Performance, scalability and Object-Orientation in Discrete Graph-based Simulation Models

K. A. Hawick and H. A. James

2005

Many interesting simulation problems in computational physics and engineering are posed on a regular data structure, such as a lattice, in two or three dimensions. There is increasing interest however in studying systems on less regular graphs that are embedded in Euclidean spaces of dimensions higher than three. We report on our experiences in attempting to formulate a highly general object-oriented framework in Java for simulating complex model systems on a generalised mesh in arbitrary dimensions and connectivity geometry. We discuss the performance and scalability issues that arose for managing large-scale complex model simulations and exploring their phase-spaces.

Keywords: Java-based simulation; scalability; OO-simulation; graph model; arbitrary geometry

BiBTeX reference:

Performance, Scalability and Object-Orientation in Discrete Graph-based Simulation Models

K.A. Hawick and H.A. James
Institute of Information and Mathematical Sciences, Massey University
Albany Campus, North Shore 102-904, Auckland, New Zealand
Email: {k.a.hawick,h.a.james}@massey.ac.nz

Abstract

Many interesting simulation problems in computational physics and engineering are posed on a regular data structure, such as a lattice, in two or three dimensions. There is increasing interest however in studying systems on less regular graphs that are embedded in Euclidean spaces of dimensions higher than three. We report on our experiences in attempting to formulate a highly general object-oriented framework in Java for simulating complex model systems on a generalised mesh in arbitrary dimensions and connectivity geometry. We discuss the performance and scalability issues that arose for managing large-scale complex model simulations and exploring their phase-spaces.

Keywords: Java-based simulation; scalability; OO-simulation; graph model; arbitrary geometry.

1 Introduction

A considerable body of work in computational physics involves posing numerical experiments to study the complex behaviour of model systems in different parameter regimes. The desired outcome is often to identify which universality class [4] the model belongs to or to study behaviours such as phase transitions or time dependent growth. These experiments generally involve exploring a model's parameter space by trying out different combinations of parameters and formulating hypotheses on what properties the model might have. Testing these hypotheses often requires computationally expensive statistical sampling of different trial configurations to capture the average or statistically significant behaviour at a particular set of parameters.

The two phases of this sort of work are therefore firstly to visualise the model and formulate some appropriate gross measurements of observable properties that some behavioural hypotheses can be formulated (or guessed). Secondly, statistically rigorous and computationally demanding numerical experiments must be run and the results carefully harvested to either show consistence with the hypotheses or to disprove them.

Much work has been done in the last 30-40 years since digital computers opened up the feasibility of this "computational science" [9]. Many of the simple systems such as the Ising, Potts, Heisenberg [2] and other models loved by computational physicists have been very thoroughly studied, mostly on regular lattices, using very large computer budgets [1]. In this paper we discuss some of the recent more complex models that are being used to explore, for example, the properties of artificial life systems [8]. Some models are being studied on irregular meshes and other more general geometries, that need to be formulated in terms of generalised graph or network models.

We describe general model properties in section 2. In section 3 we lay out our formulation ideas for simulation objects on relationship graphs and describe our object-oriented framework code in 4. We present some performance for the system in 5 and our analysis and conclusions in section 6.

2 General Model Properties

In general the models of interest to us in this paper, have some sort of state variable such as a magnetic spin or energy value or some other generalised "degree of freedom" that is associated with each node or mesh point. There are relationships between these variables, and we can think of the relationship objects as being associated with the arcs of the graph. In general, we have some update or evolution procedure that time steps or evolves the variables, based on their current values and usually some finite sub set of all the other variables - such as their nearest neighbouring values.

It is this localised concept of nearest neighbours that is opening up a wealth of new properties and behaviours to study, even on relatively well understood basic models. New geometries as might arise e from the study of quantum gravitational systems [10] or more complex semi-local relationships as might arise from the study of small world effects have some exciting implications.

It has been our experience that in formulating many physics and artificial life models, we have had to implement the same computational infrastructure again and again. This is inefficient for performance and prone to coding errors. We have therefore attempted to construct a generalised model framework based on objects and an elaborate computational geometry harness to allow a rapid investigation of many different systems with minimal recoding.
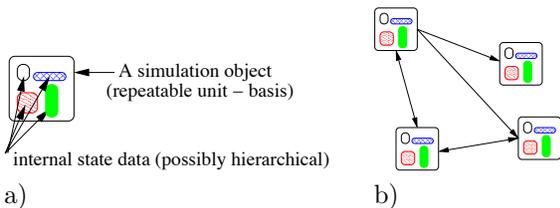
3 Simulation Object Graphs

a)

b)

Figure 1: A single simulation object, encapsulating the fundamental variables or degrees of freedom of the physical model.

The basic graph and object concepts we need to introduce are shown diagrammatically. Figure 1a) show the core concept of a model simulation object. This encapsulates the degree of freedom for a particular model. In the case of the Ising model it might be single bit, in a artificial life model it might require many fields to specify an agents energy, health, intentions and memory.
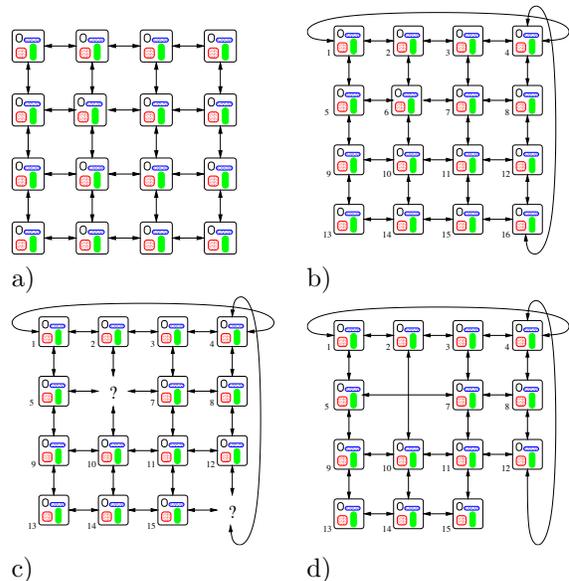

a)

b)

c)

d)

Figure 2: a) The relationship graph may be a simple regular mesh. b) The relationship graph can have short-cuts between particular objects as shown here extra links between objects 1 and 4, and between 4 and 16 have been made. These need not necessarily embed in simple Euclidean geometry. c) When object are created or destroyed we introduce some ambiguities in the meaning of the relationship graph. d) Healing the vacancies in the relationship graph can bypass or omit simulation objects.

Figure 1b) shows how simulation objects are linked together by a set of relationships. The relationship network forms a general graph of single or two way arcs. In the case of a traditional simple regular model in physics, the relationship graph might be a square mesh as shown in figure 2a). This idea obviously extends to 3-dimensions for a cubic mesh or the more realistic lattices such as face centred cubic or body centred cubic found in real matter such as crystals.

A topic of recent interest is that due to "small-world" effects or shortcuts applied to mesh models [6]. Figure 2b) shows a well-studied type of short cut that is often applied to such models. Cyclic boundary conditions can be applied to that "space

wraps around". This is often useful as then all the nodes see a similar local geometry even although the global properties of the geometry have been changed. This is often useful to avoid contamination of statistical simulation results by edge effects.

We have recently been investigating the effects of defects in systems as might occur when a simulation object (representing an atom in a crystal) is "missing". This concept is shown in figure 2c) where two nodes or vertices in the graph have been removed. This might be studied statically, or there might be a dynamic set of operations that leads to nodes disappearing and reappearing as part of the models evolutionary processes. Figure 2d) show some possible ways of resolving or "healing" the gaps in the relationship graph.

These key objects-on-a-graph ideas can be used to formulate a general simulation framework that is general enough to support many interesting models.
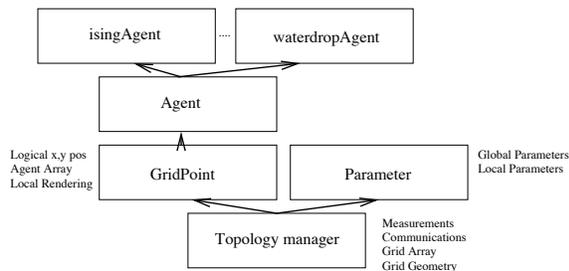
4 Generalised OO Simulation

Figure 3: Architecture diagram of our generalised mesh simulation framework. As can be seen, many different types of objects must be traversed in order for the framework to update each object even once; while providing almost complete generality, this is a major source of inefficiency when running long or complex simulations.

Our initial attempt at a mesh-based simulation framework turned out to be too general to achieve worthwhile compute performance. The architecture is shown in figure 3. At the very top of the architectural hierarchy there is a `Topology` object. This is used to create the size and *geometry* of the system to be studied. The current version of the simulation framework can site grid points using a square, triangular or hexagonal mesh. The Topology manager's job is not only to create the size and

geometry of the system, but also to calculate where each grid point is in relation to its neighbours. The Euclidean distance between grid points varies with different topologies.
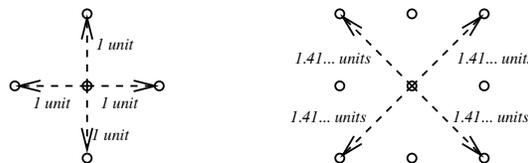

Figure 4: Our framework allows users to select the Euclidean distance within which points should be classified as neighbours. In this square mesh system, the distance between the central point and the north, south, east and west neighbours is "1" unit. The next nearest neighbours are $\sqrt{2}$ units away from the central point.

The user can instruct the Topology manager what distance should limit the consideration of neighbours. For example, in a square mesh the distance to grid points in the cardinal directions is, say, 1 unit, whereas the next nearest neighbour distance is $\sqrt{2}$. This is illustrated in figure 4. Our system also supports small-world networks [6].

The Topology manager maintains an array of global `Parameter` items that each `Agent` or `GridPoint` is able to consult whenever necessary during the simulation evolution. The Topology manager also manages communications mechanisms to allow Agents to exchange messages.

Each simulated grid point is represented by a `GridPoint` object. When the neighbours for each grid point are computed they are added into an array that is internal to the GridPoint object. This was done to relieve the Topology manager from the job of re-computing the neighbours for each point at every step. Obviously the neighbours would have to be re-computed if the framework were to support grid points that could move, or stretch elastically. `GridPoint` objects are initialised with the shape of the mesh they're inhabiting, their logical x and y position within the grid, and a list of agent types that will be located at each point. It is the responsibility for each grid point to calculate their position in real (display) space and to organise for the painting (and repainting) of their display. The Topology manager then calculates the centre of each grid point for the purposes of computing the neighbouring points. `GridPoint` objects main-

tain their own array of neighbouring grid points. This is especially useful when next nearest neighbours (distance of $\sqrt{2}$) are being used or the grid point is at the edge of a row or column in a simulation. `GridPoint`s also maintain an array of agents that inhabit the point. Our initial implementation used a Java `Vector` to store the agents but this was changed upon reflection of how often agents really moved between grid points.

Agents in our simulation framework are represented by an abstract `Agent` base class. Each agent maintains some internal information, such as its name, current state (and hence how it should render itself), the (simulation) time it arrived on a grid point, and whether the state needs to be changed in the next time-step due to external influences. Each agent in our system must extend the `Agent` base class by providing implementations of how it evolves, what type of data it produces as output and also *what* is produces as output. Output is used to make global measurements on a system. For example, in an Ising system one measurement may be the magnetisation of the system. In other models, for example artificial life systems, some agents (like passive "grass") may produce no output to be measured. Implementations of the `Agent` class can contain more than just the basic information in the superclass. They may also contain variables that can be changed so as to customise the behaviour of each agent on different grid points in the system, represented in a local `Parameter` object. This is useful, for example, when modelling fire-fronts, to change the relative probability of the front spreading to simulate different types of vegetation, etc. Java's introspection mechanism is used for this purpose.

When a `GridPoint` is instantiated with a list of agent types, it attempts to load an instance of each agent and site it within the grid point. As all agents that currently reside on a grid point are maintained within an array, it is necessary to nominate a type for the array. The array type is `Agent`, the superclass of each discrete agent definition. The main problem with this approach is that whenever one wishes to use one of the methods that is particular to the type of agent (as opposed to a method defined in the superclass), one must use the Java introspection mechanism to interrogate the object. This, unfortunately, is not a fast process, and the continual need for this introspection process considerably slows down the potential speed of the simu-

lation framework.

When the system is instructed to evolve (or move time one step onwards) by the user, this involves a message being sent from the Topology manager to **each** grid point to evolve. Evolution is completed in a two-phase process: evolution, where each grid point requests its agents prepare their next state from their current state and the local/global parameters in the system, and then synchronisation, in which the current state is replaced with the new state. The evolution of the system was implemented this way so as to avoid any sweeping effects [5] that may have occurred if agents were updated *in situ*. Each grid point cycles through its list of contained agents, requesting they update themselves.

In our generalised framework, agents are able to query their containing grid points' neighbours' agents to see whether the state of those agents will affect the evolution of this grid point. This is best illustrated by a model such as the well known Conway's Game of Life [7] where a cell (agent on a grid point) will spring to life or die if a certain other number of neighbouring cells are alive. This constant checking of grid point neighbours, and the agents contained within them, introduces considerable computational overhead.

In an attempt to model longer-range behaviour, we introduced the ability for agents to send messages between grid points. The communications mechanism was maintained by the Topology manager, which simply received a message on behalf of the destination grid point and delivered it to each of the agents in that grid point during the synchronization phase of the system evolution.

In some of our models, agents had the ability to move between grid points. This was achieved by an agent requesting to its containing `GridPoint` that it be moved to a destination grid point. The `GridPoint` would then de-register the agent from itself and register it with the new `GridPoint` object. One had to be very careful in this process that an agent was not accidentally allowed to evolve twice consecutively before being synchronised by the system.

As a result of introducing mobile agents, we observed that now not all grid points contained agents all the time. In an attempt to improve the efficiency of the system, we added a new array in the Topology manager so that grid points could register and de-register themselves from updates based on

whether they contained any agents. We observed this had a small effect in increasing the speed of the system.

The following pseudocode shows the event architecture of our framework. In order to set up the system the following operations take place:

```
1. Topology viewer instantiated with geometry shape notifier, size of
   lattice and which agents will be inhabiting simulation system
2. For each grid (lattice) point in size
   1. Create grid point
   2. Instantiate shape of grid point
   3. Instantiate agent(s) to inhabit grid point. For each agent:
      1. Instantiate internal data representation
      2. Inform agent where it is in lattice
      3. Set any global parameters in each agent
      4. agent sets its current (initial) state and any graphical
         representation of that state
   4. Grid point adds itself to the topology manager's list of those points
      containing agents
   5. Agent reports back to topology manager it's dimensions and position in
      an x-y space (used to realize neighbour relations)
3. Iterate through neighbours, setting neighbour relations
   1. Grid points keep internal information on neighbouring points
4. If any grid points do not require their agents (c.f. the game of life
   model, amongst others) then the agent is removed. the grid point removes
   itself.
```

The system is now able to undergo evolution, as illustrated below:

```
1. User requests system evolve.
2. Topology viewer creates copy list of grid points with agents
3. Topology viewer randomizes order of list (to avoid sweeping effects)
4. Topology viewer requests each grid point, in turn, evolve itself
   1. grid point consults list of agents on that site, requests each evolve
      itself by invoking per-agent evolve procedure (may use object introspection)
      1. agent evolves itself. may involve requesting from the encapsulating
         grid point the list of neighbours. may involve requesting, from each of the
         grid point's neighbours, the state of a similar object, used to determine
         the effect of evolution. may also involve the movement of an agent between
         grid points. agents create a 'new state' they will adopt upon
         synchronization.
5. once each grid point (and each agent has evolved), the system
   synchronizes the newly computed state of each agent. For each grid point
   1. grid point requests each agent synchronize its state
      1. each agent confirms its new state as its current state. agents may
         die permanently or may move between grid points at this step. A change in
         state may cause the agent's graphical representation to change
      2. agents may also allow some measurement to be computed from their
         state, in which case introspection is used to extract the data
6. any graphical display is updated to reflect the change in agents' states.
```

The object support mechanisms in Java made it very easy to implement and experiment with these ideas in our prototype framework.

5   Performance Data

Table 5 shows that the performance of our Java topology viewer and simulation engine is at least an order of magnitude slower to create a topology than the optimised C version of the Ising simulation code. Table 5 also shows that the Java version is at least two orders of magnitude slower than the optimised C code for evolving the system between states. However, the Java implementation provided us the ability to experiment graphically with the behaviour of our simulation system. It also provided us with the ability to easily create topologies consisting of meshes of different shapes (e.g. triangle and hexagonal) to experiment with different numbers of nearest neighbours, and even allows us

to site more than one type of agent at a single lattice point, all of which are not easily possible with the optimised C code. The final feature that we incorporated into the Java simulation framework was the ability for agents to communicate in a long distance fashion.

On the other hand, we have incorporated some quite advanced book-keeping code into the optimised C code and have switched off the majority of the checking one would normally do to ensure internal consistencies are maintained, because we completely understand the parameters and tolerances within which we are working. Hence, the code would require quite significant modification were we to incorporate a fraction of the features that were developed into the Java code.

Array size (square)	Java Top. viewer (w. graphics)	Java Top. viewer (no graphics)	Optimised "imperative" code
40^2	7.3	3.6	0.1
48^2	8.3	4.0	0.1
64^2	12.5	4.9	0.1

Table 1: Time (seconds) taken to load and create the nominated size of Ising cells using our Java-based Topology manager and our optimised C program. (Data accurate to 2 significant figures)

Array size (square)	Java Top. viewer	Optimised "imperative" code
40^2	102	0.5
48^2	103	0.6
64^2	105	1.1

Table 2: Time (seconds) taken to evolve the pre-loaded Ising configuration for 100 Monte-Carlo steps.

The data shows the very drastic (order-of-magnitude) differences in gross wall-clock time obtained for different codes tackling the models in different ways. All data is presented for the same hardware and operating system platforms. Key observations are that the introspective nature of the object manipulation is computationally slow. The use of graphical object manipulation is also dramatically

slow. We experimented with both imperative style code in Java as well as C. The C code is certainly faster, but that is not the dominant effect and is irrelevant to our main conclusions about the use of a generalised object framework.

6 Summary and Conclusions

Other simulation and modelling framework tools such as Matlab and Mathematica [12, 13] are also very useful in rapidly developing and visualising the sort of complex systems models we discuss here. These packages suffer from similar performance limitations due to their over generality. Simulation standards such as the DoD High Level Architecture [11] makes significant use of object ideas, and while these ideas are incredibly useful in rapid development of models, the sort of statistical analysis we are interested in requires some crafted optimised simulation codes.

We speculate that a model specification language for complex simulation models language would be a great asset, supported by general but perhaps slower tools for early model conceptualisation, but supportable by fast compiled codes too. It may be possible to develop some automated tools for compiling model specification languages into fast simulation codes using appropriate run time libraries.

We conclude that the key operations in establishing an agent framework as we describe involve iteration over the simulation objects in a manner that is appropriate to the generalised geometry and connectivity amongst agent object. Object technology is very useful up to a point, but we believe general tools (our own included) lack the computational speed to obtain useful statistical analyses on many of todays interesting simulation problems.

Acknowledgements

References

[1] Baillie, C.F., Gupta, R., Hawick, K.A., Pawley, G.S., "Monte Carlo Renormalisation Group Study of the 3d Ising Model," Physical Review B 45 (1992) 10438-10453.

[2] Kadanoff, L.P., "Statistical Physics Statics, Dynamics and Renormalization," World Scientific, 2000, ISBN: 981-02-3764-2.

[3] Gould, H. and Tobochnik, J., "An introduction to Computer Simulation Methods, Part 1 Applications to Physical Systems," Addison-Wesley, 1988, ISBN 1-201-16503-1.

[4] Jullien, R., Peliti, L., Rammal, R. and Boccara, N. (Eds.), "Universalities in Condensed Matter", Springer Proceedings in Physics 32, Springer-Verlag, 1988, ISBN 0-387-50445-1.

[5] James, H.A., Scogings, C.J., and Hawick, K.A., "Parallel Synchronisation issues in Simulating Artificial Life," in Proc IASTED Parallel and Distd Comp Sys, pp 815–820, Boston, MA, November 2004.

[6] Watts, D.J., and Strogatz, S.H., "Collective dynamics of 'small-world' networks," Nature, vol 393, pp 440–442, 1998.

[7] Conway, J.H., Game of Life in Gardner, M., "Mathematical Games," Scientific American, October 1970

[8] Levy, S. (1992). Artificial Life The Quest for a New Creation. Penguin. ISBN 0-14-023105-6.

[9] IEEE Computer Society and American Institute of Physics, "Computing in Science and Engineering" magazine, Available at http://www.computer.org/portal/site/cise/

[10] Baillie, C.F., Hawick, K.A. and Johnston, D.A., "Quenching 2d Quantum Gravity," Physics Letters B 328 (1994) 284-290.

[11] Dahmann, J.S., "High Level Architecture for Simulation," 1st Int Workshop on Dist Interactive Sim & Real-Time Apps (DIS-RT '97) January 09 - 10, 1997 Eilat, Israel, pp. 9.

[12] The Mathworks, Matlab, Available from http://www.mathworks.com

[13] Wolfram Research Inc., Mathematica, Available from http://www.mathematica.com