CPS 713

MONTE CARLO SIMULATION FOR STATISTICAL PHYSICS

Paul Coddington

Northeast Parallel Architectures Center

January 1996

STATISTICAL MECHANICS

Statistical Physics

Some physical processes, especially quantum processes such as nuclear reactions, are inherently probabilistic, and can only be described statistically.

Many large classical systems (e.g. a volume of gas) have so many variables, or *degrees of freedom* (positions and velocities of every molecule), that an exact treatment is completely intractable, and in any case, not really useful. A statistical approach works very well, since the number of degrees of freedom is extremely large.

Statistical mechanics can be used to model both these types of systems.

Stochastic (probabilistic) Processes

Example: Flux of neutrons in a nuclear reactor

Reactor is simulated by tracking a neutron with random initial conditions (position, momentum) through the system. There are different *probabilities* for absorbtion, scattering, triggering nuclear reactions to produce more neutrons, etc., for the different materials in the reactor. A statistical ensemble of such neutrons gives the required flux.

Systems with Many Degrees of Freedom

Example: Thermodynamic properties of gases

It is totally impractical and not useful to know the exact microstate (positions and momenta of all molecules) of a large volume of gas.

The useful properties are statistical: average energy of particles (temperature), average momentum change from collisions with walls of container (pressure), etc.

The law of large numbers implies the error in the averages decreases as the number of particles increases. Macroscopic volume (state) of gas has $O(10^{23})$ molecules. Thus a statistical approach works very well!

A given state or *configuration* of the system occurs with a particular *probability*, subject to a *probability distribution.*

Applications of Statistical Mechanics

Statistical Mechanics is applicable to:

- Radioactive processes, nuclear reactions
- Quantum processes in semiconductors, superconductors, etc.
- High energy physics, reactions in particle accelerator collisions
- Thermal properties of matter (solids, liquids, gases), phase transitions, heat conduction
- Magnetic and electronic properties of matter
- Metallic alloys
- Diffusion and percolation

Applications of Statistical Mechanics (cont.)

Statistical mechanics techniques are also applicable to:

- Quantum field theories of electromagnetism (electrons, photons) and strong nuclear force (quarks, gluons)
- String theories and quantum gravity
- Quantum chemistry (structure and interactions of molecules)

Calculations in Statistical Mechanics

Analytic methods: almost always approximations. Very few exact solutions. Good for understanding basic physics, but often break down in regions of interest such as phase transitions. Little knowledge or control of errors, not easily systematically improvable.

Monte Carlo methods: calculations usually involve sums or integrals over very large number of dimensions or configurations. Monte Carlo is the obvious choice. Can compute quantities for any parameter values (including where analytic methods don't work). Errors can be estimated, and are systematically improvable by using more sample configurations and larger systems (i.e. more computer power!).

SPIN MODELS

Example: Spin Models of Magnetism

This example will be used as the basis for most of the discussion of simulation in statistic physics, for a number of reasons:

- Simple models of real physical systems
- Relevant to a variety of other systems linked by the theory of critical phenomena and phase transitions, e.g. thermodynamics and quantum field theory
- Analytic solutions exist for some simple models, which is useful for checking numerical techniques
- Classic example of the use of Monte Carlo methods
- Different algorithms exist
- Interesting application for parallel computation – a lot of work has been done on parallel computers (and has helped drive their development)

Magnetic Phase Transitions

Many magnetic crystals have a phase transition from a magnetized (ordered) state to a demagnetized (disordered) state. This is not just a simple change in magnetization – it is characterized by a divergence of certain quantities (for an infinite system) at the Curie Temperature T_c.

Simple Model of Magnetism

Unpaired electron spins couple and align. The sum of their magnetic fields gives macroscopic magnetism.

$$
\begin{array}{rll}
\text{low temperature} & \Rightarrow & \text{order} \\
 & \rightarrow & \text{alignment} \\
 & \rightarrow & \text{large magnetization} \\
 & & \\
\text{high temperature} & \Rightarrow & \text{disorder (thermal fluctuations)} \\
 & \rightarrow & \text{random alignment} \\
 & \rightarrow & \text{spins and fields cancel} \\
 & \rightarrow & \text{no magnetization}
\end{array}
$$

The Ising Model

Spins have only two states, $+1\,(\uparrow)$ or $-1\,(\downarrow)$. Energy is given by

$$E = -J \sum_{\langle i,j \rangle} S_i S_j$$

S_i = spin at site i of crystalline lattice
$\langle i, j \rangle$ = nearest neighbors in the lattice
J = interaction strength

$\uparrow\uparrow$, $\downarrow\downarrow$	spins aligned	$\Rightarrow E = -J$
$\uparrow\downarrow$, $\downarrow\uparrow$	spins not aligned	$\Rightarrow E = +J$

If $J > 0$, the state of lowest energy is when all spins are aligned. This is called a ferromagnet.

$J < 0$ is an antiferromagnet, which also occurs in real systems (will come back to this later).

Ising Ferromagnet

For this model, statistical mechanics tells us

low temperature ($T \to 0$)	$\to$	low E
	$\to$	spins aligned
	$\to$	large magnetization
high temperature ($kT \gg J$)	$\to$	large thermal energy
	$\to$	spins randomly aligned
	$\to$	no magnetization

Here magnetization $M = \sum_i S_i$.

So low T and high T behavior for the Ising model is the same as a real ferromagnet.

Solution of the Ising Model

This model was proposed by Lenz in 1923 as a problem for his graduate student, Ising, who showed that the 1-d model had no phase transition, i.e. the magnetization varied continuously and slowly from $T = 0$ to $T = \infty$, with the susceptibility χ staying finite.

Onsager solved the 2-d problem exactly in 1944, showing that it had a phase transition of the type seen in real ferromagnets. This led to great advances in the theory of phase transitions and critical phenomena.

The 3-d problem also has a phase transition, but is not exactly soluble. Results come from analytic approximations and Monte Carlo simulation, which is the usual case for more complicated spin models.

Statistical Mechanics of Spin Models

How does one go about solving the Ising model?

Want to work out thermodynamic averages of physically measurable quantities such as the magnetization M and the energy E as a function of the temperature T. The thermal average of M is

$$\langle M \rangle = \sum_C p(C)\, M(C)$$

where $M(C)$ is the magnetization of the configuration C, $p(C)$ is the probability distribution for the configuration as a function of T, and the sum is over all configurations.

A fundamental result in statistical mechanics, which we will not derive here, but accept as a physical law, is that p is the *Boltzmann distribution.*

The Boltzmann Distribution

The Boltzmann probability distribution is defined by

$$\begin{aligned} p(C) &= \frac{1}{Z} e^{-E(C)/kT} \\ Z &= \sum_C e^{-E(C)/kT} \end{aligned}$$

Here E is the energy (or, more generally, the *Hamiltonian*) of the configuration, T is the temperature, and k is Boltzmann's constant. For convenience we often choose units so that $k = 1$, in which case T is measured in energy units.

Z is called the *partition function*, and is the fundamental quantity in statistical mechanics. All quantities of interest can be extracted from Z, so an analytic formula for Z implies an exact solution of the model. The sum will actually be an integral if the space of possible configurations is continuous rather than discrete.

MONTE CARLO METHODS

Review of Monte Carlo Integration

$$
\begin{aligned}
I &= \int_a^b f(x)\,dx \\
&\approx \frac{(b-a)}{N} \sum_{i=1}^{N} f(x_i)
\end{aligned}
$$

Instead of choosing x_i at regular intervals, just choose them at random. Error is purely statistical, $\propto 1/\sqrt{N}$ *independent of the dimension of the integral.* Other integration methods have error $\propto 1/N^{1/d}$.

Example is the numerical estimate of pi, using simple 2-d Monte Carlo (dartboard method).

$$
\begin{aligned}
\pi &= \int_{-1}^{1} \int_{-1}^{1} p(x,y)\,dx\,dy \\
&\approx \frac{4}{N} \sum_{i=1}^{N} p(x_i, y_i) \\
p(x,y) &= 1 \qquad x^2 + y^2 \leq 1 \\
&= 0 \qquad \text{otherwise}
\end{aligned}
$$

Monte Carlo Simulation

To calculate sums (or integrals, if the spins are continuous-valued) over such a large number of degrees of freedom, we resort to Monte Carlo techniques.

As with the Monte Carlo integration mentioned earlier, we could just generate configurations at random, and approximate the real thermal averages, such as $\langle M \rangle$, by Monte Carlo averages:

$$\overline{M} = \frac{1}{N} \frac{\sum_{i=1}^{N} M(C_i)\, e^{-E(C_i)/kT}}{\sum_{i=1}^{N} e^{-E(C_i)/kT}}$$

The problem here is that because of the rapidly varying exponential function in the Boltzmann distribution, most randomly chosen configurations will make a negligible contribution to the sum, since E will be relatively large.

Calculating the Partition Function

Suppose we want to calculate the exact partition function Z numerically. Need to do this for all T, but let's start with just one temperature. For a real system, want $O(10^{23})$ spins, but let's start small and try to solve for a 32×32 lattice, i.e. $O(10^3)$ spins.

Number of configurations in the sum $= 2^{32\times 32} \sim 10^{300}$.

Suppose we had a gigantic parallel supercomputer, with 10 million processors. Each processor could generate a configuration C, calculate $E(C)$ and the Boltzmann factor $e^{-E(C)/kT}$ and add it to the sum over configurations in one nanosecond (less than the time for a single instruction for the fastest modern computer). We run this calculation for the age of the universe. This gives

10^7 procs $\times\, 10^9$ configs/proc/sec $\times\, 10^{14}$ sec/yr $\times\, 10^{10}$ yrs
$\sim 10^{40}$ configs (not real close to what is needed)

Importance Sampling

In order to get sensible, accurate results when simulating statistical systems with a rapidly varying Boltzmann distribution, it is vital to use the idea of importance sampling in Monte Carlo integration.

Clearly the ideal situation would be to sample configurations with a probability given by their Boltzmann weight $p(C)$, which gives a measure of their contribution to the sum total. Then the Monte Carlo average for M would just be:

$$\overline{M} = \frac{1}{N} \sum_{i=1}^{N} M(C_i)$$

This is great! Except that the sampling probability $p(C_i) = e^{-E(C_i)/kT}/Z$ depends on the partition function Z, which is basically what we are trying to calculate in the first place!

If we don't know what $p(C)$ is, how can we do this?

Markov Processes

Let us set up a so-called *Markov chain* of configurations C_t by the introduction of a fictitious dynamics. The "time" t is computer time (marking the number of iterations of the procedure), NOT real time – our statistical system is considered to be in equilibrium, and thus time invariant.

Let $P(A,t)$ be the probability of being in configuration A at time t.

Let $W(A \to B)$ be the probability per unit time, or *transition probability*, of going from A to B. Then:

$$P(A,t+1) = P(A,t) + \sum_B \left[W(B \to A)\, P(B,t) - W(A \to B)\, P(A,t) \right]$$

At large t, once the arbitrary initial configuration is "forgotten," want $P(A,t) \to p(A)$.

Detailed Balance

Clearly a sufficient (but not necessary) condition for an equilibrium (time independent) probability distribution is the so-called detailed balance condition

$$W(A \to B)\, P(A,t) = W(B \to A)\, P(B,t)$$

This method can be used for any probability distribution, but if we choose the Boltzmann distribution

$$\begin{aligned} \frac{W(A \to B)}{W(B \to A)} &= \frac{p(B)}{p(A)} = \frac{e^{-E(B)/kT}}{e^{-E(A)/kT}} \\ &= e^{-\Delta E/kT} \\ \Delta E &= E(B) - E(A) \end{aligned}$$

N.B. Z does not appear in this expression! It only involves quantities that we know (k, T) or can easily calculate (E).

The Metropolis Algorithm

This dynamic method of generating an arbitrary probability distribution was invented by Metropolis, Teller^2, and Rosenbluth^2 in 1953 (supposedly at a Los Alamos dinner party).

There are many possible choices of the W's which will satisfy detailed balance. They chose a very simple one:

$$\begin{aligned} W(A \rightarrow B) &= e^{-\Delta E/kT} & \text{if } \Delta E > 0 \\ &= 1 & \text{if } \Delta E \leq 0 \end{aligned}$$

So, if $E(B) > E(A)$

$$\frac{W(A \rightarrow B)}{W(B \rightarrow A)} = \frac{e^{-[E(B)-E(A)]/kT}}{1} = e^{-\Delta E/kT}$$

and if $E(B) \leq E(A)$

$$\frac{W(A \rightarrow B)}{W(B \rightarrow A)} = \frac{1}{e^{-[E(A)-E(B)]/kT}} = e^{-\Delta E/kT}$$

Monte Carlo Algorithms

So we have a valid Monte Carlo algorithm if:

- We have a means of generating a new configuration B from a previous configuration A such that the transition probability $W(A \rightarrow B)$ satisfies detailed balance
- The generation procedure is *ergodic*, i.e. every configuration can be reached from every other configuration in a finite number of iterations

The Metropolis algorithm satisfies the first criterion for all statistical systems. The second criterion is model dependent, and not always true (e.g. at $T = 0$).

Monte Carlo References

Here are some references to books with introductions to Monte Carlo methods, especially for statistical physics:

- S. Koonin, *Computational Physics*
- H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods, Vol. 2*
- O. Mouritsen, *Computer Studies of Phase Transitions and Critical Phenomena*
- M.H. Kalos and P.A. Whitlock, *Monte Carlo Methods, Vol. I. Basics*

These are a bit more advanced:

- K. Binder ed., *Monte Carlo Methods in Statistical Physics*
- K. Binder ed., *Applications of the Monte Carlo Method in Statistical Physics*
- D.P. Landau et. al, eds. *Computer Simulation Studies in Condensed Matter Physics: Recent Developments*
- K. Kremer and K. Binder, *Comput. Phys. Reports* **7**, 259 (1988).

COMPUTER SIMULATION

The Metropolis Update

Note that the Metropolis algorithm does not specify *how* the changes to the configuration should be made — it just says that any *proposed* change to the system should be *accepted* with a certain probability that depends on the change in energy.

How the changes are made depends on the variables and the model being studied. The only constraints on the update procedure are:

- it should be ergodic
- it should not be biased in such a way as to violate detailed balance

Another issue is efficiency — the procedure should sample the configuration space as effectively as possible. There is often some freedom in tuning the algorithm to improve efficiency and performance.

Metropolis Algorithm for the Ising Model

For the Ising model, the obvious change in the configuration is to try to update (or "flip") a spin, i.e. flip the sign (or direction) of the spin variable. If we try to change many spins at once, ΔE will be large, so the probability to accept the change, $e^{-\Delta E/kT}$, will be small. So update a single spin at a time.

For a single spin flip, ΔE depends only on the spin values at the site and its nearest neighbors, i.e. the update is *local*.

If $\Delta E \leq 0$, we make the proposed change.

If $\Delta E > 0$, we make the change with probability $e^{-\Delta E/kT}$.

Programming the Metropolis Algorithm

Update a single site at a time. Choose sites at random to ensure no bias (in most cases, this is not really necessary, and a simple contiguous loop over lattice sites is sufficient).

A loop over all sites is referred to as one Metropolis *sweep* (or iteration).

The probabilistic part of the algorithm is done using a random number generator (hence the name – Monte Carlo – in reference to games of chance). Generate a random floating point number r in [0,1). A process with probability p is performed if $r < p$.

The Metropolis algorithm is very simple to program.

Code for Metropolis Algorithm (in Fortran 77)

See Fig. 1.

Measurements

We generate configurations of the Ising model using a Monte Carlo algorithm, such as Metropolis. Then what do we do with them? We want to numerically measure quantities of interest, such as:

energy	$E = -\frac{J}{V} \sum_{\langle i,j \rangle} S_i S_j$	V = volume (# sites)
magnetization	$M = \frac{1}{V} \sum_i S_i$	
correlation function	$\Gamma(n) = \frac{1}{V} \sum_i S_i S_{i+n} - \left(\frac{1}{V} \sum_i S_i \right)^2$	
correlation length	$\Gamma(n) \sim e^{-n/\xi}$	
specific heat	$C/V = \frac{1}{V} \frac{\partial E}{\partial T} = \left\langle (E - \langle E \rangle)^2 \right\rangle = \langle E^2 \rangle - \langle E \rangle^2$	
susceptibility	$\chi/V = \frac{1}{V} \frac{\partial M}{\partial T} = \left\langle (M - \langle M \rangle)^2 \right\rangle = \langle M^2 \rangle - \langle M \rangle^2$	

Quantities such as these are measured for each configuration, and the averages and statistical errors calculated.

Phase Transitions

Phase transitions are described by an *order parameter* which differentiates the two phases. Usually it is zero in the disordered phase, non-zero in the ordered phase, e.g. the magnetization for a magnetic system.

Phase transitions are classified in part by their *order*:

- First order — discontinuity in the order parameter or energy, i.e., first derivative of the partition function.
- Second order — divergence in the susceptibility or specific heat, i.e., second derivative of the partition function.
- Third order, etc.

First order transitions are classified by quantities such as the *latent heat*, the discontinuous change in the energy at the critical point T_c.

Critical Exponents

Second order transitions are classified by their *critical exponents*, which measure how quantities diverge at the critical point. N.B. These are asymptotic ($T \to T_c$) results.

$$\begin{aligned} M &\sim |T - T_c|^{\beta} && \text{magnetization} \\ \chi &\sim |T - T_c|^{\gamma} && \text{susceptibility} \\ C &\sim |T - T_c|^{\alpha} && \text{specific heat} \\ \xi &\sim |T - T_c|^{\nu} && \text{correlation length} \end{aligned}$$

In many cases, these exponents $(\alpha, \beta, \gamma, \nu)$ are *universal*, i.e., they do not depend on details of the model, but only on gross features such as the dimension of the space and the symmetries of the energy function. This explains the success of very simple spin models like the Ising model in providing a quantitative description of real magnets with complex interactions.

PROBLEMS, SUBTLETIES AND TRICKS OF THE TRADE

Problems in Monte Carlo Simulations

Although Monte Carlo simulation (using the Metropolis algorithm, for example) appears very straightforward, there are in fact many problems and subtleties that can trap the unwary and produce unreliable results.

The following are some examples of common problems that arise in Monte Carlo simulations, along with examples of techniques to attempt to overcome them.

Thermalization

Before taking measurements, we must be sure that the Markov process has *thermalized*, or reached *equilibrium*, i.e. $P(A,t) = p(A)$ and the arbitrary initial configuration has been "forgotten."

Approach to equilibrium is exponential $\sim e^{t/\tau}$. The *autocorrelation time* τ can be very large. You could thermalize for 100,000 sweeps at a certain T, but if τ is of the same order, then the measurements are suspect.

There are 3 main techniques for checking for thermalization:

Binning: Check to see whether measurements are converging to a constant average value. Can do this by *binning* the data, i.e. splitting measurements up into a number of large contiguous segments (*bins*). Averages are taken over each bin, and data from initial deviant bins is discarded. Binning can also be used to calculate errors (see later).

Thermalization

Autocorrelations: Measure the autocorrelation time τ over the whole simulation, and then throw away data for at least the first 10τ sweeps.

Hot and cold starts: Start from hot (random) configuration and also a cold (ordered) configuration. When the two have converged to the same results, then the system is equilibrated. (But what about first order phase transitions at T_c? Usually will not get convergence — in fact this is a good way to look for first order transitions.)

Ergodicity

Can be very difficult to prove that the method of choosing new configurations is ergodic (especially at $T=0$). Can get trapped in subsets of configuration space, such as periodic loops.

May also have updates that are *quasi-ergodic* — there may be a high energy barrier between subspaces, so it is theoretically possible to reach all possible states, but highly improbable in a finite time simulation.

Check: Compare results from multiple runs with different initial configurations.

Fitting

Once measurements are taken, generally need to fit the data to find parameters such as critical exponents, using a multi-parameter fit to data which usually has large errors. This is very difficult!

For example $C = A + B\,|T - T_c|^{-\alpha}$ requires a four parameter fit, and often there is so much freedom that a wide range of values are possible. Usually need to assume $A \approx 0$, and get T_c from other measurements.

The above formula is only true "asymptotically close" to T_c. How close is that? Not always clear. May see "apparent" or *pseudo-critical* exponents, which change or disappear as the size of the system is increased and we move closer to T_c.

There may also be $\log|T - T_c|$ corrections to the asymptotic formula, which are hard to distinguish from a small power-law exponent. In this case may need input from theory.

Finite Size Effects

When studying any macroscopic system with a very large number of degrees of freedom, invariably make an approximation and simulate a smaller and/or discretized model system. This introduces systematic errors called *finite size effects*. Have to understand these, and be able to extrapolate to an infinite system, usually by doing a number of simulations at different system sizes.

For spin models, we have a finite d-dimensional lattice of $V = L^d$ sites. But only get a true phase transition (i.e., divergence) when $L \to \infty$. For a finite system, get rounded peaks rather than divergences. The peaks narrow and increase in height as L is increased, and the location of the peak shifts slightly.

Many problems require an empirical extrapolation to an infinite system. But for phase transitions in statistical mechanics, some elegant and useful theoretical results exist.

Illustration of Finite Size Effects

Substitute for scanned figure ??

Finite Size Scaling

In some instances, e.g. second order phase transitions, the form of the asymptotic (large L) behavior is known for finite size systems.

$$C_{\max}(L) \sim L^{\alpha/\nu} \qquad C_{\max}(L) \text{ is height of specific heat peak}$$
$$|T_c(L) - T_c| \sim L^{1/\nu} \qquad T_c(L) \text{ is where } C_{\max}(L) \text{ occurs, } T_c = T_c(\infty)$$
$$M(T_c) \sim L^{-\beta/\nu} \qquad \text{Magnetization at } T_c \text{ is zero only for } L \to \infty$$
$$\chi(T_c) \sim L^{\alpha/\nu} \qquad \text{Susceptibility diverges only for } L \to \infty$$

Example of simple (non-rigorous) "proof" (for specific heat):

$$\xi_{\max}(L) \sim L \qquad \text{correlation length cannot exceed length of lattice!}$$
$$\xi_{\max}(L) \sim |T_c(L) - T_c|^{-\nu} \qquad \text{from definition of } \nu$$

Therefore

$$L^{1/\nu} \sim |T_c(L) - T_c|$$
$$C_{\max}(L) \sim |T_c(L) - T_c|^{-\alpha}$$
$$\sim L^{\alpha/\nu}$$

Finite Size Scaling

These relations are extremely useful, and even allow us to extract exponents and $T_c(\infty)$. But only if we are at "large enough" L. If ξ is very large (but not infinite), may get incorrect "pseudo-critical" exponents.

This is often the case at a "weakly" first order transition (very small latent heat). So it is often very difficult to distinguish between a first and second order phase transition, or even to say for sure if there is a phase transition at all — is $\xi = \infty$, or just $\xi \gg L$? May need extremely large system sizes to get correct results.

In general, we don't know the form of the finite size scaling, so must try to fit data to a *scaling function*, or go to large enough systems so that results $\rightarrow$ constant or can do a simple linear extrapolation.

Boundary Conditions

Generally use *periodic boundary conditions*, where the lattice wraps around on itself to form a torus. This has been shown to give the smallest finite size effects.

But then must be careful that in measuring the correlation function, can only take measurements to $L/2$ for a lattice of size L, since a distance n is equivalent to a distance $n + L/2$ on a lattice that wraps around on itself.

Tuning the Acceptance

For the Ising model, there is only one possible change to the state: $S_i \to -S_i$. More generally (e.g., continuous spin models) we are free to make the change in the variable as large or small as we like.

A small change means ΔE is small and so the *acceptance* (proportion of proposed changes that are accepted) is close to 1. But then we move slowly through configuration space. Large changes usually have large ΔE and thus low acceptance, which again means that configuration space is not sampled efficiently.

General practice is to do a tradeoff and tune the size of the change so that the acceptance ≈ 0.5.

Random Number Generators

Generating "good" random numbers can be a major problem in Monte Carlo simulations. Many basic random number generators (RNGs) supplied with computers (especially parallel computers) are not sufficient. Particularly true in the past, but unfortunately still true far too often today!

There are many statistical tests for uniform distribution, correlations, etc. Monte Carlo measurement of the energy of the 2-d Ising model is a good test — the result is known exactly for L^2 lattice. Some RNGs pass statistical tests, but fail the Ising model test!

Even some "good" RNGs having adequate randomness properties are not good enough for Monte Carlo simulation because their period of repetition is too small.

Random Number Generators

Multiplicative linear congruential generators (MLCG) using 32-bit integers have a period of at most $2^{31} \approx 10^{10}$. This many random numbers can be generated in seconds on a modern workstation.

Period must be much greater than the number of random numbers used in the simulation, or else the results can be incorrect. Modest Ising model simulation, 1024^2 lattice, 10^7 sweeps, uses $\sim 10^{13}$ random numbers.

Using 64-bit words, or combining two 32-bit MLCGs, can give periods $\sim 10^{18} \sim$ # nanoseconds per year. These are adequate for largest simulations done today (Gigaflop-year). Other generators (lagged Fibonacci, etc.) have even longer periods. These or combined 64-bit MLCGs will be required for Teraflop-year simulations.

Random number generators will be the subject of a later lecture.

Critical Slowing Down

A problem with Monte Carlo (MC) simulation is that the error decreases only as $1/\sqrt{N}$, so need to do 100 times more work to get one more significant figure accuracy. Unfortunately, things are even worse. Error is only $1/\sqrt{N}$ for N *independent* measurements. But MC configurations are usually correlated — C_{i+1} depends on C_i.

Can check this by measuring the *autocorrelation function* (similar to correlation function except in computer time, not real space) for an operator (measurement) A:

$$\rho(t) = \frac{\langle A(t)A(0)\rangle - \langle A(0)\rangle^2}{\langle A(0)^2\rangle - \langle A(0)\rangle^2} \sim e^{-t/\tau_{\mathrm{exp}}}$$

τ_{exp} is the (exponential) *autocorrelation time*, and is equivalent to the τ which measures equilibration time of the Markov process. Need to thermalize for time $t \gg \tau_{\mathrm{exp}}$.

Critical Slowing Down

The *integrated autocorrelation time*

$$\tau_{\mathrm{int}} = \frac{1}{2} + \sum_{t=1}^{\infty} \rho(t)$$

is a measure of how correlations affect statistical errors. It is normalized so that $\tau_{\mathrm{int}} \approx \tau_{\mathrm{exp}}$ if $\rho \approx e^{-t/\tau_{\mathrm{exp}}}$. It can be shown that for MC measurements

$$\begin{aligned} \text{error in mean} &= \sqrt{\frac{2\tau_{\mathrm{int}}}{N}} \cdot \sigma \\ \sigma &= \sqrt{\text{variance}} \end{aligned}$$

A major problem for MC simulations is that τ_{int} **diverges** at T_c!! This is known as *critical slowing down.*

Dynamic Critical Exponents

To decorrelate a configuration, need to make changes on a scale of the spatial correlation length. For local MC algorithms (such as Metropolis) the effect of a change is only seen by nearest neighbors. Since the update is a stochastic process, the changes should propagate through the system like a random walk, so the number of iterations required to propagate a distance ξ goes like ξ^2. At the critical point, $\xi \sim L$, so we expect $\tau \sim L^2$. This is a major problem for MC simulations, since we need large L.

More generally, $\tau(L) \sim L^z$ where z is called the *dynamic critical exponent*. For a local algorithm $z \geq 1$ and usually $z \approx 2$. One of the active areas of research in MC simulation is trying to find new algorithms with $z < 2$. These are usually non-local, multi-scale algorithms, c.f. multigrid methods for solving PDEs, where the same type of problem occurs.

OTHER MONTE CARLO ALGORITHMS

Acceleration Algorithms

A number of different methods have been used to attempt to "accelerate" the dynamics of Monte Carlo updates (i.e., reduce z).

- Over-relaxation
- Multigrid
- Fourier acceleration
- Cluster algorithms

The first two are commonly used to improve numerical solutions of PDEs. In some cases, over-relaxation can reduce z to ≈ 1. Multigrid Monte Carlo and Fourier acceleration work well in certain cases. Cluster algorithms work very well, but currently are only applicable to a small class of spin models.

Cluster Algorithms

The idea behind cluster MC algorithms is very simple. Local MC algorithms perform poorly because they update only one spin at a time. Cluster algorithms use a clever way of finding large clusters of sites which can be updated at once. (Remember that trying to update a group of sites chosen at random gives a large ΔE and thus a small acceptance).

Cluster algorithms were invented by Swendsen and Wang in 1987, for Ising and Potts spin models. They have since been generalized to other models, and other algorithms.

The Potts Model

The Potts model is an obvious generalization of the Ising model to an arbitrary number, Q, of discrete states. Proposed by Domb to his grad student, Potts.

$$E = -J \sum_{\langle i,j \rangle} \delta(s_i, s_j)$$

$$\begin{aligned} \delta &= 1 \quad \text{if } s_i = s_j \quad \Rightarrow E = -J \\ &= 0 \quad \text{if } s_i \neq s_j \quad \Rightarrow E = 0 \end{aligned}$$

As for Ising model, if $J > 0$, the energy is lower if neighboring spins are the same (aligned).

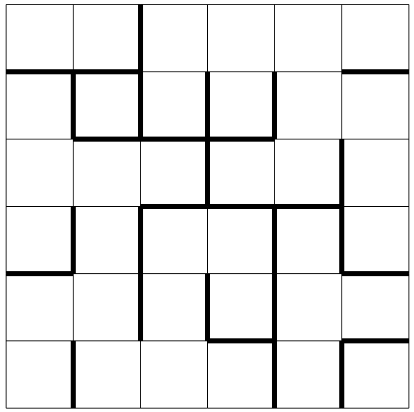
2-state Potts $\Leftrightarrow$ Ising.

$Q > 2$ model has different critical exponents, interesting phase structure. $Q \geq 5$ has first order transition in two dimensions.

Percolation

Swendsen and Wang derived their algorithm from a simple relation which transforms a Potts spin model into a bond percolation model. What is percolation?

Suppose we have a two-dimensional lattice of sites, and we put a bond between every pair of neighboring sites with probability p. Sites are then bonded together into connected clusters.



At a critical value of p, will get *percolation*, i.e., one cluster will span the lattice, meaning that there is a connected path across the lattice through a single *spanning cluster*. For an infinite two-dimensional lattice, $p_c = \frac{1}{2}$.

The Swendsen-Wang Algorithm

Swendsen-Wang (SW) found that by placing bonds between neighboring sites with the same spin value, the Potts model can be written in terms of interacting *clusters* of connected spins.

For the particular choice $p = 1 - e^{-\beta}$ ($\beta = \frac{J}{kT}$) the interaction energy is zero, so the clusters are independent. This means we can update all the clusters (by changing the current spin values to a random new spin value) completely independently.

Since the clusters can be very large, the update is highly non-local, and z can be very small (even zero).

The Wolff Algorithm

Wolff introduced a variant of the SW algorithm, in which a site is chosen at random, and a single cluster is grown around the site. This tends to favor larger clusters, and thus smaller autocorrelation times. Wolff also generalized the cluster algorithms to continuous spin models.

Cluster algorithms work amazingly well for some models, e.g., 2-d Ising model, where $z \approx 0$. Even for a large 512^2 lattice, $\tau \approx 10$, which is 0(1000) times smaller than τ for the Metropolis algorithm.

Correlations

Correlations fig goes here.

Connected Component Labeling

The cluster algorithms are very different to Metropolis. The main computational task is to take local information (bond connections) and work out global information (clusters of sites).

This is an example of *connected component labeling*, which is a standard graph problem — given a graph of vertices and edges, the problem is to identify those vertices which form a connected set, where a vertex is in the set if it has an edge connecting it to another vertex in the set. For cluster algorithms, the vertices are the lattice sites, the edges are the bonds, and the connected components are the clusters.

Component labeling is commonly used in image processing, to join neighboring pixels into connected regions which are the shapes in the image. Efficient sequential algorithms exist for this procedure, however implementing an efficient parallel algorithm is difficult.

Parallel Monte Carlo Algorithms

Metropolis algorithm for spin models is easy to parallelize efficiently, since it is regular and local.

- Regular data structures ⇒ simple domain decomposition
- Regular algorithm ⇒ lots of parallelism and perfect load balance
- Local interactions ⇒ local communications

Swendsen-Wang algorithm is difficult to parallelize efficiently, since it is irregular and non-local.

- Clusters are irregular (fractal!) in size and shape ⇒ hard to load balance, lots of non-local communication

Wolff algorithm is almost impossible to parallelize efficiently.

- Grow a single, irregular cluster starting at a random lattice site ⇒ little parallelism, very hard to load balance

PARALLEL METROPOLIS ALGORITHMS

Parallel Metropolis Algorithm

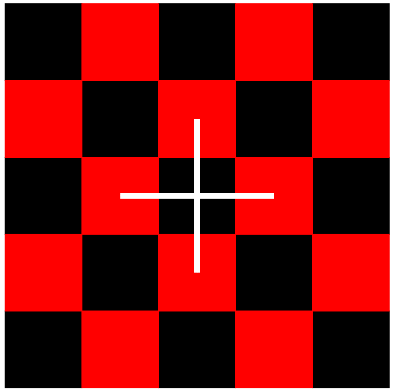
The Metropolis algorithm for a spin model is well suited to parallelism, since it is

- **regular** — so can use standard data decomposition of the lattice onto the processors and get good load balance
- **local** — the update of a site depends only on its nearest neighbors, so only local communication is required

However, cannot update all sites at the same time. This would violate detailed balance, since a site and its nearest neighbors are dependent, and thus must be updated separately.

Red/Black or Checkerboard Update

Instead, the lattice is partitioned into a checkerboard of alternating red and black sites (note this can also be done for $d > 2$ dimensions). Can then update all the black sites in parallel (since they are independent), followed by all the red sites in parallel.



This is known as a *red/black* or *checkerboard* update.

Note that unless there is more than one site per processor, the efficiency is at most 50% since half the processors will be idle.

Parallel Metropolis – Data Parallel

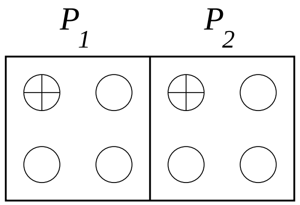
Parallel Metropolis using a checkerboard update is a data parallel SIMD algorithm. It is very simple to code in Fortran 90 and High Performance Fortran (HPF).

- Use standard domain decomposition and distribute the 2-d lattice among processors using `(BLOCK,BLOCK)` form.
- Use a *logical mask* (`black`) which only activates the appropriate (abstract) processors which deal with black sites. Then change the mask `(black = .NOT.black)` and update the other sites.
- To get data from neighboring sites, use periodic shift operations `(CSHIFT)` for periodic boundary conditions.

Metropolis Code in a Data Parallel Language

See Figure 2.

Parallel Metropolis – Message Passing



If the number of lattice sites per processor is $> 2^d$, need not use the checkerboard scheme, since cannot update neighboring sites at once.

This algorithm, when written using a message passing language such as MPI, looks **exactly** the same as the sequential algorithm! The only difference is how the `shift` subroutine, which finds neighboring spin values, is implemented:

- sequential algorithm — `shift` handles the boundary conditions.
- parallel algorithm — `shift` handles boundary conditions and does message passing if the neighboring spin is on a different processor.

Also, for the parallel algorithm, the `size` variable refers to the size of the sub-lattice on each processor, not the total lattice size.

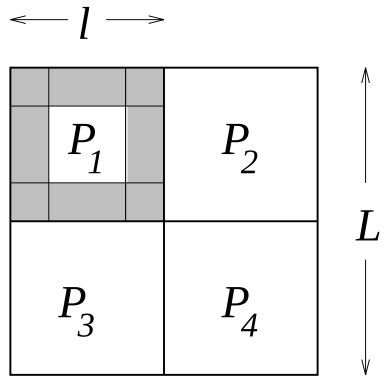
Efficiency of the Parallel Algorithm

Since the Metropolis algorithm for spin models is local and regular, it should parallelize very efficiently, even for the Ising model, which has very little computation.

For a 2-d grid of $N = P \times P$ processors, use a `(BLOCK,BLOCK)` distribution of the $V = L \times L$ sites of a 2-d lattice over the processor grid so that every processor has an $l \times l$ sub-lattice ($l = L/P$).

Communication time $\propto l$
(# edge sites of sub-lattice, i.e. perimeter).

Calculation time $\propto l^2$
(# sites of sub-lattice, i.e. volume).

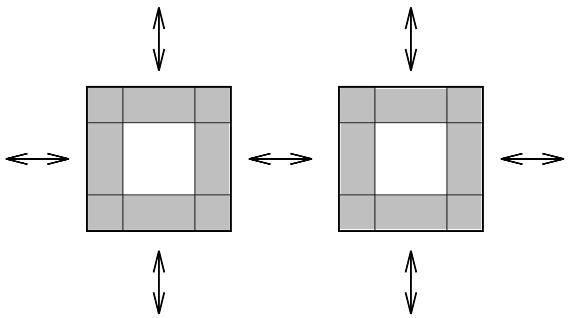


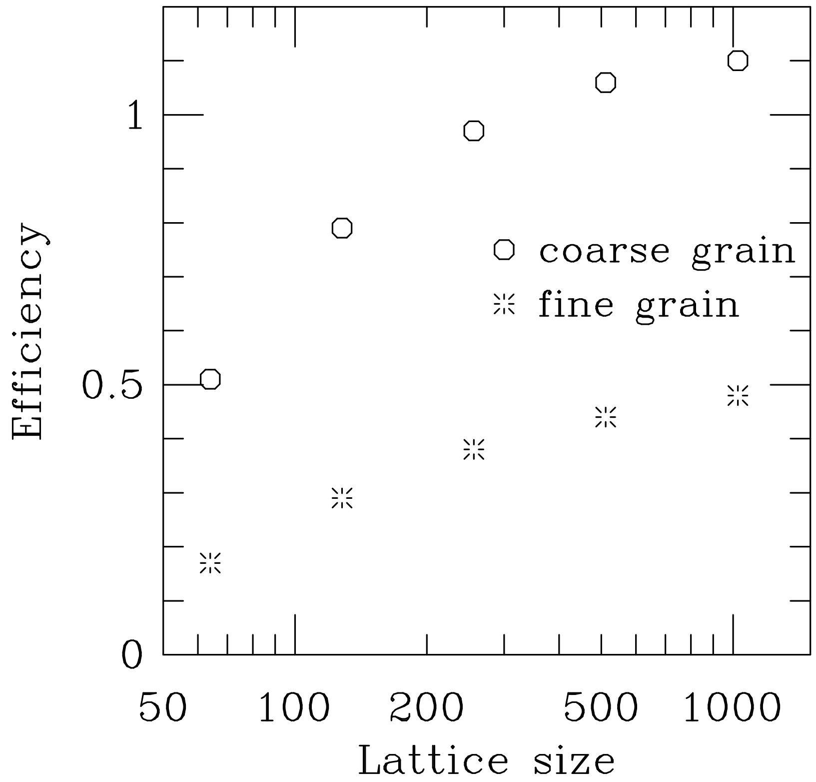
Thus, as long as l is large enough, the communication/calculation ratio will be small, and the efficiency will be near 1.

Coarse Grained Algorithm

On a coarse grained MIMD machine, there is a more efficient parallel Metropolis algorithm. This requires a checkerboard partitioning of the lattice. The data communication is done as large blocks rather than single values, by passing a block of edge data to neighboring processors after each red or black update.

This *blocked communication* greatly reduces the latency time for communication, thereby improving efficiency. It is also possible to overlap communications with computation, by computing edge values first, and sending them while the interior points are being computed.





Efficiency of coarse and fine grain Metropolis algorithms on 16 nodes of the nCUBE/2.

Measurements

Measurements of standard quantities such as energy and magnetization are easily done in parallel.

1. Calculate the quantities locally (e.g., energy per site), which can be done completely in parallel (no checkerboarding required)
2. Do a partial sum over sites in each processor
3. Combine the partial sums on each processor, using for example and `EXCOMBINE` in Express C or `KXCOMB` in Express Fortran.

In data parallel languages such as HPF, the last two steps can be implemented using the `SUM` primitive. For example,

```
energy = SUM (energy-per-site) / volume
```

Non-local Measurements

Some measurements are non-local, e.g., the correlation function $\Gamma(n) = \sum_i S_i S_{i+n}$. This can also be easily calculated in parallel, by continually shifting a copy of the spin variables one site at a time.

1. After the first shift, every (abstract) processor calculates the local value of $\Gamma(1)$, i.e., $S_i S_{i+1}$, at each site, and the sum over sites is taken (e.g. using `SUM`).
2. After the second shift every processor calculates the local value of $\Gamma(2)$, and the sum over sites is done.
3. This is continued until the spin has been shifted halfway around the lattice (only halfway due to periodic boundary conditions).

For message passing languages it is more efficient to only do the partial sums on each processor after every shift. The global sums are done at the end on the vector $\Gamma(n)$ rather than each of the individual values $\Gamma(1), \Gamma(2), \ldots$ in turn, so as to reduce latency.

Relation to PDE Solvers

The Metropolis algorithm for a 2-d spin model is similar in many ways to numerical methods for solving differential equations, such as Laplace's equation $\nabla^2\phi = 0$. This can be discretized onto a 2-d grid, with the update depending only on nearest neighbor points, e.g., an iterative scheme to solve this equation would replace ϕ_i by the average value of its four neighbors.

It is possible to iteratively update all sites at once (Jacobi algorithm). However, using a red/black update gives better convergence (like Gauss-Seidel). This is parallelized just like coarse grain parallel Metropolis algorithm.

- standard block data decomposition
- red/black or checkerboard update
- local (blocked) communications

PARALLEL CLUSTER ALGORITHMS

Parallel Cluster Algorithms

For the Swendsen-Wang cluster algorithm, all sites need to calculate their cluster labels, so can use standard domain decomposition.

The measurements can be done in parallel, just as for Metropolis.

Updating the spins can easily be done in parallel, as can setting up the bonds — all the sites just look at their nearest neighbors in the positive directions, and put a bond between them with probability $p = \left(1 - e^{-\beta}\right) \delta\left(s_i, s_j\right)$ for the Potts model.

The only difficult problem in parallel is the connected component labeling, that is, assigning to all the sites a cluster label that is unique for each cluster. So implementing an efficient parallel Swendsen-Wang cluster algorithm is basically equivalent to the problem of implementing an efficient parallel connected component labeling algorithm.

The Need for a Parallel Algorithm

The reason we want to use cluster algorithms is that they offer such a large reduction in computation time over the traditional Metropolis algorithm. For example, for the Ising model on a 512^2 lattice, the number of Metropolis iterations required to generate an independent configuration is of order 1000, whereas for the cluster algorithms it is of order 10, i.e., a 100-fold improvement in speed.

However, early attempts at running cluster algorithms on parallel supercomputers gave speed-ups only of order 10, whereas the Metropolis algorithm can easily have speed-ups of 100–1000 on vector and parallel supercomputers.

So on modern supercomputers, the improvement of the cluster algorithm is lost unless an efficient parallel algorithm can be found.

The first attempts at parallelizing irregular cluster algorithms got poor speedups compared to regular Metropolis algorithms. Figure taken from A.N. Burkitt and D.W. Heermann, "Parallelization of a Cluster Algorithm", *Comp. Phys. Comm.* **54**, 210 (1989).

Regular and Local vs. Irregular and Non-Local

The same type of problem occurs in many different guises in computational physics. In many applications, there are standard algorithms that are regular, simple, often local, and thus parallelize very efficiently. However, it is this very regularity and locality that makes them poor algorithms.

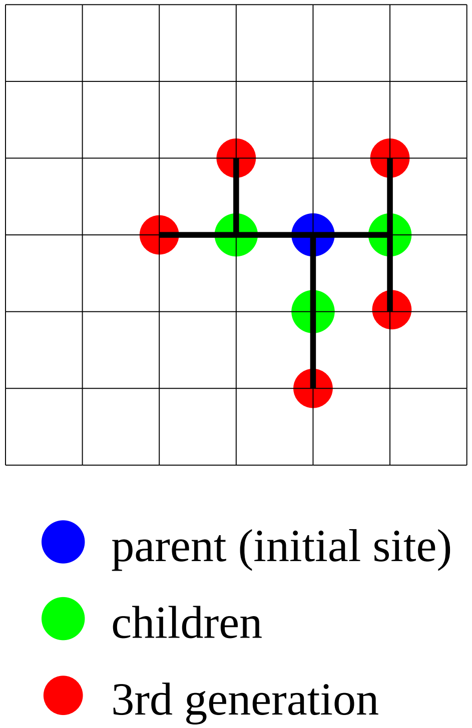
Complex physical problems and physical systems are usually irregular, often non-local, and change rapidly with time, so simple regular and local algorithms tend to have problems, such as critical slowing down.

More complex algorithms, such as non-local, irregular cluster algorithms; multiscale and multigrid methods for PDEs and spin models; adaptive, irregular grids for finite element calculations; hierarchical, adaptive N-body solvers; etc., all work much better, but are much more difficult to parallelize!

Labeling or Growing a Single Cluster

First look at how sequential component labeling algorithms work. Start with the Wolff algorithm, which “grows” a single cluster. Can liken this to the growth of an ant colony, in the following manner:

1. Pick a site at random and place an ant on it.
2. The ant reproduces by placing a child on the four neighboring sites (in 2-d) with probability p.
3. This “generation” of ants also looks at each of its four neighbors in turn, and if it is not already occupied, places a child of its own there with probability p.
4. The ant colony continues to grow until the final generation produces no children.



Depth-First Search ("Ants-in-the-Labyrinth")

A Wolff cluster is grown in the same way as the ant colony analogy. Placing a baby ant on a new site corresponds to putting a bond between the two sites (with probability p), and an ant colony corresponds to a cluster.

This is straightforward to program — just need to keep track of which sites are in the cluster, and which are the newly labeled sites (latest generation of ants) to be looped over in the next iteration.

This algorithm can also be used to do the connected component labeling for all sites in the Swendsen-Wang algorithm, given the configuration of bonds, by just adding an extra loop over all sites. If a site has already been labeled (assigned to a cluster), go to the next site. If not, increment the cluster label counter and give it to a new parent ant on that site, which then propagates the label through the labyrinth of bonds connecting sites in the cluster.

Breadth-First Search (FGHK)

First proposed by Fisher and Galler, implemented in a more efficient manner by Hoshen and Kopelmann for percolation studies.

Uses the idea of a *rooted tree.* Each site has a pointer to another site in the cluster, and so on down a tree-like structure. The site that is the root of the tree has the label for the cluster. Sites find their labels by traversal of the tree. Periodically pruning the tree (making shortcuts) makes the algorithm efficient.

The breadth-first search is better suited to a parallelism than the depth-first search, since it involves an outer loop over all sites, which can be parallelized using standard domain decomposition.

Local Label Propagation

The simplest and most obvious parallel component labeling algorithm proceeds as follows, assuming one lattice site per (abstract) processor.

1. Assign a unique label to each site (e.g., the processor number).
2. Every site looks at its neighbor in the positive x direction.
   If there is a bond between them, and the neighbor's cluster label is smaller than the label at the site in question, then it sets its label to be the same as its neighbor's.
   If the neighbor's label is bigger, then it gets the smaller label.
   This is repeated in the y direction (and other directions in higher dimensions).
3. Repeat the previous step until none of the labels has changed, in which case the clusters have been correctly labeled.

SIMD Local Propagation

Notice that this algorithm is purely SIMD, since every processor does the same thing at the same time. Since it propagates the label only one lattice spacing at a step, the number of steps required to label the clusters is the maximum path length (along bonded sites) between any two sites in the same cluster. For cluster algorithms this will be order L (the length of the lattice), since at T_c there is one large cluster.

This method will work better for labeling small connected components, and worse for labeling objects such as spirals. The method is much too inefficient for cluster algorithms, basically because the parallel algorithm suffers from the same kind of "critical slowing down" (trying to label a large cluster using local steps) as does the sequential Metropolis algorithm!

Need a <u>non-local</u> parallel algorithm.

MIMD Local Propagation

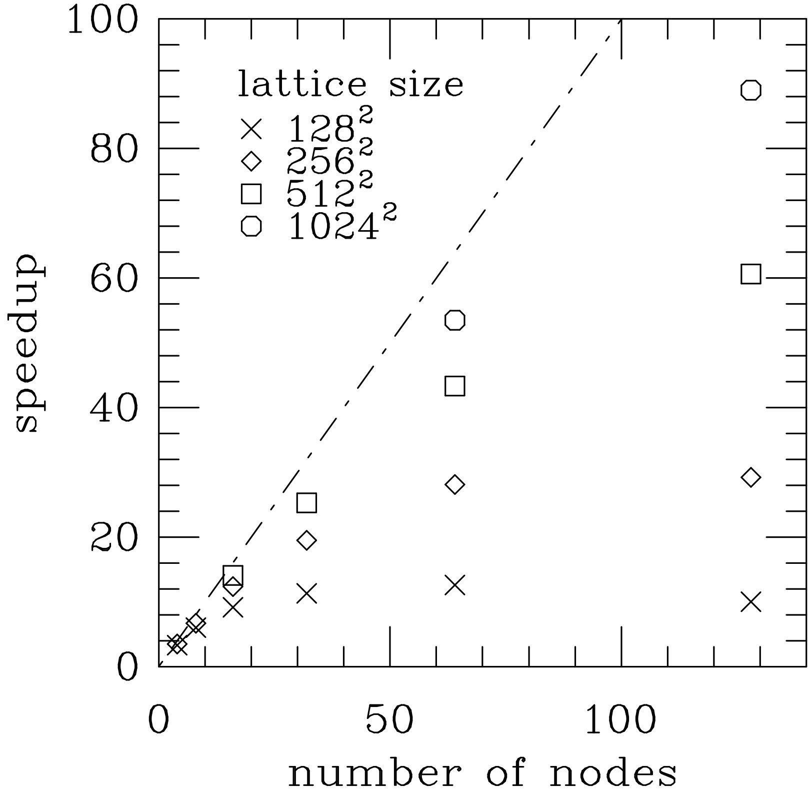
On a coarse grain MIMD machine, this algorithm is much more effective. In this case, we can label the clusters on the sub-lattice on each processor using a fast, efficient sequential algorithm. The local label propagation is now done only on the edges of the sub-lattices, to match up labels of clusters that cross processor boundaries.

Now have good load balancing, and the label of the largest cluster is propagated across the number of processors, not the number of sites. For an L^2 lattice on a P^2 processor array:

$$\text{communication} \sim P\left(\frac{L}{P}\right) = L \qquad \text{calculation} \sim \frac{L^2}{P^2}$$

$$\text{comm/calc} \sim \frac{P^2}{L}$$

As long as $L \gg P$, will get good efficiencies for this method.



Efficiency of MIMD local propagation algorithm on the nCUBE/2.

Parallel Labeling for Image Processing

There is a large literature of parallel algorithms for pixel connectivity and connected component labeling applications in image processing. But many of these are useless for spin models:

1. Pixel connectivity is a special case of component labeling, and some of the algorithms cannot be generalized.

2. Most computer science analysis of algorithms concentrates on *worse case* complexity. We are interested only in *average* time complexity to label a large number of configurations.

3. Algorithms with optimal asymptotic efficiencies are <u>not</u> necessarily optimal on real machines. For one "asymptotically optimal" parallel component labeling algorithm, need $\sim 10^6$ processors to get a speedup of 1!

4. Some algorithms may work well on small, regular objects, but not on large, irregular spin model clusters.

Global Equivalencing

Cluster labeling can be looked on as a special case of an equivalence problem: given a list of equivalences (site $i \equiv$ site j), put all the elements into equivalence classes (cluster labels). There are many algorithms known for solving this problem.

Parallel implementation:

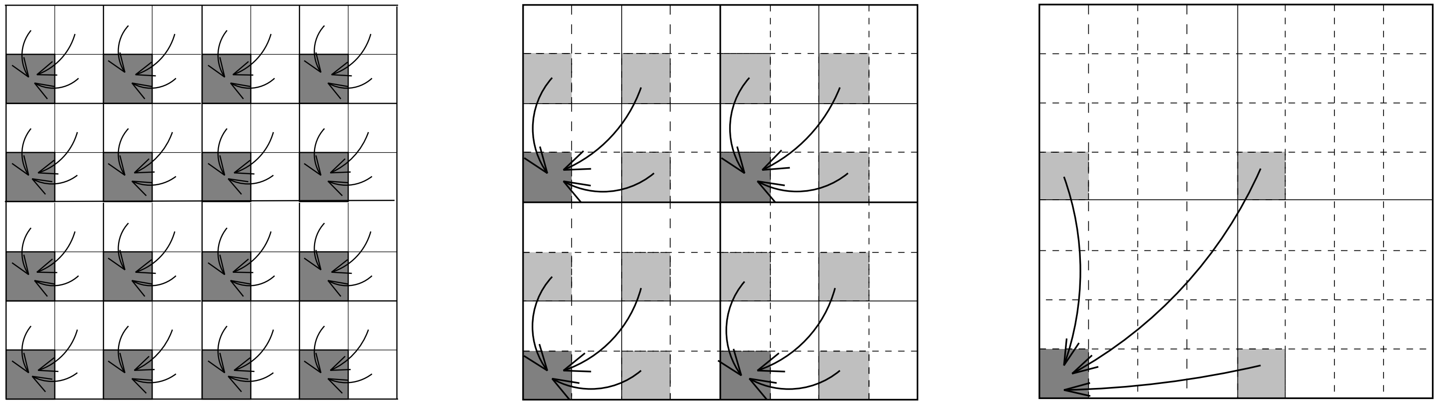
1. Label sub-clusters sequentially for sublattice on each node.
2. Each processor looks at the edges of its neighbors and creates a list of equivalences.
3. Lists are sent to one of the nodes, which creates the global equivalence classes and returns the results to all nodes.

Hierarchical Method

Problems with global equivalencing:

- Equivalence step is a sequential bottleneck
- One node needs enough memory for all cluster labels

These problems are alleviated by hierarchically merging the labels, using a quad-tree approach as shown below. Takes $\log P$ steps rather than P. More complicated to implement, a number of optimizations are required to get good performance.



Multigrid Algorithm

Instead of just looking at neighboring sites, look at sites a distance 2^m away (this is fast on a hypercube).

If the label is the same at any point, put a connection between the sites. If there are connections $A \xrightarrow{2^m} B \xrightarrow{2^m} C$, make a connection $A \xrightarrow{2^{m+1}} C$.

Now changes in the labels can be propagated long distances in one iteration. Expect labeling in $\sim \log_2 L$ iterations, each of which takes $\sim \log_2 L$ multigrid steps, each taking time $O(1)$ on hypercube. So time complexity of $O(\log L)^2$.

Get/Send Algorithm (Shiloach-Vishkin)

Another non-local SIMD algorithm that has been implemented is a method similar to an algorithm due to Shiloach and Vishkin. It is based on the rooted tree approach of the FGHK algorithm (breadth-first search).

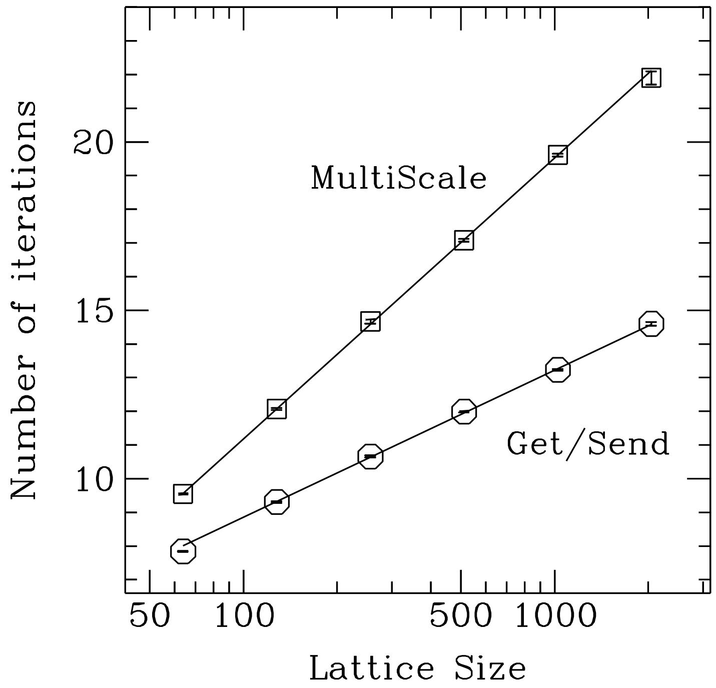
This "Get/Send" algorithm again has $\sim \log L$ iterations (rather than L), and $O(\log L)^2$ time complexity, since non-local communications take time $O(\log L)$.

This algorithm is faster in practice than multigrid, both of which are much faster than local label propagation. Currently these are only implemented on SIMD machines, where performance is quite poor due to load balance problems. Should be much better on MIMD machines.

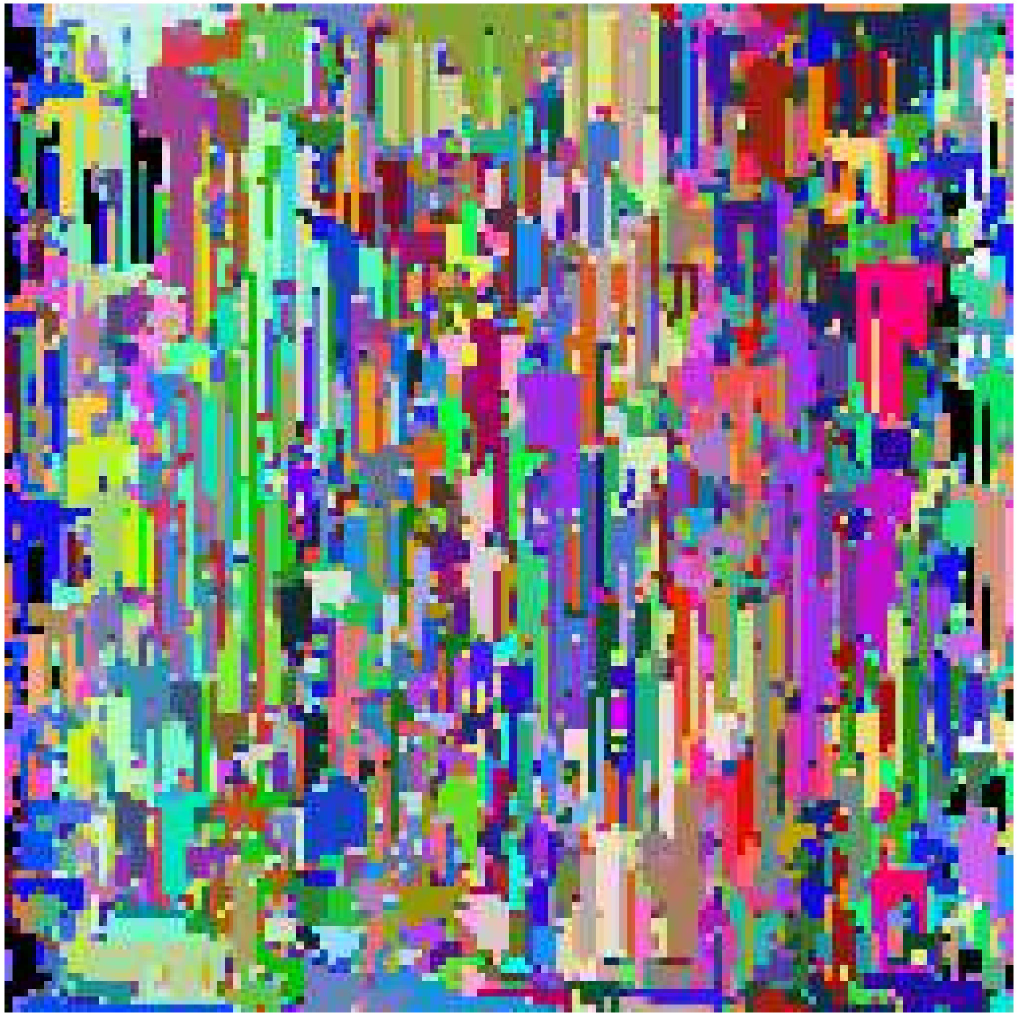
Get/Send Algorithm (cont.)

Start with every site (abstract processor) having the processor number as its unique label (i.e., the root of its own tree). Each iteration consists of the following steps:

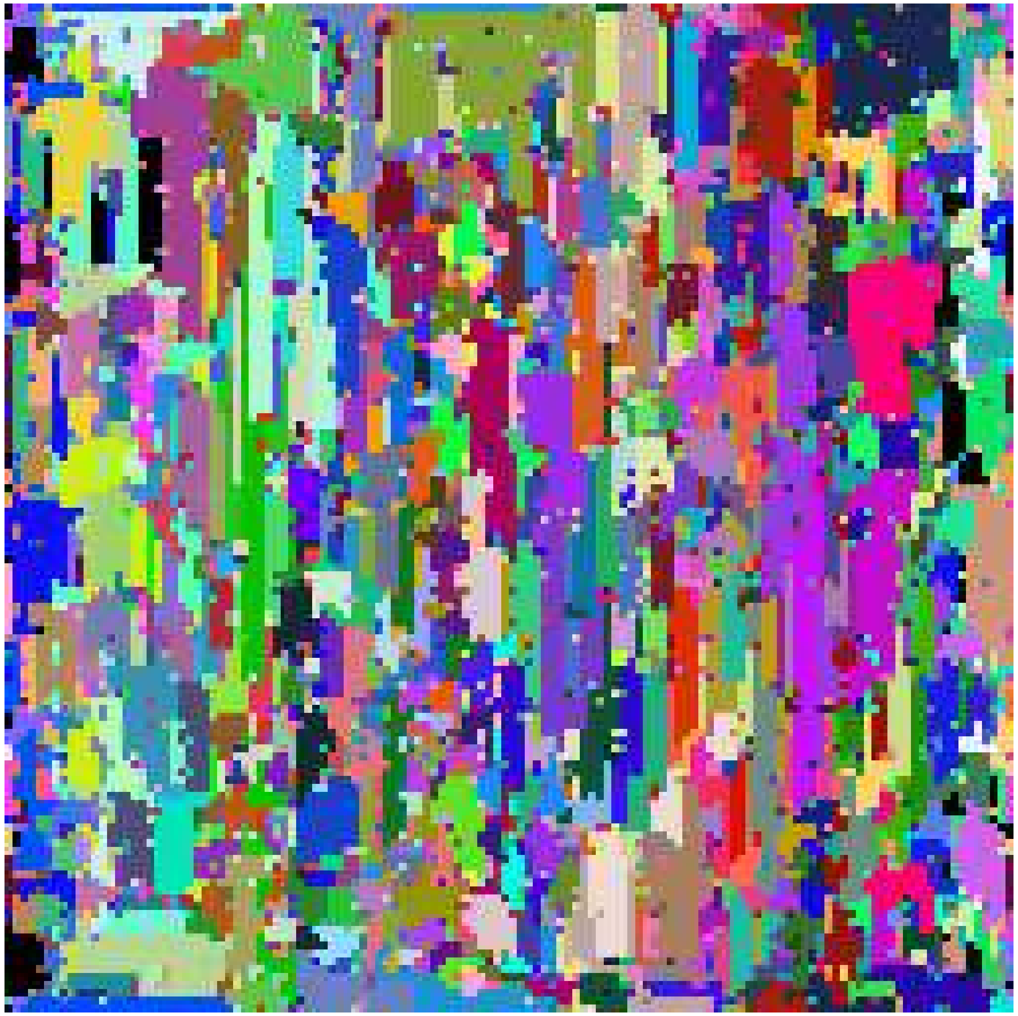
1. **Local label propagation**
2. **Send** — if a processor's label changes during step (1), it sends the new label to the root of its current sub-cluster (i.e., the processor number given by its old label), which picks the smallest of all the labels it receives as its new label.
3. **Scan** (optional) — propagate labels fast along rows and columns, using *scan* or *parallel prefix* operations.
4. **Get** — every site gets the (possibly new) label from the root processor for its current label.
5. **Check for completion** — continue if any labels have changed.



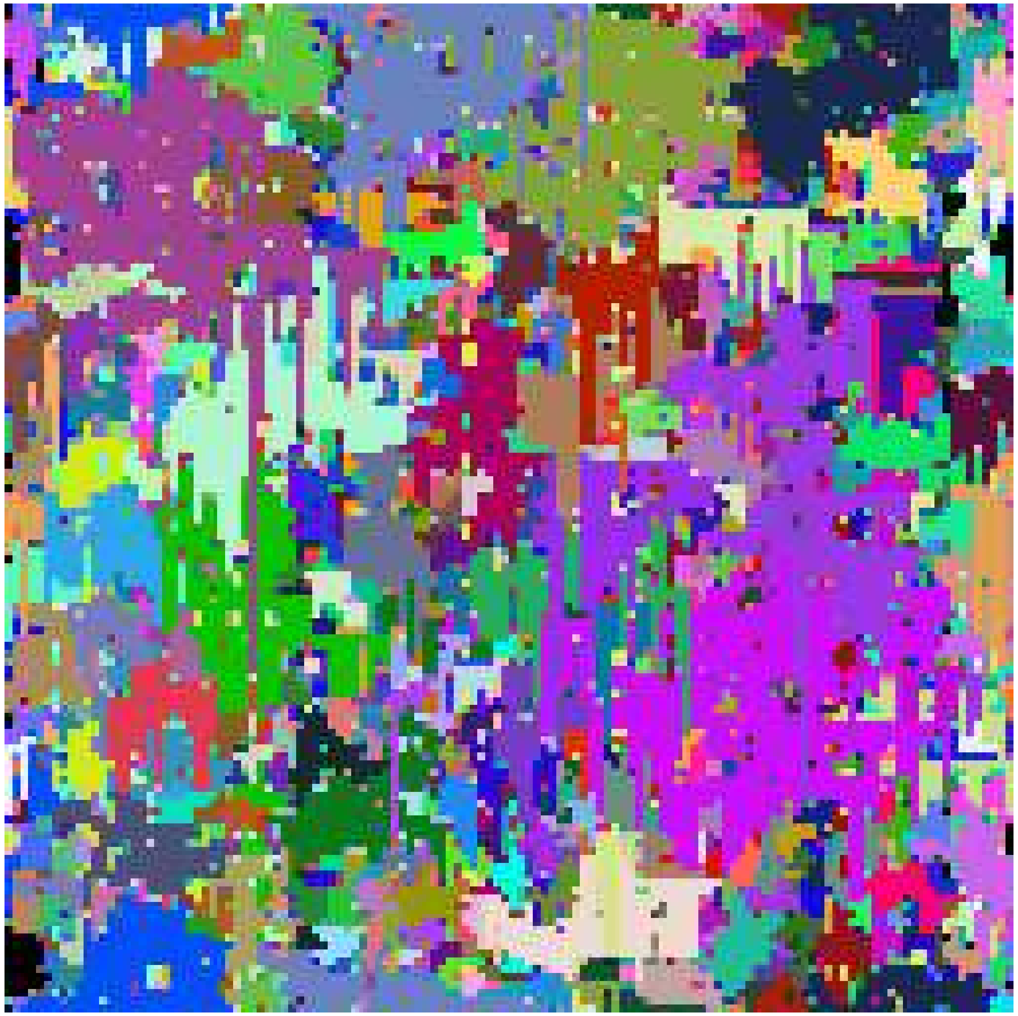
The Get/Send and MultiGrid algorithms are logarithmic in the lattice size L, unlike the simple local label propagation algorithm which usually takes time of order L, but can be order L^2 for complex clusters like spirals.



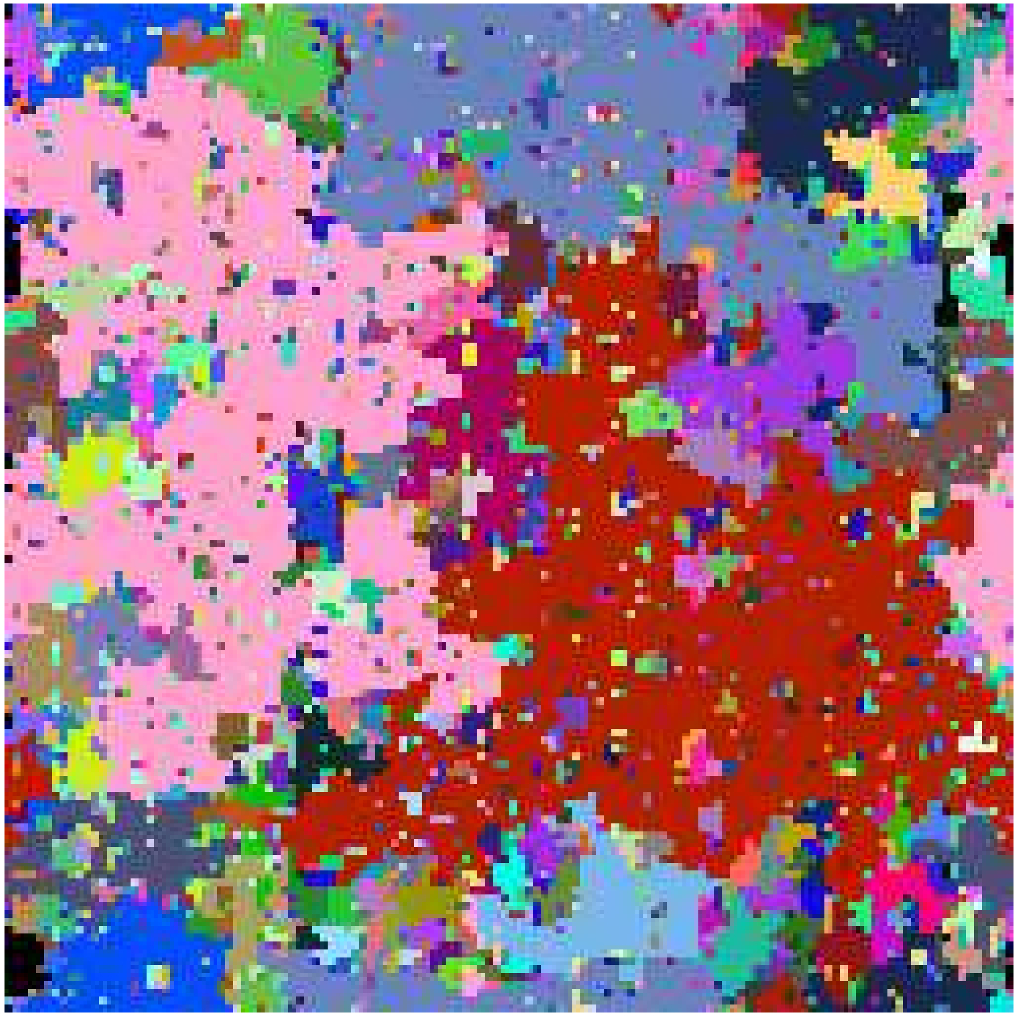
1. Get/Send algorithm after local label propagation and a scan operation. The colors represent different cluster labels, and the algorithm starts with random labels at each site.



2. Get/Send algorithm after a send.



3. Get/Send algorithm after a few iterations.



4. The final cluster labels.

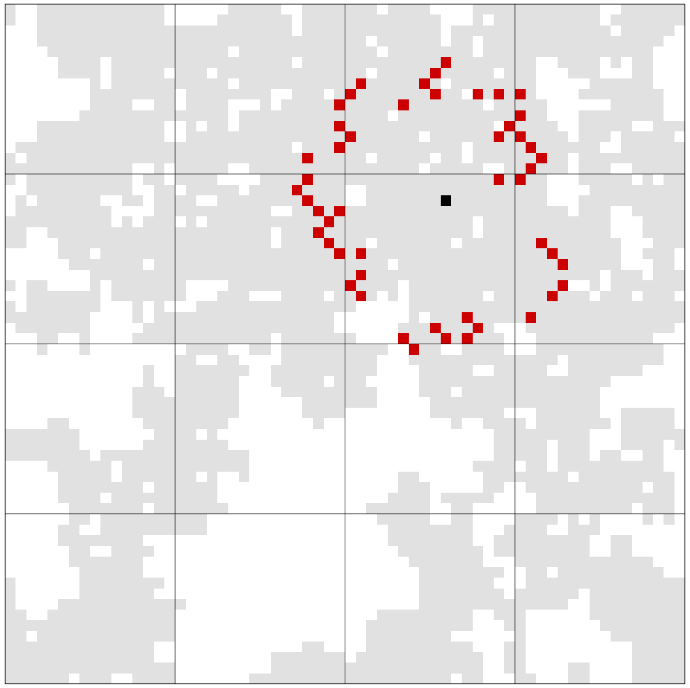
Parallel Wolff Algorithm

To parallelize the Wolff algorithm, need to use a parallel depth-first search ("ants-in-the-labyrinth") algorithm, since that is how the Wolff cluster is grown. Unfortunately, this is a sequential process — each ant generation has to follow the previous one.

The only parallelism is within a generation, i.e. over the "wavefront" of the expanding cluster. The amount of parallelism is very limited, especially if the cluster being grown is small.

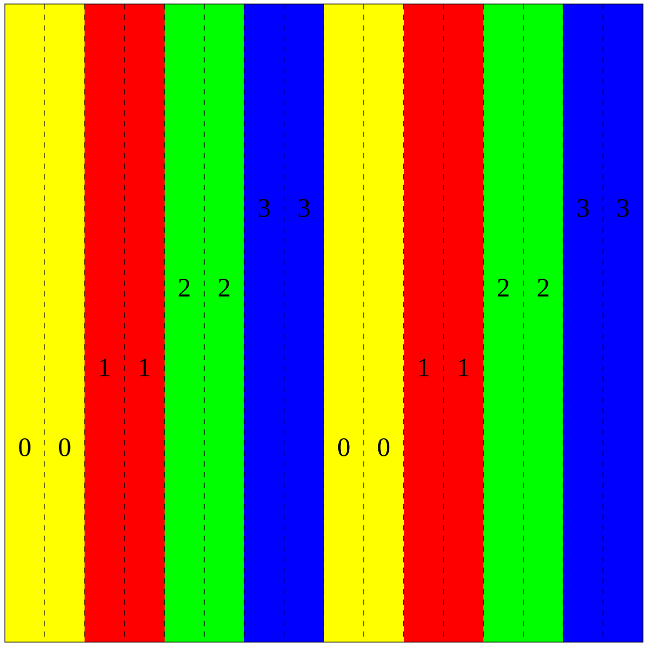
A major problem in parallelizing depth-first search methods is that the load balance can be very poor, since the computation is usually confined to a particular (contiguous) section of the data. Can sometimes alleviate this problem by using *scattered* or *cyclic* data distribution, rather than the standard block distribution. Howver there is a tradeoff — the data is no longer local so extra communication is required.

Problems with Parallel Ants

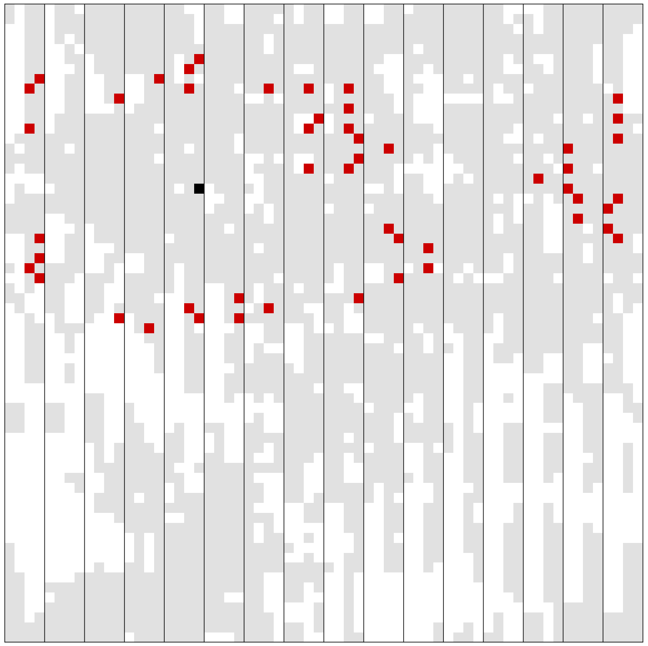


The Wolff cluster is shown here in gray, with the initial site in black. A particular generation of ants is shown in red. The black squares represent processor boundaries for a standard `(BLOCK,BLOCK)` data distribution. Note that the load balance (number of red sites per processor) is very poor.

Parallel Ants With Cyclic Data Distribution



In a column-cyclic distribution `(*,CYCLIC(2))` data is dealt out to processors (indicated with different colors) like cards to card players, 2 columns at a time.



The Wolff cluster shown in the previous figure but now with a column-cyclic distribution of data to processors. Note that the load balance is much improved by this non-local data distribution.

Independent Parallelism

For a data parallel Wolff algorithm, it is not possible to get the high efficiences that can be obtained for regular (e.g. Metropolis) algorithms, even using a cyclic data distribution. However highly irregular Monte Carlo methods, such as the Wolff algorithm, *can* be parallelized efficiently, since domain decomposition is not the only level of parallelism available for MC algorithms.

Remember that the basic idea of MC simulation is to approximate an extremely large or infinite sum (or large or infinite dimensional integral) by a finite sum over important configurations. The error is predominantly statistical, that is, proportional to $1/\sqrt{N}$, where N is the number of independent configurations. So here is another way the simulation can be done in parallel — if different processors are generating independent configurations to add to the sum.

Independent Monte Carlo Simulations

This is very easy to do. Just set up different random number streams on each processor (which is done in any case on a parallel machine, but here different processors could be different workstations in a network), set up different random initial configurations on each processor, and then run completely independent simulations on each processor. This gives a parallel efficiency of 100%!

This simple idea works very well for cluster algorithms, as long as the lattice is small enough that all the data will fit into the memory of a single processor. This was a problem for early parallel machines, but is usually no problem for workstation networks and machines like the IBM SP2. Note that this only works for MIMD machines, since cluster algorithms are irregular.

"Hybrid" Parallelism

Parallel Swendsen-Wang cluster algorithms for moderate sized lattices ($\sim 512^2$) work fairly efficiently for moderate numbers of nodes (~ 64) on a MIMD machine. However, this does not prevent us from using massively parallel MIMD machines with ~ 1000 nodes for this problem.

Can use a hybrid of domain (data) parallelism and independent parallelism (over different MC runs). For example, can run 8 independent simulations of 64 nodes each on a 512-node machine.

This gives much better performance than running parallel cluster algorithms on vector supercomputers, since these applications do not vectorize well.

Message Passing vs Data Parallel

Coarse grain parallel algorithms using message passing on MIMD machines give excellent performance for regular algorithms such as Metropolis, and fairly good performance for irregular cluster algorithms. This is because the efficient sequential algorithms can be used on the sub-domain on each processor.

Data parallel languages such as High Performance Fortran handle regular algorithms very efficiently, and are much easier to program that explicit message passing languages. However they do not perform as well for irregular cluster algorithms. This is mainly because the slower SIMD algorithms must be used even for the domain within each processor. There are also load balancing problems.

Irregular Problems in HPF

These types of irregular, non-local problems provide a real challenge to efficient implementation (for algorithms and compilers) in data parallel languages such as HPF.

It may be that good performance can only come by using library routines, e.g. for connected component labeling in this case, which would be implemented using an efficient message passing routine.

HPF can handle independent and hybrid parallelism using the `INDEPENDENT` construct.

RANDOM NUMBER GENERATORS

Pseudo-Random Numbers

Monte Carlo simulations are inherently probabilistic, and thus make frequent use of programs to generate random numbers, as do many other applications. On a computer these numbers are not random at all – they are strictly deterministic and reproducable, but they look like a stream of random numbers. For this reason such programs are more correctly called *pseudo-random number generators*.

A standard pseudo-random number generator aims to produce a sequence of real random numbers that are uncorrelated and uniformly distributed in the interval [0,1). Such a generator can also be used to produce random integers and sequences with a probability distribution that is not uniform.

Essential Properties of a Random Number Generator

Repeatability – the same sequence should be produced with the same initial values (or *seeds*). This is vital for debugging etc.

Randomness – should produce independent uniformly distributed random variables that pass all statistical tests for randomness.

Long period – a pseudo-random number sequence uses finite precision arithmetic, so the sequence must repeat itself with a finite period. This should be much longer than the amount of random numbers needed for the simulation.

Insensitive to seeds – period and randomness properties should not depend on the initial seeds.

Further Properties of a Good Random Number Generat

Portability – should give the same results on different computers.

Efficiency – should be fast (small number of floating point operations) and not use much memory.

Disjoint subsequences – different seeds should produce long independent (disjoint) subsequences so that there are no correlations between simulations with different initial seeds.

Homogeneity – sequences of all bits should be random.

TYPES OF GENERATORS

Multiplicative Linear Congruential Generators

One of the simplest, most widely used, and oldest (D.H. Lehmer, 1948) random number generators is the (multiplicative) linear congruential generator (MLCG or LCG). The generator is specified by integer constants A, C and M, and produces a sequence S_i of random integers via

$$S_i = (A * S_{i-1} + C) \bmod M \tag{1}$$

To generate real numbers in [0,1) we can just use $r_i = S_i/M$, so M should be large (near 2^{32} for 32-bit real numbers).

M is the maximum period for this type of generator, but the period can be much less. A, C and M must be carefully chosen in order to produce sequences that are random and have a long period.

Choosing the Parameters for LCGs

The period and randomness properties of linear congruential generators are highly dependent on the choice of the parameters A, C, and M. Poorly chosen values can give very small periods and bad randomness properties.

One common choice is $C = 0$, which can speed up the generator on some computers (although on most machines a combined multiply/add is no slower than a multiply). This is called a multiplicative linear congruential generator (MLCG). In this case we must be careful to exclude zero as a seed (or the period of the generator would be 1!). The maximum period is thus $M - 1$.

Since LCGs are mathematically very simple, analytic tests of randomness, such as the *spectral test*, can be used to find good parameter values.

Some Good Parameter Choices for LCGs

32-bit

A=69069, C=0 or 1, M=2^{32} (VAX)

A=1664525, C=0, M=2^{32} (transputers)

A=16807, C=0, M=$2^{31}-1$ (some IBM systems)

A=1103515245, C=12345, M=2^{31} (UNIX rand routine)

48-bit

A=$5DEECE66D_{16}$, C=B_{16}, M=2^{48} (UNIX drand48 routine)

A=5^{15}, C=0, M=2^{47} (CDC vector machines)

A=$2875A2E7B175_{16}$, C=0, M= 2^{48} (Cray vector machines)

64-bit

A=13^{13}, C=0, M=2^{59} (Numerical Algorithms Group)

Problems with Linear Congruential Generators

Linear congruential generators are very efficient, are theoretically quite well understood, and work well for many applications. However they have some drawbacks (which can generally be improved by increasing M):

- The maximum period is $M-1$, which is much too small for 32-bit generators where $M \leq 2^{32} \approx 10^9$, since this can be exhausted in a few minutes on a workstation. Need to use integers with at least 48 and preferably 64 bits. On most machines this requires multiple precision arithmetic which can be slow.
- d-tuples of successive random numbers show a regular lattice structure when plotted in d-dimensions. This is the *scatter plot* test.
- These generators have been proven to have correlations between numbers that are 2^n apart in the sequence. This can be a major

problem for applications using a regular grid or lattice.

Combined Linear Congruential Generators

Simple LCGs obtain the new random number solely from the previous number in the sequence. One might expect that correlations would be reduced by combining more than one previous value.

Empirically it has been shown that all the drawbacks of LCGs can be overcome by combining two LCGs.

$$\begin{aligned} x_i &= (A_1 * x_{i-1} + C_1) \bmod M_1 \\ y_i &= (A_2 * y_{i-1} + C_2) \bmod M_2 \\ S_i &= (x_i + y_i) \bmod max(M_1, M_2) \end{aligned} \tag{2}$$

Again, the parameters must be chosen carefully. A good choice is

$A_1 = 40014$, $C_1 = 0$, $M_1 = 2147483563$

$A_2 = 40692$, $C_2 = 0$, $M_2 = 2147483399$

Properties of Combined LCGs

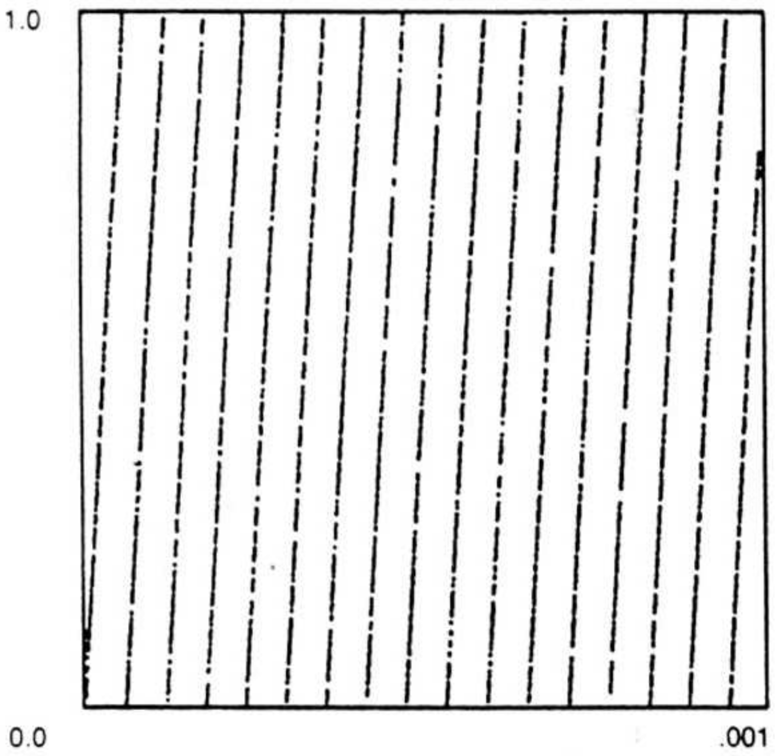
The period of this generator is $M_1 * M_2$, which can be of order 10^{18} for combining two 32-bit LCGs, which is adequate for current computers. A Gigaflop-year is 3×10^{16} flops.

Empirical tests show no d-tuple lattice structure or 2^n correlations in the sequence for the combined generator.

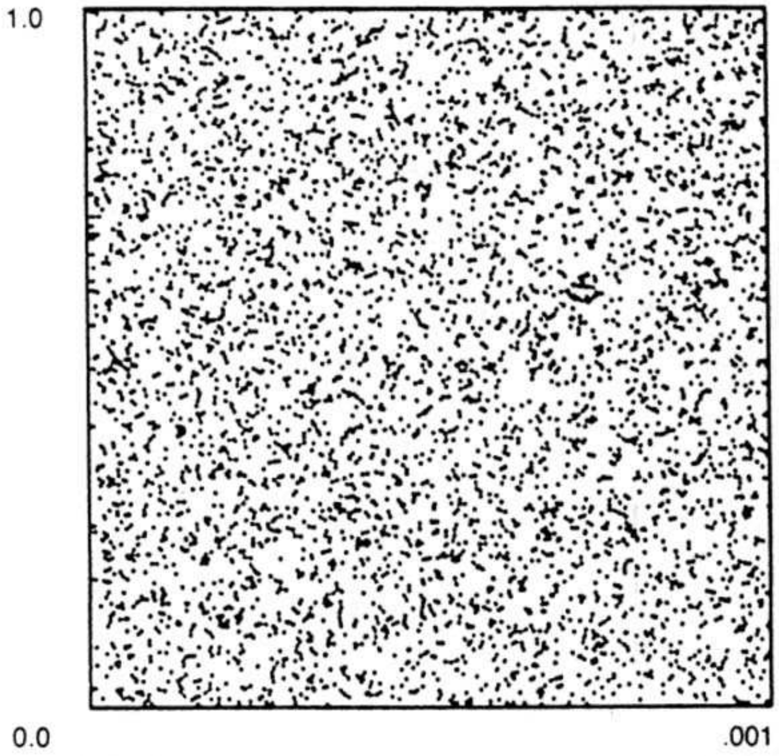
The period and randomness properties could be improved by using two 48-bit or 64-bit LCGs instead of 32-bit generators.

The authors of the Numerical Recipes books suggest that this generator, combined with a *shuffling* procedure to further reduce any possible correlations, has "perfect" randomness properties, with perfect defined as "we will pay $1000 to anyone who convinces us otherwise (by finding a statistical test that [this generator] fails in a non-trivial way, excluding the ordinary limitations of a machine's floating point representation)."

Scatter Plot Tests



Scatter plot for an LCG shows unwanted regular structure.



Scatter plot for a combined LCG shows desired randomness.

Lagged Fibonacci Generators

Lagged Fibonacci Generators (LFGs) attempt to improve on LCGs by using more than one previous value in the sequence, in order to reduce the correlations and increase the period. This is similar to combined LCGs, but in this case the numbers are taken from a single sequence, rather than two independent sequences.

We could combine the previous two numbers in the sequence, to produce something based on a Fibonacci sequence $S_i = S_{i-1} + S_{i-2}$. A better method is to use *lagged* Fibonacci sequences, where each number is a combination of any two previous values:

$$S_i = (S_{i-p} \odot S_{i-q}) \bmod M$$

where p and q are called the *lags*, and $\odot$ is any arithmetic operation, such as $+, -, *$ or $\oplus$ (the bitwise exclusive `OR` function `XOR`). These operations are done modulo some large integer value M. Multiplication is done on the set of odd integers.

Implementation of Lagged Fibonacci Generators

Need to store a table of the previous p numbers in the sequence, where p is the largest lag. This is known as the *lag table*.

For LCGs, need only one or two initial seeds to start the recursion relation. For LFGs, need to generate a *seed table*, i.e. all p initial values in the lag table. This must be done using a different random number generator. It is crucial that the numbers in the seed table are not correlated, or these correlations can be propagated throughout the sequence.

The lag table is accessed and updated using a circular list technique, where pointers to the p^{th} and q^{th} previous values are stored and updated, rather than moving the positions of all elements of the lag table after every iteration.

Properties of Lagged Fibonacci Generators

For b-bit precision seeds, the period is $(2^p - 1)2^{b-1}$, or $(2^p - 1)2^{b-3}$ for multiplication, for suitably chosen lags. This maximum period is only obtained when the lags p and q satisfy certain mathematical properties. Lists of suitable lags have been published.

Note that the period can be made *arbitrarily large* by increasing the largest lag (i.e. the size of the lag table). This is a very useful property for large-scale simulations. Empirical tests have shown that the randomness properties of these generators are also improved by increasing the lag.

Another very useful property of LFGs which use the operations $+$ or $-$ is that we can do all the computations modulo 1 on real numbers in the interval [0,1), and save having to do a multiplication to convert between integers and reals. These generators are consequently very fast.

Problems with Lagged Fibonacci Generators

Very few mathematical results have been derived about the randomness properties of these generators, so little is known about their theoretical properties. We are forced to rely on results of empirical statistical tests.

When the arithmetic operation used is `XOR`, the LFGs are more commonly referred to as generalized feedback shift register generators. These have often been used for simulation because they are extremely fast, since `XOR` can be done with simple bit operations. However the randomness properties of these generators are quite poor unless the lag is extremely large.

It has recently been shown that LFGs perform poorly for certain types of applications, such as random walks and percolation clusters, unless very large lags are used.

PARALLEL RANDOM NUMBER GENERATORS

Requirements for a Parallel Generator

For random number generators on parallel computers, it is vital that there are no correlations between the random number streams on different processors. For example, we don't want one processor repeating part of another processors sequence.

This could occur if we just use the naive method of running a LCG on each different processor and just giving randomly chosen seeds to each processor.

In many applications we also need to ensure that we get the same results for any number of processors. This is certainly the case for programs written in High Performance Fortran (HPF).

Parallel Algorithm using Leapfrog

A simple way of generating a sequence of random numbers in parallel is for processor P of an N processor machine to generate the sub-sequence $S_P, S_{P+N}, S_{P+2N}, \ldots$, so that the sequence is spread across processors. This is known as the leapfrog technique.

Multiplicative LCGs are mathematically simple enough that we can calculate analytically an arbitrary element in the sequence, by recursively applying equation 1:

$$S_{P+N} = (a * S_P + c) \bmod M \tag{3}$$

$$a = A^N, \quad c = \sum_{k=0}^{N-1} A^k = (A^N - 1)/(A - 1)$$

Thus moving through the sequence with stride N, rather than 1, is simply accomplished by specifying a different multiplier and additive constant for the generator on each processor. Note this will give the same results for any number of processors.

Parallel Algorithm using Splitting

If we can easily compute an arbitrary element of the sequence, we can also parallelize the algorithm by splitting it into contiguous sub-sequences separated by a large distance D, one on each processor. For instance, D could be the period divided by the number of processors. Splitting into N such sub-sequences for N processors will mean that

Processor 1 has sub-sequence $S_1, S_2, S_3, \ldots$

Processor 2 has sub-sequence $S_{D+1}, S_{D+2}, S_{D+3}, \ldots$

Processor k has sub-sequence $S_{(k-1)D+1}, S_{(k-1)D+2}, S_{(k-1)D+3}, \ldots$

This method will give the same results on different processors if we take N to be the number of abstract (rather than physical) processors. i.e. the number of elements in a data parallel array.

Parallel Lagged Fibonacci Generators

LFGs can be implemented very easily in parallel, by just running independent generators with different seed tables on different processors. This works very well *provided* the seed tables are set up to be independent and uncorrelated. If not, any correlations can persist throughout the simulation. The seed tables should be initialized using a different generator, such as a parallel linear congruential generator.

The problem with this simple approach is that it will give different results on different numbers of processors. This can be avoided by having a different generator on each virtual or abstract processor, i.e. each element of a data parallel array (e.g. in HPF or CMFortran). However this requires a lot of memory to hold the lag table (which must be large for good randomness properties) for each array element, which can be infeasible.

Parallel Lagged Fibonacci Generators II

Alternatively, can parallelize over the lag table, but this requires communication and is thus unacceptably slow compared to other parallel random number generators which have no inter-processor communication.

If `XOR` is used, the iterative formula is simple enough to allow analytic calculation of any term in the sequence, so leapfrog or splitting methods can be used. However these generators are rather suspect and do not have good randomness properties unless very large lags are used.

TESTING RANDOM NUMBER GENERATORS

Statistical Tests of Randomness

A number of statistical tests of randomness have been developed as tests for random number generators (see Knuth).

- Uniform distribution
- Serial correlations
- Run
- Collision
- Chi squared

Any test for which it is possible to calculate analytically what a truly random distribution would produce can be used.

Monte Carlo Tests of Randomness

Statistical tests are vital, but passing such tests is no guarantee that a generator will perform well for a certain application. Tests that are specific to certain applications can be more effective.

In the case of Monte Carlo simulations, there are a number of problems (e.g. 2-D Ising model, simple percolation and random walk models) for which exact results are known. It is thus possible to compare Monte Carlo results with exact results, and check that they agree within statistical error.

These "Monte Carlo tests" provide some of the best empirical tests of randomness.

Recommendations

Never use the default random number generator supplied by the computer vendor without first identifying the algorithm it uses. In many cases the algorithms are very poor and you will have to use another generator.

Always check your results by doing independent runs with different random number generators.

Good pseudo-random number generators:

- Combined LCGs
- Good 48-bit or 64-bit LCGs such as drand48
- Lagged Fibonacci generators using +,-,* as long as the lag is very large (at least 1000 for + and -, at least 100 for *).

More work needs to be done on developing and testing parallel random number generators.

References

D. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Methods*, (Addison-Wesley, Reading, Mass., 1981).

W.H. Press *et al.*, *Numerical Recipes in C*, (Cambridge University Press, Cambridge, 1992).

F. James, "A review of pseudorandom number generators," *Comp. Phys. Comm.*, **60**, 329 (1990).

P. L'Ecuyer, "Random numbers for simulation," *Comm. ACM* **33:10**, 85 (1990).

S. Anderson, "Random Number Generators on Vector Supercomputers and Other Advanced Architectures," *SIAM Rev.* **32**, 221 (1990).

P.D. Coddington, "Analysis of Random Number Generators Using Monte Carlo Simulation", *Int. J. Mod. Phys. C* **5**, 547 (1994).

SPIN GLASSES

Spin Models Revisited

Recall that the Ising model has energy

$$E = -J \sum_{[i,j]} S_i S_j \quad S = \pm 1 \text{ or } \uparrow\downarrow$$

For $J > 0$ we have a ferromagnet. Energy is smallest when neighboring spins are aligned, either $\uparrow\uparrow$ or $\downarrow\downarrow$. Ground state ($T = 0$ lowest energy state) is when all spins are aligned, or parallel (all $+1$ or all -1).

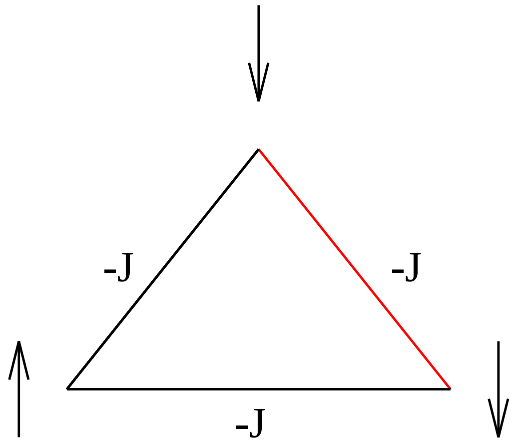
For $J < 0$ we have an antiferromagnet. Energy is smallest when neighboring spins are opposite, either $\uparrow\downarrow$ or $\downarrow\uparrow$. Ground state is when all spins are antiparallel, i.e. a checkerboard configuration for a square 2D lattice.

Thus there are only two possible ground states for Ising ferromagnets and antiferromagnets on a square lattice.

Frustration

Consider an Ising ferromagnet defined on a triangular lattice. This is similar to the square lattice case, except now every site has six nearest neighbors instead of four. The triangular lattice ferromagnet has a phase transition with the same critical exponents as the square lattice ferromagnet.

This is **not** the case for the triangular lattice antiferromagnet. The behavior is very different, since not all neighboring sites can simultaneously have the lowest energy, so some links (in red) are *unsatisfied*. The model is said to be *frustrated*.



Dealing with frustration is a difficult problem!

A Frustrated Spin Model

Generalize the interactions in an Ising spin model, so interaction strength becomes a variable, dependent on the lattice site:

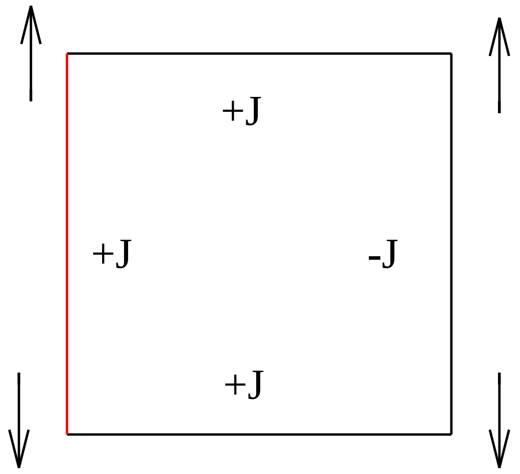
$$E = -\sum_{\langle i,j \rangle} J_{ij} S_i S_j$$

Consider the model where the strength is constant in magnitude ($|J_{ij}| = J$), but varying in sign, i.e.,

$$J_{ij} = +J \quad \text{or} \quad -J, \qquad J > 0$$

Suppose we choose the sign at random for every i, j link. This will introduce frustration, since not all links around certain plaquettes can be satisfied.

Randomness and frustration are the hallmarks of real systems called *spin glasses.*



Spin Glasses

Certain metallic alloys have magnetic interactions which oscillate in value as a function of the separation of the different atoms.

These alloys can be created in an *amorphous* state — the positions of the atoms are random and disordered like a glass, rather than regular and ordered like a crystal.

Since the distances between atoms in the glass varies, and the interaction oscillates in value as a function of distance, the magnetic interactions between spins are thus sometimes ferromagnetic, sometimes antiferromagnetic. This is what is known as a spin glass.

Spin glasses have a number of physically and mathematically interesting properties, and are the subject of much research.

Properties of Spin Glasses

- transition to a "glassy" phase at low temperatures
- susceptibility has a cusp rather than a divergence at T_c
- *remanent magnetization*, i.e. very long equilibration times – logarithmic rather than exponential relaxation to equilibrium, $M \sim 1/\log t$
- very large number of metastable states (local minima of the energy) in spin glass phase
- degeneracy (or near-degeneracy) of ground state, with many configurations having (nearly) the lowest energy value
- finding ground states is a difficult (NP hard) problem
- main properties of real spin glasses (e.g. amorphous alloys) can be described fairly well by $\pm J$ spin glass Ising model on a regular crystalline lattice

Monte Carlo Simulation of Spin Glasses

The interesting behavior in a spin glass occurs at very low temperature, so we want to do a Monte Carlo simulation of an Ising spin glass at low T.

Suppose we adopt the usual MC approach — generate an initial random configuration of spins, and then start running the simulation at the temperature T where we want to study the system. After some thermalization time τ, the system will reach thermal equilibrium, and we can start taking measurements as usual.

Right?

Well, not exactly.

Quasi-Ergodicity

Recall that a Monte Carlo algorithm will work only if it is ergodic, that is, it is possible to reach any state from any other state in a finite number of iterations.

The Metropolis algorithm is ergodic for $T > 0$ since there is always a finite probability $e^{-\Delta E/kT}$ of flipping a spin. However, the algorithm may not be ergodic at $T = 0$, since moving from one ground state to another in a frustrated spin model may require higher energy intermediate states.

For T very small, the probability of climbing out of a local minimum is non-zero, but may be so small that in any real simulation it will never happen. The system is then only *quasi-ergodic*.

Thermalization of Spin Glasses

Thus, for a system like an Ising spin glass with a rough "energy landscape" with many local minima, starting from a random initial configuration and running at a very low temperature T, the system will invariably get trapped in a local minimum, and never reach a thermalized state (at least not in the finite time of a real simulation).

This is analogous to *quenching* a metal, i.e., starting from a very hot (disordered) state, and rapidly cooling to a low temperature state (by plunging it into cold water, for example). For real metals, this will also produce a local minimum of the energy, leading to a brittle, non-crystalline state.

SIMULATED ANNEALING AND OPTIMIZATION

Simulated Annealing

In order to avoid the metastable states produced by quenching, metals are often cooled very slowly, which allows them time to order themselves into stable, structurally strong, low energy configurations. This is called *annealing*.

Annealing gives the system the opportunity to jump out of local minima with a reasonable probability while the temperature is still relatively high.

We can adopt the same approach in Monte Carlo simulations. We start with a random configuration at a very high temperature, and then reduce the temperature "very slowly" until we reach the desired low temperature. This should result in a thermalized (equilibrium) configuration.

Finding Ground States

In many cases, we are interested in finding the ground state of a spin model, i.e. the zero temperature, lowest energy state. In many regular models, it is possible to deduce the form of the ground state configurations. However, for some disordered frustrated systems, such as spin glasses, finding a ground state is very difficult – in fact it is NP hard, i.e. there is no polynomial time algorithm for finding a solution.

In practice, ground states are found using simulated annealing. This involves starting from a number of different random initial configurations, and slowly cooling them down to zero temperature. This produces local energy minima that are close to, and possibly equal to, the global minimum.

Optimization

Finding the ground state of a spin glass is an example of an *optimization problem*, i.e.

> Find the minimum (or maximum) value of a *cost function* $C(s)$ for all possible states s.

For the Ising spin glass, the cost function is the energy, and the states are all 2^N possible configurations of an N site lattice.

Kirkpatrick et al. realized that the method of simulated annealing that they were using for finding ground states in spin models could be generalized to solving *arbitrary* optimization problems by introducing a fictitious temperature and associating the cost function with the energy.

Optimization Using Simulated Annealing

For a general optimization problem, the temperature is just a parameter that governs the probability of increasing the cost function at any step, via the usual Metropolis algorithm form $e^{-\Delta C/T}$, where ΔC is the change in the cost function due to a change in the configuration (the parameter values).

Zero temperature corresponds to a *steepest descent* type algorithm, where only changes that do not increase the energy are accepted. Just as for the spin glass, having a non-zero temperature allows the procedure to jump out of local minima.

Simulated annealing works well in practice for many combinatorial optimization problems.

Finding an Optimal Solution

We expect that as long as the temperature is reduced "slowly enough," i.e. in small enough increments, and allowing time for thermalization at each T, then simulated annealing should allow us to reach the ground state (the optimal solution). But what is "slow enough"?

Geman and Geman showed that if the temperature is reduced as

$$T_k = T_0 / \log k$$

for T_0 large enough (so configurations at T_0 are close to random), then we are "statistically" guaranteed to find the optimal value.

This means that if we anneal slowly enough, we have a non-zero probability of obtaining the optimal solution. Thus, if we perform enough different annealings, starting at different random initial configurations, we will eventually obtain an optimal solution.

Advantages of Simulated Annealing

Although Geman & Geman's result may seem like a rather weak statement, guaranteeing a statistically optimal solution for arbitrary problems is more than other optimization techniques can claim.

Simulated annealing:

- can deal with arbitrary systems and cost functions
- statistically guarantees finding an optimal solution
- is relatively easy to code, even for complex problems
- generally gives a "good" solution

This makes annealing an attractive option for optimization problems where heuristic (specialized or problem specific) methods are not available.

Disadvantages of Simulated Annealing

- Repeatedly annealing with a $1/\log k$ schedule is very slow, especially if the cost function is expensive to compute.
- For problems where the energy landscape is smooth, or there are few local minima, SA is overkill — simpler, faster methods (e.g., gradient descent) will work better. But generally don't know what the energy landscape is for a particular problem.
- Heuristic methods, which are problem-specific or take advantage of extra information about the system, will often be better than general methods, although SA is often comparable to heuristics.
- The method cannot tell whether it has found an optimal solution. Some other complimentary method (e.g. branch and bound) is required to do this.

EXAMPLES OF OPTIMIZATION PROBLEMS

Graph Partitioning

In designing computer chips, there are a number of elements to be connected. If there are too many elements to fit on one chip, they need to be partitioned onto (for example) two chips. Since wires between chips are expensive, the number of connections between chips should be minimized. Since silicon area is also expensive, the number of elements on each chip should be about the same.

This is an example of a *graph partitioning* problem. If we consider the elements as vertices and the wires as edges of a graph, the problem is to partition the graph into two equal sets while minimizing the number of edges between each set. This is known to be an NP-hard problem.

This was the application to which simulated annealing was first applied (see the paper by Kirkpatrick et al.)

A Graph Problem Disguised as a Spin Model

Introduce a variable S_i which is +1 if the element is on the first chip, and −1 if it is on the second. Introduce a connectivity matrix $C_{ij} = 1$ if i and j are connected, 0 otherwise.

The penalty for connections between chips is

$$H = -\sum_{ij} C_{ij} S_i S_j$$

We want roughly the same number of elements in chip 1 ($S_i = +1$) and chip 2 ($S_i = -1$), so we want to make $\sum_i S_i$ as close as possible to zero. Can do this by adding a penalty term to the cost function

$$\begin{aligned} H &= -\sum_{ij} C_{ij} S_i S_j + \mu \left(\sum_i S_i \right)^2 \\ &= \text{constant } -\sum_{ii} J_{ij} S_i S_j \qquad \text{where } J_{ij} = C_{ij} - 2\mu \end{aligned}$$

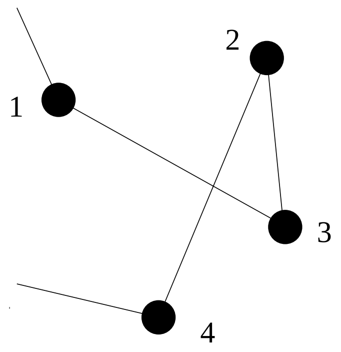
The problem now looks like an Ising spin glass! As we might expect, SA works well for this problem.

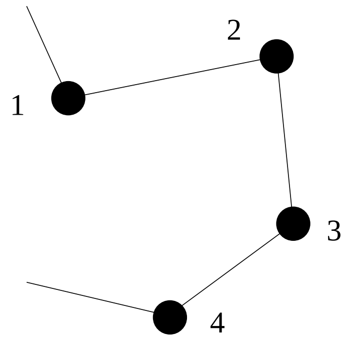
The Traveling Salesman Problem

This classic optimization problem can be very simply stated — a salesman has to visit N cities, and wants to take the shortest possible route. Here, the cost function is the length of the tour.

Changing a configuration (a particular tour) is not as simple as a single spin flip in the spin glass problem, or moving an element from one side to the other in the graph partitioning problem.

Each tour can be presented as a permutation of the numbers 1 to N, which represent the cities. The simplest change to the tour is to swap pairs of cities, and measure the change in the tour path.





IMPLEMENTING ANNEALING ALGORITHMS

The Update Step

For the TSP, the update of the configuration is performed by swapping the order of two adjacent cities in the tour.

There is no reason why we have to swap adjacent cities in the tour — larger, non-local swaps may be more effective at sampling the different configurations, especially at high T. At low T, large moves are not so useful since they will mainly be rejected. The type of move can depend on the temperature.

Choosing effective, ergodic updates is very important for SA. The updates should explore the parameter space as efficiently as possible. As in a Metropolis Monte Carlo simulation, we aim to choose the update so that the acceptance ratio is neither too large nore too small.

The Annealing Schedule

To quantify how fast we perform the annealing, we need to specify:

T_k : the range of temperatures

N_k : the number of Metropolis iterations at each T_k

This is known as the annealing schedule. T_k and N_k are parameters that can be tuned to improve the performance of SA for different problems. Finding a good annealing schedule is key to producing good solutions in a reasonable time.

Often don't need true optimal solution — just a "good" solution, say within 1% of the lowest value. There is a tradeoff between quality of solution and time to find it.

Geometric Annealing

The Geman & Geman result is a bound on the annealing schedule in order to guarantee an optimal solution. For some problems, we can get the same result with a faster annealing schedule. Also, in many cases where we want good but not optimal solutions, faster annealing is more effective.

A popular annealing schedule is one that is exponential (or geometric) rather than logarithmic:

$$T_{k+1} = \alpha \, T_k, \qquad 0 < \alpha < 1 T_k = \alpha^k \, T_0 \; = \; T_0 \, e^{-Ck}, \qquad C = -\log \alpha$$

This is known as geometric annealing (sometimes referred to as simulated quenching). Usually, this gives quite good results as long as α and N_k are chosen appropriately.

See figure tsp.ps.

Results of geometric simulated annealing for the TSP.

Determining the Annealing Schedule

Ideally, one would like a method for determining the annealing schedule based on the behavior of the specific system. We can obtain some guiding principles by considering the statistical mechanics of the problem.

One good rule of thumb is that we would like to minimize the variation from equilibrium. During the annealing, we can measure the energy (cost) E and specific heat (variance of the energy) C_H at each temperature value. These values can be used to help determine the annealing schedule.

Note that since we are constantly changing T, it is very difficult to measure equilibrium values of E and C_H for each temperature.

A "Physical" Annealing Schedule

N_k **:** We would like to choose N_k so thermal equilibrium is reached at T_k before moving to T_{k+1}.

Can check this using standard methods for testing for equilibration (binning). Or alternatively, can use clever techniques for estimating τ_{exp} from measurements of E, and then set $N_k \approx \tau_{\mathrm{exp}}$.

T_k **:** To avoid moving too far from equilibrium, could choose T_{k+1} so that fluctuations in $E\,(T_k)$ overlap those of $E\,(T_{k+1})$.

$$C = \frac{dE}{dT} \approx \frac{\Delta E}{\Delta T}$$

$$T_{k+1} \approx T_k - \frac{\Delta E}{C_k}$$

Parallel Simulated Annealing

SA can be easily parallelized on a coarse grain machine by using independent parallelism, that is, using a different random initial condition and different random number streams on each processor. Each processor will find an independent local minima, and we choose the smallest of these as our best solution. Alternatively, we may want to pass information between processors during the annealing, for example, to replicate "good" configurations.

For large problems, we may just want a good approximate solution to be generated quickly, so we parallelize the problem. For example, we could use domain decomposition for the Ising spin glass, or map a TSP tour onto a ring of processors (see Fox et al. books).

Simulated Tempering

Simulated tempering was invented by Parisi and Marinari to study disordered Ising spin models (spins in a random external magnetic field). These models have a first order phase transition, so there are two coexisting states at T_c — a high energy and low energy state. The standard Metropolis algorithm tends to get stuck in one of these states.

Tempering does a Monte Carlo update of the *temperature* — i.e. try to change T, and do a Metropolis accept/reject depending on $\Delta(E/kT)$.

Constantly changing T in a range slightly above and below T_c, moves the configuration in and out of high and low E states, thus providing a correct sampling of the configurations at T_c.

The repeated heating and cooling is like tempering in metals.

Tempering for Optimization

One of the main problems with simulated annealing is that it is difficult to find an optimal annealing schedule, and that changing T drives the system out of equilibrium.

If we use tempering instead, the system is always in equilibrium. Also, we can determine the temperature changes (the equivalent of the annealing schedule) automatically, by varying the ΔTs so that the acceptance of a change is $\sim 50\%$.

A lot of work still needs to be done in order to perfect his method. ΔTs may be too small, a random walk in T may not be optimal — this all has yet to be studied in detail.