Parallel simulation of the Ising model

G. T. Barkema and T. MacFarland
Laboratory of Atomic and Solid State Physics, Clark Hall, Cornell University, Ithaca, New York 14853-2501
(Received 7 March 1994)

Methods for parallelizing Ising model simulations are presented. A parallel single-spin Metropolis algorithm [J. Chem. Phys. **21**, 1087 (1953)] has been implemented with a speedup of 27 on 50 processors of the KSR-1 parallel computer. A parallel Swendsen-Wang algorithm [Phys. Rev. Lett. **58**, 86 (1987)] obtains a speedup of 3.2 on nine processors of the same computer. Both of these simulations were carried out on 200×200 lattices. The parallel local cluster algorithm [Phys. Rev. Lett. **71**, 2070 (1993)] has been implemented with an almost linear speedup. We also discuss ongoing research using the parallel local cluster algorithm.

PACS number(s): 02.70.−c, 05.50.+q, 64.60.Cn, 75.10.Hk

I. INTRODUCTION

In this paper we explore the parallelization of three algorithms that are commonly used for the simulation of the Ising model: the single-spin Metropolis algorithm [1], the Swendsen-Wang algorithm [2], and the local cluster algorithm [3]. The Metropolis algorithm has been used extensively to study both equilibrium properties and nonequilibrium properties of the Ising model and many similar models (see Ref. [4] and references therein). It suffers however from *critical slowing down*: near the critical temperature, correlation times scale with a power of the system size. This problem has largely been solved with the introduction of the Swendsen-Wang algorithm and the related Wolff algorithm [5], although application of these algorithms is limited to the study of equilibrium properties of a restricted class of Ising models. Recently, the local cluster algorithm was introduced. This algorithm combines the computational efficiency of the Wolff algorithm with a wider applicability.

Faster simulation methods for the Ising model have many applications. The motivation for this research is the study of the Ising model in a porous medium for varying strengths of interaction between the two components and the pore walls, as well as phase separation in porous media. The substantially different length scales appearing in these systems necessitate large-scale computation. The lattice size must be substantially larger than the pore size, which in turn must be large compared to the lattice spacing. Since we would like to study pore sizes of from 4 to 40 lattice constants our systems will consist of up to 256^3 lattice sites. Without efficient simulation methods on parallel computers, this research is impossible because of the large computational demands imposed by the system size.

Our simulations were carried out on a KSR-1 parallel computer. The KSR-1 that we use consists of 128 processors, each comparable in speed to a fast workstation. The processors are partitioned into four groups of 32 processors. Processors within one such group are connected by a zero level communication ring. These zero level rings are themselves connected by a level 1 ring. Each processor has 32 megabytes of memory. Memory management is handled by the hardware and low level operating system, so that the user accesses the local memory as if the machine has one shared memory. We used the shared memory of this machine for communication [6], but the algorithms are also suited for parallel computers without global memory.

In this paper, for simplicity we have applied our algorithms to the simulation of the two-dimensional Ising model. Each of them can be readily extended to higher dimensionality.

II. A STRAIGHTFORWARD PARALLEL METROPOLIS ALGORITHM

In the single-spin Metropolis algorithm, a lattice site α is selected, and the proposed move is to flip σ_α, the spin located on this site. If the total energy of the configuration will decrease by this move, the move will always be accepted. If the energy increases, the probability that the proposed move will be accepted is given by

$$A_{A\to B} = \min(1, \exp[H(A) - H(B)]), \tag{1}$$

in which A is the initial configuration, and B is the configuration after execution of the proposed move. H is the Hamiltonian

$$H/kT = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j, \tag{2}$$

in which J is the coupling constant, and the summation runs over all nearest-neighbor pairs (links). If the move is accepted the new configuration is B, otherwise it is A. For a parallel implementation of the Metropolis algorithm, the lattice can be divided into domains. Conventionally, periodic boundary conditions are imposed. In the following section, a division of the lattice into slices

 ©1994 The American Physical Society

is assumed, although the generalization to other divisions of the lattice is straightforward. Each slice has periodic boundary conditions in one direction, and in the other direction is connected to neighboring slices. The slices are organized in a ring structure, which preserves the periodic boundary conditions of the original lattice.

Each processor selects sites in its domain and accepts or rejects spin flips on these lattice sites. It is desirable to select the site on which a spin flip is proposed in a random way, as this avoids the introduction of an anomalous dynamical behavior and reduces biases due to the *pseudorandom* nature of the random number stream [7]. A straightforward parallel implementation of this algorithm introduces difficulties. A conflict arises if two different processors select adjacent lattice sites α and β at the same time. A straightforward implementation might cause the spin values at sites α and β to be updated, both based on the old spin values at sites α and β. Thus a "combined" step is achieved, that unfortunately does not fulfill detailed balance and yields biased results. Circumventing these "combined" steps by starting a spin flip by locking the spin value of a selected site and releasing it after it is updated might result in a deadlock. The correct way to deal with this situation is to handle the spin flips sequentially in a random order. This requires additional communication, which slows down the algorithm.

Such situations are simply avoided by using the same sequence of random sites on each processor. Processors then select sites that are periodic images of each other, and which therefore are never adjacent. This selection method causes no bias in equilibrium properties. Although the sites are not selected strictly at random, we found no bias in the dynamical behavior, and expect this to be the case whenever the domains are not extremely small.

One side effect of using the same sequence of random numbers on each processor is that, if the generation of the random numbers for site selection is itself parallelized, a *superlinear* speedup will be achieved: the time for generating these random numbers is proportional to $1/H$ and L^2/H random numbers have to be generated for one Monte Carlo (MC) step per site. Here H is the number of domains. Thus the (wall-clock) time required for a small part of the code, the random number generation for the site selection, scales as L^2/H^2.

On a KSR-1, we carried out simulations of a system with 200×200 sites at critical temperature ($J = 0.44068$), for a varying number of processors. In a sequential run, one additional attempted spin flip per site takes 0.803 s, which corresponds to a time per spin flip of about 20 μs. The circles in Fig. 1 give the measured speedup compared to this timing result, as a function of the number of processors. The speedup increases to a maximum of about 17 for 40 processors and then begins to decrease.

For one MC step per site in the parallel implementation, each processor has to make L^2/H spin flip attempts. In our parallelization by domain decomposition the memory requirement per processor decreases as the number of processors increases, resulting in a higher cache and sub-

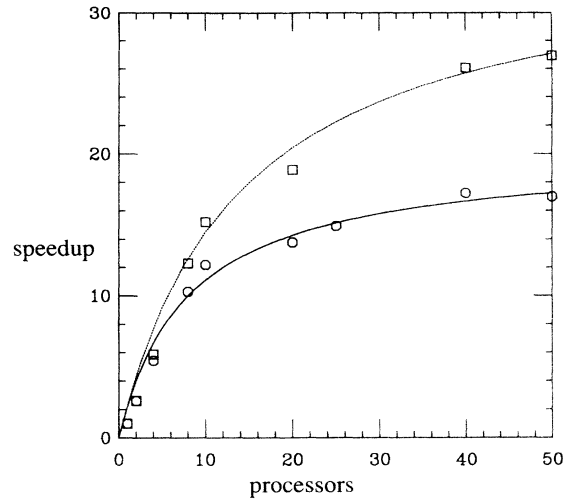

FIG. 1. Speedup of the Metropolis algorithm as a function of the number of processors; the circles denote the straightforward parallel Metropolis algorithm, while squares denote the "smart" version of that algorithm. The smooth curves are our scaling predictions for the speedup.

cache hit ratio. Because of this, the time per spin flip in the parallel implementation decreases with the number of processors to about 8 μs on four or more processors.

Synchronization must take place if a border point is selected. If B sites are located on the border of each domain, statistically we have to synchronize B times for one MC step per site. In our implementation we divide the lattice in one direction; thus $B = 2L = 400$. The speedup S is given by

$$S = \frac{L^2 \times 20\ (\mu\text{s})}{(L^2/H) \times 8\ (\mu\text{s}) + B\tau_{sync}}, \tag{3}$$

in which τ_{sync} is the time required for synchronization. We get a good fit to the data if we take $\tau_{sync} = 100\ \mu$s, resulting in the solid line in Fig. 1. The KSR-1 that we used is configured so that, if one requires less than 16 processors, one can allocate a set of processors which occupy the same communication ring. When 16 or more processors are used, processors may not share the same communication ring, resulting in slower communication. This is reflected in our measurements.

III. A SMARTER PARALLEL METROPOLIS ALGORITHM

The number of times that processor synchronization has to take place can be limited substantially by the following algorithm. Each processor oversees its own domain, and additionally has copies of the spin values on the adjacent border strip of the neighboring domains. As before, all servers generate identical random numbers that determine the domain sites on which a spin flip is proposed. If an internal site is selected, the server can

accept or reject the flip and no communication is necessary. After a border site is selected the copy of that site residing on the adjacent neighboring processor may be out of date. Even in the absence of communication, a processor can keep track of which copies of border spin sites are still up to date, because of the synchronized site selection. As long as the information required for accepting or rejecting a move is up to date, no synchronization need take place. Once communication can no longer be postponed, synchronization between all processors takes place, after which all copies of neighboring border spins are up to date. Keeping track of these copies reduces the frequency with which synchronization must take place, resulting in a higher speedup.

As an illustration, in Fig. 2 each process selects the lattice sites 1 through 9 sequentially. When lattice site number 4 is selected, its spin value is updated based on the copy of the spin value of its neighbor across the domain border and the field denoted by "4" that resides on the same process is marked to be out of date. Up to now, no communication has been necessary. At the moment that lattice site 9 is selected, information is required that may not be current, since one of the neighbors of lattice site 9 may have been changed (because the spin value at site "4" might be changed). Synchronization takes place at this point, after which copies of all border sites are current. This reduces the number of times that we have to synchronize during one MC step per site to $\sqrt{2B/\pi}$ [8], instead of B.

We have also implemented this "smart" Metropolis algorithm on the KSR-1, and measured the speedup compared to the same sequential implementation. The results are plotted in Fig. 1 as squares. The speedup S is now given by

$$S = \frac{L^2 \times 20\ (\mu\text{s})}{(L^2/H) \times 8\ (\mu\text{s}) + \sqrt{2B/\pi}\tau_{sync}}. \tag{4}$$

We get a good fit to the data if we take $\tau_{sync} = 1.2$ ms, resulting in the dotted line in Fig. 1. We observe that a synchronization takes longer in the "smart" algorithm,

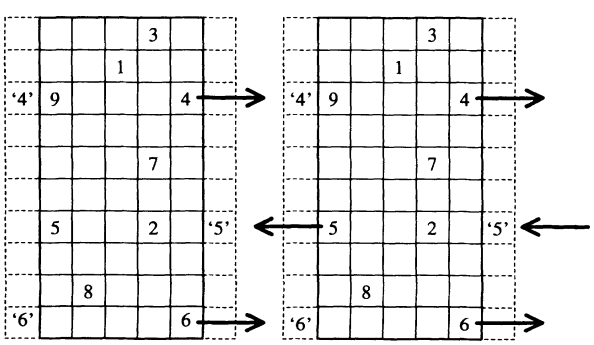

FIG. 2. Illustration of the "smart" parallel Metropolis algorithm. The dotted sites indicate copies of spin values located on adjacent domains. Sites 1–9 are selected sequentially. When site 9 is selected, communication must take place, since the copy of the spin value at the site marked "4" may have changed.

which we think is due to the imperfect load balancing over the larger time interval between synchronizations. On a large number of processors, the "smart" Metropolis algorithm outperforms the simple Metropolis algorithm by about a factor of 1.6.

Other machine-specific schemes have been mentioned in the literature over the past several years, for the ETA-10 [9], the M64/60 [10], the Cray YMP [11], the Connection Machine [12], and for the CDC CYBER 205 [13]. Also, special purpose computers have been designed for simulation of the Ising model with the Metropolis algorithm [14,15].

IV. A PARALLEL SWENDSEN-WANG ALGORITHM

The single-spin Metropolis algorithm is highly local. Therefore, near the critical temperature where the correlation length is of the same order as the size of the system, it can take a prohibitively long time to obtain uncorrelated configurations. Unfortunately, it is near this temperature that the behavior of the system is most interesting. If the dynamical behavior of the system is not of interest, uncorrelated configurations can be generated much more efficiently with methods in which elementary moves consist of flipping groups of spins, or *clusters*. The first widely used cluster algorithm was proposed by Swendsen and Wang in 1986 [2]. In this algorithm, all sites are grouped into clusters to which new spin values are assigned. We present a parallel version of this algorithm.

One lattice sweep in the sequential Swendsen-Wang algorithm consists of three stages.

(1) Visit all links connecting nearest-neighbor spins σ_i and σ_j. If $\sigma_i = \sigma_j$, with probability $P_{act} = 1 - \exp(-2J)$ the link is activated. Activated links are also called *bonds*.

(2) Construct clusters of lattice points. Two points belong to a cluster if and only if they are connected by a path of activated links.

(3) For each cluster, pick a random spin value and assign that to all the spins of the cluster.

The proof that this algorithm satisfies detailed balance and ergodicity can be found in [2].

Mino [16] proposed a vectorized implementation of the Swendsen-Wang algorithm based on the storage of clusters in a tree structure. Since this approach is better suited for vectorization than for parallelization, we propose a solution along the lines set out in the previous section.

In our parallelization of the Swendsen-Wang algorithm, the lattice is again partitioned into domains, each of which is allocated to a different processor. Each processor is responsible for constructing clusters and assigning spins within its domain. Under this scenario, conflicts between the processors could arise from the fact that clusters may cross borders between domains. In this case one processor could start assigning a spin value to its side of a cluster, while a different processor could assign a different spin value to another part of the same cluster. To avoid these conflicts, a master processor assigns spin values to

clusters containing border sites ("border" clusters). The assignment of spins to clusters entirely contained within the interior of one domain ("local" clusters) can be independently made by the processor to which the domain is allocated.

In our parallel Swendsen-Wang algorithm, the task of a processor to which a domain is assigned consists of the following steps.

(1) Visit all links connecting nearest-neighbor spins σ_i and σ_j, where sites i and j are within the processor's domain. If $\sigma_i = \sigma_j$, with probability $P_{act} = 1-\exp(-2J)$ the link is activated.

(2) Construct all clusters consisting of lattice sites connected by links activated in the previous step. A distinction is made between local clusters and border clusters. All border clusters are numbered. The border cluster number of each border site is stored in an array. This array is copied into global memory, and a synchronization counter is incremented.

(3a) For each local cluster, pick a random spin value and assign it to its sites.

(3b) When the master processor has completed the assignment of spins to the border clusters, assign these spin values to the sites within these clusters.

The task of the master processor for one lattice sweep consists of the following steps.

(m_1) With probability P_{act}, activate the link between aligned spins located on the borders of adjacent domains.

(m_2) After receiving the arrays of cluster numbers of the border sites, group the border clusters that are connected by links activated in the previous step into global clusters.

(m_3) Assign spin values to all global clusters (and thus to all border clusters), and increment a global synchronization counter.

Let us define the time required for step i by $\tau(i)$. The total time of one sweep τ_{par} is then approximately given by

$$\tau_{par} = \tau_{sync} + \max\left[\tau(m_1), \tau(1) + \tau(2) + \tau(3b)\right] + \max\left[\tau(3a), \tau(m_2) + \tau(m_3)\right]. \quad (5)$$

As a consequence of the way in which we divide the lattice, all times required by the master processor are proportional to the number of border sites $BH = 2LH$. All times required by the other processors are approximately proportional to the number of points in the domain L^2/H. Empirically we determined prefactors for these times, resulting in $\tau(m_2) + \tau(m_3) = 2LH \times 37$ μs, $\tau(1) + \tau(2) + \tau(3b) = \frac{L^2}{H} \times 22$ μs, $\tau(m_1)$ and $\tau(3a)$ are small, and $\tau_{sync} = 30$ ms.

One sweep in a sequential implementation for a 200×200 lattice at critical temperature on a KSR-1 takes $\tau_{seq} = 0.87$ s. We measured the speedup of a parallel implementation with respect to this timing result; the results are plotted in Fig. 3 as circles. In the same figure, we plotted a theoretical estimate of the S, given by $S = \tau_{seq}/\tau_{par}$, in which we used the estimated times mentioned above. The maximum speedup that we obtained was about 3.2, on nine processors (one master processor and eight other processors). The rather low number

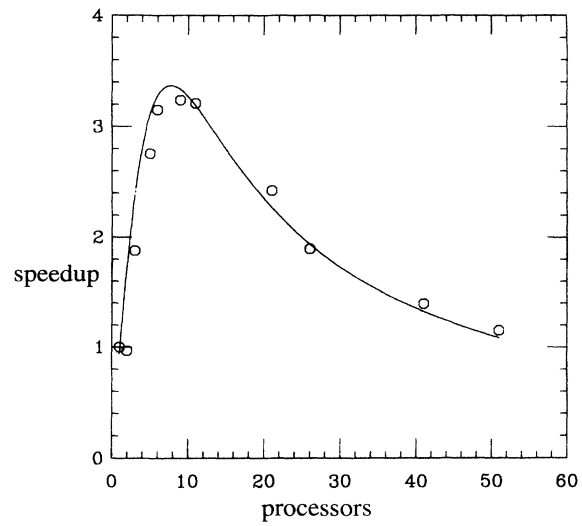

FIG. 3. Speedup of the parallel Swendsen-Wang algorithm as a function of the number of processors. The circles indicate the measured speedup. The smooth curve is our scaling prediction for the speedup.

of processors that can be used efficiently arises from the serial nature of the work carried out by the master processor. When the number of processors increases, the workload of the master increases due to the increasing number of borders in the system, whereas the workload of the other processors decreases. To use a large number of processors efficiently, the task of the master processor can be parallelized.

Wolff [5] developed a single cluster modification of the Swendsen-Wang algorithm. According to his algorithm, first a lattice point is picked at random. Only the cluster to which this point belongs is constructed. Finally, as in the Swendsen-Wang algorithm, a new spin value is assigned to all the spins of this cluster. This modification was found to be slightly superior to the conventional Swendsen-Wang method, especially in higher dimensions, due to the fact that on the average larger clusters are treated. As the difference in efficiency between the Wolff algorithm and Swendsen-Wang algorithm is small, a parallel implementation of the latter may still outperform the former.

An approach to parallelization of the Swendsen-Wang algorithm, similar to the one we propose here, has been published by Heermann and Burkitt [17]. Their approach is not based on the existence of one master process, and thus avoids the bottleneck that our scheme has for a large number of processors. Rather than investigating more complicated schemes along this line, we pursued parallelization of the local cluster algorithm (as described in the next section).

V. A PARALLEL LOCAL CLUSTER ALGORITHM

The recently proposed local cluster (LC) algorithm [3] can be implemented in a way that conserves the high

efficiency of the Wolff algorithm but, in contrast to that algorithm, is very suitable for parallel processing. In this section, we first describe the particular implementation of the LC algorithm that we used in our parallel program. Next, we present timings for this algorithm. Finally, we discuss differences in the dynamics of the Wolff and the LC algorithms.

A general description of the LC algorithm can be found in Ref. [3]; in this section we describe only the specific implementation for which we obtained timings. Parallelism is obtained by domain decomposition: the $L \times L$ lattice is divided by horizontal and vertical gridlines at a regular spacing into domains of size $(L/\sqrt{H}) \times (L/\sqrt{H})$, where H is the number of processors. Each processor makes cluster moves that are limited to the interior of its domain; this requires no knowledge of the spin configurations on other processes. One LC cluster move consists of the following steps.

(1) Each processor selects at random a site r, located in the interior of its domain, to be the first spin in the cluster C.

(2) Consider the links connecting $x \in C$ to its neighbors $y \notin C$, that have not been considered before; these links are activated with probability $\delta(\sigma_x, \sigma_y)(1-\exp^{-2J})$, in which case site y is added to C.

(3) Repeat step (2) until all links from sites in C have been considered, and flip the spins of all sites in C. However, if at any moment a site is added to C that is located on the border of the domain, cease the construction of C and restore the original spins to all sites in C.

To satisfy ergodicity, the gridlines that divide the lattice into domains are shifted periodically. The time required for one local cluster move is roughly proportional to the number of sites that are covered by the generated cluster (coverage). In order to avoid a large load imbalance, each processor runs up to a certain cumulative coverage, after which the processor copies the configuration in its domain to global memory, the gridlines are shifted, and the processor copies the configuration of its new domain into local memory, etc.

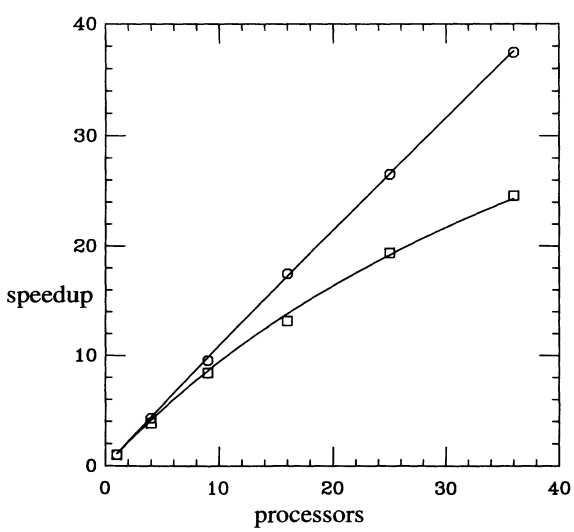

FIG. 4. Speedup of the parallel local cluster algorithm as a function of the number of processors. The circles indicate the measured speedup for a CCPS of 10. Squares indicate the same for a CCPS of 1. The smooth curves are our scaling prediction for the speedup.

We carried out simulations on a 120×120 lattice, with H =1, 4, 9, 16, 25, and 36, for communication after cumulative coverages per site (CCPS) of $\theta = 1$ and $\theta = 10$. Both the equilibration and correlation times are roughly proportional to the wall-clock time required to obtain a certain CCPS; we define the speed v of the simulation as the CCPS per second. In Fig. 4 we plot the speedup of the simulation as a function of the number of processors H, for the case in which we communicate after a CCPS of 1 (circles) and 10 (squares).

In a sequential program, the time elapsed between two synchronization points is equal to $\tau = \theta\alpha L^2$; we measured $\alpha_{seq} = 14$ μs for the sequential case. Because of a higher cache hit ratio when domain sizes are smaller, α_{par} is smaller by about 10%. In a parallel program, the communication time is in approximation described by a constant plus the amount of data to be communicated divided by the bandwidth: $\tau = \tau_0 + (L^2/H)/w$. Thus, the speedup S is given by

$$S = \frac{\alpha_{seq}\theta L^2}{\alpha_{par}\theta L^2/H + \tau_0 + (L^2/H)/w}. \tag{6}$$

The data are fitted well if we assume $\tau_0 = 9$ ms and w large; assuming these values, Eq. (5) is plotted in Fig. 4 as solid lines for a CCPS of 1 and 10.

Moving the gridlines takes time, and hence should not be done more often than necessary. If the CCPS before the gridlines are moved is too large, the interior of the domains will be equilibrated long before the border sites (that equilibrate only after several gridline moves); this leads to a long tail in the autocorrelation function of the quantities observed and thus to inefficiency. Typically, this problem does not occur if the CCPS before the gridlines are moved is 1 or smaller; in these cases, many sites would not have been visited in the sequential algorithm either. In practice, this long tail becomes particularly noticeable above a CCPS of 10. Therefore, we do not present timings for this regime.

VI. CONCLUSIONS

Methods are presented for parallel and distributed processing of the Ising model. The Metropolis algorithm with random site selection can be implemented on the KSR-1 with a speedup of up to 27 on 50 processors for a 200×200 lattice. A parallel implementation of the Swendsen-Wang algorithm is presented that obtains a speedup of a factor of 3.2 on nine processors of the same computer. The local cluster algorithm can be implemented with an almost linear speedup, and is for this reason more efficient on a parallel computer than the Wolff and the Swendsen-Wang algorithms.

ACKNOWLEDGMENTS

John Marko and Geoffrey Chester are gratefully acknowledged for their help with this research. The work of G.T.B. was supported by the National Science Foundation under Contracts Nos. ASC-9310244 and DMR-91-21654 through the MRL program, the Materials Science Center, and the Cornell National Supercomputer Facility. T.M. is supported by a Department of Education Graduate Grant No. P200A10148-93. This research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation, and New York State. Additional funding comes from the Advanced Research Projects Agency, the National Institutes of Health, IBM Corporation, and other members of the center's Corporate Research Institute.

[1] N. Metropolis *et al.*, J. Chem. Phys. **21**, 1087 (1953).
[2] R.H. Swendsen and J.S. Wang, Phys. Rev. Lett. **58**, 86 (1987).
[3] G.T. Barkema and J.F. Marko, Phys. Rev. Lett. **71**, 2070 (1993).
[4] *Applications of the Monte Carlo Method in Statistical Physics*, edited by K. Binder, 2nd ed. (Springer-Verlag, New York, 1987).
[5] U. Wolff, Phys. Rev. Lett. **62**, 361 (1989).
[6] In our programs, variables are declared as either globally shared between processors or local to each processor. All communication between processors takes place through the globally shared variables. If a processor writes to global memory, it increments a counter. If a processor requires data from global memory, it waits until the appropriate counter has been increased, ensuring synchronization between processors. The propagation of the contents of these global variables between processors is carried out by the operating system and by the underlying hardware.
[7] A. Compagner and A. Hoogland, J. Comput. Phys. **71**, 391 (1987).
[8] G.T. Barkema, Ph.D. thesis, Utrecht University, 1992.
[9] D.W. Duke, R. Salvador, and D. Sandee, in *Supercomputing, Vol. II: Science and Applications* (IEEE Computer Society Press, Los Alamitos, CA, 1989).
[10] K.J.M. Moriarty and J. von Neumann, Int. J. High Speed Comput. **1**, 505 (1989).
[11] H.-O. Heuer, Comput. Phys. Commun. **59**, 387 (1990).
[12] J.G. Amar and F. Sullivan, Comput. Phys. Commun. **55**, 287 (1989).
[13] M.Q. Zhang, J. Stat. Phys. **56**, 939 (1990).
[14] G.R. Aiello, M. Budinich, and E. Milotti, Comput. Phys. Commun. **56**, 141 (1989).
[15] H.W.J. Blöte, Int. J. Mod. Phys. C **2**, 246 (1991).
[16] H. Mino, Comput. Phys. Commun. **66**, 25 (1990).
[17] D.W. Heermann and A.N. Burkitt, Parallel Comput. **13**, 345 (1990).