

Computational Data Structures for High-Performance Complex Network-Based Small-World Simulations

H. A. JAMES
and
K. A. HAWICK
Massey University

The Watts and Strogatz model of small-world network systems has been successfully applied to explain features in several data sets. We have carried out a set of extensive simulation studies of stochastically generated small-world systems. We report on the data structures, memory and run-time complexity of our simulation codes and discuss how these ideas may be applied to similar simulations studies. Our packed k -index mechanism encodes physical spatial information and supports interpolation between pure list and multi-dimensional array models. The k -index mechanism allows most procedures in simulations of this ilk to be written in a dimensionally independent fashion.

Categories and Subject Descriptors: I.6.3 [**Simulation and Modeling**]: Applications; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms: Algorithms, Performance

Additional Key Words and Phrases: small-world model; simulation; graph model; complex network

1. INTRODUCTION

Custom-written computer simulation programs provide an important means of tackling very compute intensive application problems. Many problems that can be formulated as numerical experiments display slow convergence and can only be addressed by extensive stochastic sampling over multi-scale parameter search spaces. We have recently carried out a study of a physical model simulation on a complex network [Newman et al. 2006] to study the effect of the small-world network parameter on the location and properties of a phase transition [Hawick and James 2006]. That work has taken over two years and made use of several computer clusters. It required a highly optimized custom simulation code, that was flexible enough to explore different dimensionalities and parameter regimes to which we had to adapt as the study progressed. In this present paper we describe the data structures and performance optimization issues we encountered in this and related codes. We present some methods, performance results and related conclusions that may be of use to other numerical experimentalists developing and running similar large scale simulation codes. We have reported on the statistical mechanical results of this work elsewhere [Hawick and James 2006].

Our particular application involves a Monte Carlo simulation on a complex network of interacting nodes. The model is based on the Ising model of a simple magnet [Onsager 1944], which has a specific Hamiltonian energy functional de-

scribing the interactions between “neighbouring” nodes. This model has already been extensively studied on regular lattice networks and the location in temperature of the critical phase transition and its scaling exponents are well known in different hypercubic lattices of 2-, 3- and 4-dimensions [Binney et al. 1992].

Of interest to us was to study the model on a highly complex irregular lattice, constructed using a small-world set of shortcuts or re-wires applied to the initial lattice.

The procedure required was:

- (1) Generate a small-world complex network for a particular parameter value p based on a d -dimensional lattice
- (2) Initialize a hot (random) starting configuration of model “spins” on the (static) network structure
- (3) Equilibrate the spin system so it is representative of a finite (quench) temperature T
- (4) Apply stochastic dynamics to the configuration to thoroughly sample the characteristic properties of the system (such as its bulk magnetization and heat capacity) at equilibration temperature T

By applying the above procedure many times with different p , d and T we build up average properties for the location of the critical temperature T_c and the scaling exponents β and ν which characterize the nature of the phase transition in terms of how measured properties such as magnetization diverge near the transition temperature. Of particular debate in this set of experiments was the change nature of the transition in terms of p for different dimensional systems.

We believe this sort of numerical experiment is not atypical in its general approach: these types of studies are becoming important due to the many research programmes in Computational Science [National Computational Science Institute 2007]. It is necessary to map out the parameter space, which in this case is logarithmic and needs to span several numerical decades so as to deduce power law or other behaviour. It is only practical to simulate relatively small systems of, for example, order 10^6 or 10^7 spin nodes. Due to the noise accrued in measurements from the finite sized model systems it is necessary to repeat the experiment at each point at parameter space, with a different random starting configuration. It is only then possible to obtain statistically reliable measurements that characterize that point in parameter space. We would typically try to sample over 10 separate starting configurations, but more are desirable in some cases (up to 100 close to phase transition).

In summary then we have a relatively straightforward simulation algorithm that must be run over as large a model system as is practical and for many many runs, and the output results suitably averaged. This sort of model can be investigated qualitatively with very small model systems using standard simulation tools or problem-solving packages such as Matlab [The MathWorks 2007] or Mathematica [Wolfram Research 2007], however it is not feasible to conduct a thorough and statistically meaningful investigation of medium to large scale model sizes without a custom simulation code [Fox et al. 1994].

2. SIMULATION CODE ARCHITECTURE

We chose to implement our code in ANSI C. Alternatives were a Fortran variant or an object-oriented language like C++. Although we have found object-orientation to be very useful for some simulations in the case of the small-world Ising system the algorithm was straightforward enough that we preferred to reap the extra performance from lower memory and addressing overheads that arises from using simple yet application-relevant data structures. Although it would have been possible to implement these in a Fortran code, we found this excessively cumbersome and the memory addressing models in C were more appropriate, particularly when attaining a compromise between making use of high memory models and ensuring algorithm integrity in the case of a variable model dimension.

Since our code had to run on all platforms available to us, it was important we use a standard and non-proprietary programming language. For the most part we used the GNU compiler [Free Software Foundation 2007] on Macintosh, Linux and Windows Operating systems on various hardware platforms.

In the course of developing this code, we realized there were in fact some obvious abstractions that could be used to implement other models on a similar complex network. Our data structures and associated library operations support construction of a number of different small-world complex networks. These could be based on changes applied to random starting networks; additions to or deletions from a hyper-cubic lattice in multiple dimensions; or an arbitrary network structure imported from an external application [Herrero 2002].

The underpinning model could be based on bit-wise spins as in our work or on more elaborate vector- or tensor- based degrees of freedom. Some of these are shown in figure 1. The figure shows three different models, the first two of which we have implemented in our framework. The figure shows the number of unique states any lattice site can be in and the minimum amount of memory required to represent each site. We are currently implementing the more general Heisenberg model [Anderson 1959] into our framework. Ultimately it would also be possible to investigate complicated agents that reside on a network. We have done some preliminary work on these areas, particularly to model simulations of load balancing on wide-area networks and grids of distributed computers.




Model	Unique States	Representation
Ising		1 bit
Potts ($Q=3$)		2 bits
Heisenberg	 (spin vector with arbitrary position)	3 d.p. precision

Fig. 1. Three different simulation models. Lattice sites in the Ising model have two discrete states; three in $Q = 3$ Potts model; and an arbitrary *vector* state in the Heisenberg model.

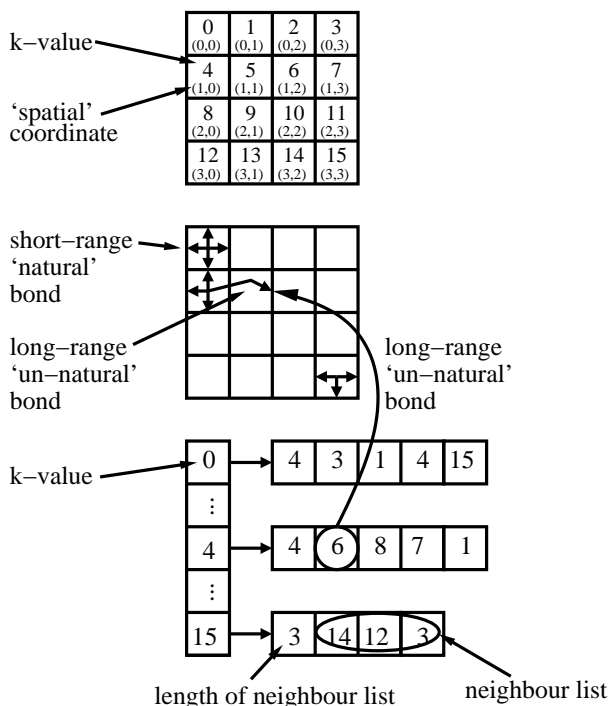


Fig. 2. Different views of a 4×4 lattice. Each lattice site has a *k*-value by which it is uniquely identified. There may also be some spatial coordinate associated with the *k*-value (top). Each lattice site also has a set of neighbours, shown graphically (middle) and as a list (bottom). The set of neighbours for a site can either be short-range ('natural') bonds or long-range ('un-natural') bonds. 'Holes' may be present in the system and thus not all lattice sites may have the same number of neighbours.

The code is written so that internally one is able to iterate through the total number of sites using a single value to uniquely reference each site. This unique reference we term the *k*-value. There will be a total number of N sites in the system, so $0 \leq k \leq N$ is always true. The use of *k*-values ensures that we are able to write the most crucial elements of our simulation code in a dimension-independent fashion: we are able to treat a multi-dimensional hypercubic lattice as if it were just a single dimension, while still being able to 'jump' back into multi-dimensional mode whenever required to look for neighbours, etc. Our code is written so as to be dimensionally independent: while there is no real limit as to the dimensionality our simulation can support, one quickly becomes compute bound for large lattice sizes in high dimensions.

A sample hypercubic lattice is shown in figure 2. The top view shows a 4×4 hypercubic row-major lattice. The *k*-values are shown in the top half of each lattice site, and (x, y) -coordinates are shown in the bottom half. Once again the use of *k*-values allow us to abstract over the fact that the sample lattice is actually two-dimensional.

In the case where the system is a hypercubic lattice we use a global array called

Algorithm 1 Construction of a hyperbrick

Require: non-null array a of integer array lengths
 $d \leftarrow$ length of the array a
malloc $Lengths$ array to be length $d + 1$ ints
malloc $LProducts$ array to be length $d + 1$ ints
 $Lengths[0] \leftarrow 1$
 $LProducts[0] \leftarrow 0$
if $d > 0$ **then**
 $Lengths[1] \leftarrow a[1]$
 $LProducts[1] \leftarrow 1$
end if
 $N \leftarrow Lengths[1]$
for $i = [2, d]$ **do**
 $Lengths[i] \leftarrow a[i]$
 $LProducts[i] \leftarrow LProducts[i - 1] * Lengths[i - 1]$
 $N * = Lengths[i]$
end for

$Lengths$ which represents the size of the lattice in each dimension. The construction of this array is shown in algorithm 1 which is implemented as a library of routines in a header file. To make the indices more intuitive, when implementing structural arrays, such as the $Lengths$ and $neighbours$ arrays we have chosen to start at position 1 instead of the C standard of 0. Thus, $Lengths[2]$ corresponds to physical dimension 2 in our code and $LProducts[2]$ corresponds to the product of all lengths up to but not including dimension 2. k -indices, however, start from 0 as they are merely a list.

Algorithm 2 $toK()$: Convert $[x_1, x_2, \dots, x_n]$ to k -value

Require: d the dimension of the system
Require: $Lengths$ array of each lattice dimension
for $i = [1, d]$ **do**
 $x[i] + = (x[i] < 0) ? Lengths[i] : 0$
 $x[i] \% = Lengths[i]$
 $index + = x[i] * LProducts[i]$
end for
return $index$ as k -value

The experimental apparatus was designed to operate with both hypercubic (see figure 2) and non-hypercubic systems, for example binary trees, ternary trees and Cayley trees [Stosic et al. 2005] (for example see figure 3). Our data structure, also lends itself to the study of hierarchical small-world graphs [Hinczewski and Berker 2006]. All internal algorithms are written so as to use k -values instead of relying on some spatial coordinate system. We have defined two simple routines to convert between the k -value and embedded spatial coordinate system when required. There are shown in algorithms 2 and 3.

Algorithm 3 *decomposeK()*: Convert k -value to $[x_1, x_2, \dots, x_n]$

Require: d the dimension of the system

Require: *Lengths* array of each lattice dimension

```

for  $i = [d, 1)$  do
     $x[i] \leftarrow k / LProducts[i]$ 
     $k \% = LProducts[i]$ 
end for
 $x[1] \leftarrow k$ 
 $x[0] \leftarrow d$ 
return  $x$  as vector

```

Algorithm 4 Construction of regular nearest-neighbour lists

Require: N the number of sites in the system

```

for  $k = [0, N)$  do
    decomposeK( $k, xVector$ )
    for  $n = [1, neighbours[k][0])$  do
         $neighbours[k][n] \leftarrow constructRdNeighbour(n, xVector)$ 
    end for
end for

```

The main area in which conversions between k -values and spatial coordinates is required is in the *construction* of a hypercubic lattice. As shown in figure 2 each lattice site has $2 \times d$ neighbours in the regular hypercubic lattice with periodic edges. In order to construct an internal representation of a site's neighbour lists we use algorithm 4. We merely decompose the k -value into a spatial coordinate vector and then use the routine shown in algorithm 5 to compute the neighbour in question. These two algorithms are valid regardless of the dimension of the system being studied. After the lattice has been constructed algorithm 5 is no longer used.

Figure 2 shows a 4×4 system that can be represented in our simulation. Starting from a regular periodic lattice there are a number of ways in which the lattice can be perturbed to simulate various effects, such as:

- (1) adding bonds
- (2) deleting bonds
- (3) swapping bonds spatially (with or without preserving z , the coordination number)

Each site in a regular lattice will have a bond with its nearest neighbours. For example, in a 2D system each site will have a bond with the neighbour above, below, to the left and right. These are what we term “short-range” or ‘natural’ bonds in the middle illustration of figure 2.

When extra bonds are added, bonds are swapped or edges are replaced, this can lead to sites that are connected not to the neighbouring lattice sites, but to those that are further away. These bonds we classify as “long range” or ‘un-natural’ bonds. Depending on the algorithm used to perturb the lattice the total number of bonds in the system may or not be preserved. Short arrows in the middle illustration of figure 2 indicate that a directional bond has the nearest neighbouring site as its

target. Longer arrows indicate an ‘un-natural’ bond.

As can also be seen in the middle illustration of figure 2, not all sites are required to have the same number of neighbouring bonds. In the figure one can see two sites with four bonded neighbours and one site with only three.

The bottom illustration in figure 2 shows the list representation of the short- and long-range neighbour bonds in the middle illustration. We index this list by the k -value of the site and then reference the vector of pre-computed neighbours. In order to cater for the situation in which there are fewer (or more) neighbours than in the regular case we store the actual number of neighbours in the zeroth element of the *neighbours* array. Once a site’s neighbour list has been perturbed, algorithm 5 can no longer be used, but it is no longer necessary anyway.

Algorithm 5 Determine the k -value of the n^{th} nearest neighbour

Require: *neighbour* code is 1,2,3,4,... $2 \times d$ odd value means previous in that dimension, even means next based on convention for neighbours storage of $d \times (-1, +1)$ pairs

Require: *xVector* the originating lattice location

$index \leftarrow 1 + (neighbour - 1)/2$

$len \leftarrow d + 1$

copy *xOriginal* to *x*

if $neighbour \% 2 == 0$ **then**

$x[index]- = 1$

else

$x[index]+ = 1$

end if

return $toK(x)$ as k -value

Of course, when the system being simulated is not a regular hypercubic lattice, such as a tree-based structure, then it is more difficult to construct the neighbour lists for each site. In order to construct the neighbour lists for, say, a hierarchical tree or Cayley tree, we write the algorithm using k -values so it never needs to make a conversion to spatial coordinates (if, indeed, it makes sense to do so). In the ternary tree shown in figure 3 one can see that there are no spatial coordinates shown in the graph of lattice sites, as it does not make sense to do so. The neighbour list, shown at the bottom of this figure shows the sites that are directly connected. In this instance, it does not really make sense to think of bonds as being distinctly short- or long-range.

Given the number of models and network variations we can simulate, the simulation code is incredibly compact at just under 2,500 lines of code. The hyperbrick header file is just over 200 lines and the Marsaglia random number generator [Marsaglia and Zaman 1987] is just under 250. The main pseudo-code for our simulation framework is shown in algorithm 6.

3. UPDATE ALGORITHMS

The core of our simulation program concerns the model update algorithm. An update algorithm is needed to first equilibrate the starting configuration, and secondly

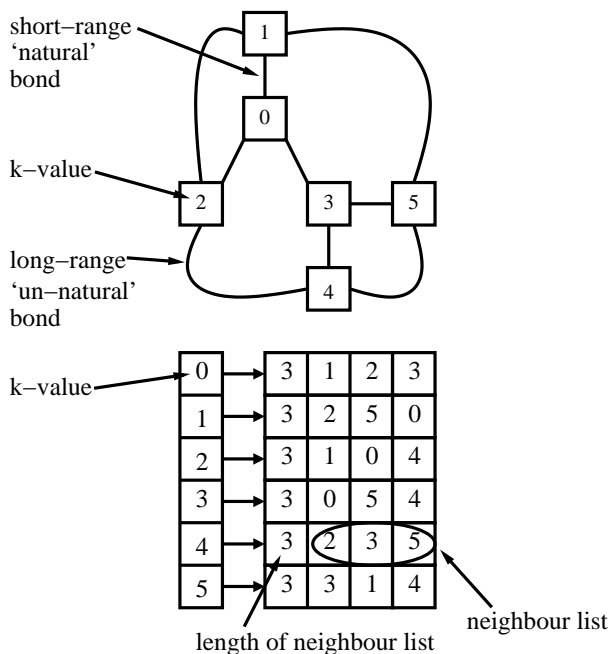


Fig. 3. Lattices in the study do not have to be hypercubic. This is an example of a ternary tree. While k -values are assigned to each site in the graph, it does not make sense to assign a spatial coordinate (top). The neighbour list is constructed from the sites in the graph that are directly connected (bottom).

to generate representative sample configurations for making measurements upon. Once a system of spin variables has been initialized and equilibrated we generate a long sequence of representative sample configurations by applying the update algorithm. This can be interpreted as applying a temporal dynamics to the system, even if the algorithm is purely stochastic in nature.

There are a number of possible ways of formulating the update process. The classic method uses either the Metropolis [Metropolis et al. 1953] or Glauber [Glauber 1963] Monte Carlo sampling method, whereby a new model configuration is constructed from the old one according to a thermodynamic probability. These algorithms are highly local and work as follows:

- (1) Choose a single spin – either randomly or in some predetermined order
- (2) Compute the change in energy (δE) of the total system if that spin variable were changed
- (3) Accept or reject the change according to a thermodynamic probability based on $e^{-\delta E/k_B T}$. This will accept decreases in energy but still allow increases albeit with a small probability. It models a form of heat-bath in thermodynamic contact with the model system.
- (4) Repeat for all spin sites

These local algorithms are unfortunately known to exhibit critical slowing down [Binder
ACM Journal Name, Vol. V, No. N, Month 20YY.

Algorithm 6 Simulation framework architecture pseudo-code

```

Require: Update algorithm
Require: Number of steps to equilibrate
Require: Number of steps to measure
Require: Thermal coupling of bonds in system
Require: Random number seed
  if resuming simulation from persistent store then
    Require: Persistent store reference
  else
    Require: System parameters (dimensions, number of spin states)
    Require: Perturbation probability
    Require: Method of perturbation
  end if
sanity-check input parameters
set all model defaults to sensible values
initialize random number generator
if resuming simulation from persistent store then
  load simulation data from persistent store
else
  create new regular model with given system parameters
  perturb model according to input parameters
  construct neighbour lists
  populate model with appropriate agents
end if
populate lookup tables
measure the state of the initial system
for number of equilibration steps do
  evolve system according to update algorithm
end for
for number of steps to measure do
  evolve system according to update algorithm
  measure system properties
end for
if saving simulation to backing store then
  save simulation to backing store
end if

```

and Heermann 1997]. This means that the new configurations are very similar to the old ones near the critical temperature. This is unfortunate as it means that measurements are highly correlated and are not good representations of the bulk behaviour of the system unless very long sequences are made. A better class of algorithm makes much more significant changes to the model system, while still generating representative individual configurations. There are a range of so-called cluster update algorithms that derive from the original work by Swendsen and Wang [Wang and Swendsen 1998]. Wolff's cluster algorithm [Wolff 1989] is one that we have found produces good measurement quality but is also computation-

ally very efficient [Baillie et al. 1992]. This algorithm works as follows:

- (1) Choose a starting individual spin
- (2) Construct a cluster of connected spin sites with a thermodynamically chosen probability for each connection
- (3) Change the entire constructed cluster at once (for example in the case of the Ising model, flip all its member spins)

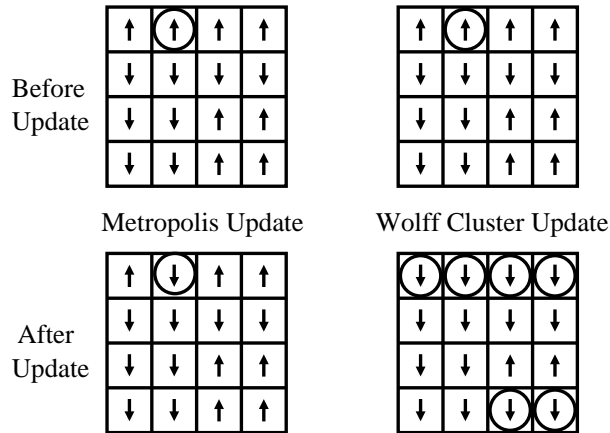


Fig. 4. Two different update algorithms. At each step Metropolis updates a single site (left) while Wolff creates a cluster of sites to be updated (right).

Figure 4 shows the local and non-local effects on a small four by four model system of applying the Metropolis and Wolff algorithms respectively. The Metropolis algorithm samples each lattice site once (on average) per update step to see whether it should be flipped. Note that we randomize the order in which sites are selected so as to avoid so-called ‘sweeping effects’. In contrast the Wolff algorithm selects a single lattice site and then builds a cluster of ‘like’ spins about this site. When no more sites can be added to the cluster it is then ‘flipped’ to update the system. The introduction of small-world links between sites can cause non-local updates to be propagated through the system very quickly.

In practice, the Wolff algorithm tends to make the system “thrash” at low temperatures. That is, it just flips the entire system as a single cluster, back and forth between states. It also performs poorly when the initial system is very hot as it is unable to build very large clusters. In practice therefore the Metropolis or Glauber algorithms are useful to equilibrate the system from its initial hot random start, and a hybrid of some Metropolis (or Glauber) steps and a Wolff cluster update are applied to generate representative samples from which measurements can be taken.

In building the simulation code we need to be able to customize the particular number of equilibration and hybrid updates to be applied for a particular model. This is managed by the job control scripts and discussed further in section 5.

The notion of local versus global site updates is also important for other simulation codes in completely different application arenas. For example, some fluid

modeling codes apply an update dynamics to a set of variables that represent finite differenced variables in a set of partial differential equations. Simple laminar flow might require only localized finite differencing operators that work using entirely local stencil operations. Turbulent flow models might require complex upwind schemes that incorporate more global information into the update of an individual model cell [Advisory Group for Aerospace Research and Development 1987].

4. PERFORMANCE CHARACTERISTICS

Figure 5 shows typical wall-clock timings for our Ising model simulation at various temperatures. The compute time is generally dominated by the time to construct Wolff clusters of spin sites. The cluster sizes that can be constructed exhibit a temperature dependence and this consequently manifests itself in the computational time. Since this physical property of the system exhibits phase transitional behaviour, there is also a marked phase transition in the timing curves. This makes it difficult to estimate how long computational jobs will take *a priori*. In a sense the job completion time has part of the problem solution information encoded in it. As the figure shows, there is a variation of up to two orders of magnitude in job completion time over the system sizes we studied.

As previously mentioned there are two main ways of running a simulation: we can either start the system from a random hot (or cold) state; or we can load a previously stored configuration and evolve the system from there. In the case of starting from a random configuration, we typically execute our simulation code using a small number of equilibration steps (heuristically determined to be approximately equal to the number of sites in the system, N) using the Metropolis algorithm and then perform measurements on the equilibrated system using the Wolff algorithm.

The Metropolis algorithm is quite slow yet effective, as it hits each site once on average per update step. It is possible to sweep through sites in deterministic order and thus avoid extra random number generator calls. We have avoided this to be sure of avoiding correlation artefacts [Hawick and James 2006]. A random site is chosen for the update and on average another random deviate is selected to decide whether to flip the site. In contrast, the Wolff algorithm selects one random site per update step and constructs a cluster of ‘like’ neighbouring sites; clusters can vary drastically in size from monomers up to, theoretically, the whole system size N .

We have extensively profiled our code using `gprof` and other operating-system specific tools. The number of equilibration steps is typically orders of magnitude smaller than the number of measurement steps. So, while the code spends most of its time in the Wolff update procedure (constructing clusters to flip), most of this time is spent generating random numbers for the update algorithms. Profiling shows that, on average, between one and three random numbers must be generated per site per update step. This is, of course, dependent on whether the system is above, below, or at the critical temperature, T_c . The relative number of random deviates generated per lattice site per update step is shown in table I. At T_c the code spends almost as much time measuring the statistics of the system as it does creating Wolff clusters to update the system.

Our code uses a k -integer index for each neighbour address, and for efficiency

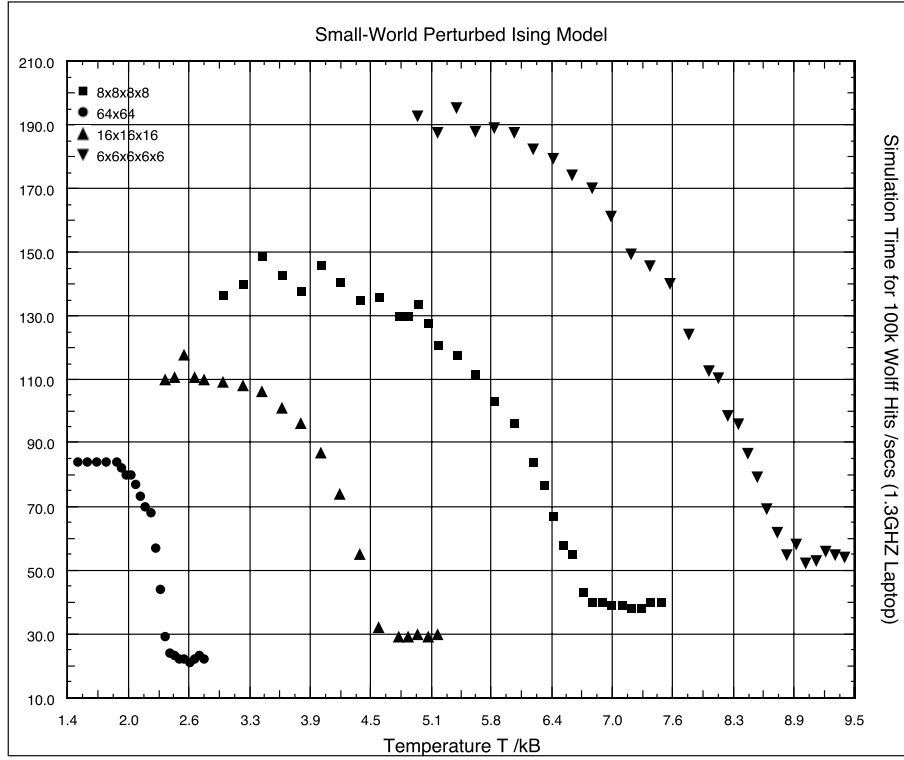


Fig. 5. Wall-clock timings (seconds) of different dimensions of Ising model code from 2-d up to 5-d. Each system has approximately the same number of lattice sites. The phase transition for each dimensional system ($T = [2.2, 4.5, 6.5, 8.8]$ for $d = [2, 3, 4, 5]$ respectively) manifests itself in the wall-clock timings.

Table I. Run-time characteristics of Ising simulations: average number of random deviates generated per site per update under different conditions (percentage of run-time consumed in generating random deviates).

Lattice	$T < T_c$	$T \approx T_c$	$T > T_c$
Unperturbed Lattice	1.3 (32%)	0.6 (20%)	0.006 (15%)
Small-world Perturbed Lattice	1.3 (26%)	1.1 (23%)	0.6 (18%)

we store both back and forward indices, even although most practical models we imagine would use bidirectional edges in the connectivity graph. Consequently the memory requirements for a model of N sites are modeled as:

$$N \times A + N \times \langle z \rangle \times B + N \times C \quad (1)$$

where A is the memory of a single degree-of-freedom variable such as a spin. Our C code uses the smallest practical addressable memory element – namely a `byte` or `unsigned char`. Other language systems might implement a packed bits mechanism, which is all we actually need for a spin bit in the Ising model. The

Table II. Minimum memory requirements for different systems shown in figure 5.

System Size	N	Minimum Memory Requirement
64^2	4096	86KB
16^3	4096	119KB
8^4	4096	152KB
6^5	7776	350KB
256^3	16777216	487MB
384^3	56623104	1.6GB
410^3	68921000	>2.0GB

parameter B is the memory needed for a k -index, and this is typically integer of standard address word size. So a 32-bit integer is sufficient for the work we describe here, although a `long long int` which implements a 64-bit integer on the GNU C compilation system could be used. Our scheme requires an average of $\langle z \rangle$ of these k -indices for each site, where z is the coordination number. In the case of a simple hypercubic lattice $z \equiv 2d$ where d is the system dimensionality. For the more sophisticated small-world rewiring models we discuss this simple relationship need not hold. The final parameter C represents a single vector of size N used to create dynamic Wolff clusters; we use a standard 32-bit integer for this vector and thus $C = B$. An illustrative table of memory requirements for models of different sizes is shown in table II.

5. EXPERIMENTAL DESIGN

A not insignificant part of the simulation architecture concerns how to manage the numerical experiment as a whole. As described in section 2 the core program can generate a suitable model configuration and can make measurements upon those configurations. Depending upon the wall clock time to carry out a run it is simpler to make a series of measurements as a single run or not. Too long a run is at risk from machine crashes, network failures and all the other interruptions that plague any compute server. These unfortunately become significant when runs take weeks to complete for some large model system sizes. Our code is therefore structured as shown in figure 6 to support loading and saving of model system configurations and the consequent chaining of measurement sequences. For shorter runs it is easier to combine the processes of model initialization, equilibration and measurements within a single program run.

We have invested considerable time and effort in the crafting of `tcsh` shell and Python scripts to manage the execution of jobs on local pools of processors. Our local supercomputer clusters use variants of the PBS batch scheduling software so we have modified our scripts to generate and manage these jobs. We have investigated the use of backing databases to help us with the considerable task of metadata management [James and Hawick 2005].

To aid in the management and analysis of measurements from our simulation framework we have standardized the way in which we name the configuration and measurement files. A base filename is used to capture the bulk characteristics of the simulation which is then appended with information particular to the system under consideration. A fixed-format encoding of parameters and run-number is used to

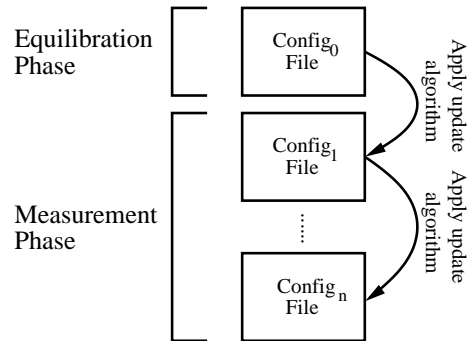


Fig. 6. Cascading configuration files. Each configuration is able to be stored in a persistent data store such as a file or database. This is especially useful when there are to be a large number of measurement steps on a given system: to combat against system failures we save the configuration at regular intervals, which can later be used as re-start points.

construct long, rich, filenames for the model system configurations.

As previously mentioned the only practical way of ensuring an extremely long experiment will succeed is to break the measurement runs into a number of manageable pieces and chain the configurations. A significant problem of file management and staging arises when the volume of data generated almost fills the available storage capacity.

6. SUMMARY AND CONCLUSIONS

Our simulation code was designed to be able to explore the properties of arbitrarily large model systems. In practice we are constrained by the size of system that will fit within computer memory but also by the computational time required to make statistically useful measurements on a model system of a particular size. The simulation model update algorithm's complexity is rather non-trivial to express because of the dependencies on the phase transitional behaviour discussed in section 4. Empirically however we are able to identify practical upper bounds on worthwhile model system sizes from the statistical noise or uncertainty obtained in the numerical measurements. Using 64-bit architecture computer platforms we were able to run model systems of more than 384^3 model spin sites, which requires more than the critical 4GBytes of memory address space normally achievable with a 32-bit processor bus architecture. However, in practice, these runs would take so many weeks of run time at the presently available processor speeds of a few GHz, that we were actually constrained to model system sizes of up to 256^3 spin sites.

This is likely to change for our particular code and model as clock speeds increase and make it feasible and useful to make use of model sizes that can only be accommodated by 64-bit address systems.

We have described the data structures and simulation methodology we developed for studying small-world network problems and their phase transitional properties. We have described our k -index spatial encoding data structure and the support algorithms that implement it. This allows the implementation of both arbitrary list and regular lattice models within a single implementation code with dimensional

independence. We have shown how the phase transitional nature of the problem intrinsically affects the performance behaviour of the simulation code itself. We believe these insights and methods may be of use for developers studying similar complex network systems.

Acknowledgments

The authors gratefully acknowledge the contribution of Helix supercomputer time by Massey University and the Allan Wilson Centre.

REFERENCES

- ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT. 1987. Special course on modern theoretical and experimental approaches to turbulent flow structure and its modeling. Tech. Rep. AGARD Report No 755, AGARD.
- ANDERSON, P. W. 1959. New approach to the theory of superexchange interactions. *Phys. Rev.* *115*, 2–13.
- BAILLIE, C. F., GUPTA, R., HAWICK, K. A., AND PAWLEY, G. S. 1992. Monte carlo renormalization-group study of the three-dimensional ising model. *Phys. Rev. B* *45*, 10438.
- BINDER, K. AND HEERMANN, D. W. 1997. *Monte Carlo Simulation in Statistical Physics*. Springer-Verlag.
- BINNEY, J. J., DOWRICK, N. J., FISHER, A. J., AND NEWMAN, M. E. J. 1992. *The Theory of Critical Phenomena*. Oxford University Press.
- FOX, G. C., WILLIAMS, R. D., AND MESSINA, P. C. 1994. *Parallel Computing Works!* Morgan Kaufmann.
- FREE SOFTWARE FOUNDATION. 2007. The GNU compiler collection. available at <http://gcc.gnu.org>.
- GLAUBER, R. 1963. Time dependent statistics of the ising model. *J. Math. Phys II* *228*, 4, 294–307.
- HAWICK, K. A. AND JAMES, H. A. 2006. Ising model scaling behaviour on z-preserving small-world networks.
- HERRERO, C. P. 2002. Ising model in small-world networks. *Phys. Rev. E* *65*, 066110.
- HINCZEWSKI, M. AND BERKER, A. 2006. Inverted berezinskii-kosterlitz-thouless singularity and high-temperature algebraic order in an ising model on a scale-free hierarchical-lattice small-world network. *Phys. Rev. E*. 066126.
- JAMES, H. A. AND HAWICK, K. A. 2005. Distributed scientific simulation data management. *J. Grid Computing* *3*, 39–51.
- MARSAGLIA, G. AND ZAMAN, A. 1987. Toward a universal random number generator. FSU-SCRI-87-50, Florida State University.
- METROPOLIS, N., ROSENBLUTH, A. W., ROSENBLUTH, M. N., TELLER, A. H. ., AND TELLER, E. 1953. Equation of state calculations by fast computing machines. *J. Chem. Phys.* *6*, 21, 1087–1092.
- NATIONAL COMPUTATIONAL SCIENCE INSTITUTE. 2007. National computational science institute. available from <http://www.computationalscience.org/>.
- NEWMAN, M., BARABASI, A.-L., AND WATTS, D. J. 2006. *The Structure and Dynamics of Networks*. Princeton University Press.
- ONSAGER, L. 1944. Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition. *Phys.Rev.* *65*, 3 (Feb), 117–149.
- STOSIC, B., STOSIC, T., AND FITTIPALDI, I. 2005. Thermodynamic limit for the ising model on the cayley tree. *Physica A: Statistical Mechanics and its Applications* *355*, 2-4, 346–354.
- THE MATHWORKS. 2007. Matlab. available at <http://www.mathworks.com>.
- WANG, J. AND SWENDSEN, R. H. 1998. Monte carlo renormalization group study of ising spin glasses. *Phys. Rev. B* *13*, 37, 7745–7750.
- WOLFF, U. 1989. Collective monte carlo updating for spin systems. *Phys. Lett.* *228*, 379.

WOLFRAM RESEARCH. 2007. Mathematica. available at <http://www.wolfram.com>.

...