ENGINEERING SOFTWARE FOR D-DIMENSIONAL SIMULATIONS WITH HYPER-CUBIC NEIGHBOURS

Ken A. Hawick
Computer Science, University of Hull, Cottingham Road, Hull HU6 7RX, UK.
Email: k.a.hawick@hull.ac.uk
Tel: +44 01482 346311   Fax: +44 01482 466666

ABSTRACT

Many important simulation models can be formulated on a d-dimensional hyper-cubic lattice where the number of dimensions $d$ is itself a parameter of the model. This leads to considerable complications even for spatially localised models as the cellular neighbourhood is a function of dimension. Neighbourhood itself can also be a necessary parameter of a model where for example different interaction distances between model degrees of freedom variables critically affects the model’s emergent behaviour. We explore programming language mechanisms for coding d-dimensional data structures and simulation models and present a number of syntactic and framework approaches for engineering software with internal or external domain-specific language features.

KEY WORDS

Software engineering; hyper-dimensional systems; simulations; code generation; domain-specific language

1 Introduction

Simulation models in material science [26], physics, chemistry [14, 21], and even sociological systems [18] are often modelled on a lattice structure. A square lattice in two dimensions or a cubic lattice in three dimensions have different numbers and geometry of neighbourhoods, but in fact the neighbourhoods can vary further by having different shells of inclusion or connectivity. For example nearest, nest-nearest, and so forth. Often such models behave very differently on different dimensional spaces. A two dimensional flatland may behave dramatically differently from a three dimensional rectilinear system independent of the size of the connecting neighbourhood.

This is shown for a variation of the well studied Ising model [25] in Figure 1 which shows that a property measured varies considerably with the dimension and in fact Ising-like systems typically have a critical dimension of $d = 2$. The dimensionality itself is often found to have a particular critical value that can sometimes only be obtained experimentally rather than analytically [10], and it is therefore greatly useful to be able to formulate and encode simulation models where dimension $d$ is in fact a parameter of the model code. Although models that relate to a real world phenomena directly are typically a flat planar two dimension system or a rectilinear three dimensional system, it is often instructive to investigate the less obviously physical four dimensional system or even higher. Often by investigating behaviours in higher dimensions one can extrapolate back and gain insights on why particular phenomena occur or do not occur in two or three dimensions. Many systems in fact have a “critical dimension” above or below which certain behaviours change.

Encoding software to manage such multi dimensional simulation models can involve considerable tradeoffs in design and implementation. While integrated problem-solving simulation packages may well offer a quick and easy solution to a known model, often such packages do not scale well to handling very large system sizes or indeed to handling arbitrary dimensions. Engineering custom software for such models obviously involves a degree of experimentation and testing and the development and testing time are considerably prolonged if one has to develop and maintain separate codes or versions for different dimensional systems and combinations of neighbour connectivities.

This article presents some design approaches and an evaluation of them against the software engineering criteria of single software code, minimizing development time and maximising code reuse. We present a nomenclature for the multi dimensional lattice that underpins many such models and present software engineering approaches for libraries and the user model codes that can employ them. This
approach also lends itself well to using data-parallelism [4, 31, 34] and other tile-oriented [3, 28] parallel computing solutions [29, 32] such as special purpose processing accelerators [22] such as Graphical Processing Units (GPUs) [27] that have special purpose hardware for manipulating regular arrays of cells [1] that can be easily mapped to a multi dimensional simulation model [16].

Alternative lattice structures such as hexagonal, triangular or face-centred-cubic and so forth can also use these techniques [9], but we focus on simple square and hyper-cubic systems here where the only parameters are the dimension and the neighbourhood shell distance.

The approach we describe is also of use for managing any hyper-cubical data sets [6] that involve localised neighbour operations such as stencils [24], finite differential calculus [15, 17, 30] or regular pattern motif analyses. For the purposes of this present paper we focus solely on lattice simulation models [8] where each cell in the hyper-cubic system holds a simple data type however.

Our article is structured as follows. In Section 2 we summarise the nomenclature we use for the d-dimensional models and this leads onto a description of software library management issues in Section 3 and the variable neighbourhood management codes in Section 4. We give some selected results of software engineering examples in Section 5 and present a discussion of the overall issues in Section 6. We conclude with a summary and ideas for further work such as the use of domain-specific language programming for this sort of application in Section 7.

2 Model Nomenclature

It is useful to formulate a data structure nomenclature for hyper-cubical models independent of any particular simulation model. The software engineering aim is that the details of any particular model can be kept separate from the hyper-cubical dimension and neighbour management framework to maximise software code reuseability.

Figure 2 shows an example of a three dimensional Ising model [25] simulation with an octant of the brick cut away for rendering purposes [5]. This was simulated using the software framework described in this present article and using software than can manage an arbitrary number of dimensions and lengths in each dimension - subject of course to available computer memory [2]. Support for this sort of model is our objective.

We usually employ a lattice structure of the regular form \( c_i \in \mathcal{L} \) where the \( N \) model variables or cells are indexed by \( i = 0, 1, \ldots, N - 1 \) and may be scalars or sometimes vectors of more than one degree of freedom. Many of the models we focus on have a very simple integer variable that can take on some number \( Q \) different states. In the case of the Ising magnetism model \( Q = 2 \) and there are only two allowable states - the quantum spin values “up” and “down.” The Potts model [13] extends this to an arbitrary number \( Q \) of pseudo-spin variables. The Sznajd opinion model [33] can also use the same structure to represent \( Q \) allowable different opinions. The Kawasaki [23] and diffuse models can use \( Q \) different atomic species held in the same way. We can therefore write \( S = c_i \) to represent a particular state of the whole model, with a definite and specific value for each of the \( N \) site variables \( c_i \).

The lattice \( \mathcal{L} \) itself can in principle have any geometry we wish and typical examples are square or simple cubic (SC), but this can be extended to hyper-cubical lattices of the form \( N = L^d, d = 2, 3, 4, 5, \ldots \) with a length \( L \) and dimension \( d \). In practice, we can also implement other structures such as triangular and hexagonal lattices in two dimensions, or the common crystallographic structures found in real three dimensional substances such as face centred cubic (FCC), body-centred cubic (BCC) or hexagonal close-packed (HCP) lattices. We focus on the hyper-cubical examples in this present article however.

The notion of spatial locality is very important to the model family we discuss. In most cases the model time \( t \) evolution can be written in the form:

\[
c_{i,t} \rightarrow c_{i,t+1} = \mathcal{F}(c_{i,t}, c_{N(i),t})
\]

where we use \( N(i) \) to denote a localised neighbourhood of sites around site \( i \). This is an important restriction and the locality of interactions - namely that the new value of site \( i \) is obtained through a formula that depends only upon nearby site values - imposes a realistic spatial structure and causality time and length scales on the model. It also provides the basis for incorporating parallelism into the model computations as it allows a simple geometric decomposition and allows sites in the whole system to be allocated in some sort of spatial patching to different processors - real or virtual.

Some models are stochastic - that is to say that the formula for updating individual sites has a random or thermal term in it. Our framework is able to supply random fields in the form of pseudo-randomly generated variates from one of several different generator algorithms.

The framework must manage the spatial geometry mapping of \( i \) to space in the form of \( x \), or \( x, y, z \) or if us-
ing a higher dimension \( d > 3 \) then to some appropriate hyper-coordinates in the \( d \)-dimensional space. The locality and neighbourhood being used can also be varied. So for instance, it is often revealing to vary a model from using nearest neighbour to next nearest, or Moore neighbourhood or some other structure such as neighbourhoods defined by a radial distance of proximity.

3 \textbf{d-Dimensional Model Template}

A simulation is typically specified by its Lattice geometry \((\mathcal{L}, L, d)\) and Neighbourhood \((N)\). To implement the above ideas in reusable software we commonly need to implement loops of the form

\textbf{Algorithm 1} 3 Common Iterative Loops needed

\begin{verbatim}
for \( i \in [0,N] \)
for \( n \in \mathcal{N} \)
\( j \in [0,|\mathcal{N}|] \)
\end{verbatim}

where we index individual sites in th model by \( i \) and neighbour either by a general iterator \( n \) or as an explicit index \( j \) into a neighbour list. We follow the mathematical convention and use round parentheses to denote an exclusive range and square brackets for inclusion.

The layout in memory of recti-linear multi-dimensional data drastically affects the computational performance of some algorithms. Often different parts of a single application require different optimal layouts. Concise and efficient data layout transformations that are easy to use from a given programming system are therefore important and useful. Specifying the semantics of a data transformation that rearranges a data set within memory has proven difficult to standardize, with different programming languages and library systems implementing different and not wholly compatible semantics and interfaces.

Algorithm 2 shows the typical outline of a computational simulation experiment where the core simulation algorithm would typically be expressed in a form like that of algorithm 3.

In this present paper we focus on being able to change the dimension \( d \) for hyper-rectangular (not necessarily hyper-cubic) lattices. It is necessary to parameterise the geometry and the neighbourhoods for an appropriate storage scheme so that this architectural scheme will work in general. The relevant aspects of Algorithms 2 and 3 are bolded to highlight the code areas we focus on to enable software engineering of \( d \)-dimensional model codes.

The standard traditional programming approach for implementing \( d \)-dimensional applications is to rely upon the programming language’s own multidimensional array syntax. We might have appropriate nested loops, so that for example a 2- or 3-dimensional code might be encoded:

Figure 3 shows how the C preprocessor might be employed to embed different static source code fragments, for each separate dimension. This is error prone as they must be kept consistent and tested separately when a site-level model change is incorporated. Many other traditional high performance programming languages such as Fortran have similar semantics but different syntax for the same nested loop encoding to manipulate a multi dimensional array. Note that in the example shown we have ordered the loops from inner to outer to suit the way \( C \) arrays are laid out in memory, to maximise cache/page performance.

It is somewhat neater from a software engineering perspective to be able to use an iterator or source code construct that abstracts over the detailed storage scheme used - particularly when managing arbitrary number of neighbouring cells.
It can be generalised to higher dimensions but as Table 1 shows in Figure 4 for the 2 and 3 dimensional cases.

The nearest neighbours can be relatively trivially managed - there are exactly two per dimension, and a useful convention to adopt is to store or reference them in list order according to negative direction then positive direction as needed. This uses the processor and its registers without disrupting cache access patterns. At the time of writing memory accesses are likely to a greater speed hindrance than processing times for a few integer arithmetic operations needed to recompute neighbour indices pre-stored for each cell is potentially expensive memory consumption, although it has been our experience that the typical state-of-the-art in feasible model sizes - limited by model computational demands - is in fact quite feasible to store in this manner in the gigabytes of memory typically available on a modern processing device. A more difficult tradeoff comes from the cache/paging miss effects when model cells are traversed in order, but which access patterns are potentially disrupted by having to index using lists of neighbour indices as indirect addresses. This can considerably slow the access and for some models and platforms it is in fact faster to recompute the neighbours of a particular cell \( k \) as needed. This uses the processor and its registers without disrupting cache access patterns. At the time of writing memory accesses are likely to a greater speed hindrance than processing times for a few integer arithmetic operations needed to recompute neighbour indices.

In this paper we only make use of plain CPUs and main memory with the usual multiple levels of cache. If specialist devices such as GPU accelerators are involved these tradeoffs may be completely different. Consequently it is important to experiment with each particular platform used and we have developed the notation and apparatus to be able to adopt any of the approaches described below.

We generally find that to avoid having to explicitly

<table>
<thead>
<tr>
<th>d</th>
<th>NN1</th>
<th>NN2</th>
<th>NN3</th>
<th>NN4</th>
<th>NN5</th>
<th>NN6</th>
<th>Moore</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>24</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>24</td>
<td>32</td>
<td>24</td>
<td>48</td>
<td>96</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 1. Neighbours in shells for up to 4-dimensions.

Figure 4. Moore Neighbourhood in 2-dimensions (left) and 3-dimensions (right) subsuming the complete surrounding hyper-cubic shell around the \( k \)'th cell.

Figure 3. C source code nested loop approach to encoding fixed number of dimensions, known at compile time.

4 Neighbourhoods and Shells

Many lattice models can be usefully formulated solely in terms of the nearest neighbours. Cellular automata lattice models or simple calculus based finite differencing approximations often make use of this. However not only do some models explicitly require second and beyond-nearest neighbour, the neighbour shell inclusion distance can be regarded as a parameter of the model itself in some instances.

The nearest neighbours can be relatively trivially managed - there are exactly two per dimension, and a useful convention to adopt is to store or reference them in list order according to negative direction then positive direction in ascending dimension. This convention allows the two dimensional routine to reuse the one dimensional routine and the three to reuse the two and so forth.

However, implementing a generalised mechanism to gather the correct indices for arbitrary neighbours of a particular cell is non trivial in hyper dimensions. The size of the neighbour shells grows in a non-trivial manner, that is a property of some complex mathematical topological geometry properties of the lattice structure.

Table 1 shows the number of neighbouring sites at successive distance shells on a hyper-cubic lattice. We list nearest neighbours as NN1, next-nearest as NN2 and so forth. We also enumerate the Moore neighbourhood as it is often employed for some models. The Moore neighbourhood can be defined as a surrounding layer of neighbours as shown in Figure 4 for the 2 and 3 dimensional cases. It can be generalised to higher dimensions but as Table 1 shows the number of neighbours in the Moore shell grows rapidly with dimensions. It is also ambiguous to define it for other than a hyper-cubic lattice.

We can encapsulate as libraries in various forms and can either precompute their indices for every cell in our model or recompute them as needed. Interestingly, it is not completely trivial to determine which approach will be fastest. Precomputing indices so that there is a list of integer indices pre-stored for each cell is potentially expensive in memory consumption, although it has been our experience that the typical state-of-the-art in feasible model sizes - limited by model computational demands - is in fact quite feasible to store in this manner in the gigabytes of memory typically available on a modern processing device. A more difficult tradeoff comes from the cache/paging miss effects when model cells are traversed in order, but which access patterns are potentially disrupted by having to index using lists of neighbour indices as indirect addresses. This can considerably slow the access and for some models and platforms it is in fact faster to recompute the neighbours of a particular cell as needed. This uses the processor and its registers without disrupting cache access patterns. At the time of writing memory accesses are likely to a greater speed hindrance than processing times for a few integer arithmetic operations needed to recompute neighbour indices.

In this paper we only make use of plain CPUs and main memory with the usual multiple levels of cache. If specialist devices such as GPU accelerators are involved these tradeoffs may be completely different. Consequently it is important to experiment with each particular platform used and we have developed the notation and apparatus to be able to adopt any of the approaches described below.

We generally find that to avoid having to explicitly
code in the multi-dimensional syntax of a particular programming language it is better to adopt a convention known as k-indexing. So each of the N cells is referenced by a unique integer index \( k = 0, 1, \ldots, N - 1 \) that are effectively pointer addresses.

We can actually use pointer memory addresses in some language such as C or C++ but it is more convenient to logically map these notional indices to integers, which we refer to as k-indices. We can pack a d-dimensional vector of coordinates up as a k-index or unpack a k-index into its constituent d-dimensional vector using simple modulo and integer arithmetic expressions and assuming we know the lattice length \( L_d \) in each of the dimensions.

This flexible k-indexing allows us to avoid hard coding of the dimensions, but means we end up with some quite verbose and unwieldy libraries and more importantly, some verbose library calls for the user/programmer to remember. One of the consequent software engineering challenges is to implement these in a way that is memorable for the model implementer and is reusable.

We also want to separate out the geometry specification from the iterative loops in the source code - so there is only one source code change needed to switch between models. This issue is discussed in the section below.

5 Software Approaches

There are many implementation details we could describe that are necessary to implement an efficient yet useable software framework for d-dimensional simulations. We focus in this section on some considerations from implementing the requisite library routines for gathering the neighbour index information for a particular cell given the various possible dimensions and neighbour shell distances and arrangements.

We present software fragments that are in C/C++ but which are similar to those we have also implemented in Java. For the most part these ideas carry over to the syntax available in most modern imperative or object-oriented programming languages. Since we use one-dimensional arrays and the k-indexing strategy these fragments are not dependent upon knowledge of whether the programming language and its run time library use row major (C/C++/Java) or column major storage (eg Fortran) for memory layout. However this does affect the optimal order for loop nesting. We show row-major examples throughout.

Figure 5 shows how semi-automatically generated code for gathering the Moore neighbourhood shell can be formulated in a C-like language. We assume the existence of some library primitives to convert between a k-index packed index for a particular cell in the lattice and the unpacked integer vector of d coordinates. We then use nested loops operating over some working variables to build up the shell of Moore neighbour indices in a list when requested for a particular centre cell \( k \). This can be done in C/C++/Java/Groovy or any related programming language. Higher level languages like Groovy, Python, Ruby and so forth allow some extra layers of code using mechanisms such as closures [9] to make these operations more compact and elegant for the model user/programmer. However the key operations as encoded in Figure 4 must still be implemented in an appropriate imperative or object library.

Figure 6 shows four different approaches to packing the d-dimensional routines in such a library. The simplest version for the user/programmer to call is the all powerful routine with multiple arguments (Case A) which allows explicit passing through of constants or variables specifying the lattice boundary conditions and dimension and sizes and so forth. Appealing though this may be it does add a considerable performance overhead since all these parameters must be passed through and particularly when the routine is called an an inner loop that traverses all cells of the model. Even if just a single clock cycle were involved in each parameter pass then this can slow down the model traversal by an unacceptable amount.

The second case B uses a common programmer approach of implementing multiple routines in the support library with some of the choices encoded in the routine name. This makes the library “messier” and harder to
Figure 6. Four approaches to packaging the d-dimensional routines - (A) passing arguments and having a single flexible and powerful library routine, or (C) some stateful setup initialiser and a simpler routine to return a list of the neighbour indices, which can be implemented using Object-oriented apparatus (D) or case (B) which is an intermediate approach with some of the options explicitly and statically encoded into the called imperative library routine.

We have described hyper-cubic lattice and various common but useful neighbourhoods such as the Margolus neighbourhood. There is also scope to implement less neighbouring shells including the commonly used Moore neighbourhood. There is also scope to implement less parallel libraries.

In effect this is the latter approach that when taken to extremes is the one use by problem solving packages - all the apparatus is hidden away from the user/programmer. Often good for rapid prototyping but often has insufficient apparatus exposed to allow speed and memory optimisations to be made - and which are still necessary for implementing large multi scale simulation models.

6 Discussion

We have seen that there are a number of tradeoff issues concerning code elegance, reuse, ease for the developer, simplicity of incremental new model deployment for the user and of course computational performance. While one expects modern optimizing compilers to continue to minimise the performance impact of having to pass parameters, at the time of writing this still matters - particularly for the inner loop of a multi scale simulation that may have \( N \approx 10^9 \) cells or more. These large numbers multiplied by present gigahertz clock frequencies mean that every wasted processor instruction transferring a parameter or invoking a conditional statement in an inner loop over all cells can translate into around a second. When this is multiplied by the many runs, many steps, and many parameter combinations that must be trialled this all leads to unacceptable performance penalties.

Our observations are summarised in Table 2. This shows the tradeoff space in terms of low, medium or high software development effort on the part of the library implementer and the user/model implementer as well as the application model complexity but also the computational performance.

It has been our experience best approach to model development - over the course of several years where one incrementally adds newly conceived models to a growing collection - is that of developing an ongoing library of framework that is accessed using appropriate high level language techniques. The fundamental hard work however is still in the underpinning library development in which a lot of care and attention and test cases are needed to ensure correct addressing and scalability of the d-dimensional neighbourhood management system. Over time this effort pays off considerably however and the incremental cost of adding a new model diminishes relative to the overall cost of the project. This is a good compromise that has paid over considerably over using separate codes for each model and especially over the use of packaged problem solving software which is still generally unable to scale up to large scale multi dimensional multi scale models.

We are only just beginning to make use of domain-specific programming language techniques layered on top of our library [30]. Very modern languages with new features such as closures [11] and other boiler-plate code generation techniques [19, 20] have great promise in making the underpinning libraries even more useable. There is scope for using apparatus such as operator overloading to simplify the specification and external domain-specific language (DSL) [7] techniques to generate whole new simulation oriented languages that still attain efficiency by using the hand optimised libraries.

We are also just beginning to incorporate parallel and accelerator optimised versions of the libraries too. An early observation is that object oriented apparatus that groups together state information that can be made thread-safe is likely to be the best approach to making parallel DSLs that use parallel libraries.

We have described hyper-cubic lattice and various neighbouring shells including the commonly used Moore neighbourhood. There is also scope to implement less common but useful neighbourhoods such as the Margolus neighbourhood. Generally any neighbourhood such notion that has a well agreed upon definition in the model simulation community can be readily added to any library and potentially to a DSL that builds upon it.
### Table 2. Tradeoff analysis of different simulation software architectural coding approaches.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Solving Environment (PSE) Package</td>
<td>None</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Custom Code</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Imperative Library (IL)</td>
<td>High</td>
<td>Med</td>
<td>Med</td>
<td>Med</td>
</tr>
<tr>
<td>Object-Oriented Library (OOL)</td>
<td>High</td>
<td>Med</td>
<td>Med</td>
<td>Med</td>
</tr>
<tr>
<td>Domain Specific Library over OOL or IL</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

### 7 Conclusion

We have described the problem of implementing arbitrary d-dimensional lattice simulations with dimension and neighbourhood as parameters of the model. We presented various software engineering issues concerned with implementing these and described our prototype libraries using various code fragments. We discussed the tradeoff space covering both software development and reusability for the library developer as well as the model user and how these tradeoff against the resulting computational performance.

In conclusion we believe the approach of developing a high performance - albeit sophisticated - underpinning library for d-dimensional and arbitrary neighbour gather-scatter operations has proved very worthwhile. This approach lowers the incremental cost of developing and experimenting with a newly conceived model and separates out the scalability and performance issues considerably from issues of model conception and associated applications science. It also allows exploration of very much larger model sizes and time scales which are beyond the reach of commonly available problem solving software packages.

In summary the generalised gather-scatter problem is a challenging one and there is still scope for extending the ideas presented here for other lattice and structures such as face centre or body centred cubic or even some of the more exotic materials structures found in graphenes and other topological materials. There is also considerable scope for further optimisations to the approaches discussed here for use on platforms with unusual or distributed memory systems such as processor accelerators.

Finally, we hope that modern programming languages and tools for domain-specific programming languages will be able to build further upon these ideas to combine high simulation performance with maximal code reuse and minimal necessary re-testing.

### References


(MSI 2011), pp. 371–378. IASTED, Pittsburgh, USA (7-9 November 2011)


