Revista Brasileira de Física, Vol. 18, nº 4, 1988

Fast Code for Monte Carlo Simulations

PAULO MURILO CASTRO DE OLIVEIRA and THADEU JOSINO PEREIRA PENNA

Instituto de Física, Univefsidade Federal Fluminense, Caixa Postal 100296, Niterói, 24000, RJ, Brasil

Recebido em 13 de Junho de 1988

Abstract We present a computer code to generate the dynamic evolution of the Ising model on a square lattice. following the Metropolis algorithm. The computer time consumption is reduced by a factor of 8 when one compares our code with traditional multiple spin codes. The memory allocation size is also reduced by a factor of 4. The code is easily generalizable for other lattices and models.

Monte Carlo simulations are important tools for studying statistical models (see ref.1). Modern applications of the method are widespread over an extensive range of fields. Enhancing the performance of these applications is a challenge for physicists and other scientists. particularly as microcomputer are ever more present in the scene. In this context. the demand for higher processing speeds and lower memory allocation requisites is crucial.

Most modern applications of Monte Carlo simulations use a *multiple spin* code (see, for instance. ref.2). in which the bits of integer type variables are used for storing the current state of the system. In the Ising model. for example. each site of lattice holds a spin value 0 or 1, and the local energy due to the interaction of this site with its neighbours ean assume integer values between 0 and the lattice coordination number, considering 0 (1) energy per parallel (antiparallel) neighbour-spin-pair. For the square or cubic lattice, the complete state of each site requires four bits, one for the spin value itself and three others for the local energy. A 32-bit processor can hold 8 spins per word, and the advantage of this storing strategy is the simultaneous update of these 8 spins by a single bitwise operation. The local energy value is needed to implement the Metropolis[3] algorithm: updating of a given spin is decided by comparing a random number with a suitable Boltzmann factor previously calculated from this energy. The calculation of the local energy is also done simultaneously for the 8 spins of a word.

In this short paper we present a computer code which improves the performance of the multiple spin code strategy beyond its usual limits. As we show

Financial support from FINEP and CNPq (Brazilian Government Agencies)

below, this procedure is especially suited for implementation in microcomputer units. owing to its simplicity and memory-saving characteristics. We use only one bit per spin. yielding a ratio of 32 spins per computer word. The local energy value is calculated simultaneously for the 32 spins. using bitwise operations. with no need of storing it. We tested our code against the traditional multiple spin code. obtaining a reduction by a factor 8 in the computer time consumption and a factor 4 in the memory size allocation. for the square lattice Ising model. As the code can be easily modified for other lattices and other discrete spin models. hereafter we will treat only this simple case.

We divide the square latice into four sublattices a. b. c and d, and store the whole spin configuration in four one-dimensional vectors corresponding to the lattices, as defined in ref.4 (only with 32-bit words instead of the 64-bits available on the Cray XMP used in that reference). The even (odd) rows are stored in vectors a and b (c and d) and even (odd) columns stored in vectors b and c (a and d). as shows in table 1. Periodic boundary conditions are adopted in both directions. taking advantage of the bit-shift operators.

Table 1 - Vector storing arrangement for the L x L square lattice Ising model. where $L = 64 \times m$ with integer m. Vectors a, b. c and d contains $mn = L^2/128$ 32-bit-integers each. The upper symbol ^{b} rneans the b^{th} bit of the corresponding word, the bracket $[j]$is the vector index, and $m2 = mn - m + 1$.

$^0d[1]$	$^0c[1]$	...	$^0d[m]$	$^0c[m]$	...	$^{31}d[1]$	$^{31}c[1]$	...	$^{31}d[m]$	$^{31}c[m]$
$^0a[1]$	$^0b[1]$	...	$^0a[m]$	$^0b[m]$	...	$^{31}a[1]$	$^{31}b[1]$	...	$^{31}a[m]$	$^{31}b[m]$
...	...		...	...		...	...		...	...
$^0d[m2]$	$^0c[m2]$	...	$^0d[mn]$	$^0c[mn]$	...	$^{31}d[m2]$	$^{31}c[m2]$	...	$^{31}d[mn]$	$^{31}c[mn]$
$^0a[m2]$	$^0b[m2]$	...	$^0a[mn]$	$^0b[mn]$	...	$^{31}a[m2]$	$^{31}b[m2]$	...	$^{31}a[mn]$	$^{31}b[mn]$

Our code is written in C-Programming-Language[S]. Declarations. definitions and the code for the whole d sublattice updating are given below . Translated to Fortran syntax code. symbols ~, —, & and A correspond respectively to NOT. OR and XOR logical bitwise operators. while A << B means the shift of the word A by B bits to the left, and >> means the same operation to the right. The sequence A = E ? X : Y is a conditional assignment operator that stores X or Y into A if E is TRUE or FALSE respectively. A symbol like x OP = y means x = x OP y, where OP is binary operator. Texts between /* and */ are comments. The variable JkT holds the coupling constant value J/kT, while m holds L/64 for a LxL lattice.

```
          /* declarations : */
const unsigned long bit31 = 1 << 31;
unsigned long r,i1,i2,i3,i4,bit[32];
unsigned long ex0,ex1,e0,e1,t,a[3201],b[3201],c[3201],d[3201]
unsigned i,j,k,ib,m,mm,mn,m1,m2,ndx;
double JkT;

          /* defined quantities : */
mm = m - 1; n = 32 * m; mn = m * n; m1 = mn - m; m2 = m1 + 1;
exl = 4294967295 * exp(-4 * JkT);
ex0 = 4294967295 * exp(-8 * JkT);
for (ib= 0; ib < 32; ib + +)bit[ib] = 1L << ib;

          /* test and update for sublattice d : */
j = 1; i1 = a[m2];
for  (i= 1; i <= n; i+ +){
  i2 = i1; i1 = a[j]; i3 = c[j];
  ndx = j + mm; i4 = c[ndx]&bit[31]?(c[ndx] << 1)|1 : c[ndx] << 1;
  e0 = (i1&i2&i3&i4&d[j])|(~ ilk ~ i2& ~ i3k ~ i48 ~ d[j]);

     /* e0 holds l-bits at sites with 0 local energy */
  e1 = (i1 ^ i2 ^ i3 A i4)& ~ (d[j] ^ ((i1&i2)|(i3&i4)));

     /* e1 holds 1-bits at sites with 1 local energy */
  t =~ (e0|e1);

     /* t holds l-bits at sites with 2, 3 or 4 local energy */
  for (ib =; ib < 32; ib + +){
  if(e0&1){r+ = (r << 1) + (r << 16); if(r < ex0)t| = bit[ib]; }
  else if (e1&1){r+ = (r << 1) + (r << 16); if(r < exl)t| = bit[ib]; }}

     /* r is a random odd integer number */
  e0 >>= 1; e1 >>= 1;
 }      /* now. t holds 1-bits at all sites that must be spin-flipped */
 d[j]^ = t;
 j+ = m;
}
if(m! = 1){
 for (k = 2; k <= m; k + +){
  j = k; i1 = a[m1 + k];
  for (i = 1; i <= n; i + +){
   i2 = i1; i1 = a[j]; i3 = c[j]; i4 = c[j - 1];
   e0 = (i1&i2&i3&d[j])|(~ i1& ~ i2& ~ i3& ~ i4& ~ d[j]);
   e1 = (i1 ^ i2 Ai3 ^ i4)& ~ (d[j] ^ ((i1&i2)|(i3&i4)));
   t =~ (e0|e1);
   for(ib = 0; ib < 32; ib + +){
    if(e0&1){r+ = (r << 1) = (r << 16); if(r < e0)t| = bit[ib]; }
    else{if(e1&1){r+ = (r << 1) + (r << 16); if(r < ex1)t| = bit[ib]; }}
    e0 >>= 1; e1 >>= 1;
   }
   d[j]^ = t;
   j+ = m;
  }
 }
}
```

The complete code was implemented for a *64x64* square lattice on an IBM-PC based on an INTEL 8088 processor. running at 4.77 MHz. The updating velocity was 4.4 x 10^3 spins/s, including the time needed to average the magnetization and susceptibility. On another INTEL 80286-processor-based PC, running at 12 Mhz, the updating velocity was 2.0 x 10^4 spins/s. Both processors only deal with 16-bit words. and the 32-bit processing is an emulation supported by the C compiler. The time consumption on a real 32 (or 64)-bit processor will be even smaller. Particularly important for the IBM-PC line, the segmentation in 64-K memory blocks imposes serious restrictions upon the size of arrays: our approach is also suitable to overcome this problem. Results are shown in fig.1, and are quite satisfactory. considering the reduced amount of time and memory used in their obtention. For comparison, the same results were firstly obtained in ref.7, with equivalent performance, on main frame computers.

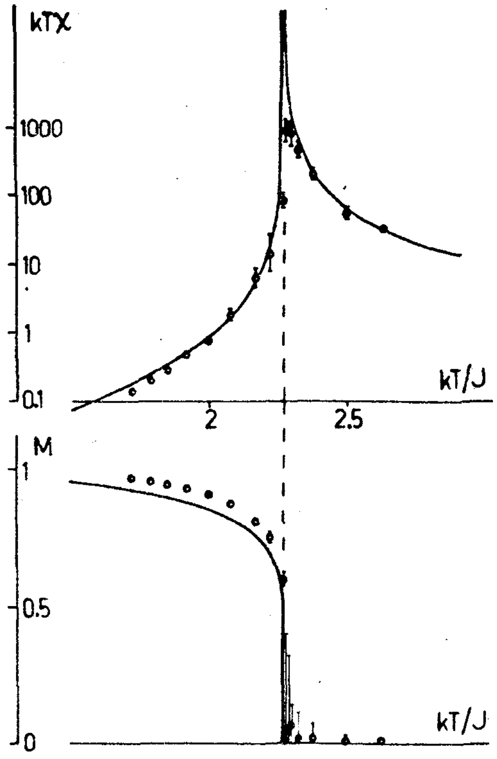

Fig.1 - Magnetization and susceptibility graphs for a 64x64 lattice. Full lines are the known exact results[6]. Each point was obtained from 10 sampies with 3000 complete Monte Carlo steps each (near the critical temperature, the full dots mean that 12000 steps were taken).

In short, we have developed a computer code for Monte Carlo simulations, whose performance is faster and more memory-saving than the usual multiple spin code. Owing to these characteristics. the code is especially suited for implementation in micro-computers. Generalizations for other spin models and lattices than this Ising model on a square lattice are straightforward; work along these lines is in progress.

We wish to thank S.L.A. de Queiroz for a critical reading of the manuscript. One of us (PMCO) is also grateful to H. Manela who introduced him to the C language.

REFERENCES

1 K.Binder (ed). *Applications of the Monte Carlo in Statistical Physics,* 2nd edition, Springer Verlag. Heildelberg 1987.

2 R. Zorn, H.J. Herrmann and C. Rebbi. Comp. Phys. Comm. 23, 337 (1981): C. Kalle and V. Winkelmann, J.Stat.Phys. *28,* 639 (1982).

3 N. Metropolis, A.W. Rosenbluth. M.N. Rosenbluth. A.H. Teller and E. Teller. J.Chem.Phys. 21. 1087 (1953).

4 H.J. Herrmann. J.Stat.Phys. 45, 145 (1986).

5 B.W. Kerninghan and D.M. Richie. *The C Programming Language.* Prentice Hall, New Jersey 1978.

6 L. Onsager. Phys.Rev. 65.117 (1944): C.N. Yang, Phys.Rev. 85808 (1952): E. Barouch. B.M. Mc Coy and T.T. Wu, Phys. Rev.Lett. 31. 1409 (1973).

7 D.P. Landau,Phys.Rev. B*13*, 2997 (1976).

Resumo

Apresenta-se um código de computador para gerar a evolução dinâmica do modelo de Ising numa rede quadrada, segundo o algorítimo de Metropolis. O tempo de computação C reduzido por um fator 8 em relação ao código tradicional de spin múltiplo. A memória necessária também C reduzida por um fator 4. O código C facilmente generalizável para outras redes e outros modelos.