Cluster Identification Algorithms for Spin Models - Sequential and Parallel

Clive Baillie
Paul Coddington

CRPC-TR90160
December, 1990

Center for Research on Parallel Computa
Rice University
P.O. Box 1892
Houston, TX 77251-1892

CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 3(2), 129–144 (APRIL 1991)

Cluster Identification Algorithms for Spin Models — Sequential and Parallel

CLIVE F. BAILLIE* AND PAUL D. CODDINGTON†
Caltech Concurrent Computation Project
California Institute of Technology
Pasadena, CA 91125, U.S.A.

SUMMARY

Monte Carlo cluster update algorithms are extremely efficient for simulating spin models near their phase transitions, where local update algorithms suffer severe critical slowing down. Unfortunately, as the cluster algorithms are highly irregular as well as non-local, they are much more difficult to parallelize efficiently. The main difficulty lies in identifying which spins belong to which cluster. In this paper we investigate a number of cluster identification algorithms, both sequential and parallel, which we have implemented on serial, SIMD and MIMD computers.

1. INTRODUCTION

Monte Carlo simulations on both sequential and parallel computers are a very important numerical technique for investigating spin models in physics (see References 1 and 2 for reviews). Unfortunately, traditional Monte Carlo algorithms are afflicted with 'critical slowing down' near the regimes of interest in these models, namely at phase transitions. We shall explain critical slowing down below; for now, it basically means that the computational efficiency of the Monte Carlo simulation goes to zero as the size of the lattice is increased. Recently, however, algorithms which alleviate this problem have been invented[3–6]. In these so-called 'cluster' algorithms, clusters of spins (rather than single spins) are changed at each step of the Monte Carlo procedure. Thus, in order to implement these algorithms, a method for identifying clusters is needed. Cluster identification algorithms have existed for some time for sequential computers, but they are fairly new for parallel computers. Herein, after summarizing the standard sequential algorithms, we go on to describe in detail several parallel cluster identification algorithms for both SIMD and MIMD machines.

In simulations of spin models, the spins σ_i are usually set up on the sites i of a d-dimensional hypercubic lattice of length L. The L^d spins form some configuration. The goal of computer simulations is to generate spin configurations typical of statistical equilibrium and measure physical observables on this ensemble of configurations. The generation of configurations is traditionally performed by Monte Carlo methods. All Monte Carlo work has the same general structure: given some probability distribution

* Current address: Physics Department, University of Colorado, Boulder CO 80309, U.S.A.
† Current address: Physics Department, Syracuse University, Syracuse, NY 13244, U.S.A.

1040-3108/91/020129-16$08.00

Received 28 June 1990
Revised 13 December 1990

p, we wish to generate many random samples ϕ from p. For a statistical mechanical system such as a spin model the probability distribution we require is the Boltzmann distribution; hence $p = e^{-\beta H(\phi)}$, where $H(\phi)$ is the Hamiltonian, or energy, of the system in configuration ϕ, and $\beta = 1/k_B T$ is the inverse temperature, or coupling (in numerical computations we shall set the Boltzmann constant $k_B = 1$). The first application of these methods to problems in statistical mechanics was by Metropolis *et al.*[7].

The trouble in practice is that the samples $\phi_i, \phi_{i+1}, \ldots$ are not statistically independent, but rather are correlated. The statistical error in the Monte Carlo calculation behaves as $1/\sqrt{N}$, where N is the number of effectively independent samples. For T *correlated* samples, it can be shown[8] that the error is given by $1/\sqrt{(T/2\tau)}$, where τ is the autocorrelation time, which is a measure of how many iterations are required to produce an independent configuration. For a spin model with a phase transition, as the inverse temperature β approaches the critical inverse temperature β_c, τ diverges to infinity, so that the computational efficiency goes to zero! This behavior is called critical slowing down. Until very recently this problem has plagued Monte Carlo simulations of statistical mechanical systems, in particular spin models, at or near their phase transitions. The new cluster algorithms manage to avoid this critical slowing down—partially or even completely—thus facilitating much better computer simulations.

2. CLUSTER ALGORITHMS

The key feature about traditional (Metropolis-like) Monte Carlo algorithms is that the updates are *local*, that is, one spin at a time is updated. Thus in a single step of the algorithm, 'information' about the state of a spin is transmitted only to its nearest neighbors. Now, in order for the system to reach a new effectively independent configuration, this information must travel a distance of order the (static or spatial) correlation length ξ. As the information executes a random walk around the lattice, one would suppose the autocorrelation time $\tau \sim \xi^2$ near criticality. This can be shown analytically to be correct for a free-field (Gaussian) model. However, in general $\tau \sim \xi^z$, where z is called the dynamical critical exponent. All numerical computer simulations of spin models have measured $z \approx 2$ for local update algorithms[9]; the new cluster algorithms reduce z by performing *non-local* spin updates.

The aim of the cluster update algorithms is to find a suitable collection of spins which can be flipped with relatively little cost in energy. Note that we could obtain non-local updating very simply by using the standard Metropolis Monte Carlo algorithm to flip randomly selected bunches of spins, but then the change in energy would most likely be large and the acceptance tiny. Therefore we need a method which picks sensible bunches or clusters of spins to be updated. The first such algorithm was proposed by Swendsen and Wang[3], and was based on an equivalence between a Potts spin model[10] and percolation models[11] for which cluster properties play a fundamental role.

The Potts model is a very simple spin model in which the spins σ_i can take q different values, and whose Hamiltonian is

$$H = -\sum_{\langle i,j \rangle} \delta_{\sigma_i \sigma_j} \tag{1}$$

For $q = 2$ this is just the well-known Ising model[12]. In the Swendsen–Wang algorithm, clusters of spins are created by introducing bonds between neighboring spins with probability

$$P(\sigma_i, \sigma_j) = \delta_{\sigma_i,\sigma_j}(1 - e^{-\beta}) \tag{2}$$

All such clusters are generated and then updated by choosing a random new spin value for each cluster and assigning it to all the spins in that cluster.

A slightly different cluster algorithm has been proposed by Wolff[4]. In this algorithm, a spin is chosen at random and a single cluster constructed around it, using the same bond probabilities as for the Swendsen–Wang algorithm. All the spins in this cluster are then flipped (i.e. collectively changed to a random new spin different from the old one). Whereas Wolff's algorithm is undoubtedly the best method on a sequential computer[6], the algorithm of Swendsen and Wang seems to be better suited for parallelization, since it involves the entire lattice rather than just a single cluster.

3. CLUSTER IDENTIFICATION

Cluster algorithms have in common the problem of identifying and labeling the connected clusters of spins. This is very similar to an important problem in image processing, that of identifying and labeling the connected components in a binary or multi-colored image composed of an array of pixels. The only real difference is that in the spin model case, neighboring sites of the same spin have a certain *probability* of being in the same cluster, while for neighboring pixels of the same color that probability is 1. Unfortunately this is a large enough difference so that some algorithms which work in image analysis will not work, or require substantial changes, for spin models, for example the algorithm in Reference 13.

First we outline three sequential methods for labeling clusters, the so-called 'ants in the labyrinth' algorithm, the commonly used (especially for cluster identification in percolation models) algorithm of Hoshen and Kopelman, and an algorithm based on equivalence classes. We then present the results of a comparison of these algorithms on a Sun 4 workstation.

3.1. Ants in the labyrinth

This method is the most obvious one for identifying a single cluster of connected sites. The reason for its name is that we can visualize the algorithm as follows[14]. An ant is put somewhere on the lattice and notes which of the neighboring sites are connected to the site it is on. At the next timestep this ant places children on each of these connected sites which are not already occupied. The children then proceed to reproduce likewise until the entire cluster is populated. In order to label all the clusters, we start by giving every site a negative label, set the initial cluster label to be zero, and then loop through all the sites in turn. If a site's label is negative then the site has not already been assigned to a cluster, so we place an ant on this site, give it the current cluster label, and let it reproduce, passing the label on to all its offspring. When this cluster is identified we increment the cluster label and carry on, repeating the ant-colony birth, growth and death cycle until all the clusters have been identified. An example for a 4×4 lattice is shown

in Figure 1. Figure 1(a) shows the bonds connecting the sites in each cluster. The ant is initially placed at the site shown in Figure 1(b). After one generation the picture is as in Figure 1(c), and by the second generation (Figure 1(d)) the entire cluster has been identified.

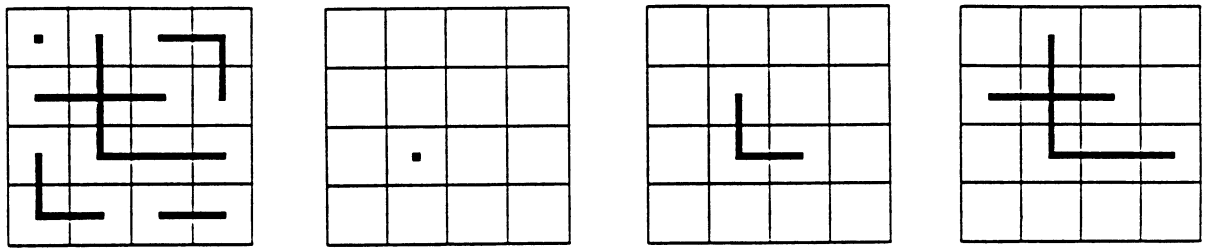

Figure 1. Example set of clusters, here identified by the 'ants in the labyrinth' algorithm: (a) the bonds connecting the sites in each cluster; (b) where the ant is initially placed; (c) after one generation; (d) after two generations

3.2. Hoshen and Kopelman

Hoshen and Kopelman[15] invented a sequential cluster identification algorithm which gives each cluster a unique label and counts the number of sites it contains. Each site i belonging to cluster α is assigned a cluster label m_i^α. The cluster α may initially be assigned several different cluster labels, which are given as a set of natural numbers $\{m_1^\alpha, m_2^\alpha, \ldots, m_s^\alpha, \ldots, m_i^\alpha, \ldots\}$. In this set only one number is regarded as the *proper* cluster label, which we designate as m_s^α. This is the smallest number in the set.

A connection between the label m_i^α at any site and the proper cluster label m_s^α is provided by an array N. This is constructed so that only $N(m_s^\alpha)$ is positive and denotes the number of sites in the cluster, while the remaining $N(m_i^\alpha)$ provide the links between the m_i^α and the proper cluster label. To be specific, if a site with label m_i^α is bonded to a site which has the proper cluster label m_s^α, then

$$N(m_i^\alpha) = -m_s^\alpha \tag{3}$$

However, if a site with label m_p^α is not directly bonded to a site with the proper cluster label, but is connected to a site with label m_i^α, then

$$N(m_p^\alpha) = -m_i^\alpha \tag{4}$$

Thus to get the proper cluster label for this site we have to go through m_i^α using (4) then (3). Similarly for higher-order indirections, we just need to iterate $-m \leftarrow N(m)$ until we reach a positive value of $N(m)$, which means that m is the proper cluster label and $N(m)$ the current number of sites in the cluster. Fortunately in most cases this hierarchy extends to one or two levels only.

In practice the algorithm works as follows. We sweep through the sites of the lattice looking at neighbors in the negative directions (there are d of them for a d-dimensional lattice). If site i has no connected neighbors (or none of its neighbors have been labeled),

then it is assigned a new label $S_i = m_t^\alpha$, and $N(m_t^\alpha) = 1$. If there is only one connected neighbor, at site n with label $S_n = m_r^\alpha$, say, then i gets the same label as n: $S_i = m_r^\alpha$, and $N(m_t^\alpha) = N(m_t^\alpha) + 1$. The good feature of this cluster labeling technique becomes apparent when site i links two or more previously labeled cluster fragments into a single cluster, that is, when it is bonded to two or more of its neighbors. No site belonging to any of these cluster fragments is relabeled (so that once a site is labeled it retains this label throughout the labeling process)—instead the readjustments occur within the $N(m_t^\alpha)$. The number of readjusted $N(m_t^\alpha)$s for a site is equal to the number of coalescing cluster fragments at that site. Let us assume that the connected sites belong to clusters $\alpha, \beta, \gamma, \ldots$ which have proper cluster labels $m_s^\alpha, m_s^\beta, m_s^\gamma, \ldots$, with m_s^α being the smallest. These clusters coalesce at i to form a single larger cluster, so we set label $S_i = m_s^\alpha$ and readjust:

$$\begin{aligned} N(m_s^\alpha) &= N(m_s^\alpha) + N(m_s^\beta) + N(m_s^\gamma) + \ldots + 1 \\ N(m_s^\beta) &= -m_s^\alpha \\ N(m_s^\gamma) &= -m_s^\alpha \\ &\ldots \end{aligned} \tag{5}$$

It is this continual readjustment which keeps the hierarchy of indirections small. As a final stage in the algorithm we can pass through the lattice a second time, setting the label at each site to be the proper cluster label m_s^α. This makes it easier to pick out the clusters for updating. An illustration of this algorithm for the 4×4 lattice used in Figure 1 is given in Figure 2. Figure 2(a) shows the labels assigned to the sites, after running the algorithm from the top left. Figure 2(b) shows the corresponding values of the connection array N. Finally, in Figure 2(c) we have the proper cluster labels; notice that (c) differs from (a) only where N is negative.

label

1	2	3	3
4	2	2	3
5	2	2	2
5	5	6	6

(a)

N[label]

1	7	3	3
-2	7	7	3
3	7	7	7
3	3	2	2

(b)

proper label

1	2	3	3
2	2	2	3
5	2	2	2
5	5	6	6

(c)

Figure 2. Hoshen–Kopelman algorithm for clusters in Figure 1(a). Note that the j axis runs from top to bottom. (a) labels assigned to the sites; (b) corresponding values of array N; (c) proper cluster labels

3.3. Equivalence class method

Identifying and labeling clusters of connected sites in a lattice is a special case of a more general problem known variously as the set union, union-find or equivalence problem, that is, given a list of equivalences between elements, sort the elements into equivalence

classes. In the context of cluster algorithms, the list of equivalences is just a list of the sites which are connected together, and the equivalence classes are just the clusters. There are a multitude of algorithms for this problem[16]; we have used an elegant and easy to code method due to Galler and Fisher[17].

To implement this algorithm, we first sweep through all the sites, creating a list of sites which are connected together. This list of equivalences is the input to the algorithm, which proceeds as follows. Let $F(j)$ be the class or 'family' label of element j (in our case, $F(j)$ is the cluster label of site j). We start off with each element in its own family, so that $F(j) = j$. The array $F(j)$ can be interpreted as a tree structure, where $F(j)$ denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all other 'family trees', then we can label each family by its most senior great-great-...grandparent, which will be the only element in the tree for which $F(j) = j$, since it has no parent. Therefore we process each equivalence of two sites j and k by:

(1) tracking j up to its highest ancestor by iterating $j \leftarrow F(j)$ until $j = F(j)$
(2) tracking k up to its highest ancestor by iterating $k \leftarrow F(k)$ until $k = F(k)$
(3) giving j to k as a new parent (or vice versa) by setting $F(j) = k$.

This process joins the 'branches' (or subtrees) containing j and k by giving them the same highest ancestor, thus putting them in the same equivalence class. After processing all the relations, we go through all the elements j and reset their $F(j)$s to their highest possible ancestors, by iterating $F(j) \leftarrow F(F(j))$ until $F(j) = F(F(j))$, so that the $F(j)$s are now the equivalence classes.

It is worthwhile noting that in step 3 we can choose to set either the first or the second label to be the highest ancestor when merging two branches of the tree. Which way we do it makes no difference to the result, but in practice it can make a big difference to the number of iterations required for each step in the algorithm, especially for a regular problem like the labeling of connected sites on a lattice, where some of the regularity is coded into the order of the elements in the list. We checked which method was faster, due to better balanced trees, and used that version.

3.4. Comparison of sequential algorithms

We have implemented the three sequential algorithms described above, and used them in a Swendsen–Wang update algorithm for the two-dimensional Potts model. As with local update algorithms, the time to update the lattice grows in proportion to the number of sites in the lattice, except for the equivalence class method, where the time is a slowly increasing function of the number of sites. The times taken on a Sun 4 workstation to update a 64^2 lattice are shown in Table 1, for the two-dimensional Ising ($q = 2$ Potts) model at various values of the inverse temperature β. The times are approximately the same for other values of q. The critical inverse temperature β_c of the phase transition for this model is known exactly: $\beta_c = ln(\sqrt{2} + 1) = 0.881374....$ Thus we know where to run in order to suffer the most severe critical slowing down. We also ran at $\beta = 0.4, 0.6, 0.8, 1.0$ and 1.2 in order to understand how the algorithms behave as the cluster size distribution changes. At large values of β, i.e. small temperatures, larger clusters are more favorable; conversely, at small β the clusters are smaller. For the Wolff

update, we have used the ants algorithm to grow the single clusters, so the update time *per site* is the same as for the Swendsen–Wang update using ants.

Table 1. Times in milliseconds for one sweep of a 64^2 lattice

β	Metropolis	ants	Hoshen–Kopelmann	F & G equiv.
0.4	179	224	263	264
0.6	184	219	263	285
0.8	192	211	286	354
β_c	198	202	301	466
1.0	198	197	326	659
1.2	200	191	318	617

We find that in our implementation, the ants algorithm is up to 50% faster than that of Hoshen–Kopelman. Although it is very general, elegant and easy to program, the equivalence class method is much slower than the other two algorithms, especially near the critical temperature. It is therefore never used to label clusters sequentially; however we will find a use for it in one of our parallel algorithms. Finally, notice that the cluster algorithms using ants have about the same update time as the Metropolis algorithm.

4. PARALLEL ALGORITHMS

Near criticality, which in most cases is where we want to perform simulations, clusters come in all sizes, from order L^2 (the number of sites in the lattice) right down to a single site. The highly irregular and non-local nature of the clusters means that cluster update algorithms do not easily vectorize, and hence give poor performance on vector machines. One processor of a CRAY X-MP runs the sequential code only about ten times faster than a Sun 4 workstation (the CRAY time is taken from Reference 18). We therefore also expect to get poor performance on SIMD computers.

On MIMD computers, by using the trivial parallelization technique of running independent Monte Carlo simulations on different processors, it is possible to do better than the standard code on a single CRAY X-MP processor using only about 20 nodes, if each node does better than about half an MFlop (such as a T800 transputer or an Ncube-2 processor). This method works well until the lattice size gets too big to fit into the memory of each node; we have actually used it to calculate the dynamical critical exponents of various cluster algorithms[19]. However, in the case of the Potts model, for example, only lattices of size less than about 300^2 or 45^3 will fit into 1 Mbyte, and most other spin models are more complicated and more memory intensive. We therefore need a parallel algorithm where a large lattice can be distributed over many processors of a parallel machine. This can be easily and efficiently done for local update algorithms, such as Metropolis; however, for cluster algorithms it is a much more difficult problem. The quality of non-locality which makes cluster algorithms so useful also makes them very difficult to parallelize efficiently, since this involves a large amount of non-local communication.

We firstly describe cluster identification algorithms for SIMD machines and then for

4.1. SIMD

It is not clear a priori whether the Wolff cluster algorithm (which requires identification of only one cluster) would be more efficient than Swendsen–Wang (which requires identification of all clusters) on an SIMD computer. For the former, the simplest cluster identification algorithm to use is ants in the labyrinth; for the latter one can use this or a rather obvious parallel algorithm (which is widely known but not, to our knowledge, published in the standard literature) which we shall call 'self-labeling'. We describe our implementation of these algorithms on the 1024-processor AMT Distributed Array Processor (DAP) 510 SIMD computer and give the performances obtained. We also mention an SIMD parallel algorithm which has been implemented on the TMC Connection Machine (CM) 2 computer.

4.1.1. Ants in the labyrinth

'Ants in the labyrinth' may be used to identify one cluster (for the Wolff algorithm) or all the clusters (for Swendsen–Wang), as described in Section 3.1. On the parallel computer, all of the ants in each generation are evolved simultaneously, thereby furnishing wavefront-like parallelism. Unfortunately the maximum number of processors in use at any time is only of the order of the square root of the total number of processors. We timed the two algorithms for the two-dimensional Ising model on the DAP, at various values of the temperature. As the Wolff algorithm updates only one cluster, whereas Swendsen–Wang does them all, we scale the Wolff times by the average cluster size divided by the size of the lattice. The times are shown in Table 2. We would expect that the update time per site would be the same for Wolff and Swendsen–Wang, since they are both using the same algorithm to identify the clusters. This is true sequentially, but not in parallel. This is because the Wolff algorithm has a larger average cluster size than does the Swendsen–Wang, and therefore more opportunity to exploit the limited parallelism available from the ants algorithm. This also explains why ants works much better at large β, where the average cluster size is large, than at small β, where the small cluster sizes severely limit the amount of parallelism which can be obtained.

Table 2. Times in milliseconds for one sweep of a 64^2 lattice on the DAP

β	Wolff ants	SW ants	SW s-l	SW s-l+comp
0.4	10480	4041	1319	1755
0.6	5244	3246	1308	1210
0.8	969	2086	1329	700
β_c	138	1157	1397	532
1.0	56	504	1330	278
1.2	47	263	1311	161

We also measured the time taken for the Wolff algorithm using ants as a function of lattice size; the results are in Table 3. Sequentially, the time to execute the ants algorithm scales linearly with the number of sites in the lattice: however this is not the case in

parallel. In fact, on the DAP, the times increase roughly as L^3 rather than L^2. We can explain this as follows. Firstly, we find that a cluster of given size takes four times longer to identify on a four times larger lattice. This is because each processor in the DAP has four times as many lattice sizes mapped into it and so takes four times longer to work through them. The other factor of two comes from the fact that on a larger lattice there are more larger clusters, which take longer to identify. Another way to think of this is that the wavefront parallelism is spread over fewer ($\log 4 = 2$ times less) processors.

Table 3. Times in milliseconds for one sweep at β_c on the DAP

Lattice size	Wolff ants	SW s-l+comp
64^2	138	532
128^2	1172	6554
256^2	10287	92327

4.1.2. Self-labeling

We refer to this algorithm as 'self-labeling', since each site figures out which cluster it is in by itself from local information. We begin by assigning each site i a unique cluster label S_i. In practice this is simply chosen as the position of that site in the lattice. At each step of the algorithm, in parallel, every site looks in turn at each of its neighbors in the positive directions. If it is bonded to a neighboring site n which has a different cluster label S_n, then both S_i and S_n are set to the minimum of the two. This is continued until nothing changes, by which time all the clusters will have been labeled with the minimum initial label of all the sites in the cluster. Note that to check termination of the algorithm a global logical OR of all the individual termination flags on each processor must be performed—an operation which is usually very efficiently implemented on SIMD computers. An illustration of this algorithm for a 4×4 lattice is given in Figure 3 for the cluster shown in Figure 1(a). In Figure 3(a) we see the original cluster labels. It takes three iterations of the algorithm (plus one to verify that we have finished) to uniquely label the clusters, shown in Figures 3(b),(c),(d). Note that our implementation firstly checks bonds to the north then bonds to the east.

We implemented self-labeling for the Swendsen–Wang algorithm applied to the two-dimensional Ising model on the DAP. We tested two versions of self-labeling, as there is actually an optimization which one can do to enhance its performance when there are only a few large clusters, i.e. at large β. To begin self-labeling we assign each lattice site a unique label, and hence we have 4096 labels. After running the algorithm we are left with, say, N labels identifying the clusters present. However, these N labels are scattered throughout the 4096 possible values. Since we do not know which values will turn up we generate 4096 new random spins. Obviously, if N is much smaller than 4096 it would be more efficient to first look through the labels and only generate new spins for the N which are being used. We call this 'compressing the labels' and the version of self-labeling which uses it 'compressed self-labeling'. Our timings for the two implementations of Swendsen–Wang using self-labeling are in Table 2.

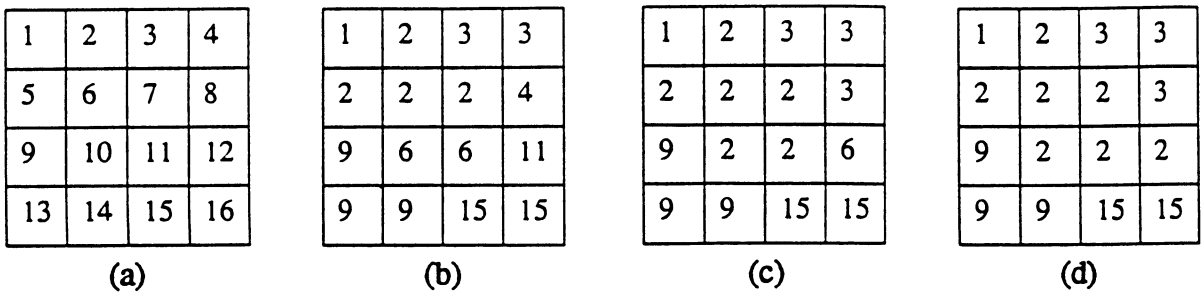

Figure 3. Self-labeling algorithm for cluster in Figure 1(a). Labels are compared firstly with the neighbor to the north, then to the east: (a) original cluster labels; (b) first iteration; (c) second iteration; (d) third iteration

Interestingly, the time to execute Swendsen–Wang using the original self-labeling implementation is the same irrespective of β. This is almost certainly because most of the time is being spent generating the 4096 new random spins. That this is the case seems confirmed by the fact that compressing the labels does indeed help at large β. In fact, at the phase transition, Swendsen–Wang using the compressed self-labeling algorithm is roughly a factor of two faster than Swendsen–Wang using 'ants in the labyrinth'; however, it is still almost a factor of four slower than Wolff using 'ants in the labyrinth'. The main reason for this is that the DAP computer is much faster handling the logical mask bits needed to identify the single cluster in the ants algorithm, rather than the integer labels required in self-labeling.

We also measured the time the Swendsen–Wang algorithm using compressed self-labeling took as a function of lattice size; the results are in Table 3. Sequentially, the time taken on a four times bigger lattice is about eight times longer, since there is a factor of L^2 for the number of sites, and the number of iterations needed to label the largest cluster is approximately $2L$. In parallel, on the DAP, we find this factor is more like 12 or 14, rather than eight. We are not certain as to what is causing this extra unexpected increase in the time required.

In the final analysis, all the cluster algorithms perform rather poorly on the DAP; in fact the Swendsen–Wang algorithms execute in less wall clock time on a Sun 4 workstation. This is primarily due to lack of parallelism in the ants algorithm and load imbalance in self-labeling, where most of the processors are waiting on the few finishing identifying the largest cluster. Of course, on the standard Metropolis algorithm the DAP is orders of magnitude faster than the Sun.

4.1.3. Other algorithms

Recently, Brower *et al.* have also investigated parallel cluster algorithms for SIMD machines[20] They have implemented a self-labeling algorithm (which they call 'local diffusion') and a non-local, multi-grid-style algorithm on the TMC Connection Machine, for the Swendsen–Wang algorithm applied to the 2-D Ising model. As expected, the non-local method takes fewer iterations, and performs better than the local algorithms. A number of other SIMD algorithms have been proposed for component labeling in image analysis applications[13,21–23], and the algorithm of Lim, Agrawal and Nekludova[21] has been implemented on the Connection Machine. This algorithm identifies the clusters

by finding the boundary of each cluster, and then giving the same label to each site on the boundary. Once this has been done, filling in the interior of each cluster can be done easily and quickly. We have not yet applied any of these algorithms to spin models; however, we are aiming to investigate them in the future, since they should perform better than the much simpler ants and self-labeling algorithms, which we have found to be very inefficient.

4.2. MIMD

On an MIMD parallel computer, such as the Symult 2010, a parallel cluster algorithm involves distributing the lattice on to an array of processors using the usual domain decomposition[24]. Clearly a sequential algorithm can be used to label the clusters on each processor, but we need a procedure for converting these labels to their correct *global* values. We must be able to tell many processors, which may be any distance apart, that some of their clusters are actually the same. Thus we need to be able to agree on which of the many different local labels for a given cluster should be assigned to be the global cluster label, and to pass this label to all the processors containing a part of that cluster. We will discuss two methods we have used for tackling this problem—'self-labeling' and 'global equivalencing'—and give timings and efficiencies for our implementation of these two algorithms on various MIMD computers. In fact, since our code is written in terms of the ParaSoft Express System[25], it runs portably on the Symult 2010, Ncube hypercube, Meiko Computing Surface and Caltech/JPL Mark III hypercube computers. We also mention some other algorithms which have been proposed for this problem.

4.2.1. Self-labeling

Self-labeling on an MIMD computer is essentially the same as on an SIMD machine. The only difference is that the termination check involves each processor sending its termination flag to every other processor after each step, i.e. a broadcast operation has to be performed. On most machines, like hypercubes, for example, this is efficiently implemented (so as to use the logarithmic combining feature of the hypercube network, for example); however, this step can still be quite time consuming. This overhead can be greatly reduced by only checking for termination after a certain number of steps, which can be taken to be slightly less than the average number of steps for the particular problem.

As described in Section 4.1.2, this is a purely SIMD algorithm, and if implemented in this way it would suffer from the same problem as for SIMD machines, which give very poor performance due to a large load imbalance, with most processors waiting for the few in the largest cluster which are the last to get the label for that cluster. However, on an MIMD machine we can improve this method by using a faster sequential algorithm, such as ants, to label the clusters in the sublattice on each processor, and then just use self-labeling on the sites at the edges of each processor to eventually arrive at the global cluster labels.

The number of steps required to do the self-labeling will depend on the largest cluster, which at the phase transition will generally span the entire lattice. The number of

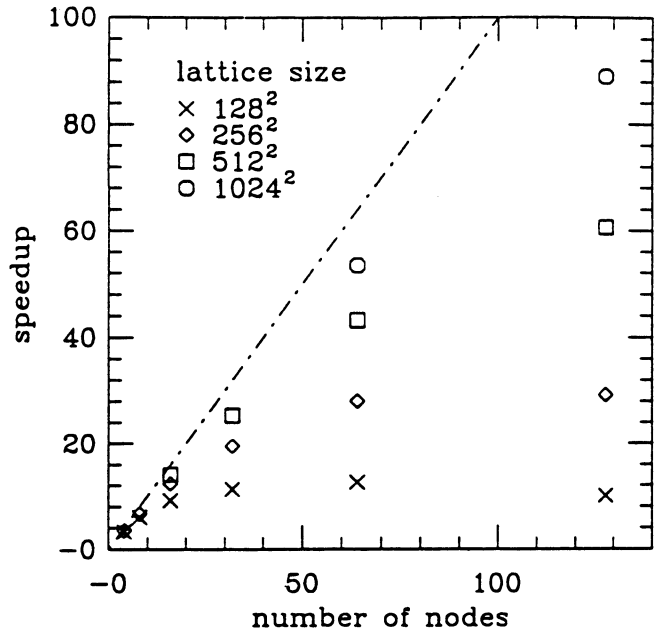

Figure 4. Speed-ups on the Symult 2010 for self-labeling

self-labeling steps will therefore be of the order of the maximum distance between processors, which for a square array of P processors is just $2\sqrt{P}$. Hence the amount of communication (and calculation) involved in doing the self-labeling, which is proportional to the number of iterations times the perimeter of the sublattice, goes like L for an $L \times L$ lattice, whereas the time taken on each processor to do the local cluster labeling goes like the area of the sublattice, which is L^2/P. Therefore as long as L is substantially greater than the number of processors (which is generally the case) we can expect to obtain a reasonable speed-up.

Table 4. Timings and efficiencies for one sweep of a 512^2 lattice at β_c using 32 processors

Machine	Time (s)	Efficiency
Ncube-1	4.45	0.79
Symult	1.39	0.79
Meiko	1.23	0.72

The speed-ups obtained on the Symult 2010 for a variety of lattice sizes are shown in Figure 4. The dashed line indicates perfect speed-up (i.e. 100% efficiency). The lattice sizes for which we actually need large numbers of processors are of the order of 512^2 or greater, and we can see that running on 64 nodes (or running multiple simulations of 64 nodes each) gives us quite acceptable efficiencies of about 70% for 512^2 and 80% for 1024^2. Using all 192 nodes of Caltech's Symult 2010 in this way gives a performance of approximately one million spin updates per second, which is about six times that of one processor of a CRAY X-MP[18], and twice that of the current best algorithm on a full sized (64K processor) Connection Machine[20]. Timings and efficiencies for self-labeling

on 32 nodes of three different MIMD computers — the Ncube-1, the Symult 2010 and the Meiko Computing Surface—are given in Table 4.

4.2.2. Global equivalencing

In this method we again use the fastest sequential algorithm to identify the clusters in the sublattice on every node. Each node then checks to see which of the edge sites of its sublattice are connected to edge sites on the neighboring nodes in the positive directions, and are therefore part of the same cluster and should be given the same cluster label. These lists of 'equivalences' are all passed to one of the nodes, which uses the equivalence class algorithm of Fisher and Galler[17] to match up the connected subclusters, and then broadcasts the global cluster labels to all the other nodes.

The problem here is that the equivalencing is purely sequential, and is thus a potentially disastrous bottleneck for large numbers of processors. The amount of extra work and communication involved goes like the number of processors P times the perimeter of the sublattice on each node, i.e. like $L\sqrt{P}$, so that the efficiency should be less than for self-labeling, although we might still expect reasonable speed-ups if the number of nodes is not extremely large. The speed-ups obtained for this algorithm on the Symult 2010 for a variety of lattice sizes are shown in Figure 5. Global equivalencing gives about the same speed-ups as self-labeling for small numbers of processors, but as expected self-labeling does much better as the number of nodes increases.

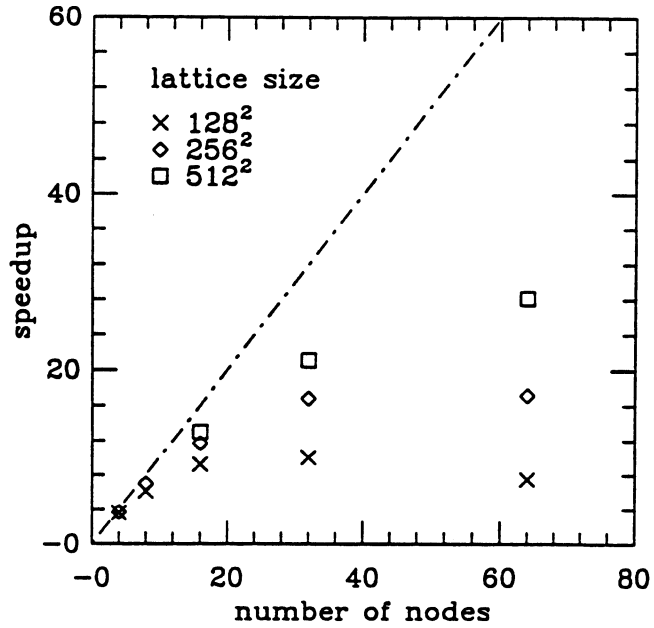

Figure 5. Speed-ups on the Symult 2010 for global equivalencing

To get around the sequential bottleneck in the global equivalencing step, we need to adopt a hierarchical divide-and-conquer approach. In this hierarchical equivalencing the processor array is divided up into smaller subarrays of, for example, 2×2 processors. Each subarray performs the global equivalencing algorithm on its section of the lattice. The results of these partial matchings are then combined on each 4×4 subarray, and this

process is continued until finally all the partial results are merged together to give the global cluster values. In this way the number of processors performing the equivalencing step is $P/4$ for the first level of the hierarchy, $P/16$ for the second level, and so on, until the final stage is done on a single processor. However, by that time most of the work has been done, so the bottleneck has been at least partially alleviated.

A very similar algorithm which uses the same hierarchical procedure has been implemented on the iPSC-2 hypercube for the image processing component labeling problem by Embrechts *et al.*[26].

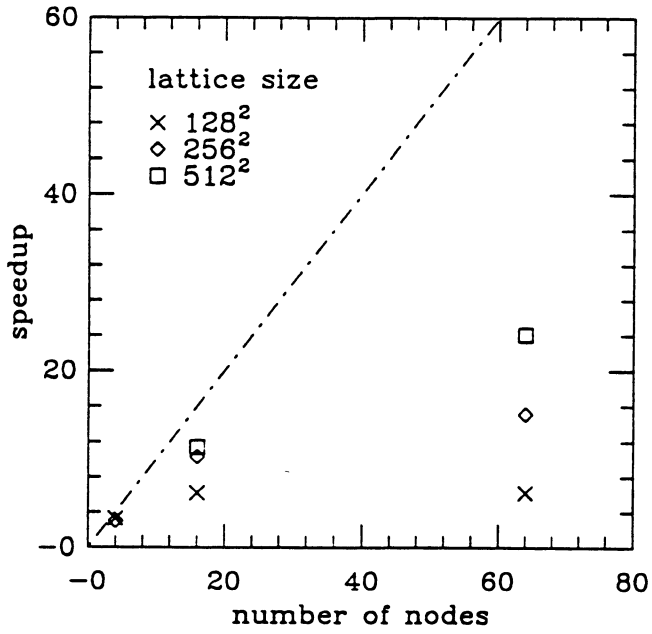

Figure 6. Speed-ups on the Symult 2010 for hierarchical equivalencing

The speed-ups for hierarchical equivalencing on the Symult 2010 are shown in Figure 6. Results are given only for 4, 16 and 64 processors, since the algorithm requires the number of processors to be an even power of two. The results are rather poor, and much worse than for self-labeling; however, this is a preliminary implementation of a quite complicated algorithm, and there are many parts of the program which could be improved upon. We are hopeful that an optimized version of this algorithm will do better than self-labeling, at least for large numbers of processors.

4.2.3. Other algorithms

Currently the only other MIMD cluster algorithm proposed for spin models is a parallel extension of the Hoshen and Kopelman algorithm[15] due to Burkitt and Heermann[27], which has been implemented on a transputer array. Their algorithm is much more complicated, and less efficient, than the self-labeling algorithm, giving speed-ups for a 512^2 lattice of approximately 11.5 and 11.0 on 16 and 32 processors, respectively.

Some other MIMD algorithms have been proposed for the component labeling problem, both for shared[28] and distributed[29,30] memory architectures. Further investigation is

needed to see if these algorithms might be applied to the problem of producing an efficient parallel cluster algorithm for spin models on larger numbers of processors.

5. CONCLUSIONS

We have described several cluster identification algorithms for both sequential and parallel computers. Of the three sequential algorithms, we have found that 'ants in the labyrinth' and Hoshen–Kopelman are much faster than the more general equivalence class method. Since 'ants in the labyrinth' is faster and slightly easier to program than Hoshen–Kopelman, we would reccommed it as the best available sequential algorithm.

Parallel algorithms may be divided into two categories—SIMD and MIMD. On SIMD computers we have found that it is indeed very difficult to efficiently implement cluster identification algorithms. However, of the three possibilities which we tried, 'ants in the labyrinth' for the Wolff algorithm (where only one cluster is identified) is the fastest on the DAP. In the future we need to investigate some of the many algorithms which have been proposed for the component labeling problem, which may provide a more efficient implementation of cluster algorithms on SIMD machines. Finally, for MIMD computers we have proposed two different types of algorithm—self-labeling and global equivalencing—with self-labeling being the more efficient of the two at the present time, although that may change since the implementation of hierarchical equivalencing can certainly be improved upon. We were able to implement the self-labeling algorithm fairly efficiently for the problem sizes of interest on up to about 64 nodes, for a number of different MIMD machines. Finding an efficient algorithm for very large numbers of processors seems to be a very difficult problem, although it can be avoided in most cases by running multiple independent simulations of 32 or 64 nodes each.

Note that all the results given above were for the simplest of spin models, the two-dimensional Potts model. For more computationally intensive spin models, or for gauge theories, the amount of calculation involved per processor should increase much more than the amount of communication for the parallel algorithms, and consequently we would expect to implement them efficiently on larger numbers of processors.

ACKNOWLEDGEMENTS

This work was partially supported by DOE Grants DE-FG03-85ER25009 and DE-AC03-81ER40050.

REFERENCES

1. *Monte Carlo Methods in Statistical Physics, Topics in Current Physics 7, 2 Edn*, K. Binder (ed.), Springer-Verlag, Berlin, 1986; *Applications of the Monte Carlo Method in Statistical Physics, Topics in Current Physics 36, 2 Edn*, K. Binder (ed.), Springer-Verlag, Berlin, 1987.
2. C. F. Baillie, *Int. J. Mod. Phys. C*, **1**, 91 (1990).
3. R. H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.*, **58**, 86 (1987).
4. U. Wolff, *Phys. Rev. Lett.*, **62**, 361 (1989).
5. A. D. Sokal, 'New numerical algorithms for critical phenomena (multi-grid methods and all that)', in *Computer Simulation Studies in Condensed Matter Physics, Springer Proceedings in Physics 33*, D. P. Landau, K. K. Mon and H-B. Schuttler (eds.), Springer-Verlag, Berlin, 1988.

6. U. Wolff, *Proc. of the Int. Workshop 'Lattice 89'*, Capri, Sept. 1989, *Nucl. Phys. B* (Proc. Suppl.), **17**, 93 (1990).
7. N. Metropolis *et al.*, *J. Chem. Phys.*, **21**, 1087 (1953).
8. N. Madras and A. D. Sokal, *J. Stat. Phys.*, **50**, 109 (1988).
9. S. Tang and D. P. Landau, *Phys. Rev. B*, **36**, 567 (1987).
10. R. B. Potts, *Proc. Camb. Phil. Soc.*, **48**, 106 (1952); F. Y. Wu, *Rev. Mod. Phys.*, **54**, 235 (1982).
11. D. Stauffer, *Phys. Rep.*, **54**, 1 (1978); J. W. Essam, *Rep. Prog. Phys.*, **43**, 830 (1980).
12. E. Ising, *Z. Physik*, **31**, 253 (1925).
13. R. Cypher, J. L. C. Sanz and L. Snyder, 'Practical Algorithms for Image Component Labeling on SIMD Mesh Connected Computers', in *Proc. 1987 Int. Conf. on Parallel Processing*, pp. 772–779.
14. R. Dewar and C. K. Harris, *J. Phys. A*, **20**, 985 (1987).
15. J. Hoshen and R. Kopelman, *Phys. Rev. B*, **14**, 3438 (1976).
16. R. E. Tarjan and J. van Leeuwen, *J. ACM*, **31**, 245 (1984).
17. B. A. Galler and M. J. Fisher, *Commun. ACM*, **7**, 301 (1964); D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1 Addison-Wesley, Reading, 1968; W. H. Press *et al.*, *Numerical Recipes in C; The Art of Scientific Programming*, Cambridge University Press, Cambridge, 1988.
18. U. Wolff, *Phys. Lett.*, **B228**, 379 (1989).
19. P. D. Coddington and C. F. Baillie, *Proc. of the Int. Workshop 'Lattice 89'*, Capri, Sept. 1989, *Nucl. Phys. B* (Proc. Suppl.), **17**, 305 (1990).
20. R. C. Brower, P. Tamayo and B. York, 'A parallel multigrid algorithm for percolation clusters', Boston University preprint, submitted to *J. Stat. Phys.*
21. W. Lim, A. Agrawal, L. Nekludova, 'A fast parallel algorithm for labeling connected components in image arrays', Thinking Machines Corporation Technical Report NA86-2.
22. D. Nassimi and S. Sahni, *SIAM J. Comput.*, **9**, 744 (1980).
23. M. Manohar, *Computer Vision, Graphics and Image Processing*, **45**, 133 (1989).
24. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, Solving Problems on Concurrent Processors, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
25. *Express: A Communication Environment for Parallel Computers*, ParaSoft Corporation, Mission Viejo, CA, 1988.
26. H. Embrechts, D. Roose and P. Wambacq, 'Component labeling on a distributed memory multiprocessor', *Proc. First European Workshop on Hypercube and Distributed Computers*, F. Andre and J. P. Verjus (eds.), North-Holland, Amsterdam, 1989; H. Embrechts and D. Roose, 'Efficiency and load balancing issues for a parallel component labeling algorithm', *Proc. Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Monterey, 1989.
27. A. N. Burkitt and D. W. Heermann, *Comput. Phys. Commun.*, **54**, 210 (1989).
28. R. Hummel, 'Connected component labeling in image processing with MIMD architectures', in *Intermediate-Level Image Processing*, M. J. B. Duff (ed.), Academic Press, New York, 1986.
29. R. Cypher, J. L. C. Sanz and L. Snyder, *J. Algorithms*, **10**, 140 (1989).
30. J. Woo and S. Sahni, *J. Supercomput.*, **3**, 209 (1989).