Parallel Wolff Cluster Algorithms

S. Bae and S. H. Ko

School of Computer Science, Syracuse University,

111 College Place, Syracuse, NY 13244, U.S.A.

P. D. Coddington

Northeast Parallel Architectures Center, Syracuse University,

111 College Place, Syracuse, NY 13244, U.S.A.

March 28, 1994

Abstract

The Wolff single-cluster algorithm is the most efficient method known for Monte Carlo simulation of many spin models. Due to the irregular size, shape and position of the Wolff clusters, this method does not easily lend itself to efficient parallel implementation, so that simulations using this method have thus far been confined to workstations and vector machines. Here we present two parallel implementations of this algorithm, and show that one gives fairly good performance on a MIMD parallel computer.

1 Introduction

Monte Carlo simulation is widely used in studying the statistical mechanics of spin models of magnetism [1]. Unfortunately, traditional Monte Carlo algorithms for these models, such as the commonly used Metropolis algorithm [1, 2], suffer from critical slowing down [1, 3], which means the computational efficiency of these methods decreases rapidly as the size of the system is increased. Algorithms have been developed by Swendsen and Wang [4] and Wolff [5] that make large-scale, non-local changes in the spin configurations, and greatly reduce critical slowing down for certain types of spin models. In these so-called cluster algorithms, clusters of spins (rather than single spins) are collectively updated at each step of the Monte Carlo procedure. The clusters are formed by generating bonds connecting neighboring sites, using a probabilistic procedure that varies between different spin models and algorithms (for reviews of cluster algorithms, see Refs. [3, 6, 7, 8]).

Here we will consider the simplest of spin models, the two dimensional Ising model [1], for which the spin at any site of the lattice can take only two values, $+1$ or -1. The parallel algorithms we will present are, however, easily generalizable to any dimension and any spin model for which a Wolff cluster algorithm can be defined.

In the Swendsen-Wang algorithm for the Ising model, bonds are placed with probability $1 - e^{-2\beta}$ between all neighboring sites of the lattice that have the same spin value, where β is the inverse temperature parameter. This procedure creates clusters of bonded sites of the same spin. The spins are updated by assigning a random spin value to each cluster.

In the Wolff algorithm, only a single cluster is created. A site is chosen at random and a cluster is constructed around it, using the same bond probabilities as for the Swendsen-Wang algorithm. All the spins in this cluster are then flipped, that is, collectively changed to the alternate spin value.

The performance of a Monte Carlo algorithm is best measured in terms of statistically independent configurations produced per unit (computer) time. Using this measure, cluster algorithms can easily outperform the Metropolis algorithm on sequential computers, with the Wolff algorithm generally better than Swendsen-Wang [5, 7, 9, 11, 12, 13, 14]. However this may not be the case on supercomputers, which have vector and/or parallel architectures. Very efficient vector and parallel programs can be written for the Metropolis algorithm, since it is regular and local. Cluster algorithms, which are highly irregular and non-local, are much more difficult to implement efficiently on vector and parallel machines. Unless efficient vector or parallel cluster algorithms can be developed, cluster algorithms may give inferior performance to the Metropolis algorithm on supercomputers [9].

2 Computational algorithms

The major computational task for the Swendsen-Wang algorithm is the identification and labeling of the clusters of connected sites, given the configuration of bonds. This is an instance of a connected component labeling problem for an undirected graph [10], where the vertices are the lattice sites and the edges are the bonds between connected sites. The goal of the component labeling algorithm is to end up with the same label on all connected sites, and different labels for all disconnected clusters.

A number of methods have been developed for implementing the Swendsen-Wang algorithm on parallel [15, 16, 17, 18, 19, 20, 21, 22] and vector [23] computers. Most of these methods are variations on parallel algorithms that were developed for the general problem of connected component labeling of a graph [24, 25, 26]. Parallel Swendsen-Wang algorithms can be quite efficient, especially for large lattice sizes on coarse-grain MIMD machines. The Swendsen-Wang algorithm seems inherently better suited to a parallel implementation than the Wolff algorithm, since clusters are constructed for all the sites in the lattice, so a data parallel algorithm using standard domain decomposition [29] can be used. This is not the case for the Wolff algorithm, for which only a (random) subset of the sites are updated at every iteration, so a standard domain decomposition would give extremely poor load balance between processors.

Constructing (or labeling) a single cluster of connected sites is done sequentially by a simple breadth-first-search algorithm [5, 10, 17, 27, 28]. Two data structures are used: a list `C` of sites that belong to the cluster;[1] and a queue `Q` of sites that have been added to the cluster but not yet used in the search. The algorithm is described by the pseudo-code in Fig. 1. The sites in the queue can be visualized as a "wavefront" expanding outwards from the initial site. It is possible to rewrite the loop over sites in the queue as a double loop: one loop over all sites in a particular wavefront, and another loop over successive wavefronts. We will refer to these successive wavefronts as *generations* of sites, with the first site being the parent (the first generation), the set of sites connected by a bond to this site being its children (the second generation), the set of sites connected to these children (and not already part of the cluster) being the next generation, and so on [17, 27]. In this algorithm, the loop over all sites in a particular generation can be done in parallel [17, 27, 28]. We implement this

[1]It is actually more efficient to store this information as a logical array with a TRUE or FALSE value for each site in the lattice, to save searching the list, but here we will consider it as a list of sites in the cluster.

parallel breadth-first-search by splitting the queue `Q` into two data structures, the current generation `G` and the next generation `G'`. The algorithm is described by the pseudo-code in Fig. 2.

For the Wolff algorithm only nearest-neighbor information is required, so the communication overhead for the parallel algorithm is not great. However this method performs poorly on the type of fine-grain, massively parallel SIMD machine for which it was originally proposed [27]. This is because at any point in the algorithm, the current generation of sites will occupy at most $O(L)$ sites of a lattice of size $L \times L$, and much less than that in the early generations when the cluster is small (see Fig. 3). If there is a single processor per lattice site, the load balance will be extremely poor. The situation is little better for a coarse-grain parallel machine with many sites per processor, as can be seen in Fig. 3. This parallel breadth-first-search algorithm is well suited to a vector machine, for which the degree of parallelism (the vector length of the machine) is much smaller [28]. However on a Cray YMP this method achieves speed-ups of less than 10 over non-vectorized algorithms that realize only the scalar performance of the machine [11, 28].

Methods for implementing the Wolff algorithm in parallel on MIMD machines have so far been confined to the trivial parallelism of running independent simulations, with different parameters or random number streams, on different processors [16, 17]. In this case the size of the system that can be simulated is limited by the memory and speed of a single node of the parallel machine. In the next two sections we introduce two methods for parallelizing a single simulation over many processors using domain decomposition. The second method performs well on coarse-grain MIMD parallel computers.

3 The single-cluster Swendsen-Wang algorithm

The Swendsen-Wang algorithm requires the identification of clusters of connected sites. This is done using a connected component labeling algorithm to give each site a number which labels the cluster to which it belongs. All of the parallel connected component labeling algorithms use an iterative procedure for updating the cluster label of each site. At the end of each iteration, a test is performed to see if the label of any site has been changed. If no change has been made, then all sites have received the label of their cluster, and the procedure terminates.

A parallel Wolff algorithm can be implemented using a simple change to a parallel Swendsen-Wang algorithm, which creates a single-cluster version of the algorithm. As in the sequential Wolff algorithm, a random site is chosen, around which the cluster will grow.

The iterative procedure mentioned above remains the same, however now the termination condition is not that the labels of *all sites* are unchanged, but rather the labels of *all sites in the single cluster containing the initial random site* are unchanged. This allows all sites of the Wolff cluster to be worked on in parallel, rather than just the expanding wavefront. However in this method the computation is being done for all clusters at the same time, not just the single Wolff cluster, so there is a lot of wasted effort.

This method was implemented using a SIMD Swendsen-Wang program [21] written in CMFortran [30] for the Thinking Machines CM-5 computer. We had planned to also implement this method in a message passing language using a MIMD programming model, which can better handle this type of irregular problem and is known to provide much better performance than SIMD algorithms [17, 21]. However we instead developed and implemented an improved MIMD algorithm.

4 The scattered strip partitioning algorithm

Parallel algorithms with local data dependencies generally use a regular domain decomposition in order to minimize communications. However for some dynamic irregular problems, the computational load tends to be concentrated in different regions of the domain at different times. In order to balance the load, some kind of dynamic data partitioning is often required. However in some cases a much simpler static approach can be adopted, in which the data in neighboring regions is scattered among the processors. This scattered domain decomposition has a higher communications cost since data locality is sacrificed, however this may be more than compensated for by the improvement in load balance when the computational load is concentrated in a certain region of the domain, since now all processors contain part of that domain. This method has been very effective in many parallel algorithms, including certain types of matrix solvers, irregular finite element problems, and problems with dynamic data distributions [29].

Scattered domain decomposition has also been effective in a parallel implementation of Lee's maze routing algorithm [31]. Maze routing is an algorithm used for routing wires in VLSI circuits [32]. The model is basically that of a regular grid where some paths are blocked and some are allowed. The problem is to find the shortest allowed path to place a wire between two points. The maze routing algorithm performs a breadth-first-search of the space, starting from the initial point, and following all allowed paths until one path reaches the final point. It is therefore very similar to the growth of a cluster in the Wolff cluster algorithm.

Fang et al. [31] found that the most effective domain decomposition for a parallel maze routing algorithm was scattered strip partitioning (this is often referred to as a cyclic or column-cyclic data distribution, for example in the High Performance Fortran language specification [33]). In this scheme, the lattice is split into N partitions, where N is chosen to be an integer multiple of the number of processors P to ensure load balancing. Each partition contains W columns of the lattice (we shall refer to W as the *partition width*). Thus for a lattice of size $L \times L$, $L = WN$ and W must be in the range $1 \leq W \leq L/P$. Partition M is assigned to processor M mod P (see Fig. 4). The mapping of data to processors is:

- Global lattice coordinate (i, j) is assigned to processor $(j \bmod WP) \operatorname{div} W$
- Local lattice coordinate (a, b) is given by

  $a = i$

  $b = (j \ \mathrm{mod} \ WP) \ \mathrm{mod} \ W \ + \ W \times \ (j \ \mathrm{div} \ WP)$

An example of the distribution of data after a scattered strip partitioning is shown in Fig. 5. The load balance is still far from perfect, but it is much better than for the standard domain decomposition in Fig. 3. As W becomes smaller, we can expect better load balancing, but a larger communications overhead due to non-locality of data. It is therefore important to find the optimal partition width W.

We have implemented this algorithm in a MIMD message-passing paradigm. The following is an outline of the steps required:

1. **Lattice partitioning**

   Do a scattered strip (column-cyclic) partitioning of the lattice, so that the portion of the lattice allocated to each processor has $L/(WP)$ partitions of width W, i.e. L/P columns and L^2/P sites in total.

2. **Selection of an initial random site**

   One of the processors (processor 0) selects the initial random site and broadcasts the information to the other processors.

3. **Local expansion of the cluster**

   Each processor maintains a queue that stores those sites in the local lattice that are in the current generation, and also maintains a communication buffer to collect all communication requests.

   Each local site in the current generation is fetched from the queue, and used as a parent site to expand the cluster.

Bonds are made with the appropriate probability between the parent site and all its neighboring sites, unless the neighboring site is already an element of the cluster.

At this time, if the parent site is on the boundary (the left or right edge) of a partition, communication with the left or right processor may be needed. Such communication requests are saved into the communication buffer for collective communication.

All local connected sites are added to the cluster and the queue for the next generation of sites, and the spin values at these sites are updated.

4. **Collective communication**

   Any communication requests are collected from the communication buffer and sent to the destination processors.

5. **Remote expansion of the cluster**

   Each site in the received buffer is checked to see whether it is already an element of the cluster.

   If not, the site is added to the local cluster list and the queue for the next generation, and its spin value is updated.

6. **Check for Termination**

   The final step checks whether there are any sites in the new generation. This can be done easily by using a global reduction function such as logical AND. If every processor has the empty queue, the algorithm halts. If not, steps 3, 4 and 5 are repeated.

The parallel code to implement this message passing program was written in C for the CM-5, using CMMD [35] reduction functions for global communications (steps 2 and 6) and Active Messages [35, 36] for point-to-point communications (step 4).

This algorithm should be easily implementable in High Performance Fortran (HPF) using a column-cyclic data distribution [33]. The HPF program would be much simpler than the message passing program we have implemented, since there would be no need to set up communications buffers and the reading and writing of messages to access non-local data – this would be handled by the compiler.

5 Results

The two algorithms have been run on a 32-node Thinking Machines CM-5 using lattices of size 128^2 to 2048^2. The results were compared with a sequential program written in C and run on a single node of the CM-5. The inverse temperature β was taken to be the critical inverse temperature $\beta_c = \log(1+\sqrt{2})/2 \approx 0.4406868$, where the clusters are most irregular in size and shape.

To measure the average execution time, we performed 100 iterations to thermalize the system, and then measured the execution time for at least 100 (and usually 1000) iterations. The time for each iteration depends on the size of the cluster that is grown, which can be any size from a single site to many thousands of sites. In order to make a sensible comparison to other Monte Carlo algorithms such as Metropolis and Swendsen-Wang, for which every site is updated at each iteration, the execution time for the Wolff algorithm is given as the time per spin update, rather than the time per iteration.

We measured the performance using both the scalar processor and the vector processors on the CM-5. As expected, using the vector units greatly improves the performance of the SIMD algorithms, so the results shown for these are using the vector units. However, for the sequential and MIMD algorithms, the vector units are not effective, and in fact the program runs slower if they are used, presumably due to the overhead of loading the data onto the vector registers. For these algorithms the times used are without vector units.

For the scattered strip partitioning algorithm, we tested various partition widths W to find the optimal width for each lattice size. Fig. 6 shows the scaled time for different values of W. The performance is very sensitive to the partition width. Larger widths mean smaller communications overhead, but greater load imbalance. Since load imbalance is less of a problem for larger generation sizes (and hence larger lattice sizes), the optimal width starts at $W = 1$ for small lattices and slowly increases with lattice size.

The timings and efficiencies (taken relative to the sequential Wolff algorithm run on a single processor) for the single-cluster Swendsen-Wang algorithm, the full Swendsen-Wang algorithm from which it is derived, and the scattered-strip partitioning algorithm with $W = 2$, are shown in Table 1 for a 32 node CM-5.

Reasonable speed-ups of around 10 are obtained for the MIMD algorithm for the larger lattice sizes. The corresponding efficiencies of around 35% are not particularly good, but perhaps better than expected for this highly irregular and dynamic problem.

Even using the vector units, the efficiencies for the SIMD algorithms are very low. This is partly because this irregular problem is not well-suited to SIMD methods, but partly because the CMFortran compiler is not very efficient at implementing this type of algorithm, for which the ratio of computation to communication is low. The same SIMD algorithms programmed in a message-passing paradigm using CMMD would be much more efficient.

The amount of parallelism that can be extracted from the scattered-strip partitioning algorithm depends on the average number of sites in each generation. Fig. 7 shows the average generation (or wavefront) size G as a function of the linear lattice size L for the 2-d Ising model. We would expect G to scale slower than $\sqrt{C}$, where C is the average cluster size. Since C for the Wolff algorithm is an estimator of the susceptibility [12, 34], we have $C \sim L^{\gamma/\nu}$, where $\gamma/\nu = 1.75$ is the finite-size scaling result for the susceptibility of the 2-d Ising model. Thus G should scale slower than $L^{0.875}$, and we measured the actual exponent to be 0.66(1).

For this algorithm, the number of processors that can be effectively used is approximately $G/2$, so on a parallel machine with 32 nodes (as we have used) reasonable efficiencies should be obtained for lattices of size greater that 512^2, which is indeed what we found. In fact these are the kinds of lattice sizes one would like to be able to run in parallel – anything smaller than this could more easily be run on a single processor, with multiple simulations on the different processors of a parallel machine [16, 17].

6 Conclusions

We have introduced two new parallel implementations of the Wolff single-cluster Monte Carlo algorithm. The first method involves a simple change to a parallel Swendsen-Wang cluster algorithm, so that the iterative procedure for identifying the clusters is halted as soon as the single Wolff cluster is identified. This method performs about as well as the parallel Swendsen-Wang algorithms on which it is based, however neither of these parallel cluster algorithms are well-suited to implementation on massively parallel SIMD machines.

The scattered strip partitioning algorithm successfully overcomes the load balance problem that is the main difficulty in constructing a parallel Wolff cluster algorithm. This is done by partitioning the lattice using a scattered type of decomposition, so that the expanding wavefront is distributed over almost all processors. This method of distributing the data produces reasonable speed-ups for the parallel breadth-first-search algorithm for growing a single cluster, as long as the linear dimension of the lattice is much greater than the number of processors, and the number of processors is not too great.

On a massively parallel machine, parallelism could be extracted by a combination of the strip partitioning algorithm and the independent (or job-level) parallelism of running independent Monte Carlo simulations with different random number streams on different groups of processors. For example, one might run 16 independent simulations, each of which use 16 processors. In this scenario, the main advantage of using the parallel algorithm is that it avoids the memory limitations of a single processor, and allows the use of larger lattice sizes.

Note that the results for the speed-ups and efficiencies of our parallel algorithms are for the simplest of spin models, the Ising model. For more complex models, for example continuous spin models such as the O(N) model, the computation required to calculate the bond probabilities is much greater, and we would therefore expect greater efficiencies due to the higher ratio of computation to communication. The speed-up also depends heavily on the average generation size, which will vary between different spin models and lattice sizes.

It is possible that this technique could also be used to improve the performance of parallel Swendsen-Wang algorithms. These algorithms also suffer from load imbalance, since at the critical point there is generally one large cluster that takes the longest time to update. With a standard domain decomposition, there are generally processors that contain little or no sites belonging to the largest cluster. These processors will be mainly idle during the latter stages of the cluster labeling. If a scattered strip partitioning of the lattice were used, this will no longer be the case, and the load will be much more evenly divided. However for the MIMD algorithm the strips will have to be much wider, in order for the sequential labeling algorithm on each processor to work effectively, so there are tradeoffs that need to be investigated to find the most effective implementation for a given lattice size and number of processors.

Simulation of the 2-d Ising spin model using the Wolff algorithm has been shown to be a very sensitive test of the randomness properties of random number generators [37, 38]. One of the motivations for this work was to create a parallel Wolff algorithm that can be used as a test for parallel random number generators. This work is currently in progress.

Acknowledgements

This work was done using the Thinking Machines CM-5 at the Northeast Parallel Architectures Center (NPAC) at Syracuse University. Work supported in part by the Center for Research on Parallel Computation with NSF cooperative agreement No. CCR-9120008 and ARPA under contract No. DABT63-91-K-0005.

References

[1] K. Binder ed., Monte Carlo Methods in Statistical Physics, (Springer-Verlag, Berlin, 1986); K. Binder and D.W. Heermann, Monte Carlo Simulation in Statistical Physics, (Springer-Verlag, Berlin, 1988); H. Gould and J. Tobochnik, An Introduction to Computer Simulation Methods, Vol. 2, (Addison-Wesley, Reading, Mass., 1988).

[2] N. Metropolis et al., J. Chem. Phys. 21 (1953) 1087.

[3] A. D. Sokal, in Computer Simulation Studies in Condensed Matter Physics: Recent Developments, eds. D. P. Landau et al. (Springer-Verlag, Berlin, 1988); A. D. Sokal, in Proc. of the International Conference on Lattice Field Theory, Tallahassee, October 1990, Nucl. Phys. B (Proc. Suppl.) 20 (1991) 55.

[4] R.H. Swendsen and J.-S. Wang, Phys. Rev. Lett. 58 (1987) 86.

[5] U. Wolff, Phys. Rev. Lett. 62 (1989) 361.

[6] U. Wolff, in Proc. of the Symposium on Lattice Field Theory, Capri, September 1989, Nucl. Phys. B (Proc. Suppl.) 17 (1990) 93.

[7] J.-S. Wang and R. H. Swendsen, Physica A 167 (1990) 565.

[8] C. F. Baillie, Int. J. Mod. Phys. C 1 (1990) 91.

[9] N. Ito and G. A. Koring, Cluster vs single-spin algorithms – which are more efficient?, to be published in Int. J. Mod. Phys. C.

[10] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, (Computer Science Press, Rockville, Maryland, 1978); G. Brassard and P. Bratley, Algorithmics: Theory and Practice, (Prentice Hall, Englewood Cliffs, N.J., 1988).

[11] U. Wolff, Phys. Lett. B228 (1989) 379.

[12] P. Tamayo, R. C. Brower and W. Klein, J. Stat. Phys. 58 (1990) 1083.

[13] C. F. Baillie and P. D. Coddington, Phys. Rev. B 43 (1991) 10617.

[14] P. D. Coddington and C. F. Baillie, Phys. Rev. Lett. 68 (1992) 962.

[15] A. N. Burkitt and D. W. Heermann, Comp. Phys. Comm. 54 (1989) 210.

[16] P.D. Coddington and C.F. Baillie, in Proc. of the 5th Annual Distributed Memory Computing Conference, Charleston, April 1990, eds. D.W. Walker and Q.F. Stout (IEEE Computer Society Press, Los Alamitos, California, 1990).

[17] C. F. Baillie and P. D. Coddington, Concurrency: Practice and Experience 3 (1991) 129.

[18] R. C. Brower, P. Tamayo and B. York, J. Stat. Phys. 63 (1991) 73.

[19] M. Flanigan and P. Tamayo, Int. J. Mod. Phys. C 3 (1992) 1235.

[20] J. Apostolakis, P. Coddington and E. Marinari, Europhys. Lett. 17 (1992) 189.

[21] J. Apostolakis, P. Coddington and E. Marinari, Int. J. Mod. Phys. C 4 (1993) 749.

[22] P. Rossi and G. P. Tecchiolli, Finding Clusters in a Parallel Environment, unpublished.

[23] H. Mino, Comp. Phys. Comm. 66 (1991) 25.

[24] Y. Shiloach and U. Vishkin, J. Algorithms 3 (1982) 57.

[25] H. Embrechts, D. Roose, and P. Wambacq, in Proc. First European Workshop on Hypercube and Distributed Computers, F. Andre and J.P. Verjus eds., (North-Holland, Amsterdam, 1989); H. Embrechts, D. Roose, and P. Wambacq, Computer Vision Graphics and Image Processing: Image Understanding, 57 (1993) 155.

[26] J. Woo and S. Sahni, J. of Supercomputing 3 (1989) 209.

[27] R. Dewar and C. K. Harris, J. Phys. A 20 (1987) 985.

[28] H. G. Evertz, J. Stat. Phys. 70 (1993) 1075.

[29] G. C. Fox et al., Solving Problems on Concurrent Processors, (Prentice-Hall, Englewood Cliffs, New Jersey, 1988).

[30] CMFortran Reference Manual, (Thinking Machines Corporation, Cambridge, Mass., 1993).

[31] Y.-Y. Fang, I.-L. Yen and R. Dubash, Improving the performance of Lee's maze routing algorithm on parallel computers via semi-dynamic mapping strategies, Technical Report CPS-93-35, Michigan State University, December, 1993.

[32] C. Y. Lee, IRE Trans. Electronic Computers EC-10 (1961) 346.

[33] The High Performance Fortran Forum, High Performance Fortran Language Specification, Center for Research in Parallel Computing Technical Report CRPC-TR92225. Available via anonymous ftp from titan.cs.rice.edu in the directory public/HPFF/draft.

[34] U. Wolff, Nucl. Phys. B 334 (1990) 581.

[35] CMMD Reference Manual, (Thinking Machines Corporation, Cambridge, Mass., 1993).

[36] T. von Eicken et al., Active Messages: a Mechanism for Integrated Communication and Computation, in Proc. of the 19th Int. Symposium on Computer Architecture, Gold Coast, Australia, May, 1992.

[37] A.M. Ferrenberg, D.P. Landau and Y.J. Wong, Phys. Rev. Lett. 69 (1992) 3382.

[38] P.D. Coddington, Analysis of Random Number Generators Using Monte Carlo Simulation, NPAC technical report SCCS-526, to be published in Int. J. Mod. Phys. C.

Lattice Size	128^2	256^2	512^2	1024^2	2048^2
Time (μs) Sequential	31.42	32.36	32.81	33.74	–
Time (μs) MIMD	6.50	5.10	3.96	3.26	2.93
Speed-up MIMD	4.8	6.3	8.3	10.4	11.5
Efficiency MIMD	0.15	0.20	0.26	0.32	0.36
Time (μs) Wolff SIMD	40.70	44.90	46.56	63.09	–
Time (μs) S-W SIMD	20.22	19.23	17.38	18.34	–

Table 1: Times per spin update, speed-ups, and efficiencies as a function of lattice size for the MIMD scattered strip partitioning algorithm, the SIMD Wolff algorithm (the single-cluster Swendsen-Wang algorithm), and the SIMD Swendsen-Wang (S-W) algorithm from which it is derived, all for a 32-node CM-5. Speed-ups and efficiencies for 2048^2 are based on sequential 1024^2 times, so are probably underestimated.

```
Chose a site i at random                             First site in the cluster
C ← {i}                                              Initialize the cluster list
Q ← {i}                                              Initialize the queue
WHILE Q ≠ {} DO                                      Search over sites still in queue
  j ← next element in Q                              Get the next site from the queue
  Q ← Q - j                                          Remove this site from the queue
  Find the set {n} of connected neighbors of j       Identify bonds to neighbors
  FOR each n ∈ {n} DO                                For all connected neighbors
    IF n ∉ C THEN                                    If neighbor is not in cluster list
      C ← C + n                                      Add it to the cluster list
      Q ← Q + n                                      Add it to the queue
    END IF
  END DO
END WHILE
```

Figure 1: The breadth-first-search algorithm for creating a single cluster. For the Wolff algorithm, the spin is also changed to the new value ($s_n \leftarrow -s_n$ for the Ising model) when a site n is added to the cluster.

```
G $\leftarrow$ {}                                     Initialize the current generation
Chose a site i at random                              First site in the cluster
C $\leftarrow$ {i}                                    Initialize the cluster list
G$'$ $\leftarrow$ {i}                                 Initialize the next generation
WHILE G$'$ $\neq$ {} DO                               Loop over generations
  G $\leftarrow$ G$'$                                 Go to new generation
  G$'$ $\leftarrow$ {}                                Initialize the next generation
  WHILE G $\neq$ {} DO IN PARALLEL                    Loop over this generation of sites
    j $\leftarrow$ next element in G                  Get the next site from the queue
    G $\leftarrow$ G - j                              Remove this site from the queue
    Find the set {n} of connected neighbors of j      Identify bonds to neighbors
    FOR each n $\epsilon$ {n} DO                      For all connected neighbors
      IF n $\notin$ C THEN                            If neighbor is not in cluster list
        C $\leftarrow$ C + n                          Add it to the cluster list
        G$'$ $\leftarrow$ G$'$ + n                    Add it to the next generation
      END IF
    END DO
  END WHILE
END WHILE
```

Figure 2: The parallel breadth-first-search algorithm for creating a single cluster.

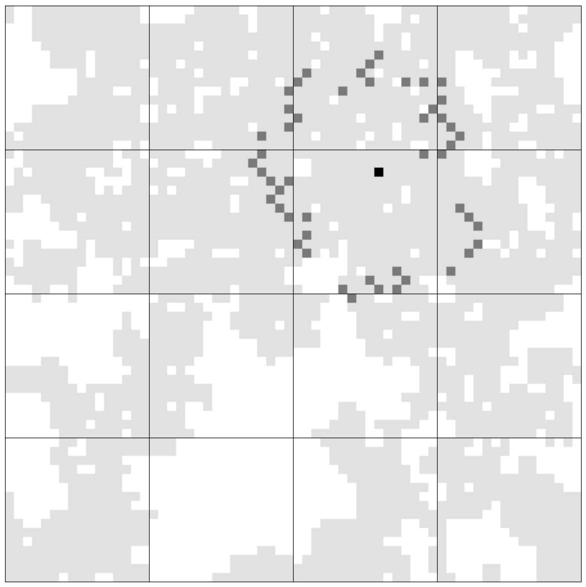

Figure 3: An example of a Wolff cluster for a configuration of the 2-d Ising spin model. The light gray sites indicate the cluster, the sites in dark gray show a particular wavefront or generation, and the black site is the initial randomly chosen site. The black lines represent processor boundaries for a standard domain decomposition of the lattice onto 16 processors.

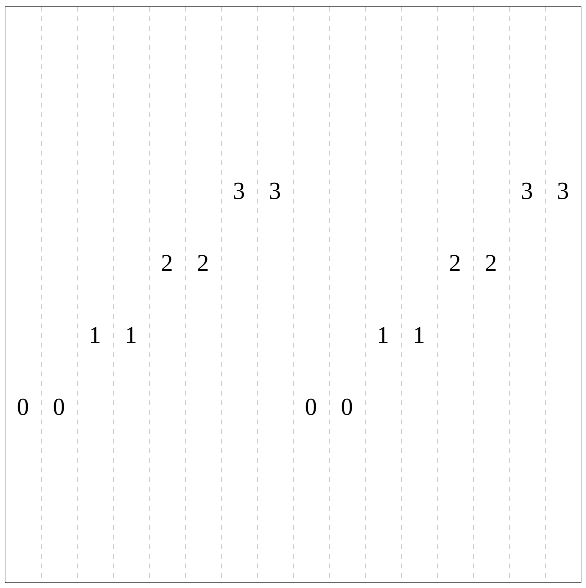

Figure 4: The mapping of 16 columns of data onto 4 processors for scattered strip (or column-cyclic) partitioning with a column width of 2.

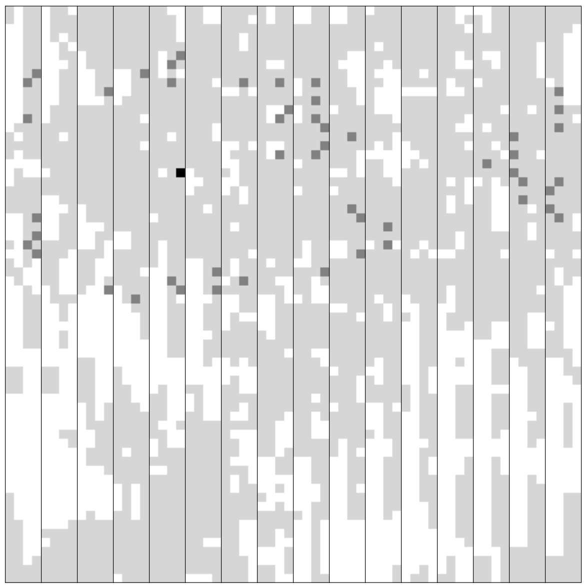

Figure 5: A scattered strip (or column-cyclic) partitioning onto 16 processors for the data in Fig. 3.

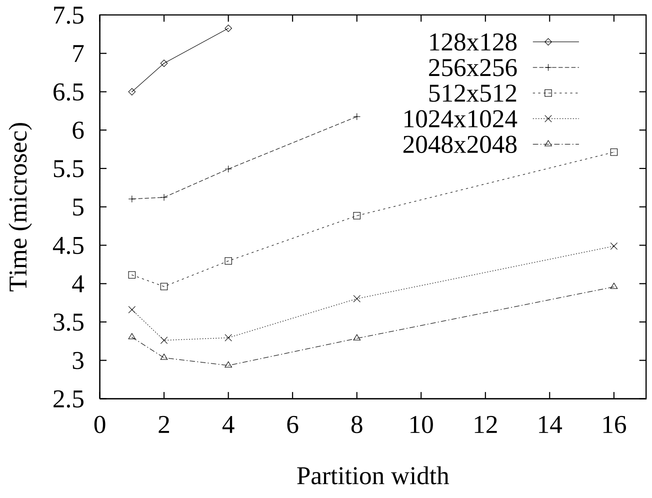

Figure 6: Average update time per spin as a function of partition width for different lattice sizes, for the scattered strip partitioning algorithm on a 32-node CM-5.

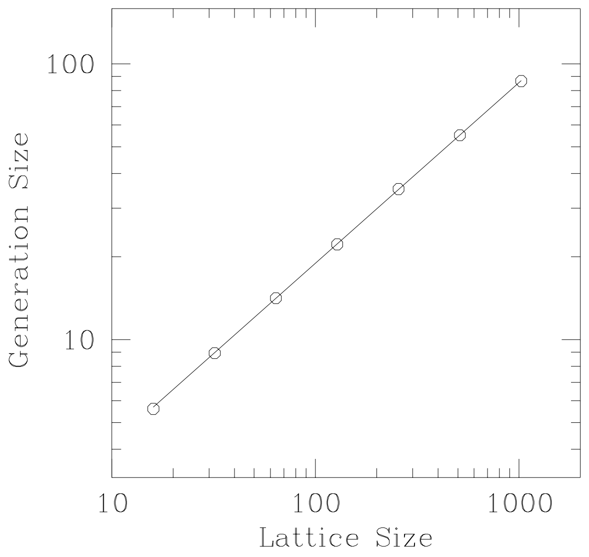

Figure 7: The average number of sites per generation in the Wolff cluster update as a function of lattice size for the 2-*d* Ising model. The line is a fit to a power with exponent 0.66(1).