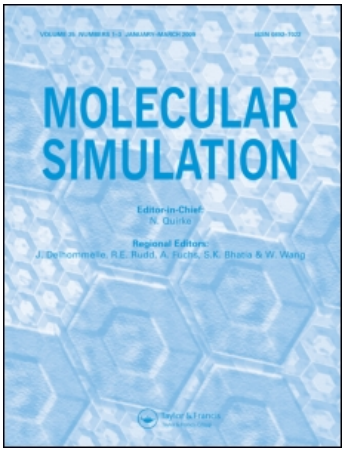

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

Cluster Identification and Percolation Analysis Using a Recursive Algorithm

T. Edvinsson[a]; P. J. Råsmark[a]; C. Elvingson[a]
[a] Department of Physical Chemistry, Uppsala University, Uppsala, Sweden

To cite this Article Edvinsson, T. , Råsmark, P. J. and Elvingson, C.(1999) 'Cluster Identification and Percolation Analysis Using a Recursive Algorithm', Molecular Simulation, 23: 3, 169 — 190
To link to this Article: DOI: 10.1080/08927029908022121
URL: http://dx.doi.org/10.1080/08927029908022121

Molecular Simulation, 1999, Vol. 23, pp. 169–190
Reprints available directly from the publisher
Photocopying permitted by license only

CLUSTER IDENTIFICATION AND PERCOLATION ANALYSIS USING A RECURSIVE ALGORITHM

T. EDVINSSON*, P. J. RÅSMARK and C. ELVINGSON

Department of Physical Chemistry, Uppsala University, Box 532, S-751 21 Uppsala, Sweden

(Received July 1999; accepted August 1999)

A recursive algorithm for sampling properties of physical clusters such as size distribution and percolation is presented. The approach can be applied to any system with periodic boundary conditions, given a spatial definition of a cluster. We also introduce some modifications in the algorithm that increases the efficiency considerably if one is only interested in percolation analysis. The algorithm is implemented in Fortran 90 and is compared with a number of iterative algorithms. The recursive cluster identification algorithm is somewhat slower than the iterative methods at low volume fraction but is at least as fast at high densities. The percolation analysis, however, is considerably faster using recursion, for all systems studied. We also note that the CPU time using recursion is independent on the static allocation of arrays, whereas the iterative method strongly depends on the size of the initially allocated arrays.

Keywords: Cluster analysis; percolation; recursion; Fortran 90; algorithm

1. INTRODUCTION

Computer simulation has proved to be a valuable tool for determining the structure and dynamics of macromolecular systems [1–4]. For polymer solutions or colloidal dispersions with attractive intermolecular interactions, clusters can form which can be complex topological objects. At a certain concentration, one can also observe percolation, *i.e.*, the formation of infinite aggregates [5]. The precise structure of these objects and the detailed information obtained by computational methods represent an "exact"

*Corresponding author.

solution for a given physical model, which then can be compared with experiment or theory.

Fast cluster counting algorithms have been developed for lattice systems [6] as well as for either lattice or off-lattice systems [7–9]. Hochen-Kopelman (HK) [6] is possibly the most widely used for lattice systems by virtue of its low memory demand and high speed. To determine if a system percolates in a continuum with periodic boundary conditions, however, requires extra information about the system in addition to the cluster statistics. The main objective of this paper is to obtain an effective and compact percolation algorithm, extracting information about the periodic boundaries that are crossed when traversing the system recursively. We will thus focus on the recursive nature of the problem, obtaining an easy way of organizing the boundary information and also use it to define a condition for percolation. The algorithm has similarities with the HK-procedure in the way the cluster information is stored but the information is sampled in a different way. In the present approach we sample the statistics clusterwise in a continuum in contrast to HK which goes through the system spacewise in a lattice. The algorithm generates extra information about the boundaries that are crossed by checking in what direction the boundary conditions are applied. This together with the algorithmic definition of percolation and the recursive implementation will be in focus throughout this paper. The efficiency of our approach is compared with the algorithm of Heyes *et al.* [10] which also utilizes extra boundary information but in an iterative way.

The basic structure in the analysis is the cluster which is here defined as a set of objects within a threshold distance from at least one other member of the set. As one can see from this definition, the concept of a cluster is inherently recursive. The notion of a cluster threshold distance, hereafter called cluster distance, is of course related to the concept of "contactness" which lies in the hands of the analyser who can arbitrarily choose the cluster distance criteria [11]. The distance, which is fixed and given, should be chosen by considering which physical observables are of interest in the analysis and the general concept of a cluster makes this analysis useful for many diverse physical systems. Percolation transitions, for instance, arise in a large variety of physical systems, such as in the swelling of a gel and gelation [12], glass transitions [13], the burning of concrete [14], conductor-insulator transitions in liquid metals [15], network formation in microemulsions [16–18], and in porous materials [19] as well as for water (hydrogen bonding) [20]. For these systems, the formation of clusters or other structures, and a possible percolation transition has a large influence on the properties of the material. When comparing experimental data with theoretical

or simulation results one must remember, however, that the interpretation is a consequence of how the criterion for a cluster and therefore also percolation is defined. If one is interested in galaxy clusters one chooses a distance on the cosmological scale, for the transport of excitations between particles one can choose the characteristic hopping distance, and if one is interested in density dependent observables one can choose the distance from beyond the first peak of the pair radial distribution function. In the present paper we will use the flocculation of a colloidal dispersion as an example. For such a system at a certain critical concentration, there is a sharp change in many macroscopic properties such as viscosity, shear modulus, and electric conductivity [21]. This is due to percolation, *i.e.*, the formation of a cluster or network spanning through the system.

Considerable progress has been achieved in understanding the universal factors affecting the macroscopic structure and the rheology of particulate systems, while much of percolation theory has focused on the critical behaviour in the vicinity of the percolation transition, using lattice models [5]. This approach is tractable when studying the universal critical behaviour and in systems where the fluid nature of the system can be approximated with a lattice. Another approach is to study the percolation geometry in close relation to material problems where the percolation transition is studied as a function of the material microstructure. In this context, the mere existence of a percolation threshold is what tends to dominate the material properties. In the present paper we will focus on the latter approach and develop an algorithm that will be useful when investigating how the properties of the individual particles determine the percolation behaviour.

We will explain the principles of the algorithm explicitly using pseudo code to close the gap between algorithm formulation and application, and for the same reasons we have also included an appendix with the core of the algorithm implemented in Fortran 90.

In Section 2 we describe a recursive algorithm for finding clusters given a spatial distribution of spherical particles. This is to outline the basic ideas of cluster counting in the present approach and will hopefully illuminate the extra information needed for a percolation analysis which is outlined in Section 3.1. There we will discuss the information needed in a cluster counting algorithm to include a percolation condition and further suggest an easy way of organizing this information. We will also briefly comment on a second alternative for percolation analysis in Section 3.2 based on a graph theoretical approach. In Section 4 some results are presented when analysing Monte Carlo generated clusters, and these results are discussed and compared with some iterative procedures. The paper ends with some conclusions.

2. CLUSTER IDENTIFICATION

The enourmous increase in computational speed using modern workstations has somewhat shifted the problem generating effective code to the problem of making a compact, transparent, and thus easily maintainable program. Fortran is still the most widely used programming language in scientific computing but the algorithms in use are often based on older Fortran versions, most of them not utilizing the new features in Fortran 90. In this paper we will show how to use one of these new features, namely recursion, which is possible due to the possibility of dynamic memory allocation. This will result in a more transparent code and, as we will see below, a recursive algorithm for determining percolation can also result in a more efficient code than using iteration.

The basic problem of cluster identification can, for small systems, be solved by Warshall's algorithm which involves a triple loop where an element of a logical matrix, (i, j), is true if objects i and j are connected. This approach is often used in graph theory applications [22]. In principle, the algorithm collects the pair-connectivity with two inner loops and then "fuses" the pair connections into the full connection matrix, containing also the indirect connections, by an outer loop. The computational cost for N particles using this algorithm is $\mathcal{O}(N^3)$ and it will in most cases of interest be too slow even if one uses the inherent symmetry. One can also fuse the pair connectivity in more effective ways and speed up the fusion by applying bit-wise logical operations as suggested by Sevick *et al.* [23]. Bit-wise logical operations can also be used in other algorithms and will not be discussed in this paper.

There are two general ideas for a cluster identification algorithm. One is to find an effective way to handle the time consuming fusing of the pair-connectivity and there exists a number of iterative algorithms solving this problem [7–9]. One cannot, however, escape the $\mathcal{O}(N^2)$ part of a cluster analysing algorithm representing the pair-connectivity information when separating the cluster counting problem into a pair-connectivity part and one part that fuses this information. The second approach is to sample the pair-connectivity and the indirect connectivity together, without separating the tasks. Hoshen and Kopelman use the latter approach and additionally reduce the $\mathcal{O}(N^2)$ dependence to $\mathcal{O}(N\log(N))$ by analysing all particles according to their positions in a lattice. This is a way of using the extra information already available by the implicit order one has in a lattice. When trying to obtain an alternative algorithm being useful for large off-lattice systems, one need not necessarily begin by sample the information about the pair-connectivity or utilize the regularity in possible positions as

in the case of a lattice. In contrast to the algorithms mentioned above, one can instead follow the connections of a cluster recursively and thus sample the direct and indirect connectivity using the inherent recursive nature of the problem of finding all connected particles. The full advantage of a recursive algorithm can be utilized when the system of concern branches into multiple paths which must all be followed. In recursive routines, the compiler takes care of the stack (*e.g.*, the list of sequencial branches) and the code can be made more compact and transparent. Tra-versing such a system using iteration on the other hand, requires that you build your own stacks in memory with *e.g.*, arrays. Moreover, for systems with an unpredictable and large number of branches, a modern compiler takes care of the stack quite efficiently when using recursion as will be seen in Section 4.

The crucial point when developing a recursive algorithm is to identify a part of the set of instructions that is general enough to be called by another part of the algorithm and also by itself. In the description below, the general part consists of Steps 4–7, which is where one recursively follows all connections of a unit, sampling all particles belonging to a cluster. The recursive part also keeps track of the particles included in a cluster by associating a logical flag with each particle. Step 2 in the main construct (Steps 1–3) then assures that no analysis is performed if the particle is included in an already detected cluster.

1. Loop over all particles (i).
2. If particle 'i' is already included in a cluster, cycle the loop (goto 1).
3. Set the present particle ('unit') to the loop variable, 'i', and a 'new-cluster' flag to false.
4. Loop over all the particles (j) except the active particle. Note that when returning from a recursive call at 7d below, the loop over the particles is continued at the current level in the recursion hierarchy.
5. If the particle 'j' is already included in a cluster, *i.e.*, if an aggregation flag has been set to true in 7a or 7b, cycle the loop (goto 4).
6. Calculate the distance between the present particle 'unit' and 'j' (using periodic boundary conditions).
7. If the distance is smaller than the defined cluster distance then:
   (a) If the 'new-cluster' flag is false, we have found a new cluster. Set the 'new-cluster' flag to true. The number of clusters is updated and the present particle 'unit' is added as the first particle in the new cluster and an aggregation flag is set to true for particle 'unit'.
   (b) Add particle 'j' to the list of particles that are included in the present cluster and set the aggregation flag to true for particle 'j'.

(c) Set 'unit = j'.
(d) Continue at 4. This is the recursive call. Now 'j' will be checked with all other particles that are not previously included in a cluster according to 5. They in turn will also be investigated if they are connected to 'j'.

The final result is collected in two vectors in a standard pointer list manner where the number of non-zero elements (minus one) in the first array gives the number of clusters and points to the second array which consecutively contains the members of each cluster.

3. PERCOLATION ANALYSIS

When using computer simulations to calculate properties of macroscopic systems, one normally uses periodic boundary conditions to reduce finite size effects [3, 4, 9]. This means, however, that the problem of finding an infinite network becomes nontrivial. To visualize the concept of periodic boundary conditions we use the notion *periodic domain* which is the simulation box seen in an environment of identical boxes. The simplistic view for identifying a percolating cluster, which would work with fixed borders without periodicity, is to look for any aggregate that would connect two opposing sides in the x, y, or z-direction. As can be seen in Figure 1, this is not, however, a sufficient condition for percolation using periodic boundary conditions.

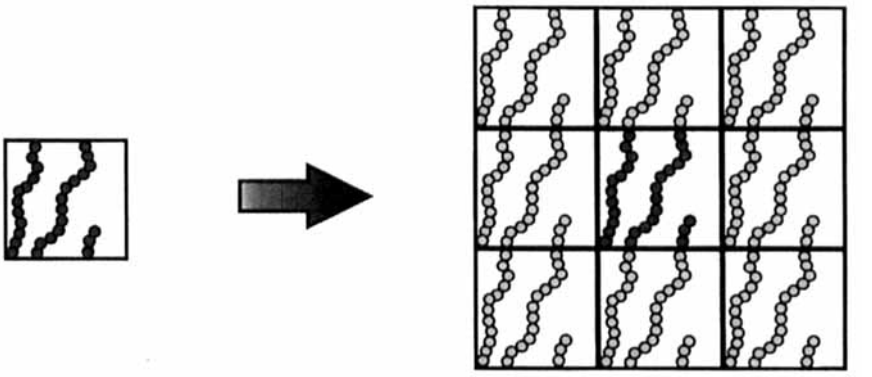

FIGURE 1 Schematic picture showing that a spanning cluster is not a sufficient condition for percolation in a system with periodic boundary conditions, even if the cluster spans over periodic boundaries.

One way of finding percolating clusters using periodic boundary conditions in off-lattice simulations has been described by Heyes *et al.*, using iteration [10]. Using recursion instead, though, one can construct an algorithm which is somewhat less cumbersome and even more efficient, especially in the density ranges where one expects percolation.

When looking for percolation in systems with periodic boundary conditions, one must find a cluster which has the property of biting its own tail. Such a combination of particles will hereafter be called a closed path. A closed path is a necessary but not sufficient condition for an infinite cluster in a periodic environment. The extra condition needed is that the cluster is only infinite if the closed path includes both the original particle and the particle as a periodic image. In Section 3.2 we formulate this condition using a graph theoretical vocabulary.

To find an infinite cluster, one must then keep track of which borders are crossed and in what direction, in order to know if the cluster returns to the same particle as it started from or to one of its periodic mirror images. This difference is illustrated in Figures 2a and 2b, where a percolating and a non-percolating cluster are shown, the latter forming a closed path but returning to the same particle in the periodic domain. In the next section we will first extend the cluster analysing algorithm presented above and define a percolation condition by introducing a box identity. We will also mention another possible approach in Section 3.2 which analyses the different closed paths. A percolation condition for this latter case is then defined by comparing the vectorial distances travelled following the different paths with the box size.

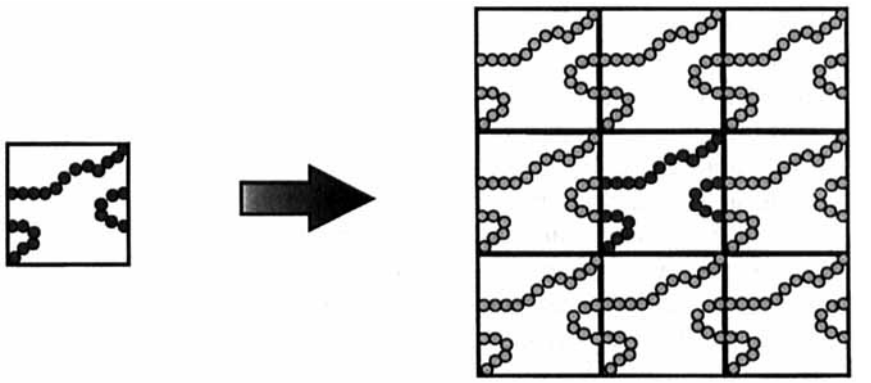

FIGURE 2a Picture showing a percolating cluster in which a particle is connected to its mirror image in the periodic domain.

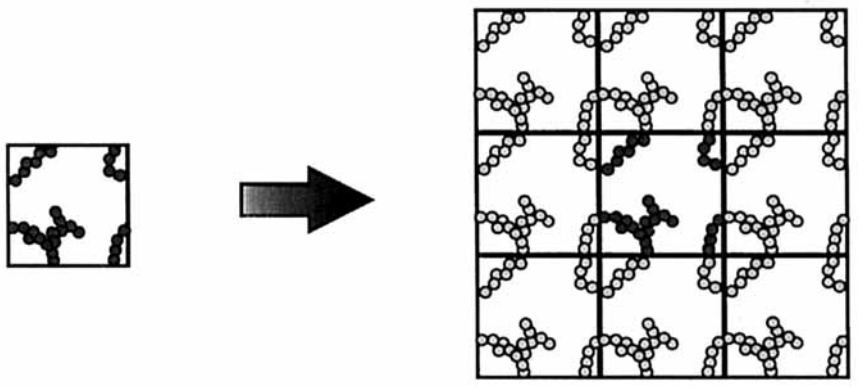

FIGURE 2b Picture showing a non-percolating cluster in which a particle is connected to itself in the same box in the periodic domain.

3.1. Extended Cluster Algorithm

One can extend the recursive cluster search algorithm in the previous section to also include percolation analysis with some minor but important modifications. First, one has to make the general part (4–7 above) even more general in the following way: Taking away Condition 5 and realizing that all other clusters are closed sets, the calling part (1–3) will not call the recursive part if a particle is already included in a previously detected cluster. By removing this condition one gets into the problem of infinite recursion, *i.e.*, a circle is also infinite at this stage. One has to extract some extra information to be able to determine if the closed path returns to the original particle or to one of its periodic images. We thus define a box identity which holds information on where we are going in the periodic domain (see Fig. 3). The integer array 'box_id' contains three elements which are the x, y, and z indices of the present box with the starting box as the origin. Furthermore, if the particle is included in a cluster, we store the present box identity. This is then utilized when defining a percolation condition. Keeping track of which borders are crossed one can neglect all non-percolating closed paths by realizing that such a path cannot be connected to itself in a periodic box. The stopping condition in the recursive part of the algorithm can now be stated: If the present particle is already part of the cluster and the box identity is not the same as the previously stored box identity then we have an infinite cluster, *i.e.*, percolation.

If the recursive routine calls itself with a particle which already has been included in a cluster, it will affect the definition of our box identity origin. This is handled by setting the box identity to the stored box identity where

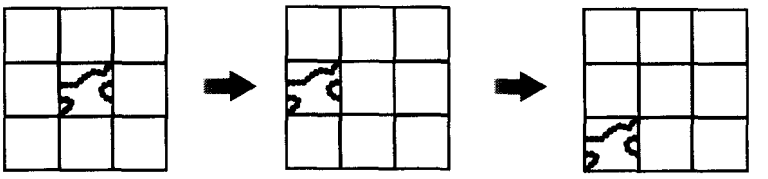

FIGURE 3 Schematic picture showing the principle of the box identity. The array 'box_id' contains three elements; the x, y, and z-index. For simplicity, we show only the plane $z = 0$. The array holds information about in what direction the periodic boundary conditions are applied and thus information about the location in the periodic domain.

the particle was found previously. This will shift the origin with respect to the latest aggregating position in the periodic domain. We have to remember that inside the recursive part of the algorithm there is no way of reaching particles in another cluster because they are all closed sets. To calculate the box identity for *e.g.*, the x-coordinate, 'box_id(1)', one can use

```
if (distance > 0.5 * box_side) then
   if (x(unit) > 0.0) then
      box_id(1) = box_id(1) + 1
   else
      box_id(1) = box_id(1) - 1
   endif
endif
```

where 'distance' is the separation between particle 'unit' and particle 'j' and 'x(unit)' is the x-coordinate of the particle 'unit'. The outer condition determines if the periodic boundary conditions are applied due to the minimal image approach [3, 4, 9]. Because this part is situated inside the distance criterion for connection, there is a connection over the periodic boundary. The inner condition then checks in what direction the periodic boundary conditions are applied. With the origin in the centre of the box, the sign of 'x(unit)' shows in what part of the box the particle is found, and thus gives the direction in which the periodic boundary conditions are applied. To improve the speed, one can instead use the construct:

$$\text{box\_id}(1) = \text{box\_id}(1) + \text{int}(\text{abs}(\text{delta\_}x)) * \text{sign}(1.0, x(\text{unit}))$$

where 'delta_x' is the distance without the periodic boundary condition. This box identity shift together with the percolation condition is the heart of the algorithm.

To increase the efficiency of the percolation analysis one can introduce two further modifications. First one knows that for an infinite cluster, a particle must connect to a periodic image, *i.e.*, a percolating cluster must cross at least one boundary. We can thus introduce a region close to the boundary represented by a variable 'box_border' and only perform the percolation analysis starting from particles in that region. Secondly, one can abort the analysis if the percolation condition has been fulfilled, which speeds up the code especially for the more dense systems. One should observe though, that these last two modifications can *not* be applied in a combined cluster and percolation analysis where we want information on all clusters. A Fortran 90 version of the percolation algorithm is given in Appendix A.

3.2. Closed Path Algorithm

An alternative way to determine if a cluster is part of a percolating structure, is to use a graph theoretical approach. In a system without periodic boundary conditions, the problem is straightforward as we have already noted and can for example be solved by finding the minimal spanning tree and determine if it is of the same size as the system [24]. Using periodic boundary conditions on the other hand, makes the task more intricate. The problem is reminiscent of the travelling salesman problem but somewhat more complicated. For the salesman the problem is to find the shortest path that connects all the cities (particles). A path that connects all cities only once is called a Hamilton path and a closed path that visits every city only once and returns to the starting city is called a Hamilton circuit [25]. The notion of a closed path (introduced earlier) is a Hamilton circuit for a certain subset of the particles. For an infinite spanning cluster in the periodic domain, the problem is thus to find any set of particles that can form a Hamilton circuit which returns to its periodic image. This means that we have a set of Hamilton circuits for different sets of particles that are all candidates for satisfying a percolation condition. Much work in graph theory has been directed to solve the problem of finding a path that connects all vertices once and returns to the starting point. Both the travelling salesman problem and the Hamiltonian path problem are NP-complete which means that no algorithm has been devised that is faster than guessing the solution and then checking, in polynomial time, that the solution works [26]. One has to remember, though, that the percolation problem amounts to a clever path/city marking algorithm that classifies all cities but not necessarily all paths. Recording all paths will not be possible for a system of any interesting size. Below we briefly describe how to find a subset of

all closed paths which can be done either recursively or by an iterative method. This is done by realizing that a closed path leading to percolation must cross at least one border. (Note that this approach is not compatible with the full cluster analysis).

An algorithm using the ideas above can be implemented as follows: Create a list consisting of the border particles connected *through* the cell and a similar list with the border particles connected *over the border*. Combine the two lists of pairs. If a pair occurs in both lists a *new* particle is introduced as the mirror particle to one of the particles in the pair. One of the problems when using periodic boundary conditions is to distinguish between the different paths connecting the same particles. This can arise when we have a path over the cell border and another inside the cell. One way of handling this is to introduce pseudo particles representing the periodic connections. Instead of two different paths from *e.g.*, A to B, we can introduce a pseudo particle C and obtain the three vectorial lengths A-B, B-C, and C-A, where the last one is zero on behalf of A and C being the same particle. This makes it possible to add the vectorial lengths A-B and B-C and see if it matches or exceeds the cell size. Using these extra pseudo particles, we can calculate the vectorial lengths for complicated paths, for instance criss-crossing over a cell border, and still get the real distance in a periodic domain. Note, though, that the pseudo particles are only introduced to distinguish between the different possible paths, and only contribute to the vector sum in the same way as the original particles. In order to remove loose ends, one also has to remove all pairs that have a particle that only occurs once in the list. Checking the lists of particle pairs, and looking for a particle which has already been checked, there is a path starting and ending with this particle, *i.e.*, a closed path passing a border, and we have a candidate for an infinite cluster. One then has to compute the vectorial lengths of the closed path including mirror particles and if the path exceeds the simulation cell length in the x, y or z direction, the cluster is infinite, *i.e.*, we have percolation.

In the Hamilton path problem the number of possible closed paths grows very fast with the number of cities (of the order of $N!$) and it soon becomes very costly to compute all of them. The same holds for a dense system of cluster forming particles even though we do not have to investigate all possible paths. A dense system is, however, more likely to percolate than a non-dense, so this algorithm is in most cases not as fast as the one described in the previous section. It should be most useful for investigating the gelation properties of polymers where percolation can occur at low volume fractions. Note that it is not the number of particles *per se* but the number of branches in the cluster that determines the speed of this algorithm.

If the particles are aligned, each connected only to the previous and the following particle, there will only be one path and this will be found quickly. The implementation of this algorithm is quite memory demanding due to the large number of possible closed paths leading to percolation in systems with high density. However, it should be of interest for those who are dealing with a relatively small system preferring a graph theoretical approach for their analysis although the resulting code is not as transparent as the implementation of the previous algorithm.

4. RESULTS AND DISCUSSION

To test the cluster identification and percolation routines discussed in the previous sections, we have performed Monte Carlo simulations of colloidal dispersions using a square well potential [3] and analysed the resulting structures. The radius of the particles is 0.25 μm, the width of the potential well is equal to 0.60 μm (*i.e.*, 0.10 μm outside the touching distance) and the depth is $-2.0kT$, where k is Boltzmann's constant and T is the temperature equal to 293 K. Each particle is moved 5000 times and the particle configuration is analysed after every 50 steps. For each set of parameter values this will give 100 sets of coordinates to analyse. The algorithms have been tested on a DEC Alpha AU500 using OSF4.0 with Digital Fortran 90 V4.1–270 and on an IBM RS/6000 3CT using AIX 4.1 with XL Fortran Version 3.2. Since we are comparing algorithms which use dynamic memory allocation with others using static memory allocation, it is important to test the algorithms on different computers with different compilers. Since the IBM is somewhat older compared to the DEC, we will mainly show results for the DEC and use the values for the IBM as comparison when explicitly stated.

We have compared the recursive method for cluster identification described in Section 2 with three different iterative routines for off-lattice cluster analysis [7–9]. For a system with $N = 1000$, the average cluster size changes from 2.1 to 994.0 in the concentration interval 0.2% to 25.0% and the number of clusters changes from 28 to 1. The recursive algorithm is slower for low volume fractions but becomes comparable to or faster compared to the iterative routines when the volume fraction is above 15% (see Figs. 4a, b). We also looked at the CPU-time as a function of system size for different volume fractions and it followed quite closely an N^2-dependence for all volume fractions studied (0.2%–30.0%), although the preexponential factor changes with volume fraction which is seen in the relative speed. Since the cluster search algorithms would be of interest mainly for systems with relatively high

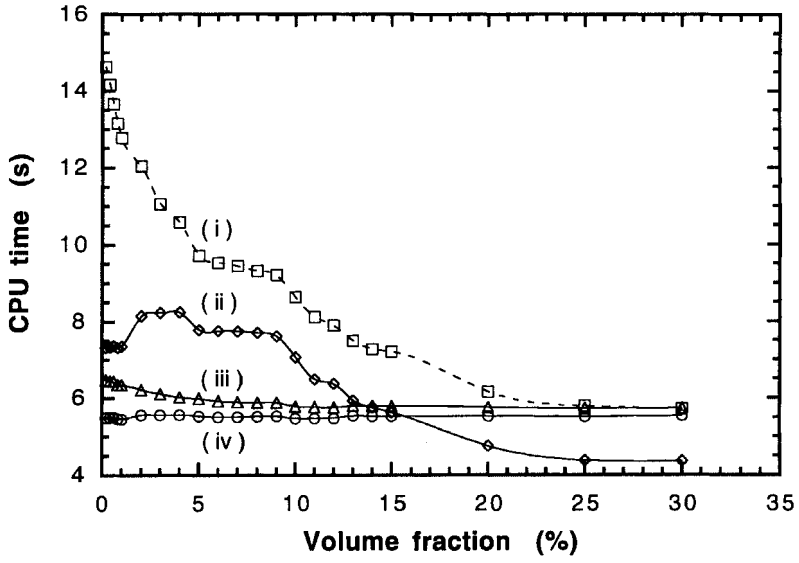

FIGURE 4a CPU-time on a DEC Alpha for cluster analysis with a system size of 1000 particles at different volume fractions using (i) Recursion as described in Section 2; the algorithms in (ii) Reference 7, (iii) Reference 8, and (iv) Reference 9.

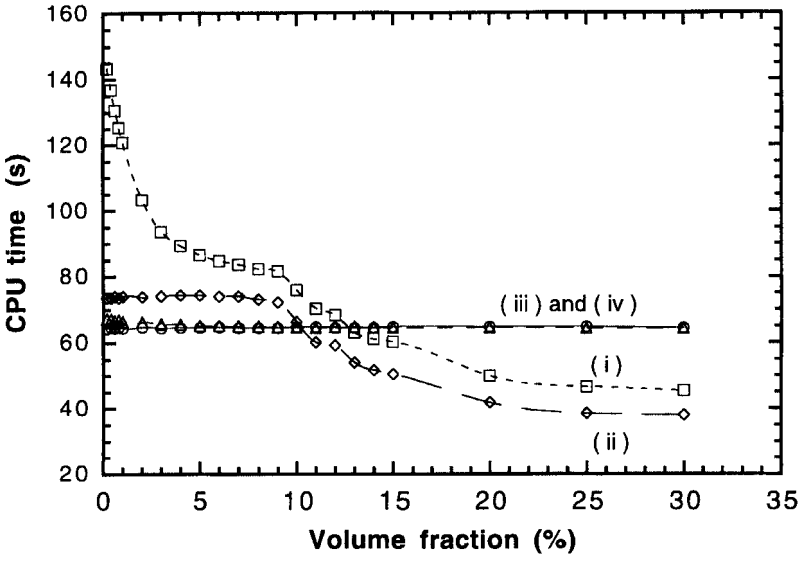

FIGURE 4b Same as in (4a) but for an IBM RS/6000.

volume fractions, recursion would still be a good choice, especially due to the transparency of the code. For lattice systems, however, HK should still be the best choice due to the low memory use and the $N \log(N)$ dependence on lattice size. Very recently, an interesting modification of the HK-algorithm has been described by Stoll which also works for continuous systems [27].

Using the same set of coordinates, we have also looked for percolation, comparing the algorithm by Heyes *et al.* [10] with the algorithm using recursion described in Section 3.1 and in Appendix A. In Figure 5 one can see that using non-optimized recursion is equal to or faster than the iterative method for a broad range of concentrations. The effect of optimizing the algorithm in Section 3.1 is shown in Figures 6a, b, where the original code has been modified by (i) starting the analysis from the particles at the border of the simulation box and (ii) end the search for percolating clusters as soon as the first one is found since in this case we are not interested in the total cluster statistics. At low volume fractions, the second criterion does not give any extra speed-up since one is below the percolation threshold which is at about 9%. When going to high volume fractions, though, terminating the analysis after having found a percolating cluster gives a substantial speed-up. By starting from the "border-particles" on the other hand, the CPU-time decreases considerably at low volume fractions. This does not, however,

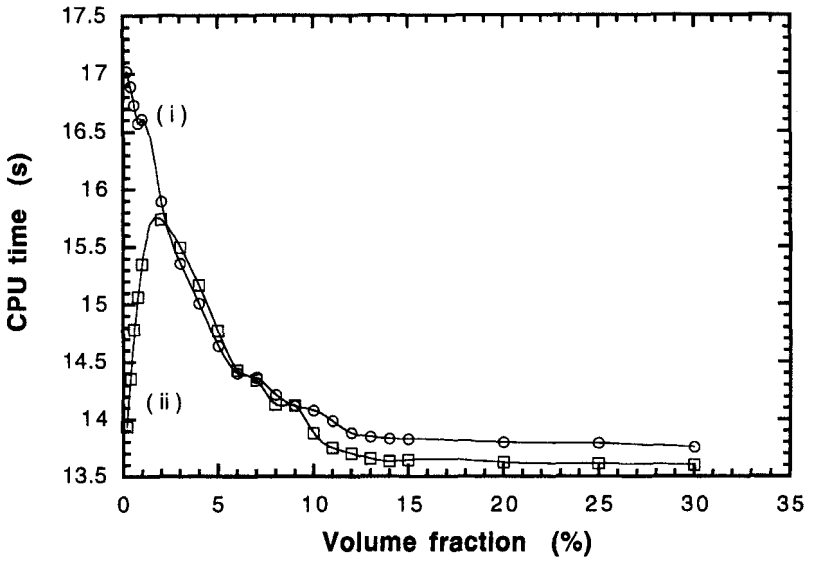

FIGURE 5 Comparing the DEC CPU time for percolation analysis with $N = 1000$ for (i) the algorithm in Reference 10 and (ii) the non-optimized recursive algorithm.

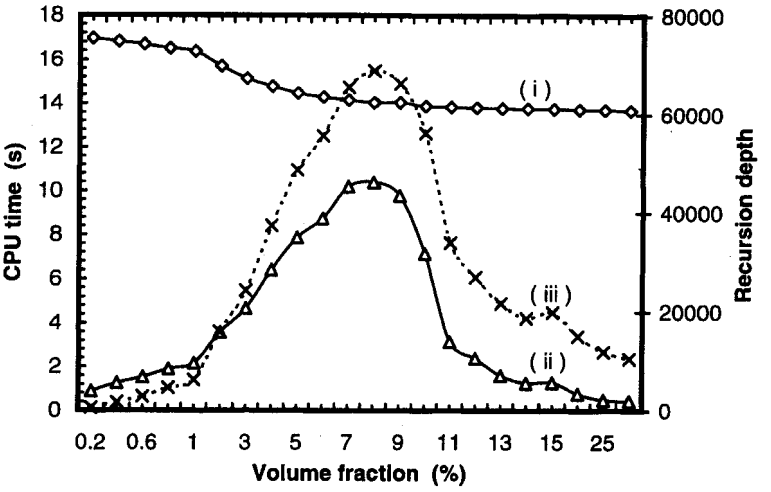

FIGURE 6a The DEC CPU time for percolation analysis with $N = 1000$ for (i) the algorithm in Reference 10 and (ii) the optimized recursive algorithm (iii) the recursive depth (number of calls) for the recursive algorithm.

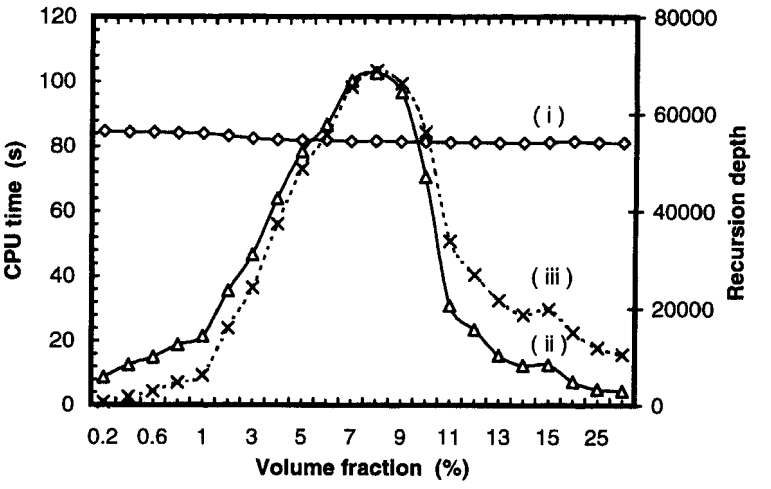

FIGURE 6b As in Figure (6a) but for an IBM RS/6000.

result in any significant decrease in CPU-time at high concentrations since most of the particles lying at the periphery of the simulation cell are now part of one (or more) percolating clusters and one has to go through essentially all particles even when starting with this smaller set. In this paper we have restricted ourselves to comparing only with the original algorithm of

Reference 10, implicitly assuming that modifications mentioned in this article also can be applied to other algorithms. Utilizing Condition (ii) above in the algorithm in Reference 10, however, results in a routine that is still seven times slower at high volume fraction compared to recursion.

When implementing either an iterative or recursive algorithm, it is important to realize how the compiler takes care of the stacks. The recursive routine performance is of course dependent on the recursive depth, mainly because the overhead needed for the stack at each level (*cf.* the recursive depth in Figs. 6a, b). To further increase the efficiency of recursive algorithms one can adopt certain depth limiting techniques but this is left outside of the scope of this paper. The computational time for the iterative approach on the other hand, depends on the size of the allocated arrays while the recursive method is largely independent on how the size of the arrays is defined. In Figures 7a, b the CPU-time for percolation analysis is shown when the size of the arrays in use exactly matches the number of particles (curves (i) and (iii)) and when all arrays are set to a system size corresponding to the maximum number of particles (curves (ii) and (iv)). It is clearly seen that the CPU-time for the iterative approach depends strongly on the exact allocation size of the arrays. This is true both for the implementation on the DEC and the IBM computer. The dependence is not

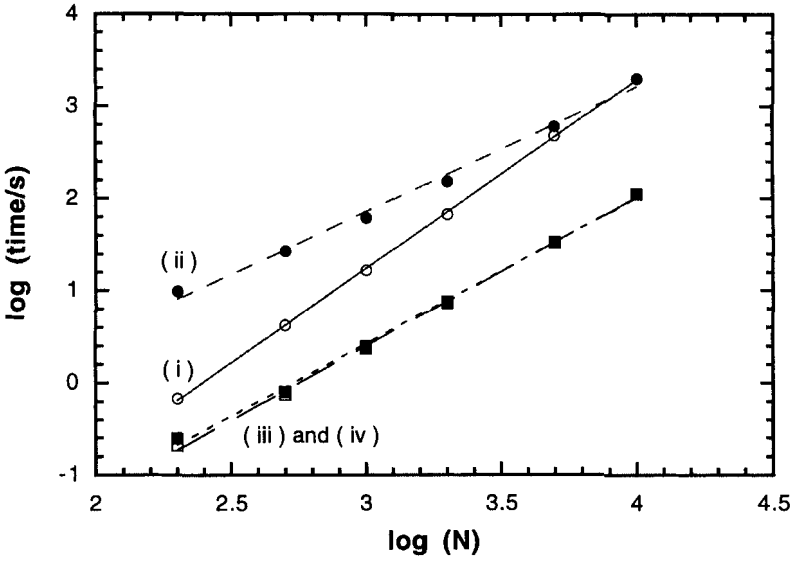

FIGURE 7a The time dependence for percolation analysis as a function of system size at 1% volume fraction. The different curves have been calculated using (i) iteration (Ref. 10) adjusting the array sizes to the actual system size, (ii) iteration with all arrays defined for a system size $N = 10000$, (iii) as in (i) but using recursion, and (iv) as in (ii) but using recursion.

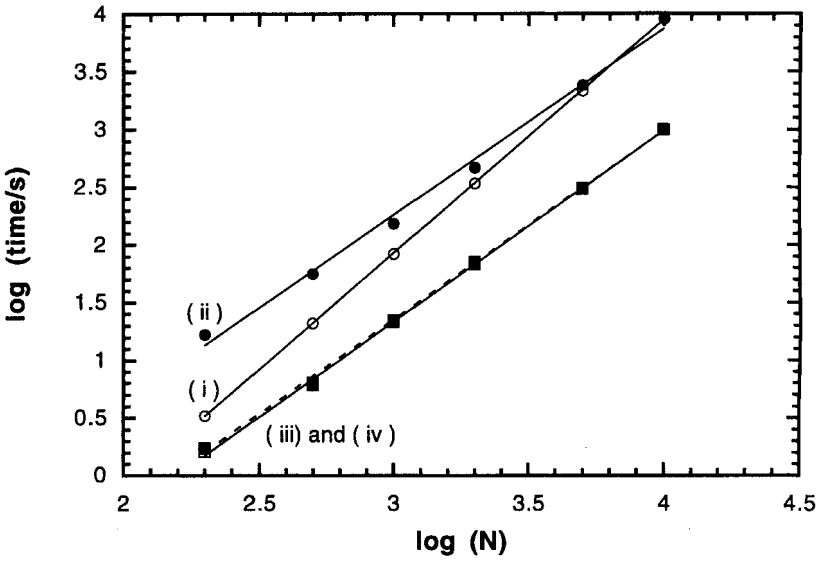

FIGURE 7b As in Figure (7a) for an IBM RS/6000.

strictly linear on a log→log scale, the time increasing even faster with increasing number of particles. One can also see that the curves corresponding to iteration merges when the allocation size approaches the exact system size, as they must. For $N = 1000$, iteration becomes four times slower when allocating arrays corresponding to 10000 particles compared to using the actual system size. In the corresponding analysis using recursion the increase is only 4%. For a 200 particle system, the increase is 1400% for the iterative approach while using recursion the increase is only 8%. Thus not only is recursion considerably faster when analysing percolation, but it is also a more robust approach since one can allocate the size of all arrays to the maximum to be used in a series of analyses, with only a very marginal effect on the CPU-time. Except for Figures 7a, b, all results presented using the algorithm by Heyes *et al.* [10], have been evaluated using the exact number of particles when defining the size of the arrays.

Finally we have also combined our cluster identification algorithm with the percolation algorithm into a single routine. The result of using this combined approach is shown in Figure 8. In this combined routine we can, of course, not use modification (i) and (ii) above because they affect the full cluster statistics. One can see, however, that even if we perform both the cluster and percolation analysis using recursion, it is still faster than or equal to the time for an iterative approach, even though the latter is only looking for percolating structures.

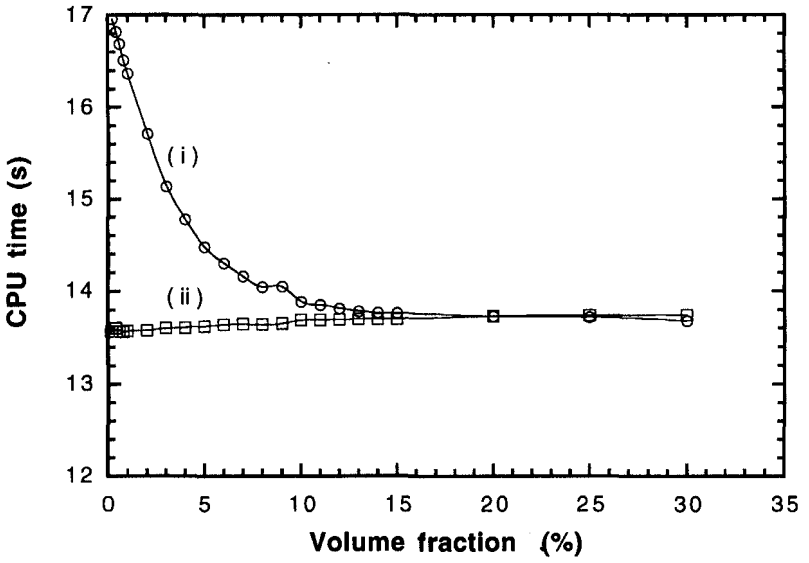

FIGURE 8 The CPU time for (i) percolation analysis as described in Reference 10 and (ii) the combined cluster and percolation analysis using recursion for a system size $N = 1000$.

5. CONCLUSIONS

We have presented an algorithm which recursively samples cluster statistics and determines the existence of percolating structures. The algorithm uses information about the boundaries that are crossed by checking in what direction the boundary conditions are applied. This information is then used to determine a percolation condition. If one is only interested in determining cluster formation and cluster size distributions, there are iterative methods that are faster at low volume fractions or running at about the same speed at higher concentrations. The cluster identification problem is, however, inherently recursive so that the present algorithm is more transparent than the iterative routines, and thus easier to maintain. The main advantage of using recursion, however, is seen when extending the cluster search routine into an algorithm for identifying percolating structures in an off-lattice system. In this case, using recursion together with the boundary information and the algorithmic definition of percolation will result in a CPU-time being several times smaller compared to an alternative iterative method for most concentrations. Recursion also results in a faster routine when combining the cluster identification and percolation algorithms. Finally we have also looked at the effect of array size allocation when implementing

the algorithms, which strongly influences the CPU time for the iterative algorithm examined in this paper, but has a negligible effect when using recursion.

Acknowledgement

This work was supported by the Swedish Research Council for Engineering Sciences.

APPENDIX A

Below is a pseudo code describing a recursive percolation analysis algorithm. The code is divided into two sections, the first is the main construct and the second is the recursive part which is called both by the main construct and the recursive part itself.

```
box_border = box_side−0.5 * clust_dist
do i = 1, tot_num_units

! Check that the particle is not already included in a cluster.
! The global logical array aggregating (i) has been set to true in
! the recursive subroutine if the particle 'i' is included in a cluster.

        if (.not. aggregating(i)) then

! Loop only over the particles that are close to the border.
! This border control and the exit and return statements below are only present
! to speed up the code and are not applicable if one wants the
! cluster statistics.

      if (abs(x(i)) > box_border .or. &
          abs(y(i)) > box_border .or. &
          abs(z(i)) > box_border) then
          box_id(:) = 0
          percolation = .False.
          CALL cluster_rek (tot_num_units, i, clust_dist_sq, percolation)
          if (percolation) then
            exit
          endif
        endif
      endif
end do
```

This part describes the recursive part which follows all connections in a cluster.

```
Recursive Subroutine cluster_rek(tot_num_units, unit, clust_dist_sq, percolation)
do j = 1, tot_num_units

    if (percolation) return
    if (j = = unit) cycle

! Calculate the distance between 'unit' and 'j'
! Save the absolute values of the distances without the periodic
! boundary condition as 'delta_x', 'delta_y', 'delta_z'
! Apply the periodic boundary condition
! Calculate the square distance, 'dist_sq', between 'unit' and 'j'

 if (dist_sq < clust_dist_sq) then

! If 'unit' is already aggregated, switch to that box

    if (aggregating(unit)) then
        box_id(1) = agg_box_id(1, unit)
        box_id(2) = agg_box_id(2, unit)
        box_id(3) = agg_box_id(3, unit)
    endif

! Shift the box identity if a border is crossed.

      box_id(1) = box_id(1) + int(abs(delta_x)) * sign(1.0, x(unit))
      box_id(2) = box_id(2) + int(abs(delta_y)) * sign(1.0, y(unit))
      box_id(3) = box_id(3) + int(abs(delta_z)) * sign(1.0, z(unit))

        if (aggregating(j)) then

! If a particle is already aggregated, check if it is the "same" 'j' or if it
! is a 'j' in another box. That is, if a particle is connected to its image in
! a periodic cell

      test = (box_id(1)/ = agg_box_id(1,j) .or. &
              box_id(2)/ = agg_box_id(2,j) .or. &
              box_id(3)/ = agg_box_id(3,j))
      if (test) then
        percolation = .true.
        Return
      endif
    else
```

```
! Else, if a particle is not aggregated

        agg_box_id(:,j) = box_id(:)
        aggregating(j) = .true.

! Now follow the connections within the present cluster recursively,
! i.e., let particle 'j' be the 'unit' to be investigated

          CALL cluster_rek(tot_num_units, j, clust_dist_sq, percolation)
        endif
    endif
End Do
END Subroutine cluster_rek
```

References

[1] Öttinger, H. C. (1996). "Stochastic Processes in Polymeric Fluids". Springer, Berlin.
[2] Leach, A. R. (1992). "Molecular Modelling". Longman, Harlow.
[3] Allen, M. P. and Tildesley, D. J. (1987). "Computer Simulation of Liquids". OUP, Oxford.
[4] Frenkel, D. and Smit, B. (1996). "Understanding Molecular Simulation". Academic Press, San Diego.
[5] Stauffer, D. and Aharony, A. (1992). "Introduction to Percolation Theory". Taylor and Francis, London.
[6] Hoshen, J. and Kopelman, R. (1976). "Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm", *Phys. Rev. B*, **14**, 3438.
[7] Stoddard, S. D. (1978). "Identifying Clusters in Computer Experiments on Systems of Particles", *J. Comp. Phys.*, **27**, 291.
[8] Fowler, R. F. (1984). "The identification of a droplet in equilibrium with its vapour", CCP5 Newsletter, No. 13.
[9] Rapaport, D. C. (1995). "The Art of Molecular Dynamics Simulation". CUP, Cambridge.
[10] Heyes, D. M. and Melrose, J. R. (1989). "Microscopic simulation of rheology: Molecular dynamics computations and percolation theory", *Molecular Simulation*, **2**, 281.
[11] Hill, T. L. (1955). "Molecular Clusters in Imperfect Gases", *J. Chem. Phys.*, **23**, 617.
[12] Coniglio, A., Stanley, H. E. and Klein, W. (1982). "Solvent effects on polymer gels: A statistical – mechanical model", *Phys. Rev. B*, **25**, 6805.
[13] Kokshenev, V. B. (1998). "Slow relaxation near structural and orientational transitions in glass-forming liquids and solids", *Phys. Rev. E*, **57**, 1187.
[14] Winslow, D. N. and Diamond, S. (1970). "A mercury porosimetry study of the evolution of porosity in Portland cement", *J. of Materials*, **5**, 564.
[15] Turkevich, L. A. and Cohen, M. H. (1984). "The Nature of the Phase Transitions in Expanded Fluid Mercury", *J. Phys. Chem.*, **88**, 3751.
[16] Mays, H. (1997). "Dynamics and Energetics of Droplet Aggregation in Percolating AOT Water-in-Oil Microemulsions", *J. Phys. Chem. B*, **101**, 10271.
[17] Mays, H., Almgren, M. and Brown, W. (1998). "Network Formation in Water-in-Oil Microemulsions Stabilised by the Amphiphilic Triblock-Copolymer $PEO_{13}PPO_{30}PEO_{13}$ in *p*-Xylene", Ber. Bunsenges. *Phys. Chem.*, **102**, 1648.
[18] Safran, S. A., Webman, I. and Grest, G. S. (1985). "Percolation in interacting colloids", *Phys. Rev. A*, **32**, 506.
[19] Perreau, M., Berthier, S., Peiro, J. and Lafait, J. (1997). "Percolation transition into random-Sierpinski-carpets", *Physica A*, **241**, 240.
[20] Stanley, H. E. and Teixeira, J. (1980). "Interpretation of the unusual behavior of H_2O and D_2O at low temperatures: Test of a percolation model", *J. Chem. Phys.*, **73**, 3404.

[21] Hunter, R. J. (1989). "Foundations of Colloid Science Vol. II", OUP, Oxford.
[22] Budd, T. A. (1994). "Classic Data Structures in C++". Addison-Wesley, New York.
[23] Sevick, E. M., Monson, P. A. and Ottino, J. M. (1988). "Monte Carlo calculations of cluster statistics in continuum models of composite morphology", *J. Chem. Phys.*, **88**, 1198.
[24] Bhavsar, S. P. and Splinter, R. J. (1996). "The superiority of the minimal spanning tree in percolation analyses of cosmological data sets", *Mon. Not. R. Astron. Soc.*, **282**, 1461.
[25] Ross, K. A. and Wright, C. R. B. (1992). "Discrete Mathematics 3rd edn.". Prentice-Hall, Englewood Cliffs.
[26] Papadimitriou, C. H. (1994). "Computational Complexity". Addison-Wesley.
[27] Stoll, E. (1998). "A fast cluster counting algorithm for percolation on and off lattices", *Comp. Phys. Comm.*, **109**, 1.