JOURNAL OF COMPUTATIONAL PHYSICS 76, 426–447 (1988)

Multitasking Case Study on the Cray-2: The Q2R Cellular Automaton

JOHN G. ZABOLITZKY*

Minnesota Supercomputer Institute and School of Physics and Astronomy, University of Minnesota, Minneapolis, Minnesota

AND

HANS J. HERRMANN

Service de Physique Theorique, CEN-Saclay, France

Received January 23, 1987; revised June 17, 1987

The Q2R cellular automaton which may be used for microcanonical simulation of the Ising model is implemented in parallel on the four processors of a Cray-2 supercomputer. The simulation speed is 4.3 GHz as opposed to 670 MHz for the previously fastest reported implementation using one processor of a Cray–XMP. We also simulated the largest Ising system ever studied, 15, 130, 968, 192 spins. A number of subtleties in implementing and optimizing the parallel algorithm are discussed, as well as problems in defining performance measurements. Performance and results from the Q2R algorithm are compared to those from the standard Metropolis algorithm, and a number of dynamic exponents are measured for the first time.

I. INTRODUCTION

The Cray Research, Inc. Cray-2 supercomputer is one of the most powerful machines available today. It offers unique features in its large 268 Mword shared (common) memory. In order to solve very large problems which need to use all of this memory it is expedient to multitask the problem, i.e., have all four processors work concurrently on the same problem. The specific memory architecture of the Cray-2 poses a number of difficulties making any optimal or near-optimal implementation a non-trivial tak. It is the purpose of the present paper to describe these difficulties, find appropriate optimization strategies and instrumentation, and implement a highly efficient problem solution. A specific problem taken from the area of computational physics will be used throughout.

* Present address: KONTRON Electronics, 8057 Eching, Federal Republic of Germany.

0021-9991/88 $3.00

The Ising model is one of the best studied problems in statistical physics. A large variety of simulation implementations of this model, using different algorithms on different computational machinery, have been reported [1–6], including three special-purpose machines [3]. The model consists of a square lattice in two dimensions, or a simple cubic lattice in three dimensions, where one binary variable is associated with each lattice site. This variable is called a "spin" and can attain the values "up" (= binary 1) or "down" (= binary 0). The simulation of this model, which is supposed to be relevant for magnetic solids, consists of initializing the system to have specific values at each lattice site and then evolving the system in time by means of one of several algorithms. Updating each lattice site once is called a "sweep" through the lattice. After a number of sweeps certain measurements are to be performed, most prominently the measurement of magnetization, i.e., counting the number of "up" spins, or equivalently, the number of "down" spins.

The microcanonical algorithm to be used here has first been described by Vichniac [7] and has been implemented by one of us (HJH) [6] in single-task mode on a Cray-XMP. This particular algorithm is equivalent to the cellular automaton Q2R. Historically, it may be interesting to note that only four years ago the fastest simulation speed [1] was 1.6 MHz (= 1.6 MFLIPS = 1.6 million spin FLIPS per second). Reference [6] achieved a speed of 670 MFLIPS earlier this year, whereas we report a speed of 4.3 GFLIPS (giga spin FLIPS per second). This represents a speedup factor of almost 3000 within four years, obtained of course by a combination of algorithmic, hardware, and programming advances to be detailed below.

Consider a square array of spins in two dimensions, of size $N \times N$ spins. In the usual red–black or checkerboard ordering scheme, this lattice is divided into two sublattices, X and Y. Every spin in sublattice X has exactly four nearest neighbours all of which are in sublattice Y and vice versa. The updating algorithm is: "If and only if spin i in sublattice X has as many up nearest neighbours as down nearest neighbours, it is flipped." This algoritm is first executed for all spins within the X sublattice and then for all spins within the Y sublattice. This sequence is iterated ad libitum. It should be obvious that this algorithm has similarities with relaxation steps for partial differential equations, and many of the results put forward below are of relevance for the latter class of problems.

The remainder of this paper is organized as follows: Section II discusses the single-task implementation and optimization of the algorithm for the Cray-2. A problem discovered in defining the performance of this implementation leads to a discussion of Cray-2 memory access parameters in Section III. The multi-tasked implementation and performance is discussed in Section IV. The calculations performed and physics results are discussed in Section V. A summary of the results and conclusions can be found in Section VI.

II. Single-task Implementation and Optimization

The algorithm to be implemented has been given in the Introduction: "If and only if spin i in sublattice X has as many up nearest neighbours as down nearest neighbours, it is flipped." An initial implementation for a Cray-XMP has been given in Ref. [6]. Let us briefly recount the most relevant aspects of this implementation, which then will be adapted and optimized for the Cray-2.

In order to exploit as much parallelism as possible within a single processor, lattice sites = binary variables are assigned to individual bits within a computer word. A 64 element Cray vector of 64 bit words therefore is taken to be a 4096 bit array. With one machine vector instruction all of these 4096 bits are treated together. This corresponds closely to ideas used in the ICL DAP implementations [4]. By assigning lattice sites to storage bits in a suitable sequence it can be arranged that the ith bit within a given computer word has its four nearest neighbours stored as ith bits within four other computer words. The algorihtm therefore can be implemented employing word-logical operations exclusively. It occurs only at the boundaries of the system that words must be shifted by one bit in order to assure periodic boundary conditions [6]. This does not pose any problem and will occur only extremely infrequently for a large system.

The relevant loop as taken from Ref. [6] is reproduced here:

```
      M = N/128
      DO 2 K = 2, M
      J = K
      DO 3 I = 1, N/2
      I1 = IA(J)
      I2 = IA(J + M)
      I3 = IC(J)
      I4 = IC(J - 1)
      ID(J) = XOR(ID(J), OR(
     * AND(XOR(I1, I2), XOR(I3, I4)), AND(XOR(I1, I3), XOR(I2, I4)) ))
3     J = J + M
2     CONTINUE
```

Here, N is the linear dimension of the spin system, i.e., we assume a square of $N \times N$ spins. N must be a multiple of 128. The above double loop updates one quarter of the spin array, minus the first line. The first line update algorithm is slightly different because of boundary conditions [6] and will not be discussed here (though it will, of course, always be included in any timing data given). For convenience in treating the boundary condition we further divide the two sub-lattices X and Y on which the algorithm is defined into two sub-sublattices each. Above loop updates one of these; the other three cases are similar.

Above double loop performs nicely if and only if N is an *odd* multiple of 128. This can easily be understood because of the banking scheme used in the Cray-2 com-

mon (shared) memory. Consecutively used elements of $IA(J)$, say, will be fetched from memory with a vector-read instruction with stride M. If M is odd, all 128 banks present within the 268 Mword vesion of the Cray-2 will be accessed before the first bank accessed in the operation will be used again. If M is even, however, only a subset of banks will be used. In the worst case, where M is a multiple of 128, only *one out of the* 128 existing banks is used for *all* elements of the vector. Since the bank busy time is of the order of 200 ns or 50 clock periods (the Cray-2 has a clock period of 4.1 ns) this unfortunate choice of system size will slow down the memory access rate from one word per clock period to one word per 50 clock periods. This aspect will be discussed in more detail in Section III.

Above loop results in an update rate of 670 MFLIPS if used in one processor of a Cray-XMP [6] for a medium-sized system, $N = 8320$, or larger. If used on the Cray-2, without a single change in the program, compiled under the cft77 compiler version 1.2 with all optimizations enabled, an update rate of 795 MFLIPS is measured on an empty machine (that is, the three other processors are idling). This rate drops down to an average of 645 MFLIPS under normal multi-user operating conditions. The reason for this is to be found in memory contention as will be shown in Section III. Memory access with stride $M > 1$ results in a non-localized memory bank busy pattern. Since many programs access vectors with unit stride it seems to be advantageous to do the same here. Unit-stride memory access tends to keep busy banks better localized in address space, and generally is the most efficient way to access any memory. Above double loop therefore is inverted to yield

```
      M = N/128
      J = 2
      DO 3 I = 1, N/2
      DO 2 K = 2, M
      I1 = IA(J)
      I2 = IA(J + M)
      I3 = IC(J)
      I4 = IC(J - 1)
      ID(J) = XOR(ID(J), OR(
     * AND(XOR(I1, I2), XOR(I3, I4)), AND(XOR(I1, I3), XOR(I2, I4)) ))
2     J = J + 1
3     J = J + 1
```

This loop structure is seen to perform identically the same operations as the previous one, but in different sequence. Results will be identically the same. Under the same conditions as stated above, we obtain a rate of 955 MFLIPS on an empty Cray-2 machine, and an average of 822 MFLIPS under multi-user normal operating conditions. For this loop to perform optimally, $M - 1$ should be some multiple of 64, or N be of the form $N = 8192k + 128$. Because of the finite 268 Mword common memory k can take on values from 1 to 15, or N from 8320 to 123,008. System sizes N^2 then range from 69, 222, 400 spins to 15, 130, 968, 192 spins.

The object code generated by the cft77 FORTRAN compiler (version 1.2) does not fully utilize the Cray-2 processor architecture. In contrast to the Cray-XMP there is no chaining on the Cray-2. Good performance is obtained by simultaneous operation of several functional units. In our case the floating-point add unit as well as multiply unit are not used at all. The logical operations employ only the logical unit. The other unit involved is the common memory port. This can be operated simultaneously with the logical unit if different vector registers are involved. Unfortunately, the current FORTRAN compiler version does not make best use of this feature. All of the memory reads are grouped together at the top of the loop, and all the logical operations follow after that. In this way only minimal use is made of the parallelism within a single Cray-2 CPU. In order to overcome this problem, the above loop has been hand-coded in Cray assembly language as given in Appendix A. This routine is carefully optimized for best possible overlap between memory-access and logical unit operations. The two loops of the FORTRAN version could be coalesced to one loop, with an intermediate conditional update of the address registers corresponding to statement 3 in the FORTRAN version. The initial address offset between the five vector streams involved is taken care of by calling the routine from a FORTRAN main program with suitably offset arguments.

With this routine, the overall performance comes up to 1512 MFLIPS on a single Cray-2 processor within an otherwise empty machine, and an average of 1257 MFLIPS under standard multi-user operating conditions.

It is quite obvious from the above loop that there are six memory accesses and eight logical operations required for updating one word or 64 one-bit spin variables. Equivalencing a logical operation with a floating-point operation, the performance figures given so far can be translated into MFLOPS (Million floating point operations per second) as given in Table I, or effectively used memory bandwidth in Mword/s. From lines 2–4 of this table it is quite clear that the performance of any program running within one CPU, measured as work performed

TABLE I

Performance Summary

Code version	MFLIPS	MFLOPS	Mword/s
XMP FORTRAN	670	84	63
Cray-2 FORTRAN	645/795	81/99	60/75
Same, inverted loops	822/955	103/119	77/90
Cray-2 assembly loop	1257/1512	157/189	118/142
Four identical tasks	1086 * 4	136 * 4	102 * 4
Two-way multitasked	2500/2770	312/346	234/260
Four-way multitasked	4300	538	403

Note. In each main column the first entry corresponds to standard multi-user operating conditions, the second one to single-user mode. Where only one entry is given, no such distinction is meaningful.

divided by *CPU-s* used, varies significantly depending upon the operating conditions, i.e., what programs if any are running in the other three processors. A further indication of this fact is given by line five of Table I, where four independent copies of the same program were put into the four CPUs. Performance drops down from the average multi-user environment performance of 1257 MFLIPS to 1089 MFLIPS. This is again due to memory contention: the program under discussion in this paper is more intense on memory accesses than the average program at the Minnesota Supercomputer Center, and so gets less effective memory bandwidth per processor if competing with itself than if competing with other users' programs. This fact will become more clear in the next section.

As a last illustration of the variance in single-task performance we give the MFLIPS rates as obtained from 100 independent measurements on a typical day at our installation in Fig. 1. Statistically we obtain 1257 ± 48 MFLIPS equals 4% standard deviation. The best case possible are the 1512 MFLIPS if the other three processors are idling. The theoretical worst-case performance is 437 MFLIPS, as will be calculated in the next section. It should be quite clear by now that any performance data measured on the Cray-2 must be given with a detailed specification of the operating conditions, since a difference factor of three is possible in principle. However, as Fig. 1 exemplifies, performance deviations of more than 15% from those measured under standard multi-user operating conditions are extremely rare occurrences. However, since multi-tasking influences memory contention in a rather singular way, design and measurement of multi-tasking jobs must take these effects into consideration carefully.

When treating very large systems one should also consider the initialization with some care. Scalar recursive initialization, assuming one microsecond per spin, would take more than 4 CPU h for the largest system. A vectorizable initialization is to set each spin "up" with probability p, and "down" with probability $1-p$ (a percolation system). The relation between the energy of the system and this probability is given by [8]

$$E(p) = 8p(1-p) - 2,$$

where E ranges from -2 (zero temperature, ground-state) to $E=0$ (infinite tempeature) with p ranging from 0 to $\frac{1}{2}$ or, equivalently, 1 to $\frac{1}{2}$. The citical temperature

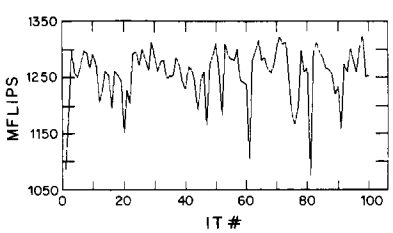

FIG. 1. Performance of single-task code in standard Minnesota multi-user environment, in MFLIPS, monitored at 100 different times ($N = 24{,}704$ system).

corresponds to the critical energy $E_c = -2^{1/2}$. This initialization is done most efficiently by generating words of random bits, then comparing the first word to the most significant bit of p, the second word with the next most significant bit of p, and so on, using only word logical instructions [9]. Special efficiency is gained when p has a small number of non-zero bits in its binary representation. $p = \frac{1}{8}$ resulting in $E = -1.125$, for example, can be realized by taking the logical AND of three words of random bits to define 64 spins within the lattice. Even with rather inefficient techniques for the generation of random numbers, this takes less than 5 ns/spin. Then, for any system, the initialization time is negligible ($\ll 1\%$ of total time).

One also needs to know the relation between energy and temperature in general. We do not measure the temperature but use the known exact relation instead [12].

The last part of the computation to be discussed is the measurement of magnetization. Here we have to count the number of 1 bits in a vector of words. The corresponding hardwae instruction exists in the Cray-2 CPU (population count), and the cft77 FORTRAN compiler uses it. Again the generated code is not optimal, but since magnetization measurement occurs not too frequently (one usually does several sweeps over the lattice before another magnetization measurement is made) this is not too relevant. FORTRAN coded magnetization measurement takes from 0.26 ns/spin (multi-user, single-task) to 0.19 ns/spin (single-user, single-task). Multitasking will be discussed in Section IV.

III. Effective Memory Bandwidth

In order to obtain more quantitative insight into the memory contention problem let us study this by itself. In a dedicated single-user environment it is possible to control the memory-access patterns of all four CPUs simultaneously and thereby obtain quantitative precise measurements. These timing measurements must consider a few important facts. A number of conflicts can slow down memory access:

1. *Register conflict.* If a memory read or write vector instruction is issued the vector register used should be free, i.e., not be reserved from some previous instruction. In particular, consecutive memory references should use *different* vector registers since vector-startup-times may be overlapped in this case. If the same vector register is used in two consecutive vector-read instructions, a conflict occurs and effective memory bandwidth degrades (example given below). Because of this fact it is difficult to perform measurements of this kind from FORTRAN programs, since one cannot in general predict what code will be generated. Assembly language routines have been used for the measurements reported here. An example is given in Appendix B.

2. *Memory bank conflict.* If a read is requested from a currently busy memory bank, that read is delayed until the memory bank becomes available.

Normally, this would not introduce severe problems. However, because of the enormous size of 268 Mwords the Cray-2 memory is constructed from slow dynamic randm-access memory chips. The bank busy time of about 200 ns is much larger than the fast system clock of only 4.1 ns. The bank conflict therefore is much more penalizing on this machine than usual, where the factor between clock cycle time and bank busy time is smaller. Memory bank conflicts can be caused by preceding fetches from the same CPU, or by fetches from other CPUs.

3. *Memory quadrant phase conflict.* The Cray-2 memory is divided into four quadrants of 67 MWord each. These quadrants are associated with the four processors in a rigidly phased mode. Assume that in clock period 0 CPU *A* has access to quadrant 0. Then, in this clock period, CPU *B* will have access to quadrant 1, CPU *C* to quadrant 2, and CPU *D* to quadrant 3. In the next clock period the asignments shift by one, i.e., CPU *A* now has access to quadrant 1, and so on. It is evident that we have another instance where sequential memory access is best: the two quadrant bits are the least significant bits of the word address.

A number of effective memory bandwidth measurements were performed. First, let us assume that three processors are idling or otherwise not issuing any memory references while the fourth processor executes the measuring program. In this case, we can avoid *any* bank conflicts by simply reading with odd strides. We measure an effective bandwidth of 210 Mword/s in this case. If register conflict occurs, this number drops down to 120 Mword/s. 210 Mword/s is slightly less than one word per cycle due to imperfect overlap of vector-startup times for the 64-element vector read instructions. Smaller vector lengths would lead to further degradation and are not studied here.

A program can create memory bank conflicts with itself. One chance to do this is to overlap the end of one 64-element vector to be read with the beginning of the next one. Assume one vector read affects banks 0 to 63, and the next vector read wants to access banks 63 to 126. The first word for the second vector then has to come from the same bank where the last word of the previous vector has come from, which is busy at this time. The effective bandwidth drops down to the same 120 Mword/s as is observed for the register conflict. This case of conflict generation might seem contrived, but this is not so. Consider the second FORTRAN loop given above. A sequence of 64 elements of the type IA(J + M) will be read, followed by a sequence of 64 elements of type IC(J): if one is not very careful about the location of arrays IA and IC in memory this conflict will very easily occur.

The conflict described last may occur in varying intensities: if the first-word-address offset between the two vectors under discussion is less than seven, no degradation occurs: these banks are not busy any more. For an offset of 7 the effective bandwith drops down to 163 Mword/s, and then gradually decreases to the worst-case (for this type of conflict) of 120 Mword/s described above.

If employing non-unit stride a quadrant phase conflict can result. If memory is accessed with stride 2, only every other cycle is the correct quadrant phase met. The effective bandwidth therefore reduces to 64 Mword/s. With stride four only every

fourth cycle is the correct quadrant phase met, resulting in a measured bandwidth of 37 Mword/s. Strides of 8 or 16 do not result in worse conflicts than four since there exist only four phases and bandwidth has already degraded so much that no busy bank is hit. With stride 32, however, the bank busy conflict takes over, resulting in 22 Mword/s. With stride 64 we obtain 11 Mword/s, and finally the worst case is stride 128 with 5.5 Mword/s. This discussion shows that non-unit-stride memory access must be considered with some care on the Cray-2.

All of the preceding discussion assumed that the other three CPUs were not issuing any memory requests. Let us consider next the case where all other three CPUs read a continuous infinite length vector stream from consecutive addresses. If the fourth CPU does the same, obviously all four CPUs will see the same effective memory bandwidth measured to be 158 Mword/s for each procesor. This is the most favourable memory access pattern. The aggregate rate of $4 \times 158 = 632$ Mword/s or 2.6 words/clock period therefore is the aggregate maximum memory bandwidth obtainable on the Cray-2. It should be noted that this is less than one word per procesor and cycle. This observation by itself should make clear that the four CPUs in any non-trivial situation will compete strongly for memory bandwidth. The decrease from 210 Mword/s (three CPUs idle) to 158 Mword/s results from bank busies generated from the other three CPUs if active.

In addition to the other three CPUs generating bank busy conflicts we can have the conflict of vector-end and vector-beginning bank addresses to overlap, i.e., the first-word-address offset of 63 discussed above. If this occurs for the CPU under study, its bandwidth decreases to 108 Mword/s.

Still more severe conflicts are generated if the other three CPUs exhibit a less-well behaved memory access pattern. The worst case possible occurs if all these three CPUs access only one and the same memory bank continuously. In this case the effective bandwidth seen by the fourth (well-behaving, unit stride, no internal conflict) CPU drops down to 41 Mword/s. Any intermediate value can be obtained by having the other three CPUs perform intermediate variations of access patterns. It has been shown, therefore, that one and the same code executing within one CPU can exhibit performance variations between 41 and 210 Mword/s, or a factor of five, solely depending upon the memory access patterns of the other three processors. For a memory-access bound computation, where all the arithmetic is hidden perfectly behind (overlapped with) the memory accesses, that factor immediately translates into corresponding MFLOPS or MFLIPS variations. This is the case for the algorithm studied here (Appendix A). If this algorithm were not involving any memory references, the maximum execution rate would be limited by the eight logical operations per word update executing at 220 MFLOPS, resulting in 1760 MFLIPS. This is only 14% larger than the rate measured on an otherwise empty machine, 1512 MFLIPS. Therefore, memory access and functional unit operations are almost perfectly overlapped for the code of Appendix A. Better overlap is inhibited by a insufficient number of vector registers and would not come to bear anyway in memory-contention limited situations, which prevail in any

realistic operating environment (either because of multi-user mode or because of multitasking).

As can be seen from Fig. 1, the worst case of 41 Mword/s, corresponding to 437 MFLIPS, does not occur under any normal circumstances. It would require quite a curious conspiration of user programs to access all of only one and the same memory bank all of the time. The performance variations of a few percent shown in Fig. 1 are much more typical.

Unfortunately the previous paragraph does not hold true for a multitasked program. In this latter case it will occur that all tasks work on the same data arrays. If someone chose to apply row-wise algorithms to matrices of dimension 128 or a multiple thereof exactly this type of conspiracy would occur, and extreme slow-downs could be observed. Multitasking in this case would lead to speedup factors much smaller than four. In general it can be said that for the Cray-2 multitasking is the more attractive the less memory-active the multitasked code is: in this case, multitasking gets rid of those other users causing memory contention, and, potentially, speedup factors much larger than four could be observed. Related observations have been made by Taylor and Bauschlicher [10].

IV. Multitasking

There exist several possible ways to multitask a computation of the present type. If one is doing small systems, one always has to do calculations for a number of identical systems, the only difference being the initial random number seed. This is required in order to reduce the statistical variance which is large for small system ($N < 8320$). The best possible (most efficient) solution then is to dynamically partition out independent systems to the CPUs available. Since systems are small no limitation due to limited storage arises. For small systems, the *first* FORTRAN loop given in Section II will be used, since the second, improved version is not yet efficient. The granularity of this scheme is very high: an individual task will take at least many minutes, if not more. Therefore, there is no measurable multi-tasking overhead. One expects speedup factors of exactly four.

Because of the discussion of memory contention in Section III one has to be very careful in defining the reference single-task performance required to calculate speedup factors. Single-task execution time is defined *only* with additional prescriptions about the work of all remaining processors, i.e., depends upon the operating conditions. In our opinion, there are three possible ways to define single-task performance, of which only one is acceptable.

1. Measure single-task execution time in a standard multi-user operating environment. This definition has the drawback that it will depend upon the installation where the measurement is made, time of day, and chance factors. It will generally not be reproducible and therefore cannot be accepted.

2. Measure single-task execution time by having all other processors idle. This definition will result in reproducible numbers. However, it represents a very significant waste of resources: no realistic large-scale calculation will ever be made under such circumstances. These operating conditions are totally artificial and contrived, and do not represent a reasonable use of investments made.

3. Measure single-task performance by having four independent copies of the same program run in the four processors. This definition will lead to reproducible results and represents a reasonable use of resources. Of course, it serves to normalize out the bulk of memory contention effects. However, as we will show below, there still is a residue of these effects left. This definition will serve to measure the overheads involved in task communication and synchronization. It does not serve

In Fig. 2 we give the MFLIPS performance of four independent calculations running in parallel, for 200 sweeps through the lattice ($N = 24704$). This graph corresponds to line five in Table I. We find 1086 ± 5 MFLIPS, a standard deviation of 0.5%. This variation is due to clock interrupts, assorted UNIX demons waking up, memory refresh, etc. Observe the typical anticorrelation between individual task performances: whenever one task gets a smaller share of memory bandwidth, the other tasks get a larger share of it, and vice versa. This type of variation occurs because memory access patterns may lock between tasks for intermediate times. Lines four and five of Table I define the three reference values discussed in 1–3 above. By definition, the multitasking philosophy given at the beginning of this section results in a speedup factor of four, therefore. If one takes the single-user, single-task value as reference, the speedup drops down to 2.9 because of memory contention. If the "Minnesota multi-user operating condition" is taken as reference, a speedup factor of 3.5 results. This last result stems solely from the fact that the current program is more memory-intensive than most others *at Minnesota*. It is not clear how much this finding has more global implications.

For large problems the multitasking philosophy described thus far is not feasible: there is not enough memory on the Cray-2. In any case, for any finite memory available, one would like to study the largest problem possible, where all of this memory is used for one problem. We therefore have to multitask a single problem solution.

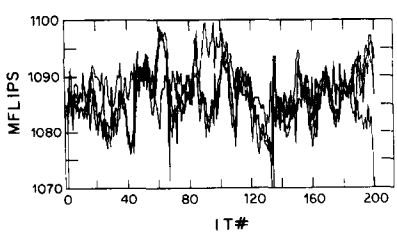

FIG 2. Performance of four identical copies of single-task code running simultaneously and independently in single-user (dedicated) mode, in MFLIPS, monitored at 200 diffeent times ($N = 24,704$ system).

Let us consider first a two-way multitasking. The original problem specification implies some amount of recursiveness: first work on one-half of the system, the "red" spins, or sublattice X. *Only after* all the X spins have been treated, is it allowed to deal with the "black" or sublattice Y spins. Therefore, some synchronization is required. The sublattices X and Y already had been broken down into two sublattices each. The two sublattices of X are totally independent of each other and may be given to two independent tasks. After both are completed, the two independent sublattices of Y may be treated. Two-way multitasking therefore comes quite naturally.

Of the various multitasking primitives the task-start and wait for task-termination routines [11] are the most time-consuming ones, i.e., incur the most overhead. It is therefore advisable to create tasks only once per program run, and use events in order to synchronize the X/Y sublattice calculations. We therefore have four tasks, corresponding to the 2×2 sub-sub-lattices. Each task is associated with two events: a start event, posted by the main program and waited for by each task, and a done event, posted by the task and waited for by the main program. The initial two sweeps through the lattice then will be executed as Fig. 3 shows. At $t = 1$, the main program posts the start events for the two X-sublattice tasks which have been created previously. At $t = 2$, these two tasks post their done events, and return to waiting for their next start events. The main program, upon receiving the two done events, posts the start events for the Y-sublattice tasks, and then waits for their done events. These tasks proceed to update the Y-sublattice and then post their done events. The main program receives these two done events, and the next sweep through the lattice can issue.

The resulting performance figures are given in the sixth line of Table I. In single-user mode, i.e., having two CPUs idle, 2770 MFLIPS are obtained, a speedup of 1.8 compared to the single-task, single-user reference. The loss of 20% of a CPU is attributed to memory contention; the synchronization calls take only a few hundred clock periods each, or about one microsecond, compared to task execution times of 100 ms or more ($N = 24,704$). Load is well balanced since the same number of

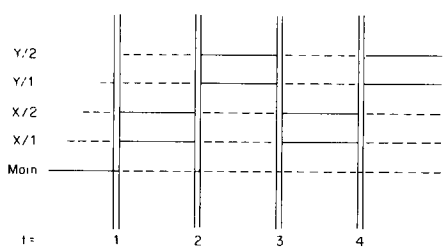

FIG. 3. Activities of tasks vs time. Broken lines, task waits for "start work" event to be signalled from master. Full lines, task executing. Two-way multitasked version. The four-way multitasked version is identical, with the exception of four tasks active for each the X and Y sublattice calculations. At the time points 1, 2, ..., the master task waits for events signalling the completion of calculations for one sublattice, and then posts events to signal the tasks for the other sublattice to begin their work.

spins are contained within each sublattice, and the same number of operations and memory accesses are performed. The number obtained under "multi-user" conditions has been obtained at a time shortly after system-startup where there were only two other tasks in the system. Under standard operating conditions this number may become arbitrarily small: in a multitasked program, measuring CPU time is meaningless as far as multitasking performance evaluation is concerned. Instead, one must measure real time. Real time for executing a program in multi-user mode, however, depends upon a number of strongly varying factors, like the number of users and the number of jobs in the system, characteristics of jobs, etc. This number of 2500 MFLIPS therefore is highly uncertain and not reproducible. If there is an increasingly large number of tasks in the system competing for CPUs, real time performance of a single task may approach zero.

In order to obtain four-way multitasking we have to split the four sub-sublattices used so far once again. In the current case, it was found easiest to simply split the outer loop, i.e., to have one processor work on approximately the first half of a sub-sub-lattice, and another processor on the second half. Because of the first line being slower as a consequence of periodic boundary conditions, the second "half" is made a litle bigger than the first "half," since the very first line requires somewhat more time per spin than all other lines. However, for the large systems considered here, the potential work imbalance would be less than 0.4% if this point were not considered. The scheme of Fig. 3 then is generalized for a total of eight tasks, four per sublattice X or Y. There are eight start events and eight done events, consequently. The synchronization logic does not change.

The spin-flip rate observed with this method is given in the last line of Table I, 4300 MFLIPS. This number is obtained by dividing the system size by elapsed wall-clock time, for one complete sweep over the lattice, averaged over 100 sweeps (there is about 0.5% statistical variation in task execution times due to reasons discussed above). Compared to our reference calculation of four independent identical tasks the speedup factor is 3.96, compared to single-task single-user mode 2.84, and compared to single-task CPU time in standard multi-user operating mode 3.42. It should be obvious by now that the last two factors are explained by memory contention: if three processors are idle, the one active processor sees a much higher effective memory bandwidth, i.e., the single task is faster, resulting in the 2.84 speedup factor. Under standard multi-user conditions, the average program at Minnesota still takes less than the maximum memory bandwidth it could get, so that the present simulation task gets more than a fair share of aggregate memory bandwidth. Comparing with the case of four independent tasks, we still have to account for 4% CPU, or 1% efficiency missing in the first speedup factor given.

The measurements reported above were taken on the $N = 24704$ system. Individual task execution time is 69 ms. The few hundred clock periods or about 1 μs required for treating the events, or fractions of a millisecond to perform a context switch, cannot explain this discrepancy. We performed a more detailed timing analysis by time-stamping each individual task start and stop event. In spite of exactly the same operation count involved in each task and all memory accesses

being stride one, we nevertheless observe a slight load imbalance of about 1% explaining the above discrepancy. This imbalance results from the different first-word-address offsets required in fetching neighbours for the various sublattices. It has been seen in Section III that consecutive vector reads from common memory with unit stride will result in varying effective memory bandwidth depending upon possible overlap in memory bank space of the vectors fetched. Investigating the offsets (like $IA(J+M)$ and $IC(J)$ in the FORTRAN code given in Section II) it turns out that unavoidable conflicts of this sort result with varying frequencies for the individual sublattices leading to the discussed work imbalance. We have therefore explained *all* performance figures given in Table I by common memory access limitations. The classical overheads usually discussed in multi-tasking investigations play no role in the present case. This is of course because of the rather large granularity of the present problem. If we were to look at smaller problems, in particular the context switch time would become observable and reduce the speedup factor. But again, for smaller problems a totally different way of multitasking would be employed as outlined at the beginning of this section.

In order to avoid reduced speedup factors due to this kind of load imbalance, dynamic load balancing [11] is strongly recommended wherever applicable. For the current case, dynamic load balancing is rather difficult to implement and would drastically reduce the granularity leading to much larger overheads; for the case of smaller systems discussed at the beginning of this section dynamic load balancing is the method of choice.

The calculation reported in the next section evaluated the magnetization of the spin array at *every* sweep through the lattice. In this rather unusual case some additional attention must be given to this usually negligible part of the computation. The magnetization calculation, which obviously can be carried out independently for any desired subset of words, has been multitasked by giving each processor one of the four sub-sub-lattices to evaluate. Furthermore, the relevant loop also has been coded in Cray assembly language to achieve better overlap between memory access and integer functional unit operation, resulting in a final performance of 0.05 ns per spin, utilizing the full machine.

V. Results

We performed a run of 2400 lattice sweeps for the 123008^2 system. The system was initialized as a percolation system with "up" probability $p = \frac{7}{8}$, corresponding to an energy of $E = -1.125$ or temperature [12] $T = 1.092349\, T_c$, where T_c is the critical temperature, and a magnetization of $M = 0.75$. The magnetization was measured at *each* iteration in order to obtain a detailed history, and is given in Fig. 4. This lattice of 15 120 968 192 spins needs 236 425 220 words of storage including two extra lines added for convenience in treating the periodic boundary conditions. The total wall-clock time used was 2 h 50 min of dedicated four-

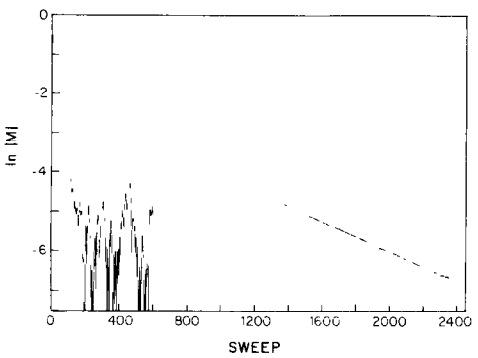

FIG. 4. Natural logarithm of magnetization measured vs number of sweeps through the lattice. The upper curves give measured data and the four-term exponential fit, for the $N^2 = 123{,}008^2$ system. The lower curve gives results for the standard Metropolis algorithm [2] applied to a 2048×2049 system, see discussion in text.

processor Cray-2 machine time. These 10, 200 s divide into 70 s for initialization (non-multitasked for ease of random-number generation), 8445 s for lattice updates, and 1685 s for magnetization measurements. The calculation could not have been performed on any other computer available today because of the required data transfer rate of 400 Mword/s. Using a Cray-XMP solid-state disk (SSD) would have slowed down the calculation at least a factor of four; using magnetic disc would have resulted in a minimal slowdown factor of 50 (assuming striping of data to 8 disks in parallel).

Because of the size of the system this data is accurate enough to support being fitted by a sum of four exponential terms. Statistical errors have been estimated by dividing the lattice into four sublattices, and recording the magnetizations individually. The first 25 sweeps have been disregarded in the fitting procedure; probably another one or two exponential terms would be required to describe these very early times. The best four-term fit found under these conditions is

$$M(t) = 0.088301 \exp(-0.001818\, t) + 0.231996 \exp(-0.003986\, t) + 0.187507 \exp(-0.010411\, t) + 0.107897 \exp(-0.040401\, t), \tag{1}$$

where the time t is measured in lattice sweeps (= Monte Carlo steps per spin). All parameters are determined to within 10%. No satisfactory fit can be achieved with only two or three terms, as is quite obvious from the significant deviations from the straight line in Fig. 4. The above expression is also shown in Fig. 4. The parameters do not exhibit any size dependence for the large systems studied here. A simulation of the $N = 57,472$ and $N = 24,704$ systems results in essentially the same parameters, though with less statistical certainty. Since the very early time regime due to the percolation initialization may be amenable to analytic treatment, the first ten magnetizations are given in Table II.

TABLE II

Early time magnetizations at $E = -1.125$, for the $N = 123{,}008$ system

Sweep	Magnetization
0	0.75
1	0.641764
2	0.630649
3	0.616336
4	0.605299
5	0.591935
6	0.588981
7	0.572937
8	0.569760
9	0.560427
10	0.555507

Note Sweep 0 is the percolation initialization.

While the MFLIPS rate of the present calculation is significantly larger than any other achieved in the past and, in particular, a factor of 50 faster than the most proficient implementation of the standard canonical Metropolis algorithm [2], the question arises if the Q2R algorithm may not need many more iterations in order to obtain physically equivalent results. At the temperature $T = 1.092\ T_c$ relevant for the present study the magnetization must decay to zero exponentially with time, as is borne out by the analytical formula given above. If this exponent is much larger for the Metropolis algorithm, the latter will have a chance to compete.

In order to investigate this idea we performed a canonical Metropolis simulation [2] of a 2D 2048×2049 system, initialized the same way as discussed above, at the temperature of $1.092349\ T_c$. The results of this simulation are given in the lower curve in Fig. 4. Because of the much smaller system size there is significantly more statistical fluctuation. However, the first 100 sweeps follow rather accurately an exponential law given by

$$M(t) = 0.6302 \exp(-0.0322\ t). \tag{2}$$

This result does not depend significantly upon system size; a 1025×1024 system, as well as a 8321×8320 system, gives the same exponent to better than 1% accuracy. It is seen that this exponent is a factor of 18 larger than the asymptotic Q2R exponent of 0.001818. That is, the Metropolis algorithm at this temperature moves a factor of 18 faster *in physical time* than the Q2R algorithm asymptotically, but only a factor of 10 faster at early times. Since the simulation speed *per sweep* (for the same system size, of course) is a factor of 50 slower, the Q2R algorithm still wins, though not by as impressive a margin, but only a factor of 2.5.

In order to find the behaviour of this penomenon with temperature, we did a

number of smaller runs at $T/T_c = 1.928443$, corresponding to $E = -0.5$, with initial magnetization $M = 0.5$ achieved via percolation initialization with $p = 0.25$. With 218 sweeps for the $N = 90, 240$ system we find leading exponents of 0.07 for the Q2R algorithm, and 0.5 for the Metropolis algorithm ($\pm 10\%$). For the latter simulation, 28 sweeps in the 12048×12049 system were used. At this rather high temperature the Metropolis methods gets 7 times as far per sweep as does the Q2R algorithm, therefore. At $T = \infty$ the Metropolis algorithm will be inferior, since every spin is flipped every time, i.e., the system never really changes but is *strictly periodic* with period length = two sweeps. This does not happen for the Q2R algorithm. Consequently, somewhere in the high temperature regime there must be a crossover point where both algorithms have the same efficiency. However, since this regime is not really interesting, we did not try to find that temperature. It seems to be a fair conclusion, however, from the present evidence as well as that of Ref. [6], that the Q2R algorithm is more strongly affected by critical slowing down than the standard Metropolis algorithm.

We also performed a number of calculations in closer vicinity of the critical temperature. At temperatures above the critical temperature any initial magnetization must decay to zero exponentially (Fig. 4). At $T = 1.003\ T_c$ this exponential decay can still be observed, though not with sufficient accuracy to measure the exponent (without significant investments in computer time), see Fig. 5. For a temperature of $T = 1.01\ T_c$ the exponential decay is already rather well pronounced (bottom curve in Fig. 5).

At the critical temperature any initial magnetization must decay to zero like a power law. For the Q2R algorithm this is demonstrated in the center part of Fig. 5,

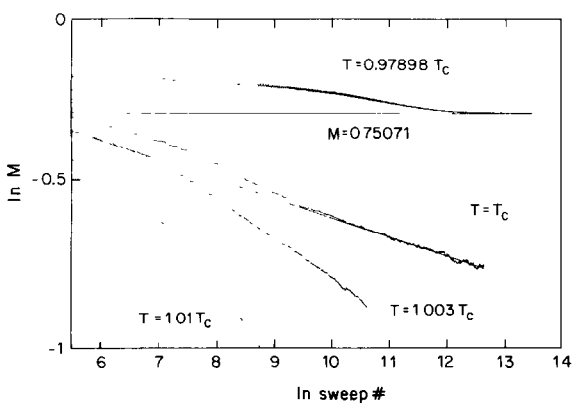

FIG. 5. Natural logarithm of magnetization measured vs natural logarithm of number of sweeps. The upper curve results for $T = 0.97898\ T_c$, where $M = 0.75071$ is the exact result. The center curve is at $T = T_c$. The straight line fit is given in Eq. (3). The two bottom curves show the typical behaviour at temperatures above T_c. All results are from the $N = 8320$ system.

exhibiting 311,000 sweeps for the $N^2 = 8320^2$ system. The asymptotic behaviour is given by

$$\ln M(t) = -0.02166 - 0.05787 \ln t, \tag{3}$$

where the time t is measured in lattice sweeps, and the coefficients are determined within 5%. For the standard Metropolis algorithm [2] the corresponding result is

$$\ln M(t) = -0.1589 - 0.05614 \ln t. \tag{4}$$

It is seen that at the critical temperature *both algorithms exhibit the same exponent*, i.e., are equally efficient per sweep. The performance advantage of the Q2R algorithm at this—the most important—temperature therefore carries over in full to the simulation of physical observables. The exponents given in Eqs. (3) and (4) do not depend upon system size in our calculations, for both the Metropolis and the Q2R algorithm, as far as the error bars allow us to tell.

A corresponding investigation for the microcanonical algorithm with demons [5] would be most useful. The study of Bhanot *et al.* [13] does not address the exponent discussed above. Their estimate as to the higher efficiency is based on CPU time comparisons. Since they do not give details of their Metropolis implementation, no comparison on efficiency per lattice sweep can be made.

At temperatures below the critical temperature the initial magnetization is to approach exponentially the spontaneous magnetization of the 2D Ising model, which is known exactly for all temperatues [12]. It has been shown in Ref. [6] that the Q2R algorithm roughly, i.e., within statistical uncertainties of several percent, reproduces these values. We performed a more quantitative test at $E = -2^{1/2} - 0.1$, or a temperature of 0.97898 T_c. The results for 7×10^5 sweeps over the $N^2 = 8320^2$ lattice are shown in the upper part of Fig. 5, together with the exact result of $M = 0.75071$. We use these rather large lattices in order to avoid any discussion of finite-size effects, which can be significant for the microcanonical Q2R algorithm [13] on smaller lattices. The measured value is $M = 0.7508 \pm 0.0004$ in perfect agreement with the exact value.

We must remark that *none* of the calculations reported here used any averaging over initial configurations. In each case, we prepared one single configuration by randomly setting a number of spins in order to achieve a system at the desired energy. All results reported come from single, long runs stepping the single configuration in time. While of course this cannot prove the ergodicity of the present algorithm we can state that no traces of non-ergodic behaviour have been found. For very low temperatures, however, non-ergodic behaviour is known to occur [14]. Also on short time scales traces of periodic behaviour may be seen, c.f. the regular oscillations superimposed on the curves in Fig. 5. This periodic behaviour is due to finite clusters with finite periods under the deterministic Q2R algorithm.

VI. Summary and Conclusions

We have demonstrated that Monte Carlo simulations of the present type can be adapted to the architecture of the Cray-2 rather well. The multitasking performance is limited by memory bandwidth considerations. Classical considerations like inter-task communication and synchronization, as well as work balance, are comparatively of little importance, due to the large granularity of the problem. The most important consideration for making optimal use of a Cray-2 therefore is reduction of memory contention. In some cases, like matrix multiply, the local memory may be used for this purpose; for the present algorithm we did not find any use of local memory.

The simulation speed per spin of the Q2R algorithm is significantly larger than the speed of the Metropolis algorithm. However, this fact is at some temperatures partially offset by a larger number of lattice sweeps required by the Q2R algorithm in order to "travel the same distance in a random walk." Overall, and in particular at the critical temperature, the Q2R algorithm is significantly faster than Metropolis.

APPENDIX A. Assembly Language Version of Inner Loop

This appendix gives an optimized version of the FORTRAN double loop of Section II, using cal (Cray assembly language). The optimization is for the Cray-2; a Cray-XMP would require an entirely different code to be optimal. A number of standard macros are used for the subroutine linkage:

```
                IDENT LOOP
* Q2R ALGORITHM INNER LOOP, GENERAL CASE
                ENTRY LOOP
LOOP            ENTER (I1, I2, I3, I4, I5, NLOOP, MDMX64), MODE = BASELVL
                BASE M
*
* REGISTER MAP
* A   HOLDS              S   HOLDS
*-------------------------------------------------
* 0           64
* 1         >IA1<
* 2         >IA2<
* 3         >IA3<
* 4         >IA4<        INNER LIMIT
* 5         >ID <        OUTER COUNTER
* 6           1                    1
* 7         >ID<OLD      INNER COUNTER
*
```

```
          ADDRESS A1, I1     ; IA1
          ADDRESS A2, I2     ; IA2
          ADDRESS A3, I3     ; IA3
          ADDRESS A4, I4     ; IA4 FIRST WORD ADDRESSES
          ADDRESS A5, I5     ; ID(K)
          LOAD S4, MDMX64; INNER LOOP TRIP LIMIT
          S7 S4
          LOAD S5, NLOOP  ; OUTER LOOP TRIP COUNT
          A6 1
          S6 A6
*
          A0 D'64
          VL A0              ; VECTORLENGTH
*
          V1 (A1, A6)        ;GET IA1
          A1 A1 + A0
          V3 (A3, A6)        ; GET IA3 *** pre-fetch ***
          A3 A3 + A0
          V2 (A2, A6)        ; GET IA2
          A2 A2 + A0
          J  BEGIN
*
LLL       (A7, A6) V4        ; STORE ID *** from previous loop trip ***
BEGIN     V0 V1\V3           ; IA1 XOR IA3
***
          A7 A5              ; SAVE CURRENT ID ADDRESS
          A5 A5 + A0         ; GET NEW ID ADDRESS
          S7 S7-S6           ; DECREMENT INNER LOOP COUNT
***
          V4 (A4, A6)        ; GET IA4
          A4 A4 + A0
          V6 V1\V2           ; IA1 XOR IA2
***
          JN S7, PROCEED  ; NO WORD SKIP
          S7 S4              ; RE-INIT COUNTER
          A1 A1 + A6
          A2 A2 + A6
          A3 A3 + A6         ; SKIP ONE WORD
          A4 A4 + A6
          A5 A5 + A6
***
PROCEED   V5 (A7, A6)        ; GET ID
          V1 V2\V4           ; IA2 XOR IA4
          V7 V1&V0           ; FIRST AND
```

```
          V2 (A2, A6)          ; GET IA2 *** for next trip ***
          A2 A2 + A0
          V1 V3\V4             ; IA3 XOR IA4
          V0 V1&V6             ; SECOND AND
          V3 (A3, A6)          ; GET IA3 *** for next trip ***
          A3 A3 + A0
          V6 V0!V7             ; OR
          V1 (A1, A6)          ; GET IA1 *** for next trip ***
          V4 V5\V6             ; XOR FLIP
          A1 A1 + A0
*
          S5 S5-S6             ; DECREMENT LOOPCOUNT
          JN S5, LLL
*
          (A7, A6) V4          ; STORE RESULT FROM LAST TRIP
          RETURN
          ENDSUB
          END
```

APPENDIX B. MEMORY-BANDWIDTH MEASURING INSTRUMENTATION

In order to avoid any register conflicts a memory bandwidth timing routine should be written in assembly language for the Cray-2. The routine given here will read 256,000 words from common memory, using the given stride and address offset between consecutive vectors. This routine should be called sandwiched between calls to the CPU timer [second( )] from a FORTRAN main program. Overhead is then negligible.

```
          IDENT TIMER
          ENTRY TIMER
TIMER     ENTER (I1, STRIDE, OFFSET), MODE = BASELVL
          BASE M
*
          ADDRESS A1, I1       ; address of array to be read from
          LOAD A6, STRIDE      ; stride between vector elements
          LOAD A7, OFFSET      ; address offset between first words
          S1 1                 ;                          of vectors
          S6 D'1000            ; read 4 * 1000 vectors ...
*
          A0 D'64
          VL A0                ; ... of length 64
*
```

```
LOOP  V0 (A1, A6)        ; LOAD TO VO WITH STRIDE
      A1 A1 + A7
      V1 (A1, A6)        ; LOAD TO V1 WITH STRIDE
      A1 A1 + A7
      V2 (A1, A6)        ; LOAD TO V2 WITH STRIDE
      A1 A1 + A7
      V3 (A1, A6)        ; LOAD TO V3 WITH STRIDE
      A1 A1 + A7
      S6 S6-S1           ; all loads done to different
      JN S6, LOOP        ;                vector registers
*
      RETURN
      ENDSUB
      END
```

ACKNOWLEDGMENTS

We are grateful to D. Stauffer for useful discussions. We thank the staff of Minnesota Supercomputer Center, Inc., and in particular R B. Walsh and K. C. Matthews for significant help in carrying out the calculations reported here.

REFERENCES

1. R. ZORN, H. J. HERRMANN, AND C. REBBI, *Comput. Phys. Commun.* **23**, 337 (1981); C. KALLE AND V. WINKELMANN, *J. Statist Phys.* **28**, 639 (1982).
2. S. WANSLEBEN, J. G. ZABOLITZKY, AND C KALLE, *J. Statist. Phys.* **37**, 271 (1984); S WANSLEBEN, *Comput. Phys. Commun.* **43**, 9 (1987).
3. R B PEARSON, J. RICHARDSON, AND D. TOUSSAINT, *J. Comput. Phys.* **51**, 241 (1983); A. HOOGLAND, J. SPAA, B. SELMAN, AND A. COMPAGNER, *J. Comput. Phys.* **51**, 250 (1983); J. H. CONDON AND A. T OGIELSKI, *Rev Sci. Instrum.* **56**, 1691 (1985); A T OGIELSKI AND I MORGENSTERN, *Phys. Rev. Lett.* **54**, 928 (1985).
4. G. S PAWLEY, D. J. WALLACE, R. J. SWENDSON, AND K. G. WILSON, *Phys. Rev. B* **29**, 4030 (1984); S. F. REDDAWAY, D M. SCOTT, AND K A. SMITH, *Comput. Phys. Commun.* **37**, 351 (1985).
5. M CREUTZ, *Phys Rev. Lett.* **50**, 1411 (1983); M. CREUTZ, P. MITRA, AND K. J. M. MORIARTY, *Comput. Phys. Commun.* **33**, 361 (1984).
6. H. J HERRMANN, *J. Statist. Phys.* **45**, 145 (1986).
7. G Y. VICHNIAC, *Physica D* **10**, 96 (1984). Y. POMEAU, *J. Phys. A* **17**, L415 (1984).
8. D STAUFFER, Institute for Theoretical Physics, University at Cologne, W.-Germany, private communication.
9. G. O WILLIAMS AND M H. KALOS, *J. Statist. Phys.* **37**, 283 (1984).
10. P. R. TAYLOR AND C. W. BAUSCHLICHER, to be published.
11. *Cray-2 Multitasking Programmer's Manual*, Cray Research, Inc., SN-2026 (unpublished).
12. G. F. NEWELL AND E. W. MONTROLL, *Rev. Mod. Phys.* **25**, 352 (1953); B. MCCOY AND T. T. WU, *The Two-Dimensional Ising Model* (Harvard Univ Press, Cambridge, MA, 1973).
13. G. BHANOT, M. CREUTZ, AND H. NEUBERGER, *Nucl. Phys. B* **235**, 417 (1984)
14. H. J HERRMANN, H. O. CARMESIN, AND D. STAUFFER, *J. Phys. A*, in press.