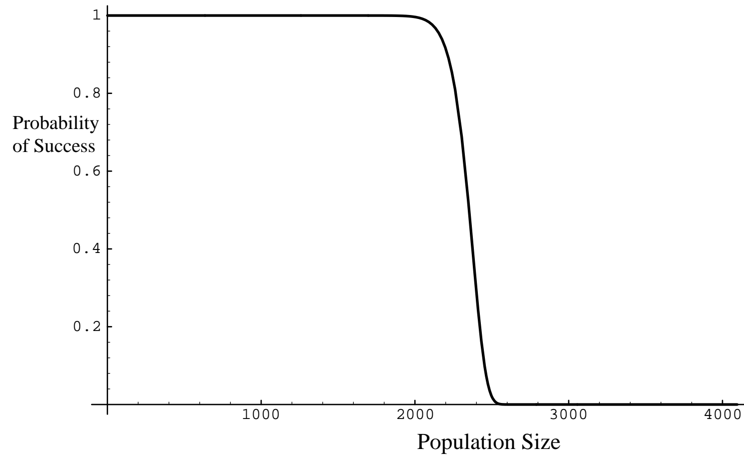


Figure 3.4: Probability of success for $K = 64$ and $B = 64$

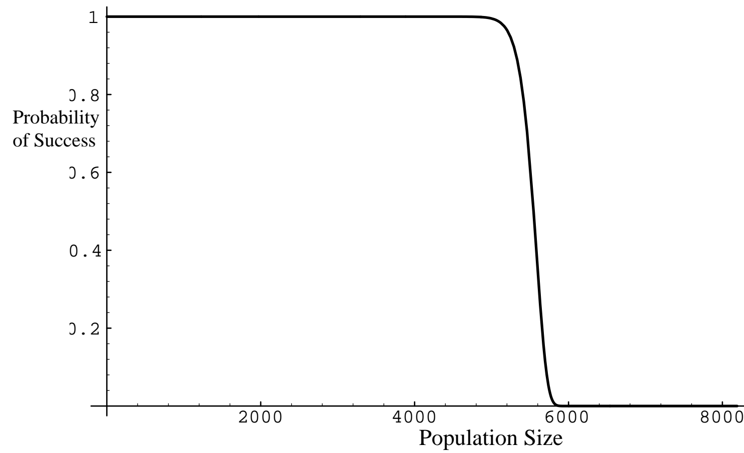


Figure 3.5: Probability of success for $K = 64$ and $B = 128$

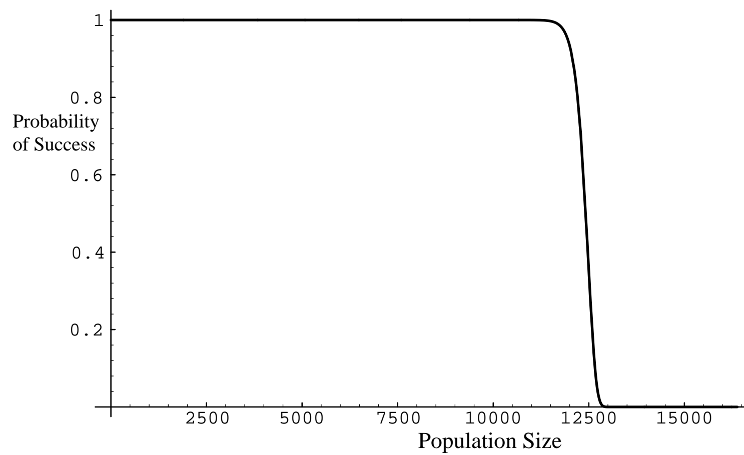


Figure 3.6: Probability of success for $K = 64$ and $B = 256$

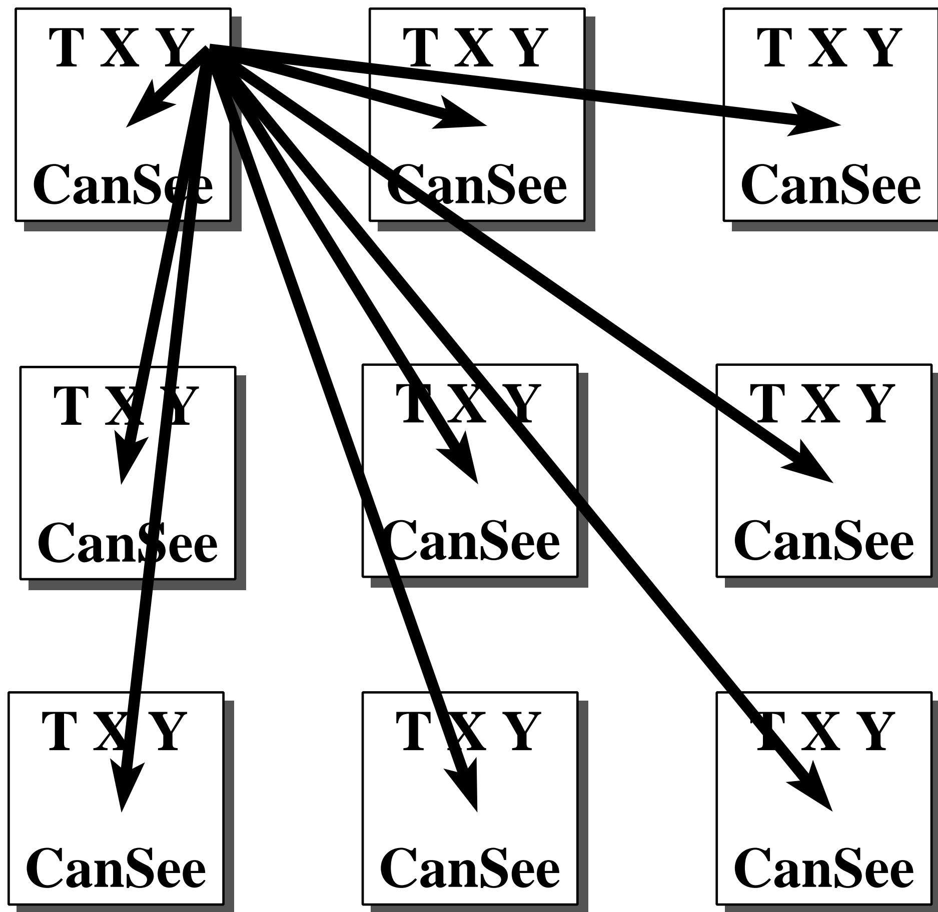


Figure 3.7: Broadcasting the Creature Map

applied in a massively parallel system of this type.

3.3.1 A Naive Implementation

Given a large number of processors, each creature may be assigned its own individual processor. The type and location of each creature is broadcast and incorporated into the world view of every other creature (figure 3.7). The creatures are then stepped, and a search for free nodes is performed by those processors which are required to produce offspring.

Performances of 15000 creature steps per second (CPS) were recorded when the machine was fully populated, with a single creature on each of the 4096 nodes (figure 3.8). For populations of this size, a naive (non-bucketed) serial implementation on a NeXT workstation could only achieve 48 CPS. This speed increase of approximately 300 times is in agreement

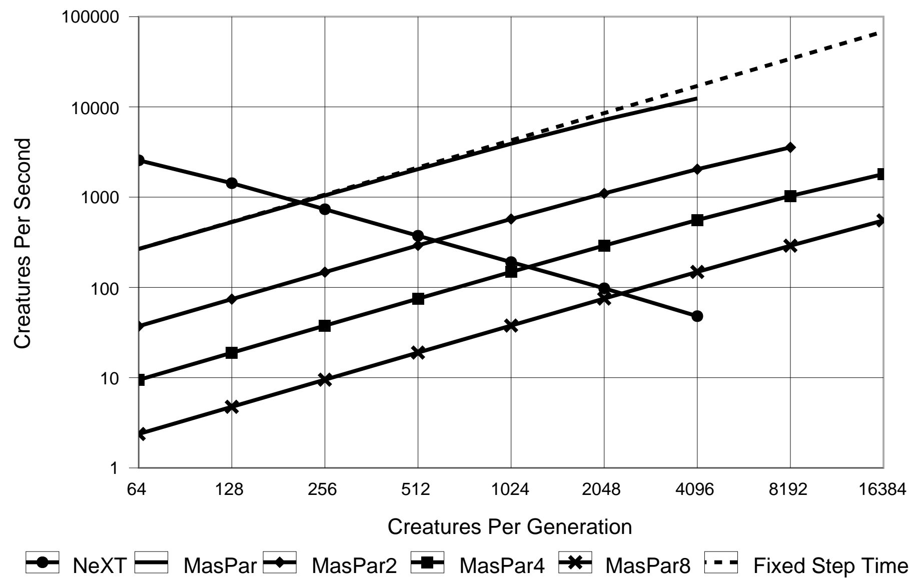


Figure 3.8: MasPar Mp-1 Performance

with the claimed performance figures for the MP1104, and NeXTstations (6400 and 16mips respectively — a ratio of 400). While the NeXT implementation was not optimised for speed, the results show that the algorithm is extracting a good percentage of the theoretical peak performance from the MasPar architecture. For small populations, the overhead of supporting a large number of processors results in poorer performance than that available from a serial machine — there are insufficient creatures to utilize the available processors. Step time is in fact almost independent of population size (the "fixed time step" line shows the CPS performance required to give a constant time per generation).

For many applications even 4000 creatures may be insufficient to produce reliable statistically valid behaviour. However once the number of creatures exceeds the number of processors the performance of even a massively parallel system begins to degrade rapidly. This is shown by the lines MasPar2, 4,and 8 on the graph (figure 3.8), which represent the systems performance with each node holding 2, 4, and 8 creatures respectively. In addition to the overhead of virtualisation, the time taken to calculate one generations increases by a factor of four for every doubling of population size due to the n^2 effect of the mapping operation — the CPS performance falls proportionally to n.

3.3.2 Bucketing on the MasPar

As previously discussed, the application of "bucketing" improves the performance of the mapping operation. However due to the MasPar's architecture certain constraints were placed on how buckets could be configured. The 4096 node machine is arranged as a 64×64 square. The system was therefore built to contain sixty four buckets of up to sixty four creatures. The hashing function used was of the form $H(x, y) = Kx + y$. Specifically $K = 9$ as this results in 7 steps in the x direction placing the creature one bucket down from its initial location, ensuring that a line of creatures either horizontally or vertically (a common degenerate case) would efficiently utilize all processors. Virtual processor space was implemented by increasing the number of buckets. This maintains a constant performance (measured in creatures per second) for an increasing population size.

Increasing the bucket size was not feasible due to the nature of the MasPar architecture — the mapping operation was performed by spreading information about creature type around the rows of the machine. Initially this was implemented by each node examining each other node within the bucket. However the communications overhead proved high due to the distance being covered. Instead each node makes a copy of its type, then replaces that copy with the copy held in the node to its right (and assimilates the new data into the creatures world view). This results in the copied data's rotation round each bucket with a minimal communications overhead (as shown in figure 3.9). Introducing virtual nodes into this operation would have been difficult, as the MPL language makes no provision for such structures — the code would have to explicitly deal with the multiplexed nodes, and the inter/intra-node communications required. Were such a system to be implemented, the step time would increase by a factor of four for every doubling in size (due to the n^2 complexity of mapping within a single bucket). However the problem of bucket overflow would be eased, allowing the simulator to be run at closer to its maximum capacity.

Statistical analysis of bucket overflow (section 3.2.3) suggested that running the simulator at approximately fifty percent capacity was likely to cause a failure during the course of the 500 steps that were used for benchmarking. This proved to be the case — provided that a population of one quarter the simulator size was not exceeded, no overflows were obtained. The performance of the bucketing system is shown in figure 3.10. This shows that around 10,000 creature steps per second may be maintained for large populations. The loss in performance due to increased simulator size when the population is small is negligible compared to the loss obtained by the naive method.

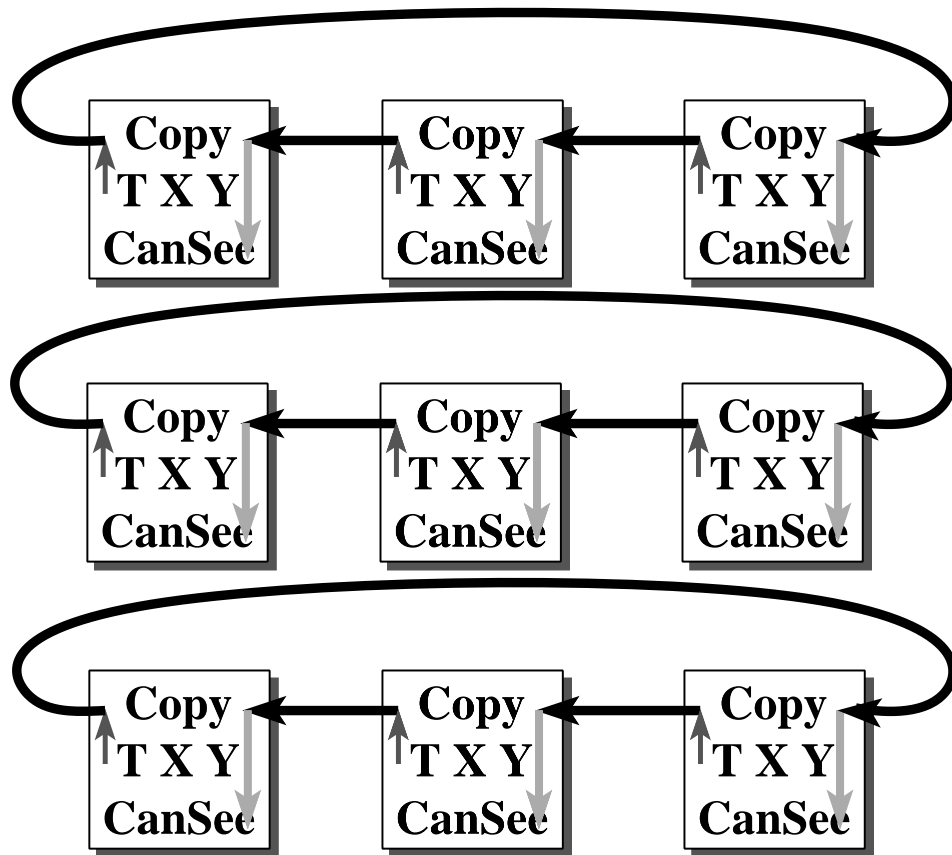


Figure 3.9: Generating the Creature Map using buckets

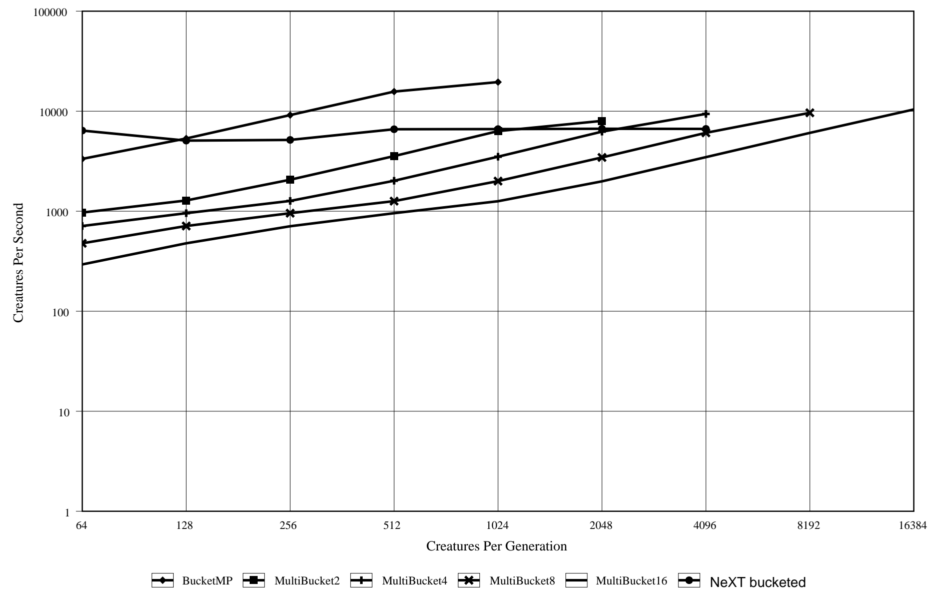


Figure 3.10: Bucketed Performance

3.3.3 Conclusions about the MasPar System

The performance of the naive method obtains good speed up compared to the naive serial implementation. However this approach is necessarily limited by the n^2 mapping problem.

The performance of the bucketed approach is unfortunately little better than that obtained by serial implementations. This can be explained by two factors:

- Dynamic Bucketing: A serial system is able to determine the optimum number of buckets and bucket size at run time, and change this dynamically as the simulation progresses. In addition to ensuring good performance it eliminates the problems of overflow.
- Movement: Profiling of the Bucketing code revealed that approximately ninety five percent of the simulation time is taken up by a single line of code — the line responsible for the movement of creatures between buckets. When a creature moves its new location is invariably, on another processor, and hence data must be copied between nodes. This problem will be further considered, and tackled later in section 3.5. For the purposes of benchmarking a worst case problem was picked where *every* creature moves to a new location at every time step. Hence the performance figure is

somewhat pessimistic, and is likely to be exceeded for real simulations.

The MasPar implementation failed to deliver the full performance that the naive system indicated should be possible, and is somewhat difficult to use (the correct bucket size for a problem must be carefully chosen). However it does demonstrate that Creatures code written in Jam may be retargetted to a new back end with little modification. Most importantly the implementation demonstrates that though bucketing solves the problems of mapping, it creates a number of problems of its own when implemented on architectures of this type. The implementation on the MasPar provides insight into how a more effective machine may be built.

3.4 Creatures on the Thinking Machines CM2

Following the completion of the MasPar implementation the bucketed MasPar implementation was ported to the Connection Machine[29][38] C*[21] environment. Such a port would demonstrate that the techniques developed to run Creatures on the MasPar were applicable in a range of situations, and reveal the relative strengths and weaknesses of the alternative underlying machine architectures. In addition, access to the MP1104 was limited to a short time period — the CM2 provided a platform which could be returned to, allowing further experiments to be performed should the need arise.

Porting the simulator proved to be a relatively minor operation requiring a minimal number of changes other than lexical translation to use C* rather than MPL keywords. C*'s more limited control structures did present some minor problems, requiring additional tests which may have had some impact on efficiency.

The Connection Machine is physically a hyper-cube but the programmer's model (when using C*) is an array of variable dimension. Communication is by wormhole routing through this $n - cube$. There is no global routing hardware as in the MasPar system. C*'s provision for virtual processors (which was lacking in the version of MPL available at the time) allowed a more complete exploration of the bucketing parameters.

The resulting performance is shown in figure 3.11. These results indicate an exceedingly poor performance on a machine that is theoretically far more powerful than the MP1. Once again the poor results may be attributed to communications overhead. The movement of Creatures between buckets may result in the transfer of data between nodes which are remote in the physical architecture (though this is difficult to determine, as the base hyper-

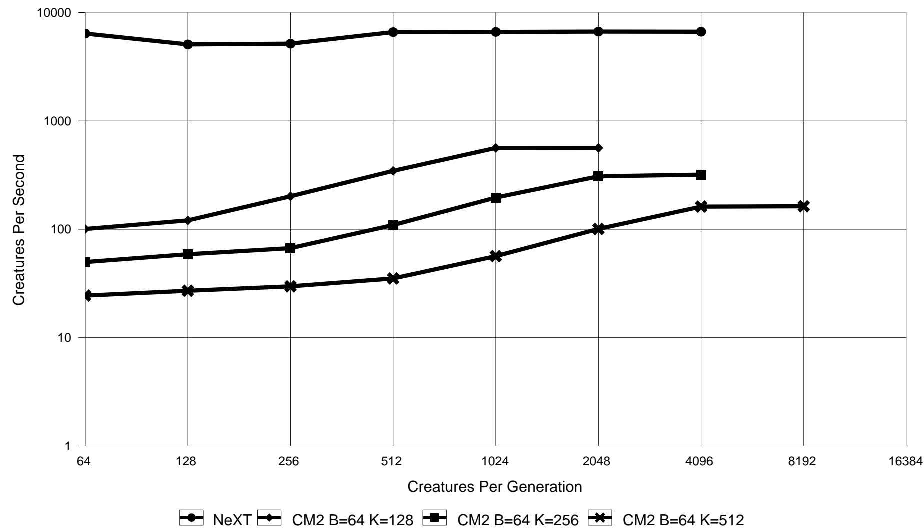


Figure 3.11: CM2 Performance

cube is totally obscured by the C* programming paradigm). These transfers are especially expensive given the local communications system used by the CM. While these transfers are also expensive on the MasPar, the provision of hardware dedicated to long range routing relieves the problem somewhat.

In order to verify that the communication overhead of creature movement is indeed the bottleneck, the CM2 was benchmarked again this time choosing the optimum scenario where every creature remains in the same location. The results of this are shown in figure 3.12. This shows a dramatic improvement in performance. Such a system operates at up to one hundred times the speed of the previous example. Clearly steps must be taken to optimise the movement of Creatures.

3.5 Spiralling — Reducing the Movement Problem

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 3.13). The last bucket is connected back to the first to form a toroidal structure. This ensures that the

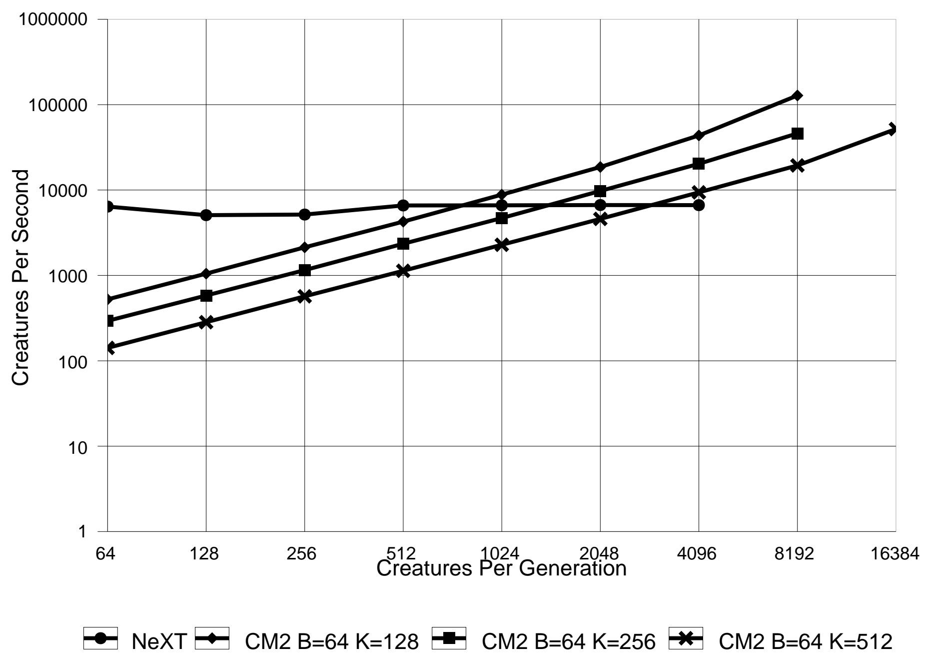


Figure 3.12: CM2 Stationary Performance

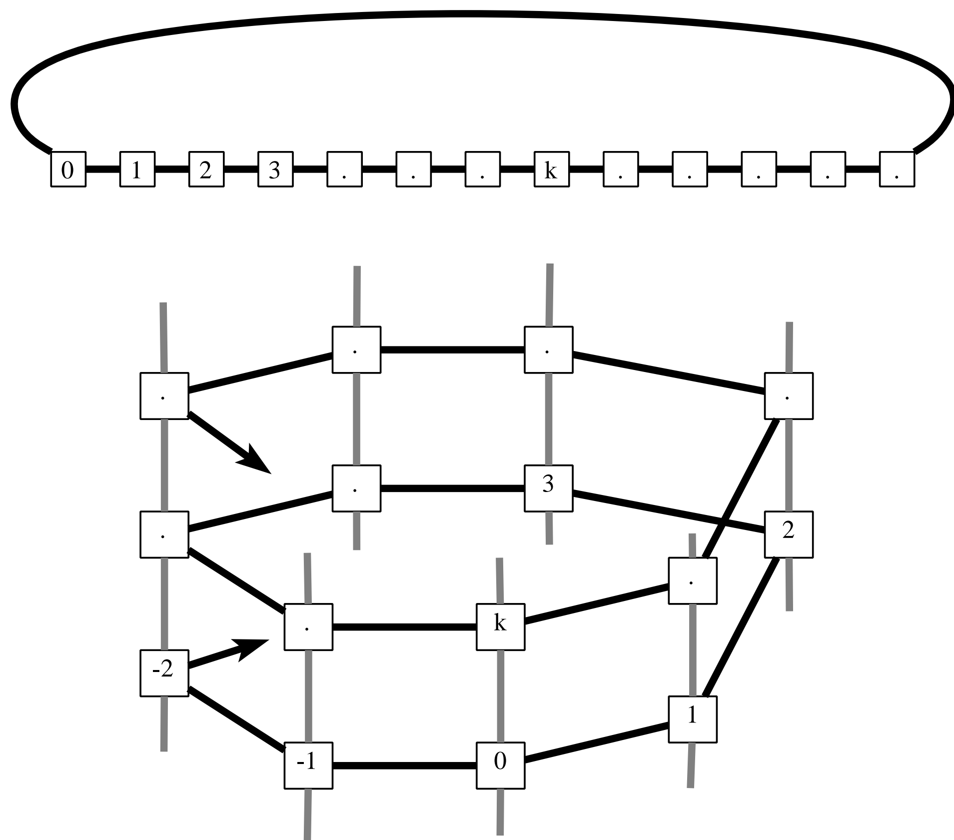


Figure 3.13: Wrapping buckets into a Spiral

small changes of x and y involved in the migration of processes result in a short communications distance between adjacent buckets, while still breaking the creatures down into easily mapped sub-populations. Load balancing for degenerate, one dimensional problems is also maintained.

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 proposed for use in systolic computation systems[56]). An offset of any size could 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 3.14. When a shift of distance o is applied in each direction, a gap appears in the grid of size o^2. This gap must be filled with additional buckets to regain a pseudo infinite, uniform surface similar to that of a traditional torus.

3.5.1 Designing A Double Twisted Torus

Given that the twisted torus architecture reduces the communications overhead produced by the simple hashing approach, it is necessary to consider what values of n (the grid size), and o (the offset) are effective. The topology is scalable, so the absolute size of the array will be limited only by practical considerations. However for a fixed number of processors a number of shapes could be build, each with differing performance. By careful selection of n and o a more efficient machine may be built.

Without a priori knowledge of the problems to be tackled by the proposed machine, few assumptions can be made about the load pattern that will be placed on the grid. However the performance of the system is highly dependent on the load upon it. For any topology it would be possible to find loads which map particularly badly. A well designed grid will be built such that loads which map badly are unlikely to occur, while common loads map well.

It must be assumed that on average loads will be symmetrical, as for any load that may be applied there is an equally likely load where the x and y axes have been interchanged. Therefore only grids such that $n_x = n_y$ and $o_x = o_y$ will be considered.

When implementing bucketing on the MasPar a desirable feature of the hash function selected was that movement in either the x or y direction pass through every node before returning to the starting 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

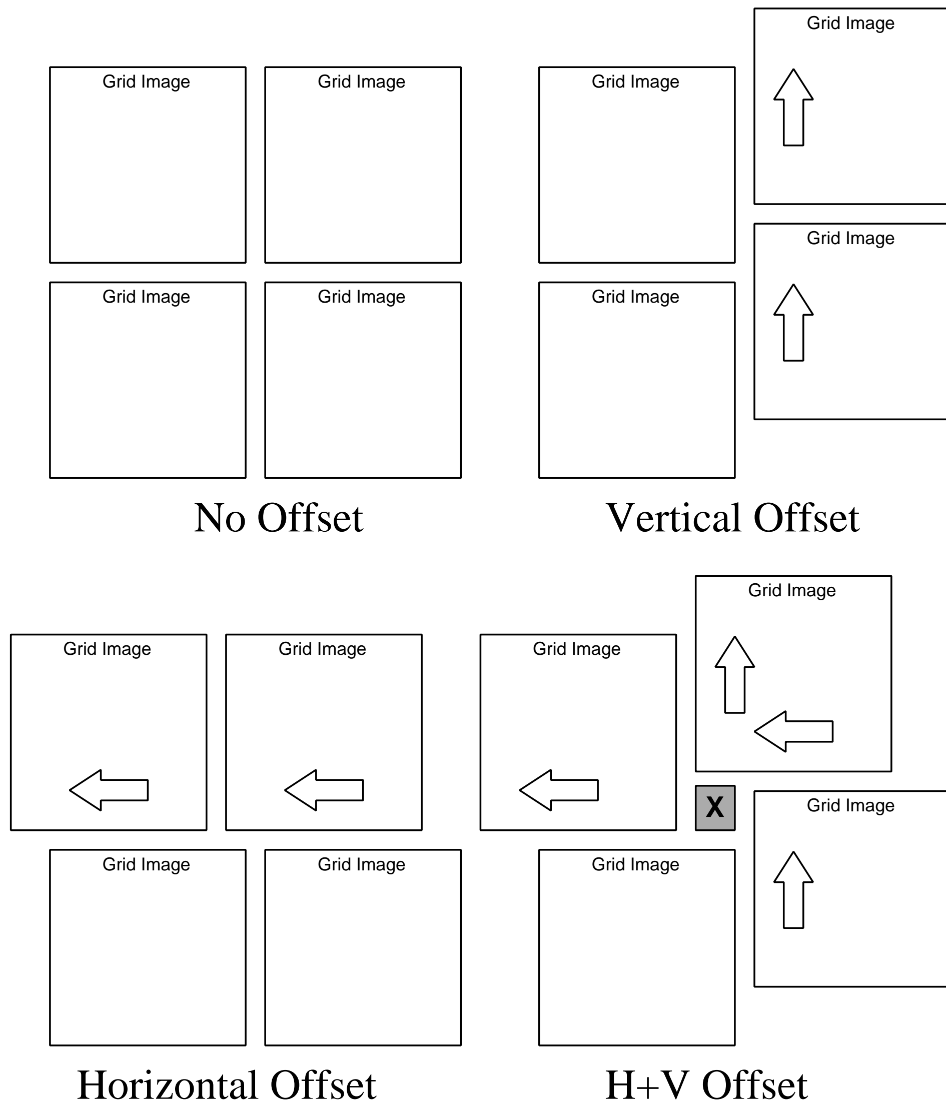


Figure 3.14: Filling the Gap

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)[52]. Given a pair of values $< n, o >$ which satisfy this condition $< n, n-o >$ will also be suitable. Offsets may therefore be categorised as large ($o > n/2$) or small ($o < n/2$).

If the previous argument is generalised to avoid clashes 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 $o = n-1$.

Any node v in a network G has an *eccentricity* $e(v)$ — the maximum distance $d(u,v)$ from any vertex u to v in G. The *radius* and *diameter* of G are respectively the minimum and maximum eccentricity among the vertices of G[52]. For a uniform surface such as the torus or twisted torus all nodes are equivalent, and hence the radius of the graph is equal to the diameter. For any network of processors a low value for the diameter d is desirable as it represents the worst case communications overhead. For a simple n by n mesh this distance is $2n$ and for a toroidal mesh the diameter is approximately n[53]. Any twisted mesh $< n, o >$ also has a diameter of n. This is independent of o. If $o = n$ the number of processors will be $2n^2$. Although the diameter of an <n,n> twisted torus is the same as an ordinary n by n torus it contains twice the number of processors.

This increase in density is achieved be reducing the redundancy, as can be seen in figure 3.15. In a standard torus there are three sets of paths from any point to any other of distance less than n. When an offset is present the redundancy is reduced, until for the final case $o = n$ there is only one set of paths to any point (the remaining redundancy is necessary to retain a regular surface — it will always be possible to travel east then north, *or* north then east). A similar result has been used to reduce the mean communications distance in a hypercube[1].

When the offset is increased the mean routing distance between pairs of points on the grid increases. When the offset is zero the mean distance between pairs of points on the grid is $n/2$. However as the offset is increased to n the mean distance tends to $2n/3$. Such a grid contains $2n^2$ processors. A regular torus of $2n^2$ processors would have a mean routing distance of $n/\sqrt{2}$. The mean routing distance for a twisted torus is therefore less than the mean routing distance for a regular torus of equivalent size.

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 should therefore be chosen in preference to smaller values.

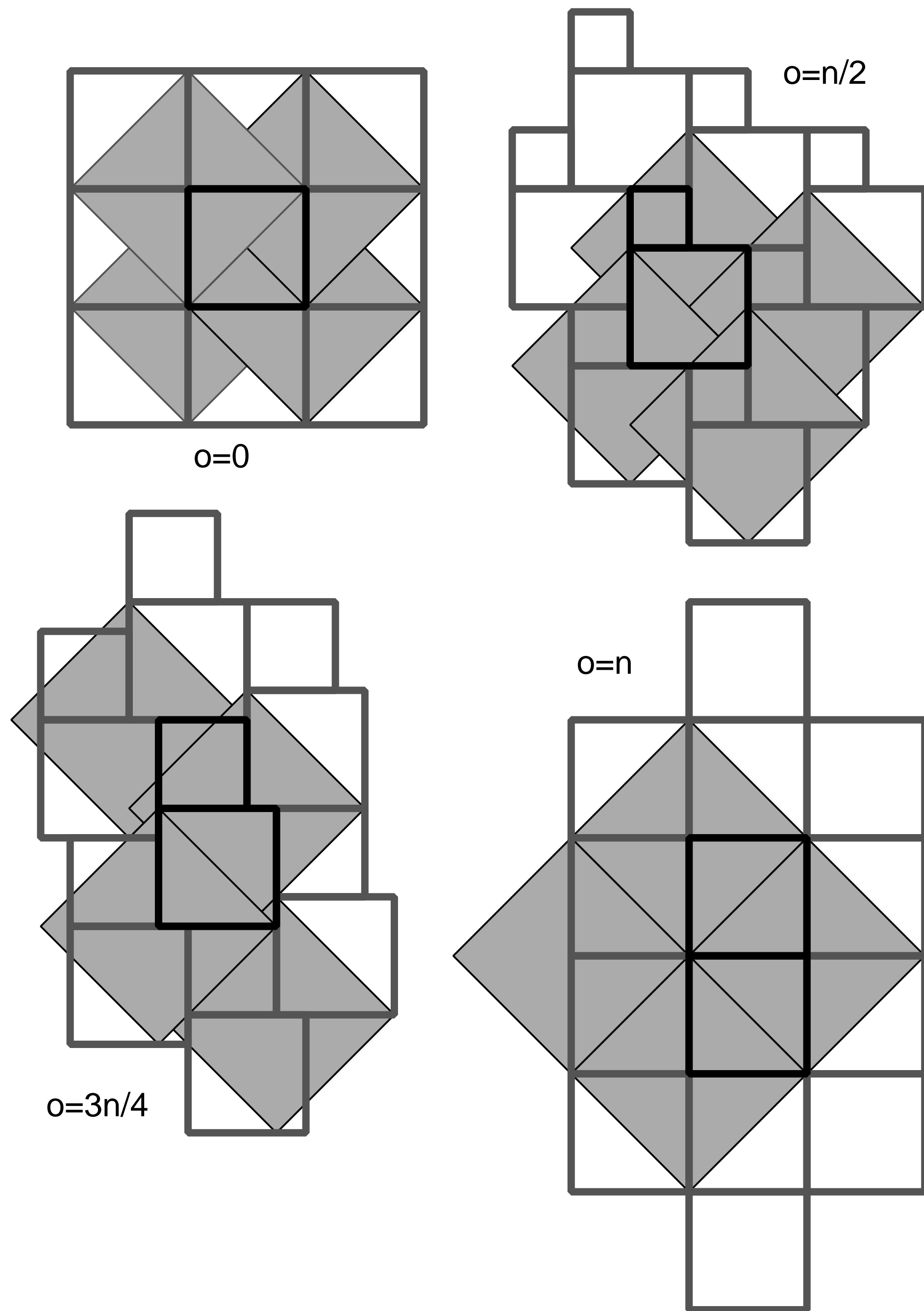


Figure 3.15: The diameter of Twisted Meshes

Grid Size: n	Small Offset:$o < n/2$	Large Offset:$o > n/2$	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 3.1: Some Viable Machine sizes

Values of o greater than n can be constructed. However in such cases the roles of n and o are interchanged: that is $< n, o > \equiv < o, n >$. It is simplest to consider only cases such that o is less than or equal to n, as no systems are excluded by such a limitation.

In order that a statistically reliable distribution of load between nodes is achieved the load should wrap around the torus as many times as possible. This demands that the total number of nodes be kept reasonable small, hence the restriction already established are sufficient to indicate which values should be considered for the development of a test system. A number of suitable machines are shown table 3.1.

3.5.2 Load Balancing

By applying the shift found in the twisted torus topology, regular patterns within loads are broken up. For the purposes of testing rectangular loads (one load unit in each virtual location within a rectangle of a chosen size) were applied to twisted and regular toroids as shown in figure 3.16. 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. This is shown in figure 3.17 where a rectangular load (randomly chosen to be 68 by 52) is applied to an 8 by 8 edge connected mesh. The mean load per node 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 the load is broken up around the surface, as show in figure 3.18. With an 8×8 grid any offset of between 2 and 7 results in an optimum balancing for the example load. Figure 3.17 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 —

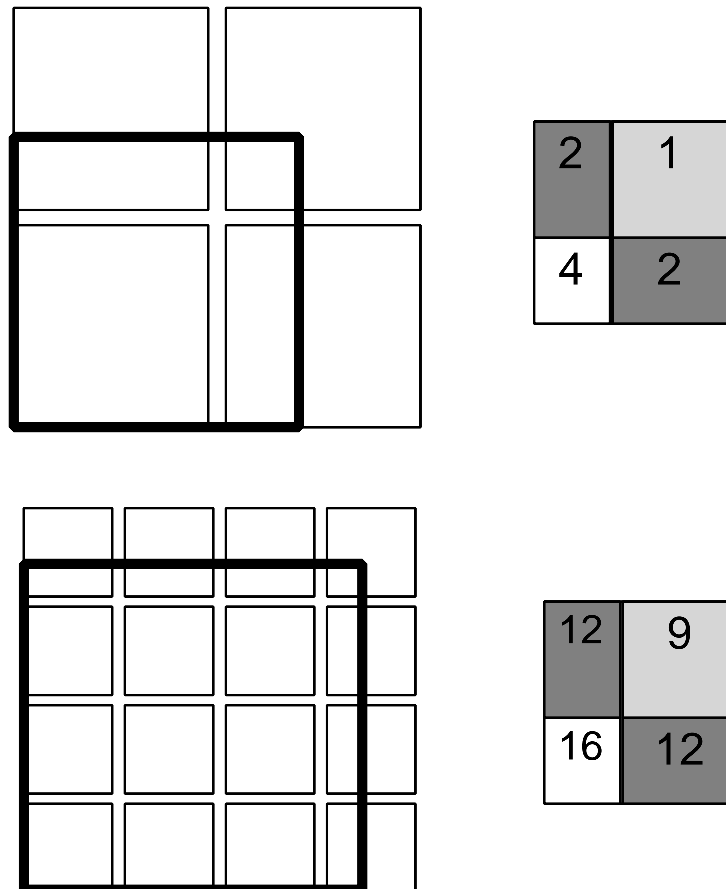


Figure 3.16: Applying a Rectangular Load to a Traditional Grid

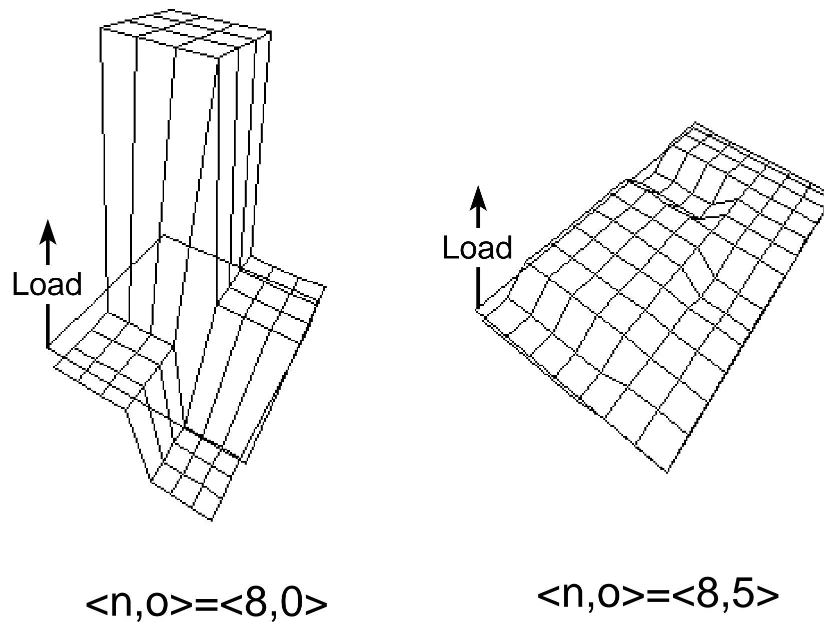


Figure 3.17: Applying a rectangular load

a dramatic improvement.

The standard deviation of the load was calculated for all rectangular loads up to forty by forty, applied to the eight by eight grid. The difference between these values and the best possible standard deviation (for that number of processors and load units) is shown in figure 3.19. Ideally this graph would be the plane $z = 0$, as this would indicate the system was optimally load balanced. However when an ordinary torus is used the surface has many high peaks representing poor performance. Only a few special cases (where the load is an exact multiple of the grid size) are well balanced.

An equivalent graph for and eight by eight grid with an offset of five is shown in figure 3.20. The surface here is a much better approximation to $z = 0$ indicating good load balancing for almost all loads. The mean load error is 0.15 when the offset is used, as opposed to 1.16 in the simple case. Despite the dramatic improvement in average performance the new grid does lack the simple perfectly balanced cases, as found in the pervious example. If the shape of the load may be selected by the programmer in an arbitrary fashion to fit the shape of the machine then there are a number of simple shapes which can be chosen which embed well in a simple torus. It is non-trivial to select such perfect shapes for a twisted

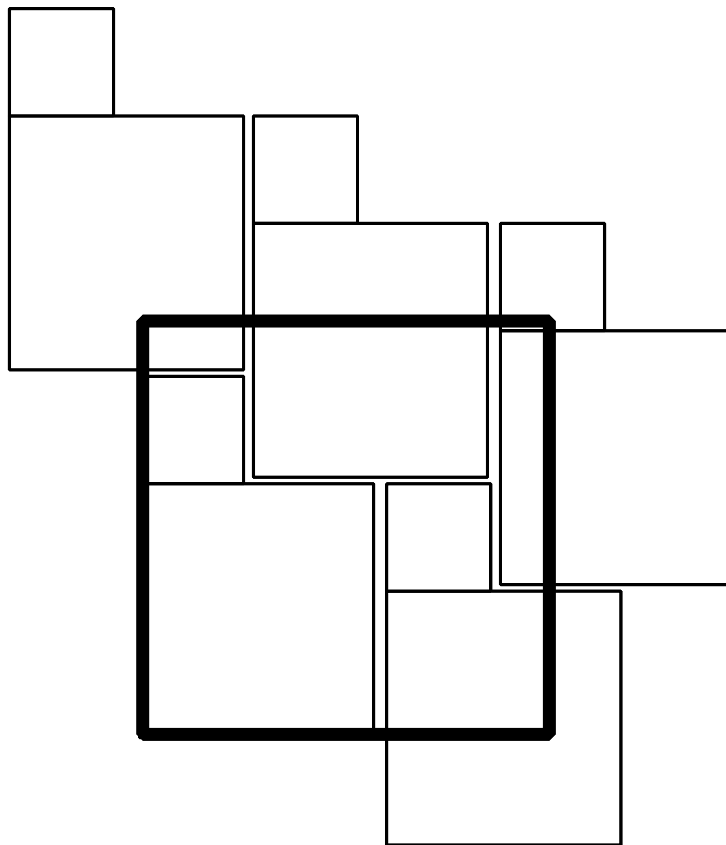


Figure 3.18: Applying a Rectangular Load to a Twisted Grid

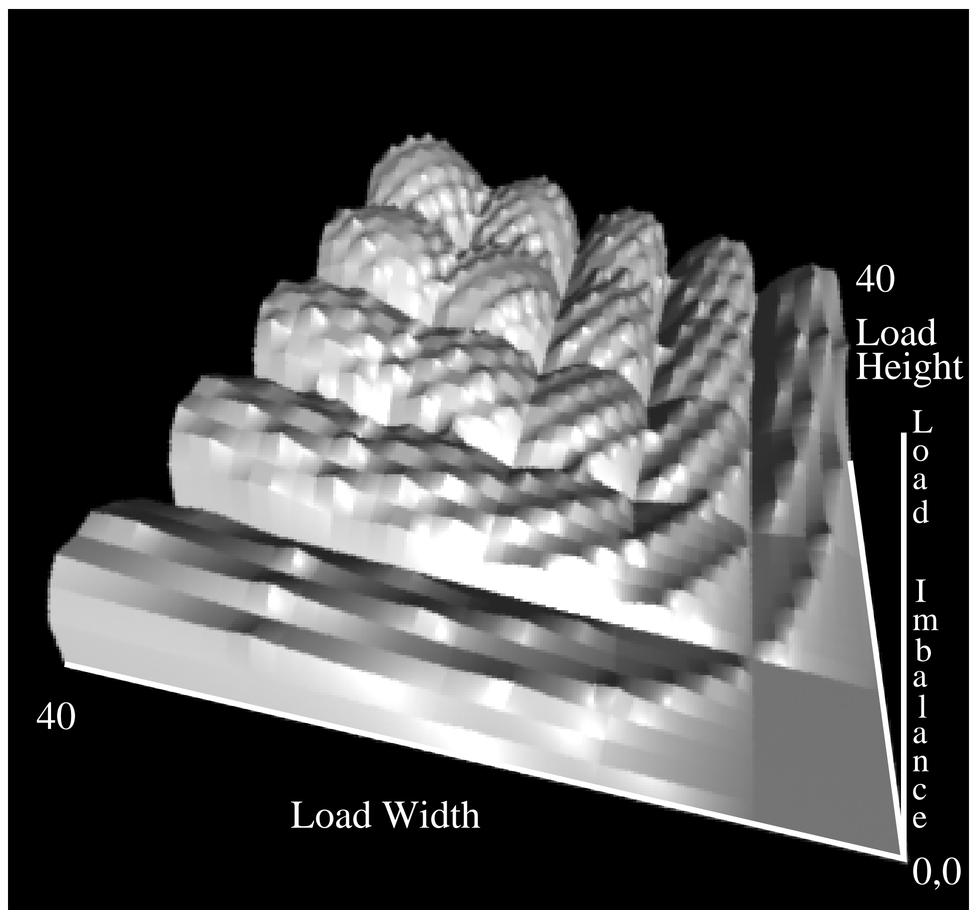


Figure 3.19: Loads applied to an 8 grid

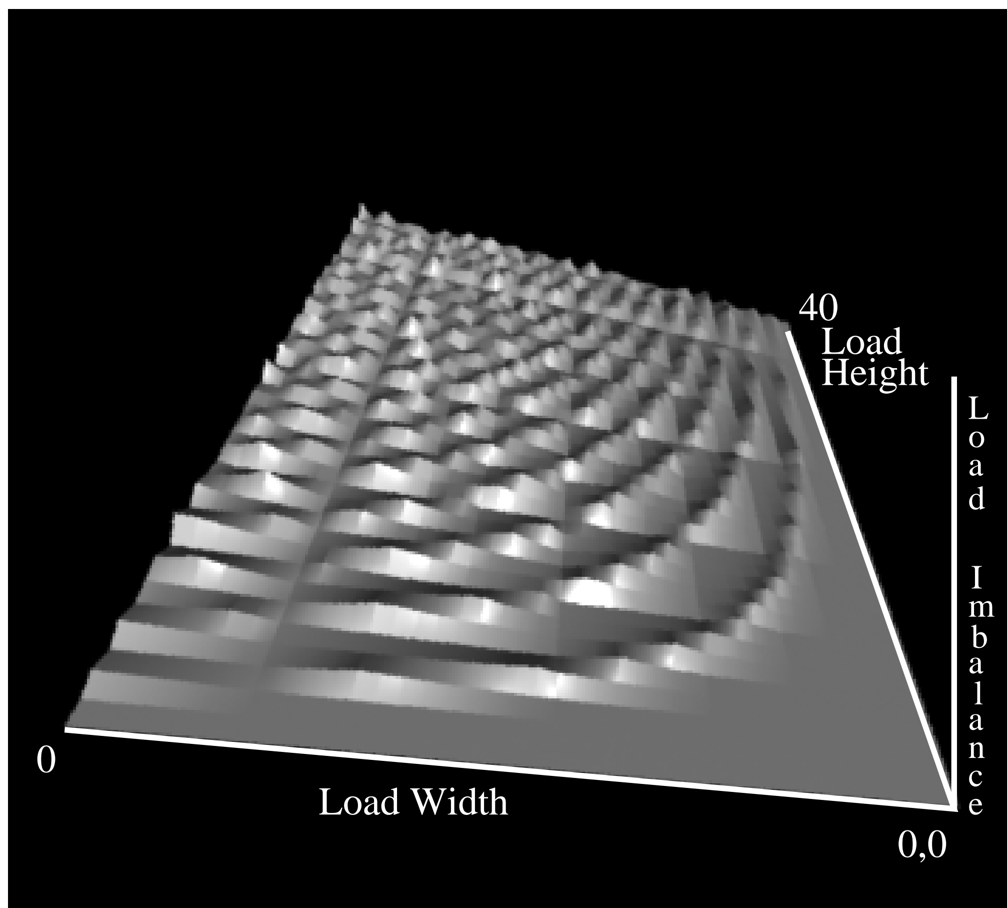


Figure 3.20: Loads applied to an $8 \times 8 + 5 \times 5$ grid

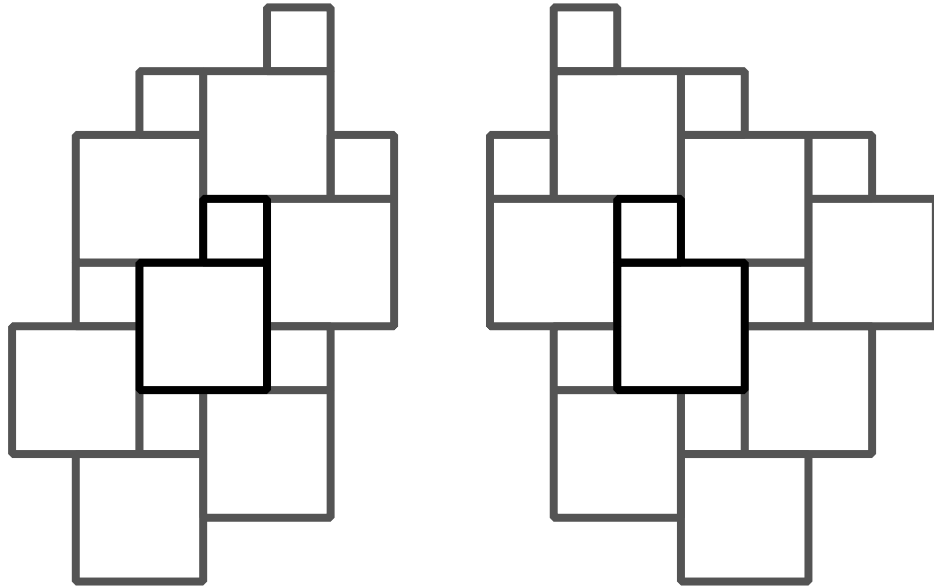


Figure 3.21: Two Similar Networks

torus. However for a generic machine running a selection of general problems of unknown shapes performance should be dramatically improved.

The above results have also been demonstrated with real loads taken from Creatures simulations. Geometric structures will be poorly mapped onto a traditional mesh, but are (generally) distributed evenly around the processor array by the spiraling technique.

3.5.3 Construction and Routing

So far spirals have been considered as being specified by two parameters — the basic size n, and an offset o. While this is sufficient to specify the computational properties, it does not completely describe a physical system as reflections and rotations may be used to transform between a number of possible implementations of a single topology. This can be seen for the two dimensional case in figure 3.21. While these are simply mirror images and hence computationally uninteresting they will change the way data must be routed around the system — a particular location in virtual space will map to a different location in real space.

When the system is implemented in two dimensional space the offset applied in either the X or Y direction must be negative (with respect to a conventional axis). This is necessary to prevent the surface overlapping with itself. While this is simply dealt with when

implementing a machine in a two dimensional space it was initially unclear how this would scale to higher dimensions. By the application of matrix techniques[61] the twisted torus topology may be described in a fashion more appropriate to higher dimensional systems.

The issues of construction and routing may be clarified by considering a number of "Origin" vectors, which map the origin of the surface onto itself. For a d dimensional space there will be d such independent vectors which will form a basis for the space. For the two dimensional examples in figure 3.21 these vectors are $\begin{pmatrix} n \\ o \end{pmatrix} \begin{pmatrix} -o \\ n \end{pmatrix}$ and $\begin{pmatrix} n \\ -o \end{pmatrix} \begin{pmatrix} o \\ n \end{pmatrix}$. These pairs may be merged into a transformation matrix T for each case: $\begin{pmatrix} n & o \\ -o & n \end{pmatrix}$ and $\begin{pmatrix} n & -o \\ o & n \end{pmatrix}$. The placement of the negative offset value determines which of the mirror image forms is produced.

The matrix T in each case would transform the unit square into a parallelogram with corners at the origins of the surface, as shown in figure 3.22. One such parallelogram may be associated with one image of the physical computational surface, and as both tessellate to fill space, must have the same area. Each parallelogram began as a unit square, and was transformed by the matrix T. The determinant of a matrix (in this 2D case) is of course the area scale factor. It is additionally known that for the surface to be continuous it must contain $n^2 + o^2$ processors (each of which may be taken as having unit area). Hence the determinant of T must be $n^2 + o^2$. This clarifies why a negative offset is required in one direction, as determinants are calculated by the evaluation of sub-determinants which are alternately added and subtracted from the total — only by applying a single negative offset will a matrix with the correct determinant be produced.

More importantly the matrix T may be used to simplify routing. In order to communicate effectively between virtual nodes it is essential that the physical node on which the virtual node resides be easily calculated[2]. Given a virtual processor, any integer combination of origin vectors may be added or subtracted, and the mapping to a physical node remains unaffected. The physical mapping may be determined by adding a combination of origin vectors, such that the virtual node comes to lie within the physical system. The exact definition of which location is the “real” location of a node is unimportant. It is sufficient that a unique location for a node may be calculated. Such a unique address may be simply arrived at by defining the physical system to exist within half an origin vector in any

[2]When implementing Creatures this information is only required to load positions into the system, and to dump them once simulation is complete. However the spiraled architecture has other potential application where communication at a distance may be required.

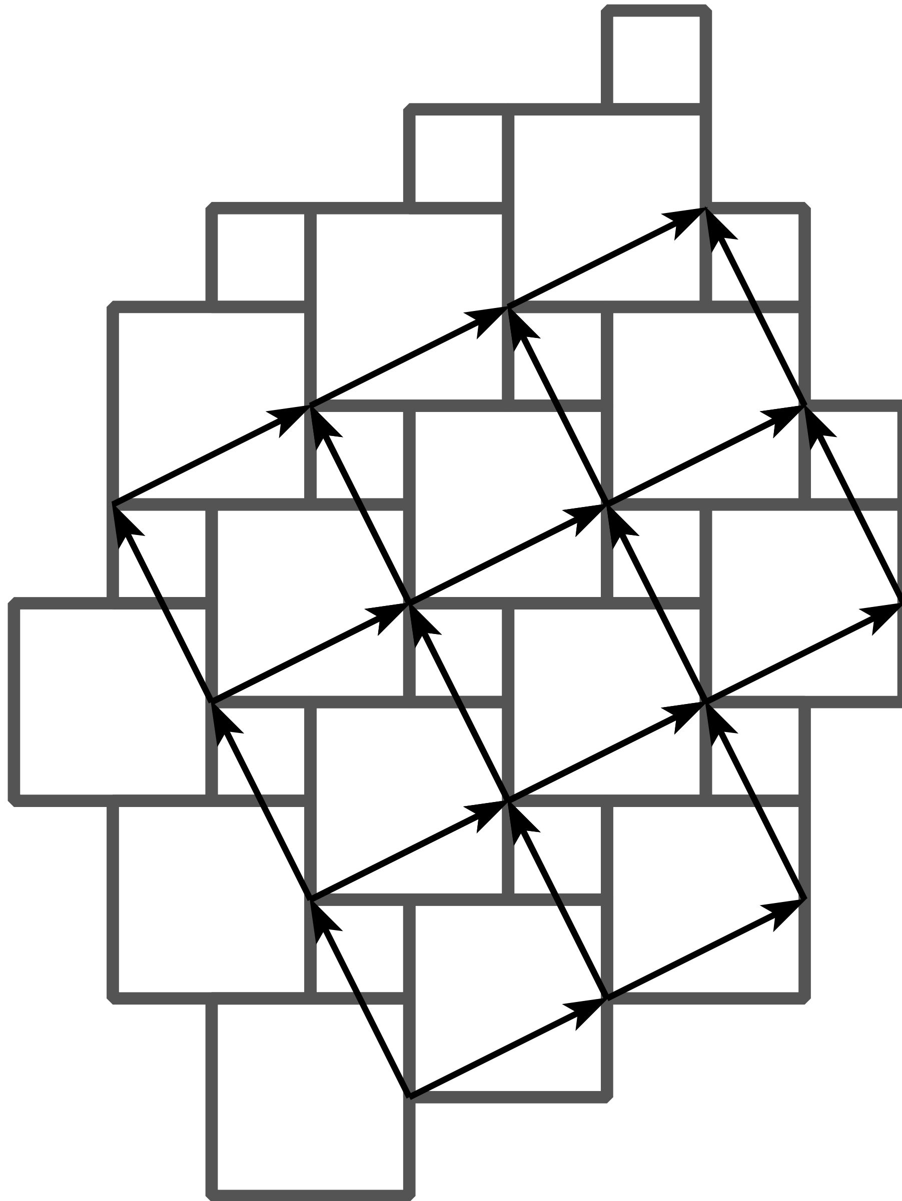


Figure 3.22: Origin Vectors Spanning Space

direction from the origin (as in figure 3.22). Unfortunately it is difficult to determine by simple inspection the correct combination of vectors which must be added to a location to move it into the physical space, as origin vectors may be non-orthogonal (in higher dimensions), and interact in a non-trivial fashion.

The matrix T maps the unit (square) vectors onto the origin vectors, and hence its inverse T^{-1} will map origin vectors onto the unit square. Once in this form the origin vectors are easily separated, and a mapping close to the genuine origin may trivially be found by taking the fractional part of each component, and then transforming back (via T) to the original coordinate system.

An optimal routing may be found between two virtual nodes by taking the virtual offset, transforming it to the orthogonal space, and choosing a new value for each component in the range $1/2 < c \leq 1/2$. This is then transformed back by T. Such an operation will produce a new offset from the original location which has had redundant moves through images of the machine removed, leaving the shortest possible route between the two points.

The T^{-1} matrix may be trivially calculated for both specific architectures, and for any required dimension. For the 2D case:

$$T = \begin{pmatrix} n & o \\ -o & n \end{pmatrix} \Rightarrow T^{-1} = \begin{pmatrix} \frac{n}{n^2+o^2} & \frac{-o}{n^2+o^2} \\ \frac{o}{n^2+o^2} & \frac{n}{n^2+o^2} \end{pmatrix}$$

This matrix is calculated once for each machine topology. At runtime, two matrix multiplications and two truncations are required to find an optimum path to the destination node. Though the technique as described required floating point operations, division is always by the determinant of the matrix which is fixed for any machine. It would therefore be trivial to develop a fixed point implementation which requires less computational resources, and could be run efficiently on the simple processors found in massively parallel systems.

3.5.4 Higher Dimensional Spirals

When machines are constructed in greater than two dimensional space the system becomes an N-cube with an additional smaller N-cube fixed in the corner of one "face" (the three dimensional case is shown in figure 3.23). As the dimension of the system increases the "face" acquires more corners, and the number of possible systems, represented by the possible choices for placement of negative offsets increases. The three dimensional case

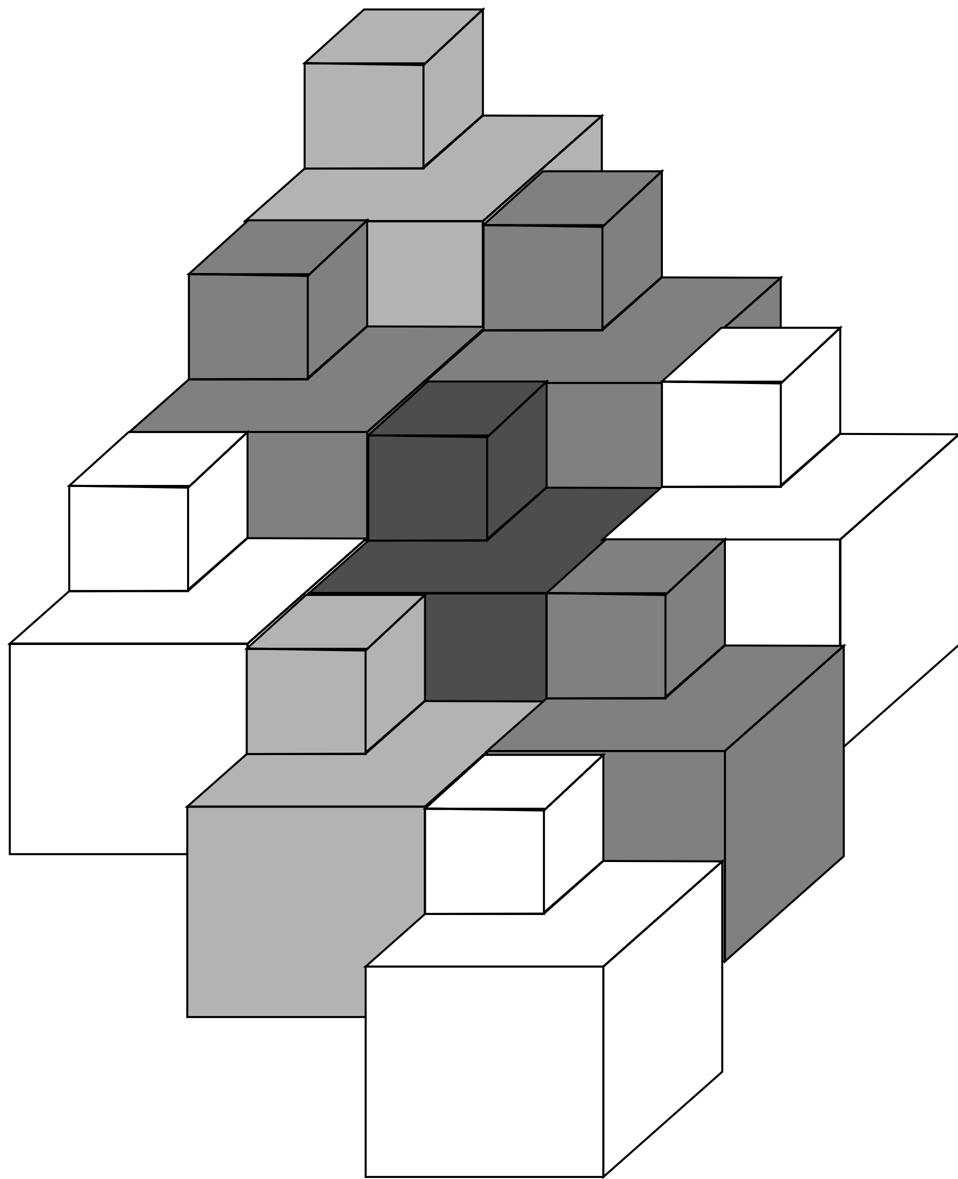


Figure 3.23: A Three Dimensional Twisted Torus

offers four unique values of T which produce the required determinant $n^3 + o^3$:

$$\begin{pmatrix} n & 0 & o \\ o & n & 0 \\ 0 & o & n \end{pmatrix}$$

$$\begin{pmatrix} n & 0 & -o \\ -o & n & 0 \\ 0 & o & n \end{pmatrix}$$

$$\begin{pmatrix} n & 0 & -o \\ o & n & 0 \\ 0 & -o & n \end{pmatrix}$$

$$\begin{pmatrix} n & 0 & o \\ -o & n & 0 \\ 0 & -o & n \end{pmatrix}$$

In fact there are others produced by considering trivial transformations of this matrix, but these produce identical shapes. In general there are 2^{d-1} interesting values of T.

These are best produced by placing n on the leading diagonal, then placing the value o immediately below. This is the simplest method of arriving at a configuration which satisfies the conditions:

- that all origin vectors have an n component and an o component.
- that the o component of a vector is orthogonal to the n component.
- that no two n components are parallel.
- that no two o components are parallel.

These specify that each image of the array shall have a single adjacent image on each "face" that is offset from that face. The direction of the offsets must then be adjusted so that none of the images overlap. It has been established that the determinant of the matrix should be the "volume" of the shape. If the images overlap they will be sharing volume, and hence the determinant will be less than $n^2 + o^2$. It is shown in the following section that for all dimensions the number of positive offsets must be odd to satisfy this condition. There are 2^d combinations of positive and negative values, half of which will have an odd number of positive offsets, hence $2^d/2 = 2^{d-1}$ interesting values of T. For $d = 2$ one negative offset is required , $d = 3$ requires either zero or two negative offsets. From these a matrix may be picked arbitarilly, and used to construct a machine.

Consider the example $d = 4$:

- Construct a diagonal matrix of dimension d with the value n on the leading diagonal:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 0 & n \end{pmatrix}$$

- Place the value o directly underneath each n

$$\begin{pmatrix} n & 0 & 0 & o \\ o & n & 0 & 0 \\ 0 & o & n & 0 \\ 0 & 0 & o & n \end{pmatrix}$$

- Evaluate the determinant:

$$\begin{vmatrix} n & 0 & 0 & o \\ o & n & 0 & 0 \\ 0 & o & n & 0 \\ 0 & 0 & o & n \end{vmatrix} = n^4 - o^4$$

- If this does not equal $n^d + o^d$ reverse the sign of any one o

$$\begin{vmatrix} n & 0 & 0 & o \\ -o & n & 0 & 0 \\ 0 & o & n & 0 \\ 0 & 0 & o & n \end{vmatrix} = n^4 + o^4$$

- Evaluate the inverse T^{-1}

$$T = \begin{pmatrix} n & 0 & 0 & o \\ -o & n & 0 & 0 \\ 0 & o & n & 0 \\ 0 & 0 & o & n \end{pmatrix}$$

$$\Rightarrow T^{-1} = \begin{pmatrix} n^3 & -o^3 & n\,o^2 & -\left(n^2\,o\right) \\ n^2\,o & n^3 & o^3 & -\left(n\,o^2\right) \\ -\left(n\,o^2\right) & -\left(n^2\,o\right) & n^3 & o^3 \\ o^3 & n\,o^2 & -\left(n^2\,o\right) & n^3 \end{pmatrix} / n^4 + o^4$$

The physical layout resulting from a particular layout is not always clear, particularly for higher dimensions. The best method of constructing a system from the matrix specification is to test for the presence of processors at a number of virtual locations (greater than the number of processors required) which are guaranteed to include all physical locations (for example $0 < x < n$, $0 < y < n + o$ for the two dimensional case). Processors are

inserted when a processor is found to be lacking at the tested location. When this procedure is completed it has been established that every virtual location within the area contains a processor, and hence every possible physical location contains a processor. Such a technique is easily automated, and could be used to develop less abstract designs for real machines.

3.5.5 A Proof of the Negative Offset Effect

It has been observed that for even dimensions it is necessary to place an odd number of offsets in a negative direction (Sequin [56] notes this as an oddity for the two dimensional case). This may be shown mathematically as follows.

The general form of the matrix T

$$T_d = \begin{pmatrix} n & 0 & \cdots & 0 & o \\ o & n & 0 & \cdots & 0 \\ 0 & o & \ddots & & \vdots \\ \vdots & & \ddots & n & 0 \\ 0 & \cdots & 0 & o & n \end{pmatrix}$$

may be reduced using the first column to find its determinant. The resulting equation will have two components — that resulting from the n in the first column, and that resulting from the o below it. The determinant will be of the form: $|T_d| = n|U_{d-1}| - o|V_{d-1}|$ Each of these components will be considered in turn.

The n component of the T matrix's determinant will be n times the determinant of the bottom right sub-matrix. This matrix has the simpler form of

$$U_{d-1} = \begin{pmatrix} n & 0 & \cdots & \cdots & 0 \\ o & n & 0 & \cdots & 0 \\ 0 & o & \ddots & & \vdots \\ \vdots & & \ddots & n & 0 \\ 0 & \cdots & 0 & o & n \end{pmatrix}$$

By considering reduction along the top row it can be clearly seen that the determinant of this matrix will have the value of n times the determinant of the bottom right matrix. This is again in the form of a U type matrix, and hence $|U_{d-1}| = n|U_{d-2}|$.

For the simple case of U_2

$$|U_2| = \begin{vmatrix} n & 0 \\ o & n \end{vmatrix} = n^2$$

(Alternatively $U_1 = n$ may considered as a trivial case). Hence $U_k = n^k$.

The first component of the determinant of T_d is therefore $n|U_{d-1}| = nn^{d-1} = n^d$.

Obtaining the second component of the determinant of T takes a similar form. This component may be considered as $-o$ times the determinant of the submatrix V_{d-1}.

$$V_{d-1} = \begin{pmatrix} 0 & \cdots & \cdots & 0 & o \\ o & n & 0 & \cdots & 0 \\ 0 & o & \ddots & & \vdots \\ \vdots & & \ddots & n & 0 \\ 0 & \cdots & 0 & o & n \end{pmatrix}$$

This is again reduced, now using the first column. The resulting submatix is again in the form of V, hence $|V_{d-1}| = -o|V_{d-2}|$.

In the simplest cases

$$|V_2| = \begin{vmatrix} 0 & o \\ o & n \end{vmatrix} = -o^2$$

(or alternatively $|V_1| = o$). Hence in the general case $|V_k| = o(-o)^{k-1} = -(-o)^k$.

The second component of T_d is therefore $-o|V_{d-1}| = -(-o)^d$.

Combining these two results:

$$|T_d| = n^d - (-o)^d$$

When d is odd this will reduce to the form $|T_d| = n^d + o^d$. When d is even $|T_d| = n^d - o^d$.

In order that the system be operable T_d must be of the first form, as previously discussed. This may be corrected for even values of d by reversing the sign of one o. Any pair of o's may additionally be negated as their effect clearly cancels out.

3.6 A Transputer Implementation

3.6.1 Hardware

A small two dimensional spiral was constructed using transputers by adapting an INMOS B042 board. It was proposed that the development of such a system would demonstrate that a spiraled system could be built, and that such a system could provide a platform for a high speed implementation of Creatures on semi-custom hardware.

The B042 board consists of 42 T8 transputers[41] connected in a two dimensional 6×7 NEWS grid as shown in figure 3.24. The unconnected edge links are available to the user,

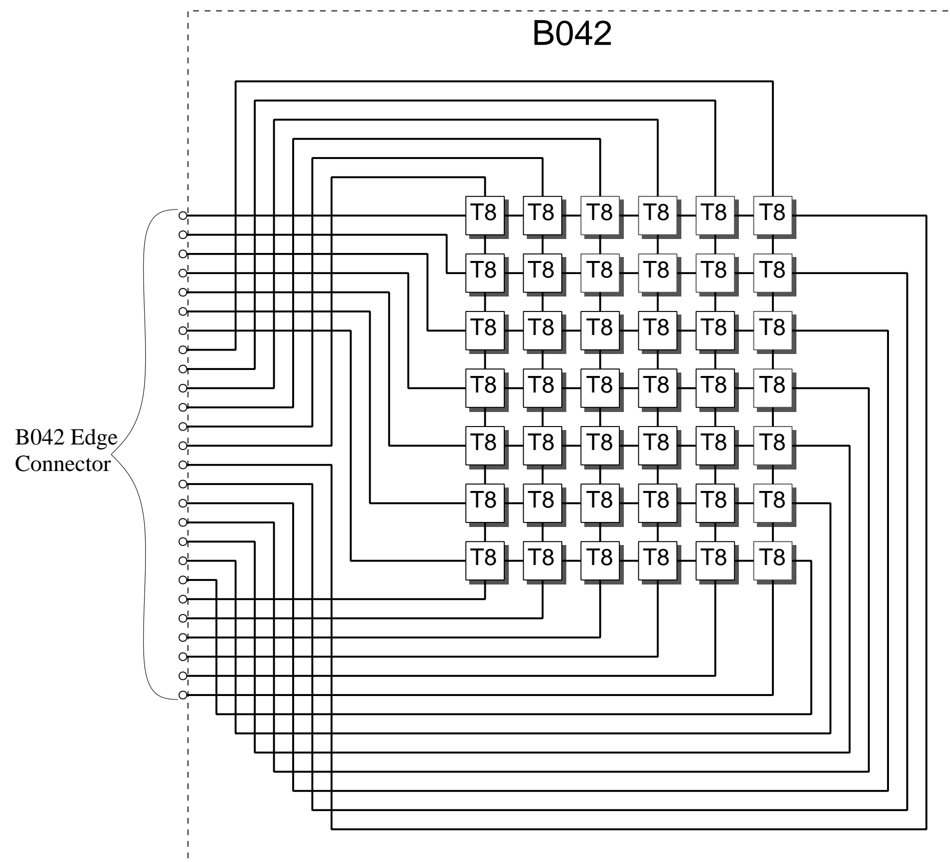


Figure 3.24: The INMOS B042 Transputer Board

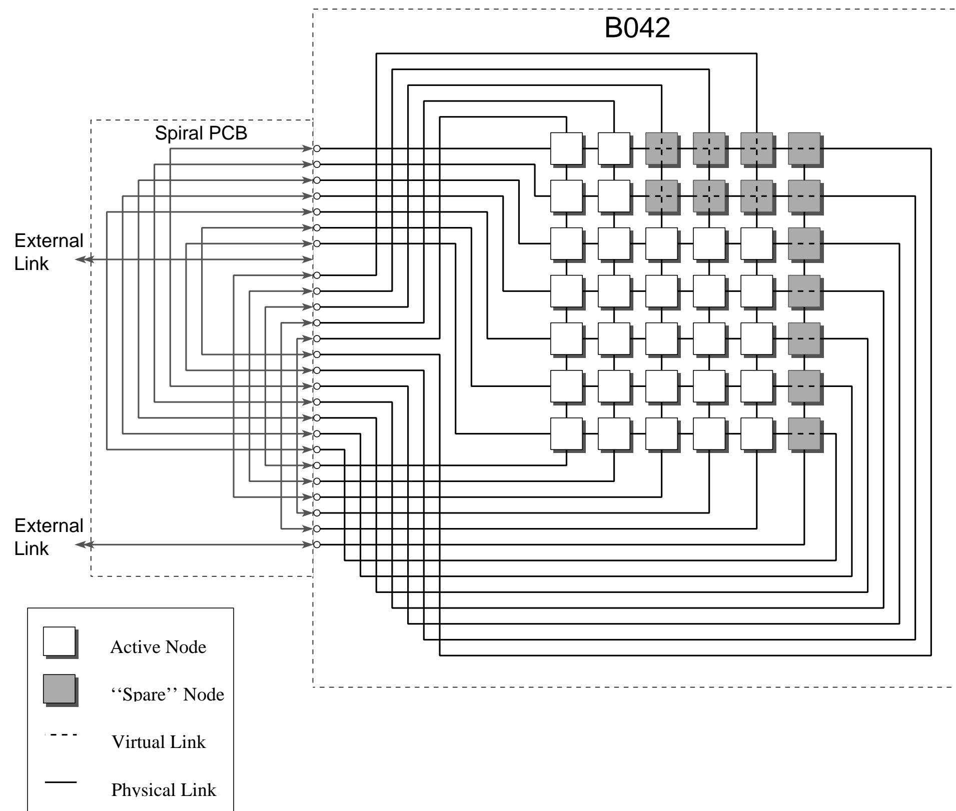


Figure 3.25: A Spiral Embedded within a B042 Board

and may be connected back into the board to modify the topology. In addition a number of auxiliary control signals are available. There is no memory on the board other than the transputers internal memory (four kilobytes per processor).

A spiral of type $< n, o > = < 5, 2 >$ was selected to be embedded into the board, as this is the smallest desirable shape (as derived in section 3.5.1), and fits within the board with a minimum of wasted nodes. The remaining "spare" transputers operate transparently while the system is running, copying the data on their links straight through without modification. The edge connector provides enough flexibility to allow this topology to be constructed, as shown in figure 3.25. While this wastes a number of transputers, it is simple to implement, and the column of transputers on the far right provides access to the array via the spare links, allowing data to be inserted into the system by routing process which may placed there.

A PCB to implement this was constructed, and the resultant hardware was attached to a B008 board via one of the spare links. This provides an additional transputer known as

the “root”, with additional memory (typically one Megabyte). One link of this processor is connected via additional hardware to a host machine. The host will typically run an “iserver” which provides IO services to the transputer board. Because of its position in the network the root transputer must handle these IO operations, and hence the root operates as a controller for the rest of the network.

In the hardware configuration used for the development of the Creatures software, the connection between root transputer and the host machine was provided by a B300 based ethernet system. While this provides a flexible development system available to many users spread across a network, in a host independent form the performance of the B300 link is very poor. Fortunately, the link is transparent to the user, aside from its poor performance and hence could be replaced by a more effective connection should it be required.

3.6.2 Software

A Creatures simulator was developed in Occam[42] using the Inmos toolkit, to run on the transputer hardware. This consisted of a communications layer which provided facilities for transferring creatures around the network, and code dealing with creatures upon each node.

An individual node operates using the same algorithm as the original serial implementation. A single list of all creatures upon the node is maintained, and a “cansee” list is calculated for each creature as it is required. The creature is then stepped using code produced by a retargetted version of pancake. The resulting creature(s) may then be passed into the communications code where they are passed to adjacent nodes as necessary.

The communications layer passes a list of new creatures to adjacent nodes at each timestep. Synchronisation is retained by terminating each list with a marker. Communication proceeds in each direction in parallel. This allows computation on adjacent nodes to be up to half a cycle out of phase with adjacent nodes. For example: one node may complete its calculation; it would then be able to enter into communications with adjacent nodes, some of which may have had less computation to perform and little data to pass, hence completing communication rapidly. These nodes may then start calculation of the next step, while other nodes may have yet to complete the previous calculation, and enter the communications phase.

Computation may only begin once all communication has been completed for that node. However this semi-synchronous coupling between nodes allows large amounts of clock

skew across the board, and hence improved performance compared to a more strictly timed system.

A “loader” process connected to the iserver inserts creatures into the system having first calculated which node their location requires they be processed upon. This calculation is preformed using the matrices described in section 3.5.3. The population of creatures is fed into the system over a number of timesteps to prevent the nodes close to the insertion point overflowing. The system then runs purly as a communications system until it can be guaranteed that all creatures have reached the correct node. Once the system is loaded the loader process starts collecting creatures which have completed their computation, and returns them to the iserver. Because of its communication with the iserver this process must be run on the root processor.

Each creature carries within it a counter which is decremented at each time step. When this counter reaches zero the node holding it will attempt to route it back to the loader. In addition each node holds a counter, and will shut down following a pre-determined number of steps. This node counter must be significantly larger than the counter within the individual creatures, as nodes are required to operate both during the preloading of the grid, and during the dumping phase.

A through router process was placed upon the unused processors. Those at the top of the board (figure 3.25) simply route for a number of steps. Those forming the column at the edge of the board must provide a more complex service as shown in figure 3.26. Each channel contains a series of creatures with end flags to mark each time step. Creatures are read from the south and merged into the west/east data stream, such that creatures remain in the correct time step. The “gen” process sends clock ticks north to feed the south input of the processor above it, as all the processors in this column are identical. Creatures from the east and the west are passed through the node unchanged unless their timer has expired, in which case they are passed south with any creatures received from the north.

The Inmos toolkit provides relatively flexible mapping between processes, virtual processors, and physical processors. The software was therefore developed to map onto the hardware, then mapped onto an identical set of virtual processors. These virtual processors could then be mapped either directly to the real hardware or to a single processors for debugging and performance comparison. However memory limitation did limit the effectiveness of this approach.

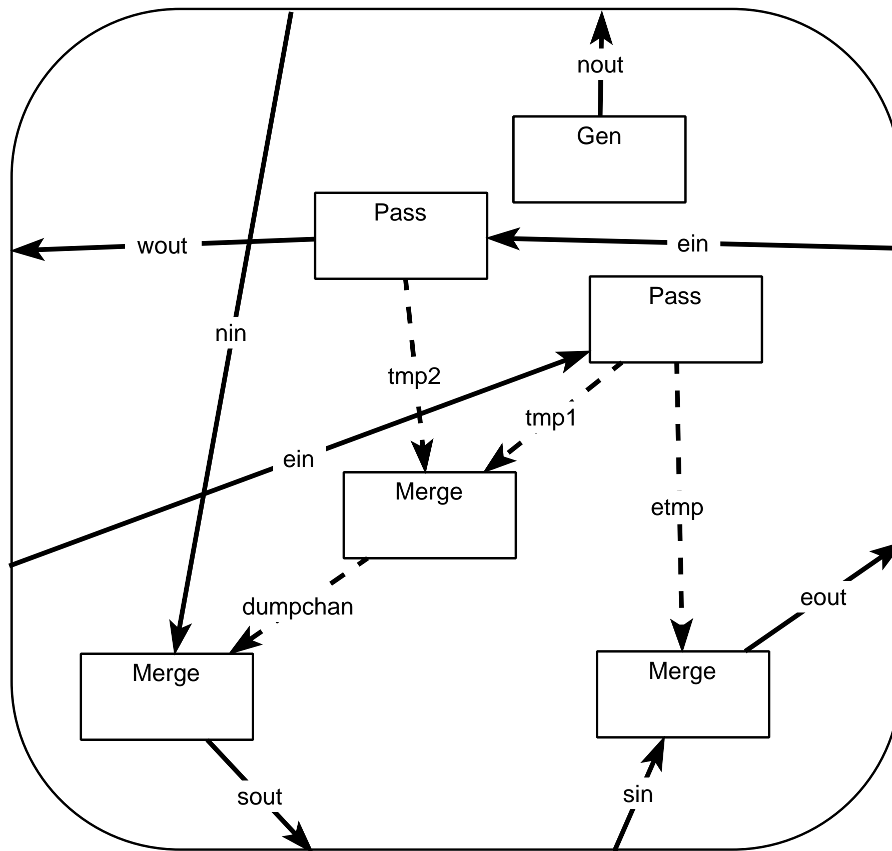


Figure 3.26: Routing In The B042's Spare Coloumn

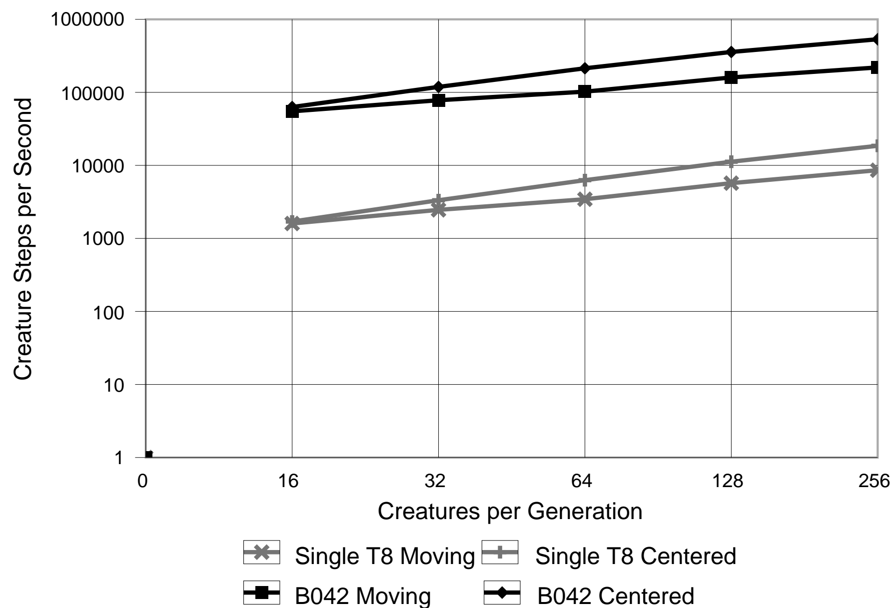


Figure 3.27: Performance of the Transputer Implementaion

3.6.3 Performance

The system described was developed and debugged, and successfully run on a single transputer. However memory limitations of the B042 system prevented the full Creatures code being run in a truly parallel fashion. In order that the performance of the software could be tested on the limited hardware available much of the auxiliary code was removed from the simulator. This code dealt with the loading and saving of creatures, and the long range communications across the network required to perform such operations. By hardcoding simple start states it was possible to produce a system capable of operating in 4K of memory. However memory limitations still restricted the population sizes which could be tested.

The restricted system was run on both a single T8 transputer (with 1Meg of RAM, capable of holding the complete network), and the B042 system. The topology of the virtual network used in both cases is identical. The system was tested with both creatures remaining stationary (to test the performance of the calculation, and the mapping operations), and moving at every timestep (to demonstrate that spiraling reduces the communications overhead). The results of these tests are shown in figure 3.27.

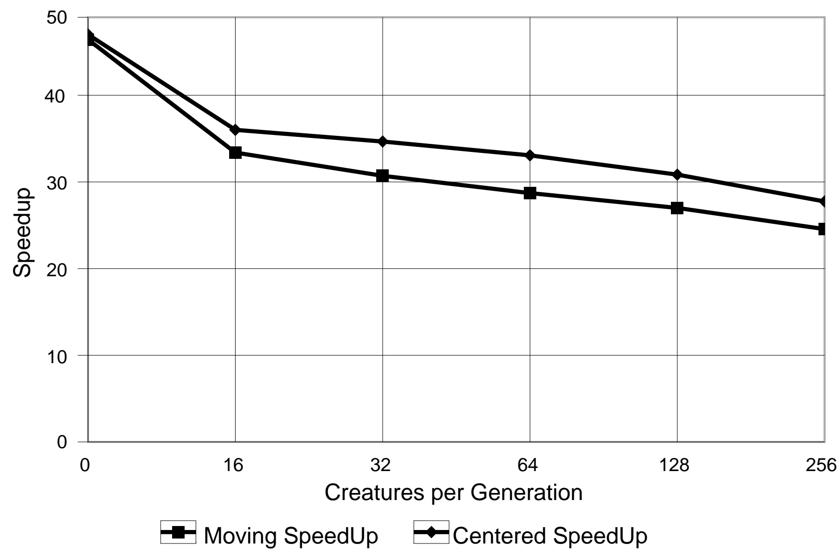


Figure 3.28: Speedup using the B042

The graph shows that as the population is increased the performance of the system improves. This is due to the reduced overheads, and the utilisation of all the slots within the simulator. Similar performance can be seen in all other parallel implementations of the Creatures model. Unfortunately due to the memory limitation the test only reveal the beginning of this curve where mapping is not significant. Even at the largest population size only ten creatures are allocated to each bucket. Were the system to be build with more memory per node larger populations could be accommodated. As population increased the mapping operation would again come to dominate, as it is still an N^2 operation within a single mode, and performance measured in Creature steps Per Second would fall off with $1/N$. In such a case performance could be improved further by using an additional hashing function within each single bucket.

The simulation operates faster when creatures are not moving, as under these conditions very little data must be passed between nodes. However the differences in performance between the moving and stationary tests is very small compared to the order(s) of magnitude measured on the Connection machine. This illustrates that the Spiralled mapping is effective at reducing communications overhead.

An interesting result may be seen be examining the speedup found in the system (shown in figure 3.28). Speedup is approximately 45 for small populations (the >1 efficiency may be

attributed to experimental error). However as the population increases the speedup falls to around twenty eight. This is counterintuitive, as it is generally expected that speedup will improve for larger populations.

The anomaly in speedup can be attributed to the way the twisted topology is embedded within the B042. As was shown in figure 3.25 the board contains 42 processors which are all used as part of the network. However only 29 of these are actually used for computation — the others providing routing services. When the load is low, all nodes require similar amounts of work, so the 42 processors may all operate effectively. However when the number of creatures is high, the load on the computing nodes is far higher than that of the routing nodes. Under such conditions a single processor can divide its time efficiently between all the nodes. However in the parallel case only 29 of the processors are performing significant amounts of work, hence the drop in speedup from approximately "42" to "29".

Examination of the speedup in this fashion indicates that despite the unusual behaviour of the system it is in fact producing a linear speedup. From this result and an examination of the structure of the system which may be seen to be free from bottlenecks it may be predicted that larger systems constructed using this topology would perform similarly well.

The peak performance from the transputer implementation was in excess of 500,000 creature steps per second. The CM2 implementation managed only 128,000. More importantly performance exceeded 200,000 steps per second when creatures were moved around the grid. Though this may appear to be a serious degregation in performance, the CM2's performance dropped by a factor of twenty when movement was introduced. Given the large number of creatures migrating round the system, and considering that in a real system not all elements may choose to migrate at every timestep, this drop in performance is considered acceptable.

Though the performance of the B042 system compares most favorably with the 10,000 CPS obtained from a serial workstation based solution, the implementation is not currently suitable for practical use, due to the severe limitations it places on population sizes. Once these have been resolved, and a suitable user interface installed, the transputer implementation should provide an excellent platform for the development of Creatures simulations.

3.7 Comparison of Performance between Implementations

The performance of each implementation is shown in figure 3.29. The values selected to

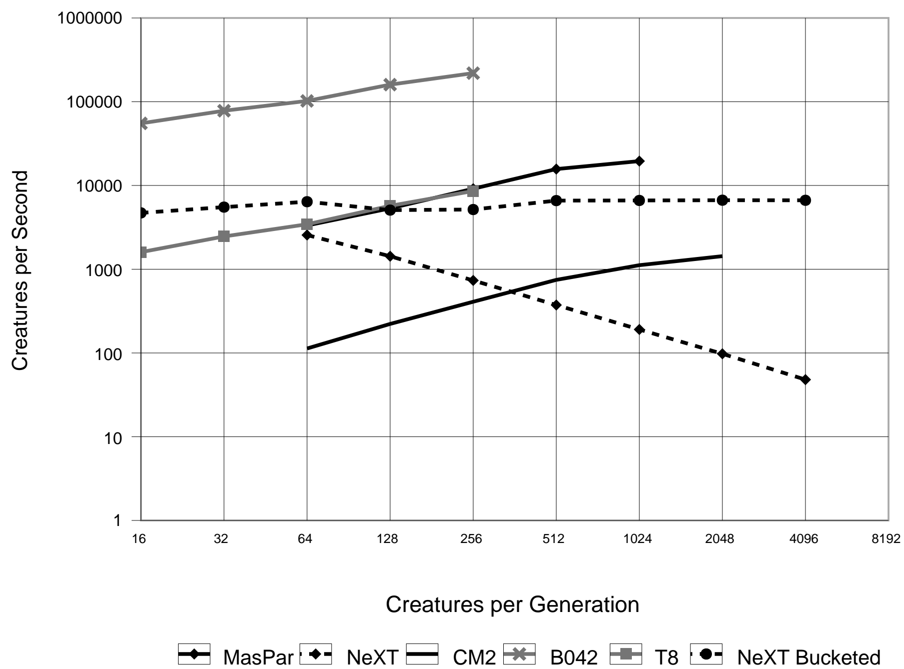


Figure 3.29: Performance of the Implementations

represent each platform are those of the fastest version of the system when all creatures were moving (this being more appropriate for practical use than the stationary case).

The bucketed NeXT implementation provides good performance over a large range of population sizes. In addition it is the easiest to use, and provides greater functionality than the other simulators. It has been extensively tested during the course of this research, having been used to develop the applications discussed in the chapter 4, and is therefore known to be relatively bug free, and stable. The dynamic nature of the implementation ensures that the system will not crash as a result of bucket overflow.

The Connection Machine implementation produced very poor results. Despite being (on paper) the fastest machine used during the project, the implementation of Creatures is slow. This is primarily due to communications overheads, the CM2's hyper-cube network being poorly suited to the Creatures problem. In addition the system was somewhat fragile, as each node has a limited capacity, so locations could easily run out of memory.

The MasPar implementation was based on identical code to that used on the Connection Machine, and hence was equally fragile. In fact the more limited configuration options of the MasPar hardware resulted in a system that was even more limited in its capacity, as the Connection Machine supports virtual processors which may be configured to best suit a particular simulation. However the MasPar's communications grid, and global routing hardware proved far more effective than the Connection Machine's wormhole routing at dealing with the migration of creatures through space. While performance was only slightly better than the serial NeXT based system, MasPar systems are available at a fraction of the cost of a Connection Machine.

Having learnt from the previous implementations, the transputer based system's performance was excellent. In addition to being the cheapest system it was also the fastest, and most scalable, providing linear speedup. It proved possible to design an implementation such that the movement of creatures had little impact on the systems performance. However the hardware available for development prevented the full Creatures system being run, and severely restricted the population size. Were a fully operational system developed, the evidence collected indicates it would perform excellently.

4

Applications

During the development of the Creatures system, a number of simulations were created, both to test the implementation, and to explore and illustrate the computational model. A number of these are presented in this chapter, showing both the application of creatures to a range of systems, and recording some of the approaches developed for implementing them.

The examples have been broadly grouped into CA style problems and "more complex" models. The former set of examples are reworkings of classic CA problems where the state of each creature is small, and behaviour is deterministic. These illustrate how the dynamic nature of Creatures produce solutions which are generally simpler, more intuitive, and sometimes more efficient than the CA equivalent.

The latter group of problems are best considered as demonstrations of how Creatures could be used as a practical research tool in a number of application fields. It is not claimed that the models developed are accurate simulations of real systems, as such validation is beyond the scope of this work. However they do illustrate how Creatures may be used to build larger systems. Typically these models are non-deterministic, and allow each Creature to hold a larger internal state.

The final problem of this second group demonstrates the complete development of a simulation from a physical system of interest, through the implementation of an abstract simulation to the statistical analysis of results obtained from the simulations. This analysis provides insight into the original physical problem.

```
NEIGHBORHOOD: vn;

TYPES: box, np, ep, sp, wp;

RULE:
    {
iam(box):      CENTER;

iam(np):
        {
        CANSEE(sp):
            {
            BECOME(ep);
            EAST;
            }
        CANSEE(box):
            {
            BECOME(sp);
            SOUTH;
            }
        true:
            NORTH;
        }
    }
```

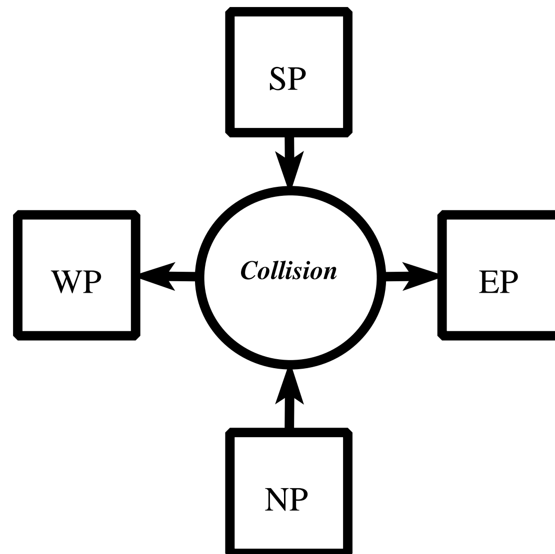


Figure 4.1: An Ideal Gas Simulation

4.1 Classic CA Problems

4.1.1 An Ideal Gas

One common use of cellular automata is in the modeling of gases [28]. Cells attempt to represent the presence (or otherwise) of gas particles at each location. Although the results are good, the techniques used to maintain data integrity, are complex (Margolis neighborhoods for example[63]). The Creatures model offers an intuitive alternative: BOX creatures are used to surround a collection of gas particle creatures which move around and bounce off each other. Figure 4.1 shows the definitions of BOX and one type of gas particle NP. The particles EP, SP and WP are similarly defined. The Creatures simulation is simpler

than CA, as it allows the behaviour of the active agents in the system (gas particles) to be described directly, rather than indirectly though the behaviour of space.

The four basic particle types used: NP, SP, EP and WP move NORTH, SOUTH, EAST and WEST respectively. Particles only interact with particles moving in the opposite direction. Upon encountering such a particle they turn right through 90 degrees. This is effectively an elastic collision, and momentum is conserved as required. In addition particles will rebound from box particles which surround the gas.

It is necessary to have four types of particle, as the interactions between particles depends on being able to examine the momentum term of the other particle. Momentum must therefore be stored in a publicly readable form, ie. as the type.

This model may be developed to include diffusion limited aggregation by adding the creature type ICE. This has the same behaviour as the creature BOX, but upon encountering a creature of type ICE the particles rebound, and then becomes ICE.

4.1.2 A Model of Digital Logic

The Creatures model may be used to build digital logic simulations. Unlike the techniques used to tackle this problem with CA[14], the methods involved in producing logic systems with Creatures are relatively intuitive. A minimal system is shown in figure 4.2.

A creature type SIGNAL is defined which will move FORWARD until it encounters a logic gate (FORWARD may be defined in an additional macro package by redefining the regular directions to store their direction in the creature's internal state. FORWARD then repeats the previous move). This creature represents a signal of logic one. Upon encountering a gate the creature DIEs, as the task of transmitting information is complete. The gate may then produce a new SIGNAL to carry information through the system to the next gate. It should be noted that SIGNALs do not interact with each other. This allows logic streams to cross in a similar fashion to beams of light.

The gates are of the OR type, as this is best suited to the Creatures environment. If an OR creature observes a SIGNAL creature it creates a new OR creature to takes its place, then becomes a SIGNAL, and starts moving in an appropriate direcion. Four types of OR gate are available, each producing output in a different direction. This allows OR gates to be used to turn a signal stream through an angle (signals do *not* run along wires). By placing two different types of OR gate at the same location a SIGNAL may be split to produce two

```
NEIGHBORHOOD:vn;

TYPES:signal,eastor,northor,southor,westor,eastnor;

RULE:
    {
    iam(signal):
          {
           CANSEE(eastor...eastnor): DIE;
           !                       : FORWARD;
           }

    iam(eastor):
        {
        CANSEE(signal):
            {
            BIRTH(eastor);
            BECOME(signal);
            EAST;
            }
        true:CENTER;
        }

    iam(northor): {...}
    iam(southor): {...}
    iam(westor) : {...}

    iam(eastnor):
        {
        CANSEE(signal)=0:
            {
            BIRTH(eastnor);
            BECOME(signal);
            EAST;
            }
        true:CENTER;
        }
    }
```

Figure 4.2: A Digital Logic Simulation

streams of SIGNALS moving in different directions.

The technique of having a creature which waits for an event, which upon being triggered creates a new creature to wait for the next event while itself becoming a messenger is frequently used in Creatures simulations. The waiting creature has little state, and therefore is simple to replace. Upon being triggered the creature acquires information it wishes to transmit — this may be difficult to pass on to an offspring. In addition were the waiting creature to produce an offspring it would need to distinguish it from any incoming signal (failure to do so is a simple method of producing a memory).

OR gates alone are insufficient to produce a logically complete system. Some form of inverter is also required, and in this simulation inversion is provided by a NOR gate. Infact the NOR gate alone is sufficient to implement any logic function, but the simpler OR gates are useful for "steering" information around the system. In a practical system further components (either logic functions or higher level structures such as counters and memories) may be provided in order to make implementation of a required circuit simpler. The system here is theoretically adequate to build circuits of arbitrary complexity. In practice propagation delays may lead to difficulties in building large circuits, though this is no more a problem here than it is in systems such as reconfigurable logic or microwave systems, where techniques have been developed to deal with significant propagation delays.

4.1.3 The French Flag Problem

The french flag problem[70] was first put forward in the mid 60's as a simplification of the mechanisms required in the construction of multi-celled organisms. The requirement is for a worm to divide itself into three equally sized portions, a `HEAD`, `BODY` and `TAIL` (or alternatively red, white and blue as in the french flag), using only local information.

The solution shown in figure 4.3 uses two chemicals, emitted by each cell. Cells are arranged in a line from East to West — it is irrelevant what type of cells are used for initialisation as they will change type immediately to a more appropriate state. One chemical migrates East, while the other moves West. By considering the density of these two chemicals a cell decides which type of cell it should be.

If at any time a cell is removed from the center of the worm, the chemicals will rebalance to produce two smaller worms with correctly proportioned heads and bodies. Information is distributed among all the elements of the system which collectively are able to behave in the desired fashion, though no single node has knowledge about the whole system. The

```
NEIGHBORHOOD: vn;

TYPES: eparticle, wparticle, head, body, tail;

RULE: {
      iam(head)| iam(body)|iam(tail)
          {
          true  :  {
                   birth(eparticle);
                   birth(wparticle);
                   become(body);
                   }
          cansee(eparticle) > 2*cansee(wparticle) : become(tail)
          cansee(wparticle) > 2*cansee(eparticle) : become(head)
          true                                     : CENTER
          }
      iam(eparticle):
          {
          cansee(head)|cansee(body)|cansee(tail)   : EAST
          !                                        : DIE
          }
      iam(wparticle):
          {
          cansee(head)|cansee(body)|cansee(tail)   : WEST
          !                                        : DIE
          }
      }
```

Figure 4.3: The French Flag Problem

failure of a cell does not unbalance the structure of the system as a whole, which is able to recover. Collective behaviour giving rise to structure and stability is an important property of many Creatures simulations, as will be seen in later more complex examples.

4.1.4 The Firing Squad Problem

The firing squad problem is another classic CA problem: it is this time required to synchronise a number of cells spread over an area of space, such that they all perform some action at the same time. This problem has parallels in biological systems, but is commonly expressed as a line of soldiers who must all fire upon receiving an appropriate command.

The Creatures solution (figure 4.4) 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 example illustrates how a creatures solution to a task can be arrived at by taking a physical approach to the problem. Though the exact details of implementation may be complex (as in the "sargent" creature), the roles of creatures represent real components in a system, and hence their basic function may be grasped by even the most naive user. Deriving the basic structure of a simulation is often the most difficult step — it is essential that the programmer has a good feel for how each component should behave. As a result of this Creatures can provide a more accessible environment than traditional programming systems.

4.1.5 The Game of Life

Perhaps the most famous of all CA problems, Conways Game of Life[22] has a very strong spatial basis, and it would at first appear that there is little to be gained by modeling it within Creatures. This implementation demonstrates how, by considering the game in a non-spatial fashion a Creatures simulation may be far more efficient than a CA simulation by directing computational effort only where it is required.

Rather than considering the space, consider that each live cell produces spores. These spread to the neighboring locations, where a cell master is elected, which counts the number of spores and if appropriate starts the next generation. The code for this is given in figure 4.5.

```
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 4.4: The Firing Squad Problem

Because a creature can only give birth at its current location the simulation necessarily has two phases — the counting where the spores decide if a cell is alive, and create new spores if appropriate, and the spreading phase where this information is passed to the adjacent locations.

The creatures implementation only examines those grid points which might be alive in the next generation — ie those that are adjacent to a live cell. This allows the system to operate on a near infinite grid, as typically only a finite subset of the cells are active (in the worst case all cells may be in this state and hence all must be considered, but this is unlikely). In the case of a CA implementation all cells must perform the same calculation regardless of the state of the system. By directing effort towards cells which might become active the Creatures system will be more efficient.

4.1.6 Langton's Ant

The mathematical curiosity known as Langtons Ant[60] demonstrates simply how ordered behaviour can emerge from apparent disorder. From a Creatures perspective it also demonstrates how a behaviour can easily be coded to meet a specification, and how complex results can emerge from trivial simple rules.

Langton's ant is a creature which turns left when it sees a black square and turns right when it sees a white square. In doing so it always reverses the colour of the square. Starting from an all white infinite grid the ant runs around apparently aimlessly for over ten thousand moves before creating a stable "highway", as shown in figure 4.6. The ant will always build a highway regardless of the starting state of the grid, and highways will be constructed even if several ants are present on the grid.

Although the behaviour of the ant is fully specified, in a very simple fashion it is almost impossible to predict the behaviour of the complete system. It is unlikely that highway building could be anticipated without running the code, though is can be proved that the ant's trajectory is unbounded. The self-similarity, yet unpredictability of the system has certain parallels with fractal systems, though here the system is discrete.

A Creatures implementation is shown in figure 4.7. The only complexity within the creatures code for Langton's ant is that the problem is based in terms of left and right while Creatures generally deals with absolute directions. Even with this complication the code remains trivially simple. Black squares are represented by the presence of a creature while white squares remain vacant. Upon seeing an ant a black square dies to become white, while upon

```
NEIGHBORHOOD: moore;

TYPES: cp, ep, nep, np, nwp, wp, swp, sp, sep;

VARS: int age;

INIT: age=0;

RULE: {
    age==0:
        {
        true      : age=1;
        iam(cp)   : CENTER;
        iam(ep)   : EAST;
        iam(nep)  : NORTHEAST;
        iam(np)   : NORTH;
        iam(nwp)  : NORTHWEST;
        iam(wp)   : WEST;
        iam(swp)  : SOUTHWEST;
        iam(sp)   : SOUTH;
        iam(sep)  : SOUTHEAST;
        }
    age==1:
        {
        iam(ep)  & cansee(cp)       : DIE;
        iam(nep) & cansee(cp...ep)  : DIE;
        iam(np)  & cansee(cp...nep) : DIE;
        iam(nwp) & cansee(cp...np)  : DIE;
        iam(wp)  & cansee(cp...nwp) : DIE;
        iam(swp) & cansee(cp...wp)  : DIE;
        iam(sp)  & cansee(cp...swp) : DIE;
        iam(sep) & cansee(cp...sp)  : DIE;

        (cansee(ep...sep)==2 & iam(cp))
        | cansee(ep...sep)==3       : birth(cp...sep);

        true: DIE;
        }
    }
```

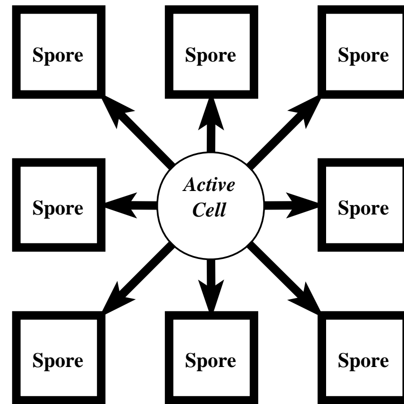


Figure 4.5: The Game of Life

Figure 4.6: Langtons Ant

```
NEIGHBORHOOD:vn;

TYPES:black,ant;

VARS:int dir;

INIT:dir=0;

RULE:  {
        iam(black) :
                {
                cansee(ant) : DIE;
                !           : CENTER;
                }

        iam(ant):
            {
            cansee(black): dir=dir+1;
            !            : {
                           dir=dir-1;
                           birth(black);
                           }
            dir<0: dir=dir+4;
            dir>3: dir=dir-4;

            dir==0 : EAST;
            dir==1 : NORTH;
            dir==2 : WEST;
            dir==3 : SOUTH;
            }
        }
```

Figure 4.7: Langton's Ant Code

finding a white square the ant turns it black. The *dir* variable is used by the ant to indicate its previous orientation, and incremented or decremented to turn left or right.

As with the life example this problem requires an infinite grid (it has been shown that the ant will *always* exceed any finite area[60]) which would not be possible with CA techniques. The simplicity of this implementation demonstrates the applicability of Creatures in the field of Alife and anti-chaos.

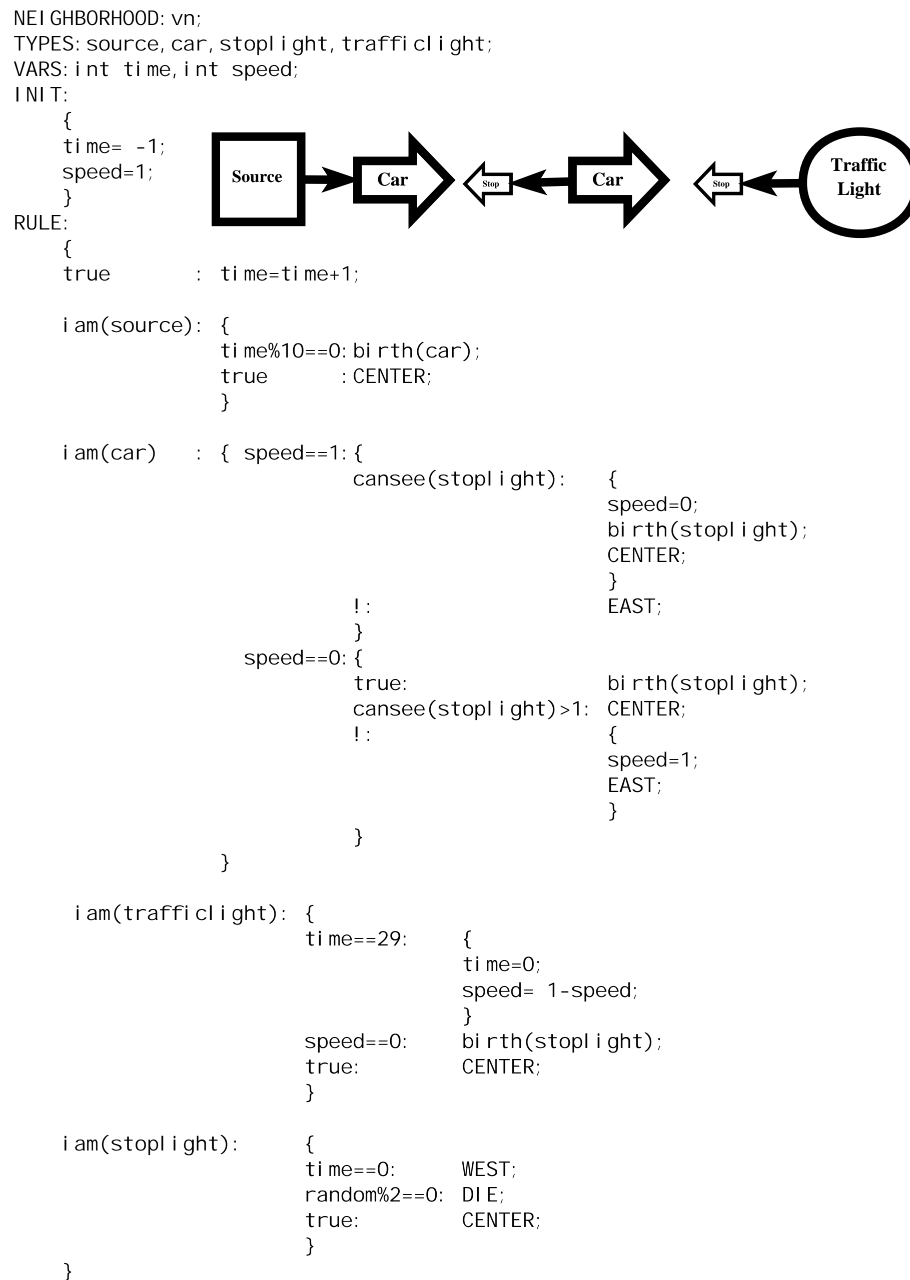
4.2 More Complex Models

4.2.1 Simulating Road Traffic Flow

Road traffic flow is traditionally modeled using statistical methods and techniques similar to those found in fluid dynamics[57] [4]. This abstract view is difficult to relate to the intuitive behaviour of real cars, and hence may be unreliable in its predictive behaviour. A Creatures simulation of traffic flow may directly represent cars as active elements within the model, scaling the well understood small scale behaviour into the complex large scale effects found in real road systems. Figure 4.8 shows a very simple model of road traffic queuing at a junction. Even this trivially simple model of cars' behaviour produces interesting results when several cars interact.

A source creature creates cars that move in an easterly direction. In this simple demonstration, their progress may be blocked by one or more traffic lights. Traffic lights may be in one of two states. When in the go state they do nothing but wait for a preset time. While in the stop state they produce "stoplight" creatures. These move west one square, and then wait for a random time interval before dieing. When a car encounters a stop light it stops moving, and starts producing its own stop lights. These in turn stop the car behind, and an orderly queue is formed.

When the traffic light stops producing "stoplight" creatures, the car at the front of the queue will start moving again, and hence the whole queue will eventually restart. However the random delay before the death of the "stoplight" creatures causes a delay in each car restarting. This leads to bunching and increased tailbacks, as found in many real situations. Frequently a block of cars will be stationary, though there is no immediate impediment to their progress — the bunch has simply grown back from a genuine obstacle until it is an obstacle in its own right.




```
NEIGHBORHOOD:vn;
TYPES:source,car,stoplight,trafficlight;
VARS:int time,int speed;
INIT:
    {
    time= -1;
    speed=1;
    }
RULE:
    {
    true        :  time=time+1;

    iam(source):  {
                   time%10==0:birth(car);
                   true      :CENTER;
                   }

    iam(car)    :  { speed==1:{
                              cansee(stoplight):    {
                                                    speed=0;
                                                    birth(stoplight);
                                                    CENTER;
                                                    }
                              !:                    EAST;
                              }
                     speed==0:{
                              true:                 birth(stoplight);
                              cansee(stoplight)>1:  CENTER;
                              !:                    {
                                                    speed=1;
                                                    EAST;
                                                    }
                              }
                   }

     iam(trafficlight):  {
                         time==29:     {
                                       time=0;
                                       speed= 1-speed;
                                       }
                         speed==0:     birth(stoplight);
                         true:         CENTER;
                         }

    iam(stoplight):      {
                         time==0:      WEST;
                         random%2==0:  DIE;
                         true:         CENTER;
                         }
    }
```

Figure 4.8: Road Traffic Flow

This model demonstrates how a simulation may be constructed successfully by describing the intuitive behaviour of small elements in the system, and then observing the results of their interactions. The bunching of cars produced by the simulation is an effect easily observable in a real system though it is not explicitly referenced in the simulation description — it could be considered as emergent behaviour.

4.2.2 Water Current Analysis

The behaviour of water current in oceans and rivers is measured by two classes of techniques[48]:

- Lagrangian: The location of particles is traced over a period of time.
- Eulerian: The strength and direction of the current is measured at many fixed points.

These two types of data are generally used together to produce a complete picture of the behaviour of currents in an area. This simulation considers the production of Lagrangian data from collected Eulerian data.

The currents at regular points in the mouth of the river Oler (in France) have been measured, both the direction and intensity of movement being recorded (Eulerian data has been collected). Storing this information in a CA would be simple, due to its static, spatial nature. However the elements of real interest may be the buoys which must move though the system (Lagrangian information is required). Such movement of buoys between locations would be somewhat difficult to implement in CA, as the concept of moving elements is not directly supported.

The currents in the bay of the river Oler were modeled by placing a current creature at each discrete location within the area of simulation. This creature holds a velocity, and direction for the water at that location. Buoys observe the current at their location, and based on that, decide to move to an adjacent location. Land creatures mark the edge of the bay. The running simulation is shown in figure 4.9. It should be noted that the simulation does not occupy a simply bounded area, as only sufficient land is required to bound the water. Were it necessary (as in CA) to fill the entire space with processes almost twice as many creatures would be required.

The number of Creature types, and the discrete space/time model limits how accurately the collected data may be represented when observed by the buoy. A current may have an

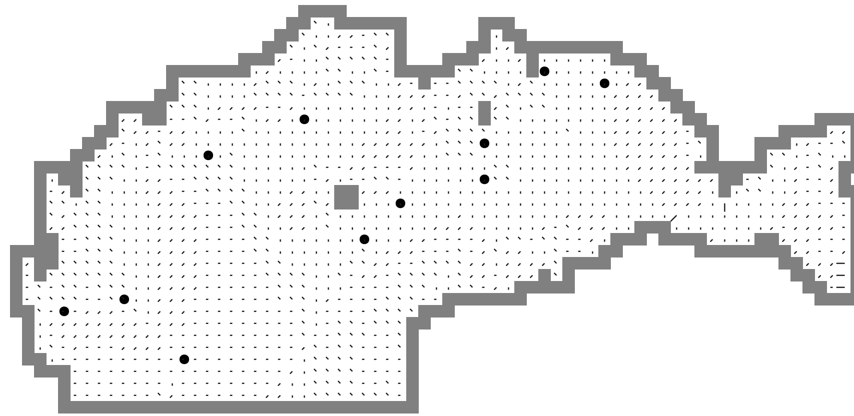


Figure 4.9: Currents on the river Oler

accurate velocity held internally, but at any one time a buoy must either move, or not move to an adjacent location. However a simulation may be built which behaves in a fashion consistent with the accurate data by the use of dithering. At any one time, the current will either be active or not, but it will hold within its private state an accurate value of its true velocity. A strong current is more likely to be active, hence the correct velocity is achieved when averaged over a number of time steps.

A similar action may be used to control direction of the current. Noise is added to the strength of the current before it is observed by the buoy. This helps to produce a more statistically likely result, as chaotic effects are accounted for. A simplified version of the code is shown in figure 4.10.

In addition to this JAM code, the full simulation used the implementation specific file “baydefs” to redefine the appearance of the simulation, and to load the velocity of the currents from the position file. Though such modifications are system specific they allow the model to produce output which displays the state of the system more accurately than the default methods. The modification to the format of the position file allowed details information about a specific system to be included. These changes were available to the end user without the need to modify the simulator.

The dithering[43] technique used in this example is frequently of use when behaviour is required which does not sit easily with the discrete nature of the system. It has been used in a number of simulations allowing creatures to move with variable speed, and in variable directions beyond the limited neighborhood and step times available in Creature (and CA)

```
NEIGHBORHOOD:moore;

TYPES:   mc,mn,me,ms,mw,buoy,land;

VARS:   int temp, int strength,int direction;

RULE:{
    true          : temp=random%10;
    iam(buoy):{
        cansee( mc )  : CENTER;
        temp==9       : CENTER;
        temp<2 :{
                cansee( mn  )      :  NORTHEAST;
                cansee( me  )      :  SOUTHEAST;
                cansee( ms  )      :  SOUTHWEST;
                cansee( mw  )      :  NORTHWEST;
                }
        temp>6 :{
                cansee( mn  )      :  NORTHWEST;
                cansee( me  )      :  NORTHEAST;
                cansee( ms  )      :  SOUTHEAST;
                cansee( mw  )      :  SOUTHWEST;
                }
        true :  {
                cansee( mn  )      :  NORTH;
                cansee( me  )      :  EAST;
                cansee( ms  )      :  SOUTH;
                cansee( mw  )      :  WEST;
                }
        }

    iam(land): CENTER;

    temp < strength:{
        direction==1 : become(mn);
        direction==2 : become(me);
        direction==3 : become(ms);
        direction==4 : become(mw);
        }
    !  :    become(mc);
    true:  CENTER;
}
```

Figure 4.10: Simulation Water Currents

systems.

4.2.3 A Model for the Spread of Sexually Transmitted Disease

The model of sexual behaviour shown in figure 4.11 is one of the most complex simulations developed using Creatures, and demonstrates some of the power (and a potential weaknesses) of the creatures technique. It models the spread of a sexually transmitted disease through a population[2].

Eight creature types are used to represent all combinations of Male/Female, Infected/Clear and Nonactive/Active members of the population, and define their interactions. Each creature also has a number of internal attributes which it uses to control its own personal behaviour. These are initialised to default values.

At each step a small fraction of the population will die from "natural" causes. An additional fraction is killed as a result of being infected by disease (represented by the letter "i" as the second character in its creature type name). Following this the time-since variable is increased for each creature, this represents the time since its last successful interaction. The confidence variable is decreased to represent of a general fall in the creatures perception of its own activity level.

The second section of code marked in figure 4.11 represents the behaviour of creatures that are not currently active. Should a creature find itself alone with a member of the opposite sex, it may, based upon the time since its last interaction choose to attempt an interaction, by becoming "active". Becoming "active" indicates to other creatures that the creature is willing to interact at the next timestep.

If a creature is marked as active, and is alone with a member of the opposite sex which is similarly signalling, an interaction is deemed to have taken place. This will be in the timestep following the decision to become active and is described by the third section of marked code. If either party was infected then the disease is passed with a predefined probability. The time-since variable is reset for both creatures. Active creatures are marked as no longer being active regardless of whether an interaction took place.

All creatures then move to a random adjacent location, using the "WANDER" operation. This is defined outside of the creatures model for convenience, though the exact nature of movement could have been coded in JAM. Although the movement is currently random, a more complex model could base movement upon the previous behaviour, and actions of the

```
NEIGHBORHOOD:vn;
TYPES:fcn,fca,fin,fia,mcn,mca,min,mia;
VARS: int timescince;
INIT: timescince=0;

RULE: {
      random%LIFEEXPECTANCYN==0: DIE;
      (iam(fin)|iam(fia)|iam(min)|iam(mia))
                      & random%LIFEEXPECTANCYI==0 : DIE;
      true:  timescince=timescince+1;

      iam(fcn)|iam(fin)|iam(mcn)|iam(min):
          {
          cansee(fcn)+cansee(fin)==1 & cansee(mcn)+cansee(min)==1
                          & random%100<timescince:
              {
              iam(fcn) :become(fca);
              iam(fin) :become(fia);
              iam(mcn) :become(mca);
              iam(min) :become(mia);
              true     :CENTER;
              }
          }

      !: {
          cansee(fca)+cansee(fia)==1&cansee(mca)+cansee(mia)==1:
              {
              iam(mca)& cansee(fia) & random%100<CONTAGEOUSNESS:
                      become(mia);
              iam(fca)& cansee(mia)& random%100<CONTAGEOUSNESS:
                      become(fia);
              true:  timescince=0;
              }
          iam(mca):  become(mcn);
          iam(mia):  become(min);
          iam(fca):  become(fcn);
          iam(fia):  become(fin);
      }

      true:         WANDER;
      }
```

Figure 4.11: A Sexual behaviour Model

creature.

While this model is a gross simplification of the system being modeled, Creatures models of slightly increased complexity have shown some interesting results. The simulation has been extended to include births, and a "confidence" variable which affects probability of interaction of the creature.

The coding of this simulation is made more complex than it need be by the scalar nature of creature type. Much of the code is repetitive, as it attempts to deal with sets of creature types in a consistent way. For example: all infected creatures have a higher probability of dying. The current system must test for each possible type of infected creature, as there is no way of structuring the creature types. It would be desirable to extend the creatures' external state to allow more complex interactions to be described easily. However this is a non-trivial problem which is considered further in section 5.2.2.

4.2.4 Taxis as a Goal Orientated Navigation Strategy

Background

Female crickets[39] locate potential mates over large distances (in the order of 10's of meters) by the chirping noise males produce during the mating season. The female apparently has the ability to distinguish the chirp from other sounds, and identify a mate of the appropriate species by properties of the song. A single male may be selected from a number of sound sources, and the female will follow this sound reliably.

Such abilities would appear to indicate some high level cognitive function, or at least some kind of decision making mechanism. However simulations using small robots[67] have demonstrated that the physical construction of the cricket's ears provide sufficient mechanism to eliminate sound sources of the incorrect type, and select a single source based solely upon its volume. Using such a simple technique of moving towards the loudest signal(phonotaxis[1]) produces what appears to be intelligent behaviour.

In response to this work it was hypothesised that such a simple strategy could also provide a mechanism for obstacle avoidance. An obstacle placed between the sound source and the observer would appear (as a result of diffraction and Huygens'[2] construction[50]) as two

[1] taxis: reflex translational or orientational movement by a freely motile and usually simple organism in relation to a source of stimulation (as a light or a temperature or chemical gradient)

[2] Christian Huygens, Dutch mathematician, physicist, and astronomer,1629-1695

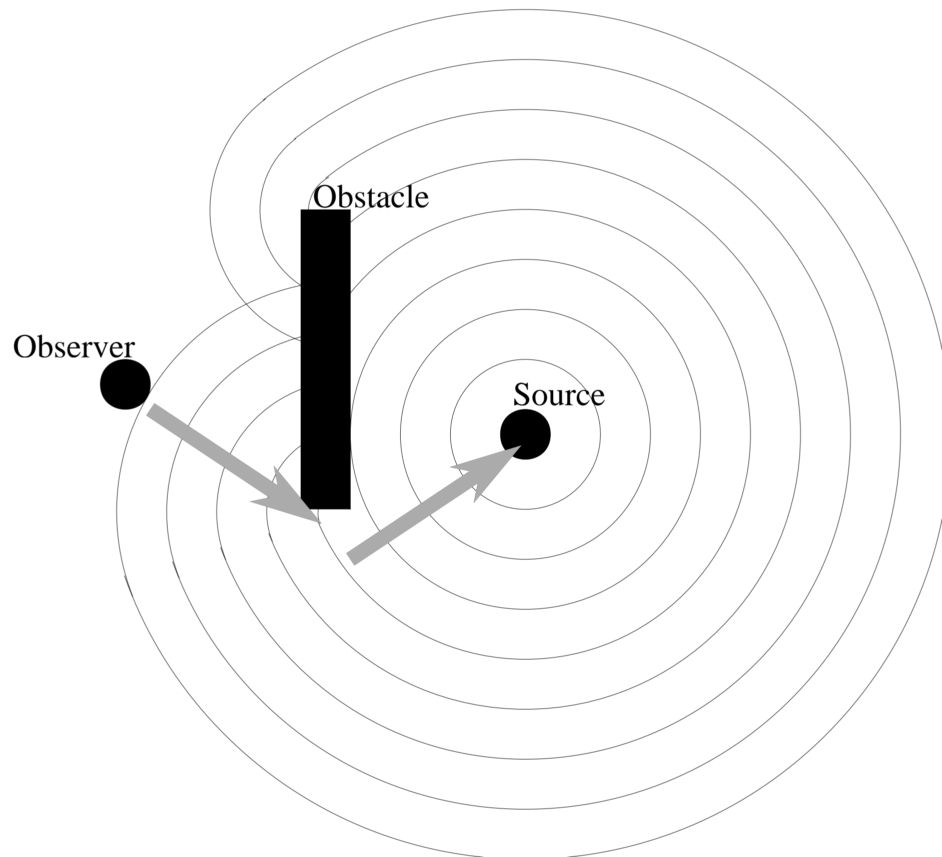


Figure 4.12: Diffraction Round an Obstacle

separate sound sources, one on each side of the obstacle (as shown in figure 4.12).

The observer would pick one of the sound sources based on its strength, and hence move towards the closest edge of the obstacle. Upon arriving at the edge of the obstacle the original sound source would become visible, and hence the observer would move towards the source by a very efficient route without either prior knowledge, training or complex calculation. Without knowledge of the creatures internal structure a third party could easily describe the behaviour as intelligent, setting sub-goals in order to reach a more difficult objective.

```
random%1000==0: DIE;
cansee(wall): {
                random%20==0: DIE;
                iam(np):      SOUTH;
                iam(ep):      WEST;
                iam(sp):      NORTH;
                iam(wp):      EAST;
                }
true: {
                true:  i=random%4;
                i==0:  {become(np);NORTH;}
                i==1:  {become(ep);EAST; }
                i==2:  {become(sp);SOUTH;}
                i==3:  {become(wp);WEST; }
                }
```

Figure 4.13: The Movement of Scent particles

Implementation

It appeared that such a system could be investigated using a Creatures model. A simulation of *Phono*-taxis would require a very large number of agents behaving in a strongly coherent fashion to simulate the movement of a wave front. While such behaviours have been implemented it was decided that *Olifi*-taxis (the following of smell) would be more appropriate to a Creatures based model. Odor gradients change more slowly over time and depend on statistical diffusion rather than the strict geometric behaviour of sound waves. The simulation was therefore built in terms of a rat attempting to find cheese in a maze.

The first task was to implement the behaviour of odor particles. In order that an observer may know which direction a particle is moving in four types of scent particle were used: np, ep, sp and wp each indicating a different direction. This however is a short term phenomena as it only indicates the behaviour on the previous move. Statistical properties are relied on to ensure that the density of particles is greatest nearest the source, and that more particles will move away from the source than towards it. Code is included to ensure that particles do not pass through walls. A scent particle's behaviour is otherwise random, as shown in figure 4.13. Particles decay with a small probability, and are removed with a much higher

```
random%1000==0: DIE;
cansee(wall): {
iam(cheese): {
             cansee(rat): DIE;
             !: {
                birth(np);
                birth(sp);
                birth(ep);
                birth(wp);
                CENTER;
                }
             }
```

Figure 4.14: The Cheese Creature

probability when they encounter a wall. This is to reduce reflections which could confuse the creature attempting to locate the target. However a small level of reflection is useful as it aids the smell in propagating round the maze.

A "cheese" creature was implemented as shown in figure 4.14. A cheese creature creates a number of odor creatures at each time step, and stays in its current location unless it sees a rat creature. Should the cheese observe a rat it dies and is considered to have been eaten by the rat.

Four different strategies were developed for the movement of the rat:

- rat: moves towards the strongest smell. If there is no strongest smell it remains stationary.
- frat(frantic rat): moves towards the strongest smell. If there is no strongest smell it moves randomly.
- prat(persistent rat): moves towards the strongest smell. If there is no strongest smell it repeats its previous move.
- rrat(random rat): moves randomly. rrats purpose is to provide a control.

The code for each of these strategies is shown in figures 4.15 and 4.16. Frat and Prat are complicated slightly by their need to avoid walking through walls! The standard rat will

```
iam(rat):    {
             true:i=4;
             (cansee(np)>cansee(sp)):i=2;
             (cansee(np)<cansee(sp)):i=0;
             (cansee(ep)>cansee(wp)):i=3;
             (cansee(ep)<cansee(wp)):i=1;
             random%2==0:
                 {
                 (cansee(np)>cansee(sp)):i=2;
                 (cansee(np)<cansee(sp)):i=0;
                 }
             }
iam(rrat):   {
             cansee(wall): i=(i+2)%4;
             !:{random%4==0:i=random%4;}
             }
iam(rat)|iam(rrat):
             {
             i==0:     NORTH;
             i==1:     EAST;
             i==2:     SOUTH;
             i==3:     WEST;
             i==4:     CENTER;
             }
```

Figure 4.15: Rat and Random Rat

```
iam(prat):  {
            cansee(wall): i=(i+2)%4;
            !:  {
                (cansee(np)>cansee(sp)):i=2;
                (cansee(np)<cansee(sp)):i=0;
                (cansee(ep)>cansee(wp)):i=3;
                (cansee(ep)<cansee(wp)):i=1;
                random%2==0: {
                    (cansee(np)>cansee(sp)):i=2;
                    (cansee(np)<cansee(sp)):i=0;
                    }
                }
            }
iam(frat):  {
            cansee(wall): i=(i+2)%4;
            !:  {
                random%4==0:i=random%4;
                (cansee(np)>cansee(sp)):i=2;
                (cansee(np)<cansee(sp)):i=0;
                (cansee(ep)>cansee(wp)):i=3;
                (cansee(ep)<cansee(wp)):i=1;
                random%2==0: {
                    (cansee(np)>cansee(sp)):i=2;
                    (cansee(np)<cansee(sp)):i=0;
                    }
                }
            }
iam(prat)|iam(frat):{
            i==0:    NORTH;
            i==1:    EAST;
            i==2:    SOUTH;
            i==3:    WEST;
            }
```

Figure 4.16: Persistent Rat and Frantic Rat

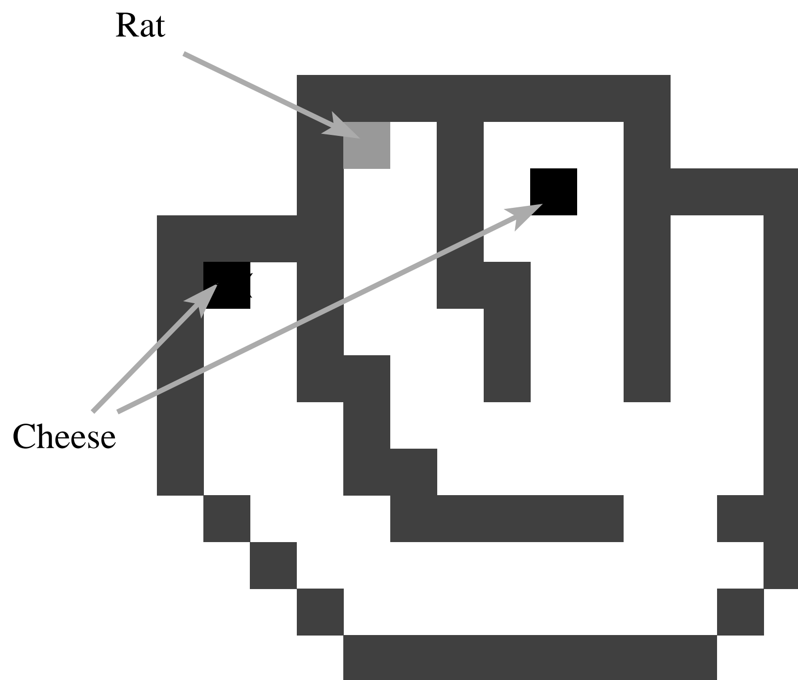


Figure 4.17: The Maze

never move onto a wall, as odor will never be greatest from that direction (due to a subtlety in the scent code). Frat and Prat may move onto a wall during one of the uncertain moves, and therefore require a small amount of code to extradite themselves from this predicament.

Results

In an initial test where a single wall was placed between a rat and the cheese, performance was generally poor — the rat found the cheese but typically took many hundreds of timesteps to complete a task which could be accomplished in approximately twenty. Examination of the distribution of scent particles indicated that although a large number of particles were being created their density was very low at even short distances. Such a density failed to provide a sufficient statistical basis from which the rat could find the cheese. A maze was therefore developed as in figure 4.17. This provided sufficient containment of the scent particles that a reasonable density could be achieved — most locations contained scent most of the time.

Following initial trials that demonstrated that the rat was capable of finding the cheese

Standard Rat	Random Rat	Frantic Rat	Persistent Rat
165, 129, 180, 140, 140, 147, 44, 36, 69, 202, 74, 88, 153, 55, 107, 144, 155, 124, 159, 63, 371, 225, 63, 131, 83, 124, 88, 82, 48, 177, 274, 54, 102, 180, 223, 63, 145, 150, 142, 334, 155, 138, 268, 56, 135, 141, 177, 86, 169, 203	249, 800, 532, 635, 540, 62, 602, 42, 225, 907, 335, 56, 490, 280, 133, 519, 94, 366, 802, 858, 979, 458, 632, 592, 592, 283, 743, 318, 282, 668, 138, 424, 80, 692, 124, 840, 476, 2019, 280, 328, 458, 632, 784, 592, 293, 342, 255, 920, 374, 86	29, 96, 99, 164, 174, 66, 96, 72, 58, 164, 92, 106, 64, 194, 70, 56, 136, 40, 136, 134, 54, 52, 257, 46, 118, 128, 164, 130, 156, 92, 142, 122, 72, 88, 76, 88, 98, 160, 226, 50, 72, 88, 76, 110, 98, 160, 226, 50, 50,124	62, 30, 86, 42, 50, 110, 121, 86, 38, 128, 134, 74, 62, 152, 132, 164, 82, 94, 50, 44, 102, 152, 138, 136, 112, 50, 128, 106, 76, 88, 100, 240, 42, 18, 48, 62, 98, 200, 170, 48, 40, 122, 106, 72, 42, 154, 96, 120, 140, 159

Table 4.1: Rat Performance: Timesteps taken to reach the cheese

in an acceptable timescale (the same order of magnitude as an optimal solution) the rat was removed from the maze, and the system was run for several thousand time steps. This ensured that the scent distribution was in a (dynamically) stable state prior to the commencement of any experiments. This position was saved, and used as the basis for all further experiments.

A rat to be tested was placed at the starting location and allowed to run the maze until it found one of the pieces of cheese. The number of time steps taken was recorded. This was repeated fifty times for each kind of rat. The results are shown in table 4.1, and graphically in figure 4.18. The mean times for Rat, RRat, FRat and Prat respectively are: 139, 484, 109 and 98 steps (the complete statistical results are shown in table 4.2). The standard deviations for the respective distributions are: 72, 341, 52 and 48. Despite the large overlaps in the distributions there is sufficient evidence to statistically differentiate these behaviours.

If a rat is more effective at solving the maze than the others it should be statistically possible to show that its results are a sample from a different distribution. This may be tested by calculating the standard error of the difference (the square root of the variance of the

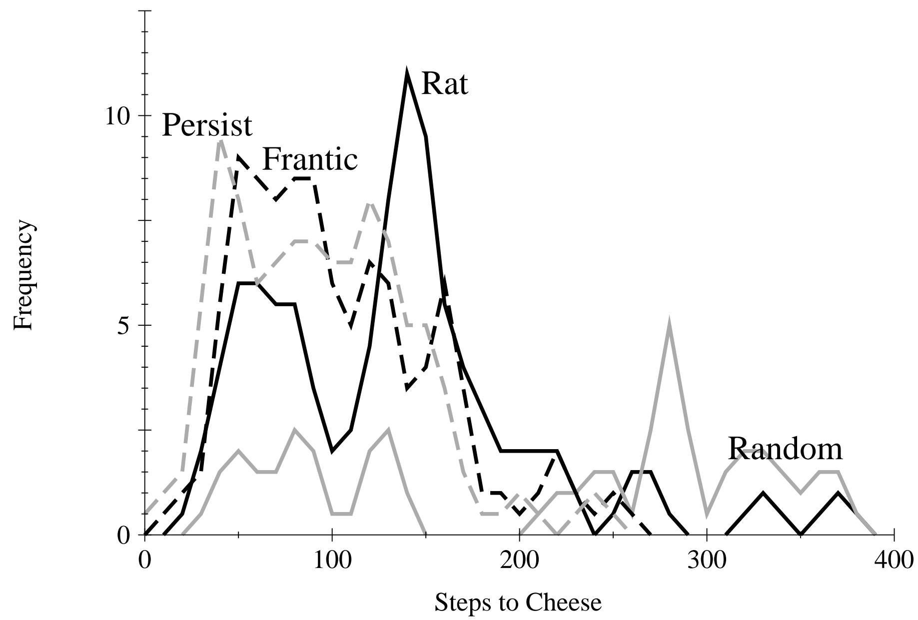


Figure 4.18: Rat Performance

	Rat	RRat	FRat	Prat
Mean	139.2	484.0	108.8	98.1
Std Dev	71.6	341.1	52.1	47.9
SEoD	Rat	RRat	FRat	Prat
Prat	12.2	48.7	10.0	
FRat	12.5	48.8		
RRat	49.2			
Deviation	Rat	RRat	FRat	Prat
Prat	3.4	7.9	1.1	
FRat	2.4	7.7	>2 is signifigant	
RRat	7.0			

Table 4.2: Statistical Results

difference) between pairs of distributions[47]. This is defined as:

$$\text{Std Error of Diff} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} \tag{4.1}$$

where σ is the standard devitation, and n the size of each population. If the difference between the means of two populations is greater than twice the standard error of the difference, then the result is statistically "significant" — ie. they are probably different. Two standard deviations represents approximately 95% certainty that the populations are different — a greater difference would indicate greater certainty. However no amount of data would ever be totally conclusive.

By inspection Rat, FRat and PRat perform better than RRat (as would be expected). When the data is statistically analysed it is found that each set produces highly significant results. There is virtually no doubt that these strategies provide some advantage over random foraging. Rat produces an SEoD of 49.2 compared to RRat and hence a difference 7.0 standard deviations between the two means. FRat produces an SEoD of 48.8 with RRat and hence 7.7 standard deviations difference. Prat gives and SEoD of 48.7 when compared to RRat — 7.8 Standard deviations.

The comparison of Rat with FRat is more interesting than the comparisons to RRat. Though FRat seems to perform better there is a very large overlap between the two distributions — it is not clear by inspection that FRat is genuinely better. The Standard Error of Difference between the two distributions is 12.5. The difference between the means is in fact 2.43 times this value, and hence the result is significant. FRat probably (with approximately 99% certainty) does perform better than Rat. More samples could be used re-inforce this result.

Prat performs 3.4 standard deviations better than Rat. However after fifty trials the comparison of Prat and Frat gives a SEoD of 10.0, and hence a difference of only 1.1 standard deviations. While Prat appears to perform better (and in fact probably does perform better than Frat), the evidence is statistically inconclusive. 1.1 standard deviations indicates approximately 75% certainty in the result — this is generally not considered sufficient evidence to establish a result.

Although Frat and Prat will find the cheese faster than Rat they will generally make more movements to get there (FRat and PRat will never be stationary while Rat may be stationary if there is nothing to do). Whether this is a useful strategy in a real situation would depend on the relative costs of movement against standing still, and whether other creatures are competing for the goal.

Prat's potential improvement over Frat is due to it trusting the previous sample, when the current sample is inconclusive. Unfortunately if the current sample is inconclusive then the previous sample may have been unreliable, so the improvement is not as great as one might hope (though in the absence of good information it may be the best thing that can be done). An improved strategy would take into account an estimate of the reliability of the previous result, and follow its directions for a limited time based upon its strength. At the simplest level the previous result could be used for a single time step, then random activity adopted. This would prevent the rat traveling a large distance as a result of a single bad sample.

Conclusions

When simulating taxis using high level programming techniques it is often difficult to make a system which *does not* work. In a perfect environment a single sample (theoretically) tells the observer both the distance and bearing of the target. Noise must then be added by the programmer to make the problem harder! The creatures simulation prevents this by enforcing strict locality, and restricting the transfer of information. No individual scent particle "knows" the location of the target (and may very likely be providing wrong information), but collectively they indicate its location. This style of simulation produces results which are more reliable than a high level implementation with artificial noise added due to its basic structural validity.

Simple taxis based strategies *are* capable of producing behaviour which to an external observer would appear intelligent. The rats do appear to set sub-goals which are solved in turn to achieve a final complex objective. The "intelligence" displayed is equal to that shown by many far more complex maze solving algorithms[15][51].

The interesting result of this experiment is not that taxis works — as a simple hill-climbing algorithm it is too basic to fail to work, given a suitable problem. The significant result is that formulating a maze solving task in this fashion produces an environment which is perfectly suited to a taxis based approach. The diffusion of particles from the goal ensures that the problem is free from local minima, and the rat may simply "follow its nose" to find an optimum route through the space. This consideration of creature and environment as a single system is frequently advocated by those attempting to simulate living systems[58].

4.3 Conclusions on the Application of Creatures

The CA style examples show in a very abstracted fashion some of the techniques that may be used when programming Creatures. In each of the cases the Creatures implementation is simpler, more intuitive, or more efficient than the traditional CA solution. These code fragments show how problems may be tackled in a physical fashion which may be easier for a non-specialist than traditional SIMD programming styles.

The second set of examples build on the techniques developed in solving the simple problems of the CA examples. While the solutions presented here are not intended to be complete simulations, they do show how the problems could be tackled using the Creatures model, and relatively short pieces of JAM code. In each case the code was easily developed and is relatively readable compared to equivalent CA code.

Of particular value is the separation of each component of the problem. It is quite simple to change the behaviour of one part of the simulation while the rest of the code remains unchanged. This is particularly evident in the final example where the behaviour of the Rats was independent of the behaviour of the rest of the system, and hence could be changed to test different strategies. In addition several Rats could be placed in a maze containing many pieces of cheese, should such an experiment be considered useful. Such options may not be available with either conventional high level, or CA approaches.

5

Discussion

5.1 Review

The Creatures model of parallel processing addresses some of the problems inherent in conventional parallel architectures:

- Complexity
- Scaleability
- Performance
- Accessibility
- Applicability

The architecture draws on the existing SIMD paradigm of cellular automaton and other simulation techniques, adapting the models to produce a simulation environment suited to the modelling of dynamic systems.

The shift of emphasis from the spatial nature of CA to active elements improves accessibility — the system is much easier to use for inexperienced users, who would be unable to operate conventional SIMD/CA systems. For an experienced user, the system allows many structures to be easily expressed that would be difficult in a traditional paradigm — an increase in applicability.

By representing active agents the model allows strong structural isomorphisms to be developed between elements in a simulation, and elements in the real world. In addition to making programming easier, this allows greater confidence on a simulation. As it is not necessary to make global assumptions about behaviour, the model is more reliable when taken beyond the limits of initial, known test data.

Performance of the systems developed is too complex to be easily sumarised. Creatures simulations are less efficient than static systems when implemented upon current hardware. The fixed data paths of CA style programming are better suited to SIMD hardware. This is only to be expected, as such hardware is likely to have been designed from CA style concepts. However much of this performance deficit may be regained by well written simulators, and simulations which exploit the dynamic nature of creatures to evaluate sparse data. If we consider the "Game of Life" example, then Creatures will give better performance for large grids which have little activity on them. CA force work to be done for all space. It may also be necessary to increase the complexity of a CA simulation to incorporate concepts, for which Creatures are better suited. In such cases the performance of CA suffers, and hence Creatures may be more efficient.

CA scale well in many respects. Information is only used on a local basis, and hence a large system does not require information to be spread further than in a small system. Locality is also strong in Creatures. However the dynamic nature of a creatures neighborhood often makes this difficult to exploit without throwing away much that is useful — particularly in terms of load balancing. Bucketing and spiraling offer a compromise between the strict locality that CA require, and the dynamic localities found in Creatures. By careful use of these and other similar techniques, scalable Creatures implementations have been built.

Creatures provides a general purpose environment suited to the simulation of many systems. Though such systems could easily be programmed in other languages, by using Creatures, the initial programming effort may be avoided. More importantly, by working within the strict formality of Creatures, the programmer is prevented from implementing a number of "bad" simulation concepts (and may also benefit from a sound theoretical basis). Creatures forces the user to implement the movement of data in a very explicit fashion. Strict locality, and limited interactions prevent the user from building, for example "a food sensor[1]"[8]. Without such restrictions it is all to easy to build a system that appears to demonstrate emergent global behaviour from locally defined action, when in fact global data is frequently being broadcast to all agents.

This strength is also the major weakness of creatures. Building large models from such simple components may prove too laborious to be practical. It may be necessary to develop methods of expressing many basic creatures operations as a single statement. Such a meta-creatures model would allow systems to be built more quickly from larger building blocks, while retaining the underlying creatures structure and integrity.

[1] It is often built into simulations of animal behaviour, that a creature will *move towards the nearest food.* This may seem reasonable, but no consideration is given to how this should be performed

Creatures most important feature is its simplicity. By restricting the model to a very limited set of operations, Creatures may be implemented, reasoned about, and programmed both efficiently and effectively.

5.2 Further Work

5.2.1 Simulator Development

The current implementations of Creatures provide a solid platform on which rules may be developed (NeXT implementation), and a range of systems which demonstrate the implementation of Creatures on parallel hardware. The parallel systems developed to date have not produced reliable, useful platforms on which a user would choose to develop code. Though the CM and MP implementations did offer some performance improvement for certain types of population this speedup was neither great enough or frequent enough to justify the high cost of the machines. However the experience gained in developing these implementations led to the creation of a transputer based system which though limited to very small populations, performs excellently. Evidence suggests that with more memory, and perhaps more processors a practical, very high speed parallel implementation could be developed with transputer hardware.

Such a simulator would allow a massively parallel system to be built and run at very high speed. However further development would still be required to make such a system accessible to end users. It would be desirable to access a parallel back end through something resembling the existing NeXT front end. It is likely that such an implementation would be limited by the data bandwidth to the front end, as the entire system must send information about its state to the front end (and perhaps respond to additional interrogation). This could be improved through the use of a transputer with a greater number of links (such as the Texas C40[62]) to form a tree structure perpendicular to the main computing surface (figure 5.1). Such a topology would provide vastly improved performance for the loading and saving of data — a potential bottleneck in the current system where creatures must be threaded through the main grid to reach their target nodes. The virtual routing available on Inmos T9000 systems would also be of assistance in implementing such a system, though only with respect to software development — real time performance would still be limited.

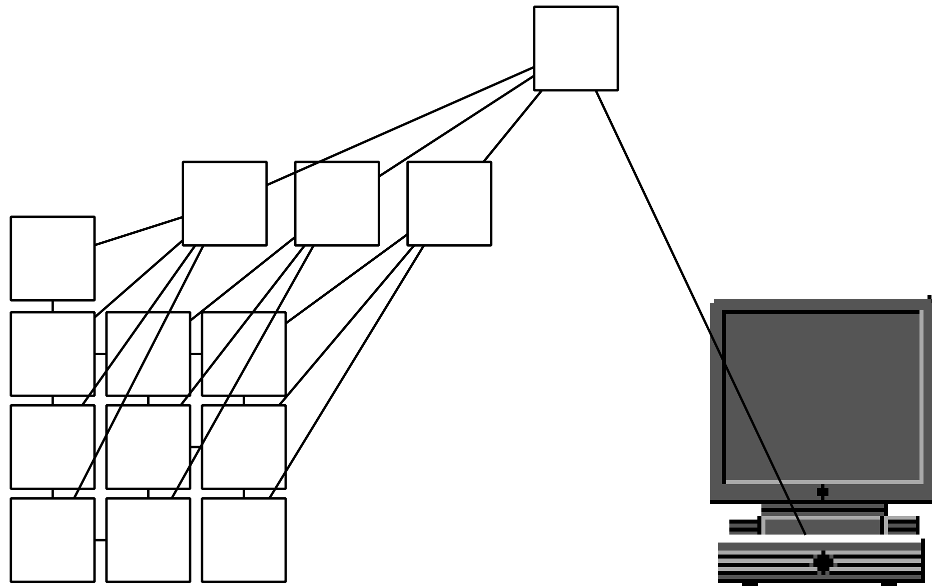


Figure 5.1: Improving I/O Performance

5.2.2 Extending the Model

The current creatures model has the important properties of being simple, yet complete. There are however a number of areas which could be investigated further.

Current implementations of the Creatures model, and the JAM language restrict the number of creature types to an enumerable set. While this is a very practical approach, it is not unreasonable to imagine systems which would require an infinite set of creature types, or at least a very large range of types (consider extensions to the sexually transmitted disease model to make other physical attributes visible). It may not always be reasonable to explicitly define behaviour for each type. Instead it should be possible to consider type as a vector quantity, allowing general boolean operations to be performed upon it. Observations in JAM are particularly weak in this area — the ranging operator being too limited, as the complexity of types increases.

It has often been considered that the *near* constraint be relaxed, and that continuous space introduced into the model. This is attractive in concept, but raises many new problems. Is the viewing radius fixed for all creatures? If not then is it a function of observer or observed (or both)? Data integrity becomes a serious problem, as the useful properties that $a \odot b \vee a \odot c \Rightarrow b \odot c$ and perhaps even that $a \odot b \Rightarrow b \odot a$ are lost. It is unlikely that a practical system will be developed that incorporates continuous space into the creatures model

without major changes to the underlying ideas. In addition to the conceptual problems, the implementation of such a system is unlikely to be practical.

A more practical proposition is the concept of continuous time. Just as DEVS brings continuous time to CA type models, so time could be incorporated into the Creatures model, bringing Creatures somewhat closer to the Mirror system. Self timed systems are more appropriate when modeling localised events, as the clock itself is a global mechanism capable of producing structured behaviour which has no inherent meaning within the simulation. By giving each Creature a wake-up time, and additionally waking it whenever its world view changes a modeling system closely related to Creatures but with discrete event style properties could be developed.

An alternative approach to time within the model would be to simply remove the global synchronisation, and allow each creature (or location) to update either at random times, or with a period slightly different to all other creatures (or locations). It has been shown[40] that certain properties of CA are dependant upon the synchronous nature of the model, and in fact that certain interesting behaviours may be masked by this. Similar results are likely to apply to Creatures. Though introducing continuous time into the Creatures model would radically change the techniques used to build simulations (and simulators), the concept is not totally alien.

5.2.3 Simulation Techniques

Much progress has already been made in learning to program the architecture. There is however much research still to be done, in investigating how structures may be built. In particular the problems of transferring information across space must be addressed. Techniques must be developed for building composite objects made up of many creatures, and sharing information between them over a range of space.

The beginnings of this may be seen in most of the simulations presented here. These ideas must be drawn together to produce an effective approach to model building. However these ideas will only surface as the result of building real simulations of real systems.

5.2.4 Applications

If the Creatures architecture is to develop it must find practical application beyond pure parallel processing research. In chapter 4 some simple systems were demonstrated, showing

how Creatures could be applied in a number of fields. Models such as these must be developed into complete simulations, and be shown to be valid when compared to the real world. This activity would provide valuable feedback into the development of simulators and the programming model.

6

Conclusions

6.1 General Aims

This research's aim was to develop an environment where simulations could be described in terms of agents which move through the system interacting on a local basis, and to demonstrate the viability of such an aproach. To achieve this the Creatures model was derived, and practical implementations of the model produced. By developing a specific model a lower bound has been placed on the abilities of agent based modeling — an "improved" agent based system would be able to do anything Creatures can, but may be able to do more. Creatures has demonstrated that a general model of computation may be produced which encompasses many of the hardcoded agent based simulations which have been produced. By exploring the specific Creatures model it has been shown that in general, agent based models exist which may be both practically implemented and usefully applied. It has been demonstrated that for certain classes of problem agent based systems are simpler to use than traditional SIMD architectures, yet retain the attractive features of CA. Clearly agent based systems have a role to play, alongside traditional data parallel programing techniques. In addition the specific model developed, and its simulators provide a platform on which practical work may be done.

6.2 Creatures

The "Creatures" model of SIMD parallel computation which has been proposed and developed in this thesis, retains the attractive features of traditional data parallel techniques while allowing dynamic systems to be described in a simple intuitive form. Systems are specified in terms of the active components of which they are made up. These small scale elements (or creatures) are then allowed to interact on a strictly local basis. Though the

small scale activity of single creatures may be simple, the large number of interactions will typically produce interesting results when the behaviour of the whole system is considered. By enforcing the strong structural parallels between the simulation code and the system being modeled, simulations may be built more easily and with greater predictive power.

The model has been implemented on a number of platforms including traditional serial machines, tightly coupled SIMD machines and loosely coupled MIMD hardware. It has been demonstrated that the model is scalable provided the implementation is constructed carefully. Use of a hashing function to partition a computation has been shown to be effective. In applying this hashing technique to the Creatures model a generalisation of Sequin's doubly twisted torus was developed which provides effective load balancing for a broad range of problems without a priori knowledge of the task. This has application beyond the Creatures system, as a simple yet powerful network topology.

The model forces the user to describe the parallelism of the problem rather than find parallelism in the solution. This not only makes the description of the problem simpler (as the real world system being described generally has inherent parallelism), but it allows the description to be translated into code for a range of parallel or serial machines in an efficient manner. The simulation is described in terms of an abstract Creatures machine rather than being tied to a specific style of hardware. As such models may be developed on a low cost, interactive platform, with only a few creatures active in the environment. The code may then be copied to a larger system and used to analyse a more realistically sized problem.

In order to test and develop the model a number of simulations were developed. These demonstrate the programming techniques which have proved effective when using the Creatures model. It has been shown that simulations developed in Creatures may be effective in understanding real systems in other fields. The simulations so developed typically required very small quantities of code to produce a meaningful simulation, and development time was typically in the order of hours for relatively complex simulations. The simulations could not have been developed with such clarity or speed in a traditional high level language. Further, by developing simulations from a known theoretical base the simulations are likely to be more reliable — the strict environment prevents the inadvertent sharing of data that could make a system appear to behave correctly when in fact the underlying mechanisms are incorrect.

The Creatures model, and its supporting software have been developed as far as is practically possible in a computing research environment. The Creatures model must now be tested against the problems of researchers in other fields who wish to build simulations to analyse

their systems, rather than to test the simulation techniques as has been done so far. Once this has been done the true strengths and weaknesses of the model will be revealed, rather than the state of the model as perceived by its developers.

References

[1] S. Abraham, 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.

[2] Anderson, May. (1992) *Understanding the Aids Pandemic.* Scientific American May 1992.

[3] Arvind, Smart, Peden. (1993) *Concurrent Discrete Event Simulation.* Edinburgh Parallel Computing Centre, Project Directory 1993.

[4] W.D.Ashton. (1966) *The Theory of Road Traffic Flow.* Methuen and Co Ltd.

[5] A. Assad. N. Packard. (1991) *Emergent Colonization in an Artificial Ecology.* Proceedings of the 1st European Conference on Artificial Life. MIT Press.

[6] J. Barnes, P.Hut. (1986) *A Hierarchical O(NlogN) Force-Calculation Algorithm.* Nature, 324, p446 (December).

[7] J. Bradley. (1981) *File & Data Base Techniques.* Holt, Rinehart and Winston.

[8] R. A. Brooks. (1991) *Artificial Life and Real Robots.* Proceedings of the 1st European Conference on Artificial Life. MIT Press.

[9] A. Burns, A Wellings. (1989) *Real-Time Systems and their Programming Languages.* Addison-Wesley.

[10] R.B.Cooper. (1981) *Introduction to Queueing Theory.* North Holland.

[11] B. J. Cox, A. J. Novobilski. (1986) *Object-Orientated Programming, An Evoloutionary Approach.* Addison-Wesley.

[12] L.Dagum. (1992) *Data Parallel Sorting for Particle Simulation* Concurrency: Practice and Experience, Vol. 4(3), 241-255 (May) John Wiley & Sons Ltd.

[13] J. L. Deneubourg, G. Theraluz, R. Beckers. (1991) *Swarm made Architectures.* Proceedings of the 1st European Conference on Artificial Life. MIT Press.

[14] A.K.Dewdney. (1990) *The Cellular Automata Programs that create wireworld, rugworld and other diversions.* Scientific American, January 1990.

[15] J.Y.Donnar, J.A.Meyer. (1994) *A Hierarchical Classifier System implementing a Motivationally Autonomous Animat.* From Animals to Animats, Simulation of Adaptive Behavior '94. MIT Press.

[16] M.Dorigo, V.Maniezzo, A.Colorni. (1994) *The Ant System: Optimization by a Colony of Cooperating Agents.* Politecnico di Milano. Submitted to IEEE Transactions on Systems, Man and Cybernetics.

[17] J.D.Eckart (1994) *A Cellular Automata Simulation System.* Computer Science Dept, Radford University.

[18] M.J.Flynn (1972) *Some Computer Organisations and their effectiveness.* IEEE Transactions on Computing, C-21 948–60.

[19] C.C.Foster, T.I.Berall. (1985) *Computer Architecture.* Van Nostrand Reinhold.

[20] Nigel R. Franks. (1989) *Army Ants: A Collective Intelligence.* In American Scientist, April 1989, pages 138-145.

[21] J.L. Frankel. (1991) *C* Language reference Manual.* Thinking Machines Corp.

[22] M Gardner. (1970) *Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life".* Scientific American, October 1970.

[23] M Gardner. (1970) *Mathematical Games: Cellular Automata, Self-Reproduction, The Garden of Eden and the Game of "Life".* Scientific American, February 1971.

[24] S.L.Garfinkel, M.K.Mahoney. (1993) *NeXTStep Programming.* TELOS.

[25] D.G.Green, T.J.Bossomaier. (1993) *Complex Systems: From Biology to Computation.* IOS Press, Amsterdam.

[26] A.Goldberg, D. Robson. (1983) *Smalltalk-80 The Language and its Implementation.* Addison-Wesley.

[27] D.Gross, C.M.Harris. (1974) *Fundamentals of Queueing Theory.* J.Wiley & Sons.

[28] B. Hasslacher, U. Frisch, Y. Pomeau. (1986) *Lattice gas automata for the Navier-Stokes equation.* Physics Review Letters, 56.

[29] W.D. Hillis. (1987) *The Connection Machine.* In Scientific American, June 1987 , pages 108-115.

[30] C. A. R. Hoare. (1985) *Communicating Sequential Processes.* Prentice Hall ISCS.

[31] R.W.Hockney, C.R.Jessope. (1981) *Parallel Computers: Architecture, Programming and Algorithms.* Addison-Wesley.

[32] P. Hogeweg. (1984) *Heterarchical Modeling.* Encyclopedia of Systems and Control. Oxford: Pergamon Press.

[33] P. Hogeweg, B.Hesper. (1984) *Socioinformatic Processes: MIRROR Modelling Methodology.* J. Theor. Biol 113. Accademic Press Inc.

[34] P. Hogeweg. (1988) *MIRROR beyond MIRROR, Puddles of LIFE.* Artificial Life, SFI Studies in the Sciences of Complexity. Addison-Wesley.

[35] P. Hogeweg. (1989) *Simplicity and Complexity in MIRROR universes.* BioSystems 23. Elsevier Scientific Publishers Ireland Ltd.

[36] P. Hogeweg. (1983) *The Ontogeny of the Interaction Structure in Bumble bee Colonies: A Mirror Model* In Behavioral Ecology and Sociobiology 12:271–283. Springer-Verlag.

[37] J.E.Hopcroft, J.D.Ullman. (1979) *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

[38] S.Horst. (1989) *Scientific applications of the connection machine.* Singapore. World Scientific.

[39] F.Huber, J.Thorson. (1985) *Cricket Auditory Communication.* Scientific American, 253:6:47–54.

[40] T.E.Ingerson, R.L.Buvel. (1984) *Structure in Asynchronous Cellular Automata.* Physica 10D, North Holland Physics Publishing.

[41] Inmos. (1989) *The Transputer Data Book.*

[42] Inmos. (1988) *Occam 2 reference Manual.* Prentice Hall ISCS.

[43] N.S.Jayant, P.Noll. (1984) *Digital Coding of Waveforms.* Prentise Hall.

[44] G.C.Lie, E.Clementi. (1986) *Molecular-Dynamics Simulation of Liquid Water with an ab initio Flexible Water-Water Interaction Potential.* Physical Review, A33, p2679, 1986.

[45] MasPar Computer Corporation. (1991) *Data Parallel Programming Guide.*

[46] R.Milner. (1989) *Communication and Concurrency.* Prentice Hall ISCS.

[47] M.J.Moroney. (1951) *Facts From Figures.* Penguin Books.

[48] G.Neumann. (1968) *Ocean Currents.* Elsevier Oceanography Series.

[49] NeXT Computer, Inc. (1991) *The NeXTStep Advantage.*

[50] H.C.Ohanioan. (1985) *Physics.* N.W.Norton and Co.

[51] A.G.Pipe, T.C.Fogarty, A.Winfield. (1994) *A hybrid Architecture for Learnign Continuous Environmental Models in Maze Problems.* From Animals to Animats, Simulation of Adaptive Behavior '94. MIT Press.

[52] A.D.Polimeni, H.J.Straigh. (1985) *Foundations of Discrete Mathematics.* Brooks/Cole Publishing Company.

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

[54] M.Resnick. (1991) **Logo Manual.* MIT Media Lab.

[55] R.Rucker. (1991) *vants* email rudy@autodesk.uucp.

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

[57] M.J.Smith. (1987) *Traffic Control and Traffic Assignment in a Signal-Controlled Network with Queueing.* 10th international Symposium on transportation and Traffic Theory.

[58] T.Smithers. (1994) *On Why Better Robots Make It Harder.* From Animals to Animats, Simulation of Adaptive Behavior '94. MIT Press.

[59] I.Stephenson, R.W.Taylor. (1994) *Creatures and Spirals: A data parallel object architecture* Proceedings of the Euromicro workshop on Parallel and Distributed Processing 1994, IEEE Press.

[60] I.Stewart. (1994) *Mathematical Recreations: The ultimate Anty-particle.* Scientific American July 1994.

[61] A.Stroud. (1969) *Mathematics for Engineers and Scientists.* Van Nostrand Reinghold.

[62] Texas Instruments. (1991) *Texas C40 Transputer DataSheet.*

[63] Toffoli, Margolus. (1986) *CAM a New Environment for Modeling.* MIT Press, Cambridge, Mass.

[64] A.M. Turing. (1937) *On Computable Numbers.* Proceedings of the London Mathematical Society,Series 2 issue 42 pp230–265.

[65] G. Y. Vichniac. *Simulating Physics with Cellular Automata.* (1984) Physica 10D, North Holland Physics Publishing.

[66] Von Neuman. *collected works Vol V:Design of Computers, Theory of Automata, and Numerical Analysis.*

[67] B. Web. (1994) *Robot Experiments in Cricket Phonotaxis.* From Animals to Animats, Simulation of Adaptive Behavior '94. MIT Press.

[68] K.Weihrauch. (1987) *Computability.* Springer-Verlag.

[69] S. Wolfram. (1986) *Theory and applications of cellular automata.* World Scientific.

[70] L. Wolpert. (1968) *The French Flag problem: A Contribution to the Discussion on Pattern Development and Regulation.* In Towards a Theoretical Biology. Edinburgh University Press.

[71] Yonezawa, Tokoro. (1987) *Object Oriented Concurrent Programming.* MIT Press.

[72] B.P.Zeigler. (1976) *Theory of Modeling and Simulation.* Wiley & Sons, New York.

[73] B.P.Zeigler. (1982) *Discrete event models for cell space simulation.* International Journal of Theoretical Physics.