Simulation of percolation on parallel computers

Diploma thesis of Daniel Tiggemann
Institute for Theoretical Physics
University of Cologne
10th May 2001

Document typeset using $\LaTeX\,2_\varepsilon$.
Typeset process controlled by `make`.
Text files typed using `vi`.
Plots generated by `gnuplot`.

Simulation von Perkolation auf Parallelrechnern

Diplomarbeit von Daniel Tiggemann
Institut für Theoretische Physik
Universität zu Köln

10. Mai 2001

Zusammenfassung

Perkolation ist ein Gebiet der statistischen Physik (und anderer Fachgebiete), das seit sechzig Jahren untersucht wird. Obwohl Perkolation ein sehr einfaches Modell ungeordneter Medien ist, hat es doch eine nahezu unüberschaubare Zahl von Anwendungen, sodass das Interesse daran nicht nur akademischer Natur ist.

Perkolation ist auch deshalb ein wertvolles Modell der statistischen Physik, weil es trotz seiner Einfachheit eine Vielzahl von Eigenschaften aufweist, die auch in anderen physikalischen Systemen eine Rolle spielen: Phasenübergang zweiter Ordnung, Skalenverhalten, Universalität.

Glücklicherweise lässt sich Perkolation einfach erläutern: Man nehme ein Gitter und besetze die Gitterplätze zufällig mit einer Wahrscheinlichkeit p, bzw. lasse sie frei mit der Wahrscheinlichkeit $1 - p$. Gruppen von besetzten nächsten Nachbarn nennt man *Cluster*. Perkolationstheorie beschäftigt sich mit den Eigenschaften dieser Cluster. Wir haben hier also eine Definition, die stochastische mit geometrischen Aspekten verbindet. Dies führt zu einem Problem:

Unglücklicherweise gibt es keine exakte Lösung für Perkolation (im Allgemeinen). In einer Dimension, auf Bethe-Gittern und für gewisse Gittertypen in zwei Dimensionen gibt es exakte Lösungen, aber für eine Vielzahl von Dimensionen und Gittertypen ist man auf numerische Näherungsmethoden angewiesen.

Aufgrund der stochastischen Natur von Perkolation sind Monte-Carlo-Simulationen ein natürliches Werkzeug. Um möglichst präzise Ergebnisse zu erhalten, ist aber die Simulation möglichst großer Gitter notwendig. Dazu möchte man natürlich die leistungsfähigsten Computer der Welt nutzen.

Während bis Mitte der 70er Jahre des letzten Jahrhunderts noch keine Algorithmen bekannt waren, mit denen man vernünftige Monte-Carlo-Simulationen hätte durchführen können, änderte sich die Situation schlagartig 1976, als Hoshen und Kopelman einerseits und Leath andererseits die nach ihnen benannten Algorithmen vorstellten. Damit wurde eine Werkzeug geschaffen, das die numerische Untersuchung von Perkolation mit hoher Genauigkeit gestattet.

Um maximale Präzision zu erreichen, ist die Nutzung sehr großer Computer nötig; die leistungsfähigsten Computer unserer Zeit sind aber ausnahmslos Parallelrechner, wohingegen der Hoshen-Kopelman-Algorithmus für klassische sequenzielle Rechner entworfen wurde. Eine Übertragung auf parallele Rechner ist keineswegs trivial.

Hauptziel der vorliegenden Diplomarbeit war es, den Hoshen-Kopelman-Algorithmus so auf einen Parallelrechner zu portieren, dass damit Weltrekordsimulationen möglich wurden (d. h. die Untersuchung von Gittern, die größer waren als alle zuvor untersuchten).

Zu diesem Zweck wurde die Methode der Gebietszerlegung (engl. *domain decomposition*) gewählt: Das Gitter wurde in Streifen zerlegt, wobei jeder Streifen einem Prozessor zugeordnet wurde.

Der Hoshen-Kopelman-Algorithmus untersucht das Gitter Zeile für Zeile (bzw. Ebene für Ebene in drei Dimensionen), wobei in einem L^d-Gitter nur L^{d-1} Git-

terplätze abgespeichert werden müssen. Um möglichst große Systeme simulieren zu können (insbesondere in $d > 2$), war es nötig, das Gitter so zu zerlegen, dass jeder Prozessor nur einen Teil der L^{d-1} Gitterplätze abspeichern musste. Dadurch wurde zwar die Implementation kompliziert (da viel Kommunikation zwischen den Prozessoren nötig wurde), aber die Effizienz des Algorithmus' litt nicht signifikant darunter.

Durch diese Methode konnten die alten Weltrekorde von 2000000^2, 10001^3 und 611^4 deutlich verbessert werden. Damit gelang auch die sehr genaue Bestimmung einiger Systemeigenschaften, z. B. die Zahl der Cluster pro Gitterplatz n_c, der Fisher-Exponent τ für die Clustergrößenverteilung und der Exponent Δ_1 für die Korrektur zum Skalenverhalten (alles am kritischen Punkt p_c):

d	L	τ	Δ_1	n_c
2	4000256	187/91	0.70(2)	0.02759791(5)
3	20224	2.190(2)	0.60(8)	0.052442(2)
4	1036	2.313(2)	0.5(1)	0.0519980(2)

Im Rahmen dieser Simulationen wurden auch andere Aspekte wie z. B. der Einfluss von (schlechten) Zufallszahlengeneratoren oder der endlichen Systemgröße auf die Ergebnisse untersucht. Dabei bewahrheitete sich die alte Weisheit, dass man Simulationsresultaten mit Vorsicht begegnen muss; umfangreiche Überprüfungen anhand von Kontrollsimulationen sind notwendig.

Durch die erfolgreiche Parallelisierung des Hoshen-Kopelman-Algorithmus' ist es jetzt möglich, innerhalb weniger Stunden Systeme zu simulieren, deren Berechnung auf einem sequenziellen Rechner Wochen dauern würde oder sogar überhaupt nicht möglich wäre. Dadurch können noch viele Aspekte von Perkolation untersucht werden, die in dieser Diplomarbeit aus Zeitgründen keine Beachtung finden konnten.

Contents

Chapter 1

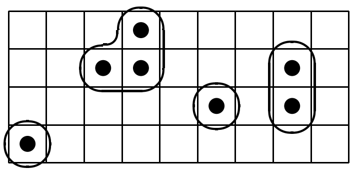
Introduction

1.1 What is percolation?

Percolation is a problem that can be easily defined, but which is difficult to solve.

Take a square lattice of L^2 sites. Each site is either occupied (with a probability p) or free (with probability $1 - p$), independent of other sites. A *cluster* is a group of neighbouring occupied sites, surrounded by free sites. Percolation theory deals with the properties of these clusters.

In the sketch below, occupied sites are marked by a bullet, clusters are marked by a surrounding line. We have two 1-clusters, a 2-cluster, and a 3-cluster. Throughout this diploma thesis, the number of sites in a cluster will be denoted by s, the number of clusters in a system that contain s sites each by n_s. Thus, below we have $n_1 = 2$, $n_2 = 1$, and $n_3 = 1$.



This is called site percolation. There is also bond percolation: the bonds between sites are occupied with probability p, and we define as clusters those sites that are connected through occupied bonds. Within this diploma thesis, only site percolation will be investigated; bond percolation differs only gradually, not principally. A good introduction to percolation theory can be found in [47], a short overview in [11, 44]. Some remarks on the history of percolation theory can be found in [19, 26].

Research in percolation started 1941 with Paul Flory investigating the gelation of polymers [17], although the term *percolation* was first coined by Broadbent and Hammersley in 1957 [9, 23, 24]; Broadbent was investigating gas masks, which sheds a first light on the diversity of applications for percolation theory (for more on applications, cf. [43]).

Because of the stochastic nature of the problem, Monte Carlo methods were a natural tool to be applied to percolation. Unfortunately, in the fifties and sixties, computers had strongly limited capabilities. Together with rather simple algorithms this yielded a situation where the Monte Carlo simulation of percolation was possible only for very small systems that did not show interesting behaviour. To cite Hammersley and Handscomb: "The direct simulation [of percolation] is out of the question" [25, p. 135]. Speed and memory capacity of computers grew exponentially over the last decades (Moore's law), but the real breakthrough for Monte Carlo studies of percolation came with sophisticated algorithms in the year 1976.

The algorithm of Leath [31, 32] generates a cluster from a seed site and uses a list of sites still-to-investigate for growing that cluster. The algorithm of Hoshen and Kopelman [27] generates a whole lattice in linear manner and uses a list of labels for accounting clusters generated on-the-fly.

The combination of modern algorithms with modern hardware made Monte Carlo studies an effective tool for dealing with percolation. It was possible to get results of high precision. This can be pushed further by utilising the modern parallel computers, which offer unrivaled performance. Unfortunately, they require new algorithms. One such algorithm will be described in this diploma thesis.

1.2 Phenomena of percolation

One reason why percolation is so popular in statistical physics is that it is a very simple, purely geometric and stochastic problem (to cite Stanley et al. [44]: "In principle, Archimedes could have studied percolation"), but shows the full set of phenomena found in other physical systems, like phase transition, scaling, and universality. Even more, the modern concept of renormalization can be easily demonstrated on percolation.

When there is a cluster that goes from top to bottom, we call this cluster "infinite", because if we increase system size, a cluster retaining this characteristic also increases in size, and for $L \to \infty$ the cluster would become infinitely large.

When we increase the occupation probability p from 0 to 1, we will recognize that for a certain probability p_c such an infinite cluster appears; below p_c there is none, above p_c there is one. Because of the sudden switch behaviour, we speak of a phase transition. p_c is the critical point.

Near the critical point, some system properties go with power laws. For example, the weight of the infinite cluster $P \propto (p-p_c)^\beta$, the mean cluster size $S \propto |p-p_c|^{-\gamma}$, or the correlation length $\xi \propto |p-p_c|^{-\nu}$.

Right at the critical point, the correlation length is infinite. This leads to the interesting situation that the system is invariant under real-space renormalization, in other words, when we rescale the system, it looks the same (a nice illustration of this self-similarity at the critical point can be found in [44]). So we expect a power-law for the distribution of cluster sizes right at the critical point, $n_s(p_c) \propto s^{-\tau}$. This power-law is modified for small cluster sizes s, as then the lattice spacing is an inherent length that breaks self-similarity; away from the critical point, the power-law is modified as the system no longer is self-similar on all length-scales. The full ansatz for the cluster size distribution is (cf. [45, 47])

$$n_s(p_c) = k_0 s^{-\tau} \cdot \underbrace{(1 - k_1 s^{-\Delta_1} + \dots)}_{\text{for small } s} \cdot \underbrace{f((p-p_c)s^\sigma)}_{\text{away from } p_c}, \tag{1.1}$$

where the second term is the correction for small clusters and the third for p away from p_c. Here we can notice another important property of percolation, which plays a central role in modern statistical physics: *universality.* This means that a special quantity, like a critical exponent, does not depend on microscopic details; i. e. it is the same for a square lattice and a triangular lattice. In eq. 1.1 τ is universal, while k_0 is not.

Scaling arguments relate several critical exponents with each other (cf. [46]): $1/(\tau-2) = 1 + \gamma/\beta$, where τ is the exponent for the cluster size distribution right at the critical point, β is the exponent for the size of the infinite cluster, and γ that for the mean cluster size (the exponents β and γ are not investigated here, as these require many simulations with slightly different p, which costs too much of the precious computing time; τ or Δ_1, on the other hand, can be extracted from a single run). The scaling function $f(z)$ is not investigated here for the same reason.

Above six dimensions, the cluster numbers n_s are expected to follow mean-field theory, for which the critical exponents are the same for all dimensions $d > 6$ (for this reason, $d = 6$ is called the upper critical dimension). Additionally, $f(z)$ is expected to be a gaussian. But recent numerical work [55] showed that i. e. seven dimensions are not yet fully understood. However, due to the limited time of one year for a diploma thesis, it was not possible to implement the algorithm presented here for more than four dimensions.

Chapter 2

Parallelizing the Hoshen-Kopelman algorithm

2.1 The Hoshen-Kopelman algorithm

The Hoshen-Kopelman algorithm examines a lattice in linear fashion, site after site. It can be used to count clusters in an existing lattice (that is, experimental data), but in Monte Carlo studies the lattice is generated on-the-fly (when we examine a site, we roll the dice and decide by this if it is occupied or free). In this case, one main advantage of the algorithm is that we do not have to store the whole lattice in memory, but only one line in two dimensions, one plane in three dimensions, and more generally: if we are examining a L^d lattice, we only have to store L^{d-1} sites, which yields a big advantage for memory consumption in low dimensions.

Let us now examine a small lattice using the Hoshen-Kopelman algorithm. Bullets mark occupied sites:

●			●
●		●	●

We now go through the lattice line by line, and within each line from left to right. Whenever we encounter an occupied site that is not connected to another occupied site to the left or to the top, we say that this site starts a new cluster and assign it a new number as cluster label, starting from 1. On the other hand, when it has an occupied neighbour to the left or top, it inherits its cluster label from that neighbour. Thus, after seven examined sites our lattice looks like

1			2
1		3	?

The eigth site has two occupied neighbours with different cluster labels, so we have to decide which of them shall be the new cluster label for the site currently in examination. When we choose label 2, we also have to renumber the sites carrying the label 3, because our assumption that they were different clusters showed as wrong. For large clusters, this would require a lot of work.

The genuine idea of the Hoshen-Kopelman algorithm is to let the sites labeled as they are and instead write down a notice that clusters 2 and 3 belong together. In practice this is done by using a seperate data structure that holds information about the cluster labels: for a direct or “root” label, it records the number of sites within that cluster, for an indirect or “non-root” label, it records to which “real” cluster label this label belongs. This distinction is made within the label list simply by the sign of the integer number.

After the whole lattice was examined, our supplementary data structure contains all information we need: Each “real” cluster corresponds to a root label, which also records the number of sites in that cluster. All non-root labels point directly or indirectly to a root label. They carry no information, as their only purpose was to spare us the costly renumbering of sites.

Because we investigate line by line and only need information for the left and top neighbour of the site in investigation, we only need memory for one line of size L^{d-1} when investigating a lattice of size L^d. Additionally, we also need memory for the supplementary data structure containing information about the labels. A lot of space within this data structure is occupied by non-root labels that carry no information, but are only a trick that speeds up simulation. When doing huge simulations, it is thus a good idea to *recycle* this wasted space by relabeling the plane currently in investigation with root labels only and then throwing away all non-root labels. Even more, we can mark all root labels within our data structure that are present in the currently examined hyperplane, and afterwards throw away all non-marked root labels, as they belong to clusters that “died out” above the current hyperplane. This method is known as NAKANISHI recycling (cf. [37]).

By using the Hoshen-Kopelman algorithm with Nakanishi recycling, it was possible to simulate percolation on impressively large lattices, as for low dimensions not computer memory, but only computer speed was a limiting factor. With the advent of powerful supercomputers, Monte Carlo techniques proved as a useful tool for studying percolation that allowed extremely high precision for the determination of interesting properties. This allows us to reverse Broadbent’s remark on Monte Carlo studies of percolation [8] “The capacity of computers is, however, insufficient for any but small lattices. This is another example of the authors’ remark that pen and paper might be better than machine work” to the computer programmers’ remark “Machine work might be better than ink and paper”.

A rather new trend in supercomputing are massively parallel computers. They emerged as a tool for general purpose computing with the beginning of the 1990s. They offer unrivaled performance for a rather low price, but they have one major disadvantage: Traditional algorithms were designed for sequential, single-processor computers and cannot simply be used on massively-parallel computers. Instead, massively-parallel processing (MPP) requires completely new or at least heavily restructured algorithms. This is the main reason why MPP is not as widespread as one would expect.

On the other hand, sometimes it is reasonable to put some effort into porting algorithms to MPP. This is true also for percolation, because more speed or more memory for simulating a larger lattice means a higher precision for determining properties of interest. There are several ways to parallelize the Hoshen-Kopelman algorithm in a reasonable way, some of them were already presented in literature [15, 16, 20, 22, 29, 51, 54]. In this diploma thesis, a new, rather complicated, but promising way was chosen. Using this algorithm, it was possible to achieve new world records in simulated system size, which substantially improved upon the old world records.

2.2 Cutting the lattice into strips

One of the major limitations for the Hoshen-Kopelman algorithm in higher dimensions is memory, or lack thereof. The old world record size for a simulation in four dimensions was 611^4 (cf. [50]), which requires approx. 1 GByte only for storing the hyperplane of investigation, aside from more memory needed for supplementary data structures. Pushing this world record further would require huge amounts of memory not available in standard sequential computers.

Fortunately, massively parallel computers offer the neccessary amounts of memory. Unfortunately, they use a programming model of distributed memory, where the whole memory is divided into partitions onto which only single processors have direct access; access by other processors has to be done by message passing, which requires explicit parallel programming.

On distributed memory machines, for implementing algorithms that operate on regular data structures like lattices, the standard method is *domain decomposition.* The lattice that shall be simulated is cut into several domains, and each domain is assigned to one processor and its local memory. When sites from one domain interact with sites from another domain, these interactions have to be programmed using message passing. Interactions within a domain are programmed like in a conventional algorithm.

For the Hoshen-Kopelman algorithm, there are several reasonable ways for decomposing the lattice. As the algorithm walks through the lattice hyperplane by hyperplane, it makes sense to classify the different resulting domains into those parallel and those perpendicular to one such hyperplane.

A decomposition into parallel (or "horizontal") strips would offer one big advantage: within each domain, all interactions would be local and no message passing is required. Only after the whole domain was investigated, communication between the domains resp. processors is neccessary. This allows for the easy implementation of the Hoshen-Kopelman algorithm, as the local part within the domain is simply the standard algorithm for sequential computers. But there is also one disadvantage: each processor has to store one full hyperplane. For high dimensions, this would require too much memory (even in three dimensions).

On the other hand, a decomposition into perpendicular (or "vertical") strips would divide the hyperplane into pieces, so that each processor has to store only a small amount of data. We could thus simulate larger lattices. Of course, this advantage comes at a price: during the simulation sites from different domains interact with each other and so message passing becomes an inherent ingredient of our algorithm. In other words: the algorithm would be much more complicated. But it is worth the effort.

2.3 Inventing a complicated algorithm

The main problem when decomposing the lattice into vertical strips is that sites from different strips can interact in a non-regular manner. For example, a cluster which was local in a strip gets in contact with a cluster from the left strip. Those two need to be joined, which makes communication neccessary. Or even worse, a cluster from the left strip and a cluster from the right strip join in the middle strip.

When designing algorithms for massively-parallel computers, it is important to keep in mind the limitations of message passing: delivering messages is about one or two orders of magnitude slower than direct access to local memory. Even worse, many small messages require much more time for delivery than one large message. It is therefore a good idea to bundle messages.

When investigating its piece of the hyperplane, each processor should defer communication until it has finished investigation; this is the local part. After this, all processors exchange the information in a regular manner. This is the reason why the algorithm becomes complicated, but this complexity is neccessary for efficency.

We introduce the notion of *local clusters* and *global clusters.* Local clusters are clusters in the lattice, whose occupied sites all lie in the same strip. Global clusters consist of occupied sites which are distributed among several strips.

The local clusters can be handled like in the sequential Hoshen-Kopelman algorithm. Only when they extend to the border of the strip, we have to find out if they become global (by means of communication). With the global clusters we have to be careful: When modifying global clusters during the local part, we later have to inform the neighbour strips (those in which parts of the cluster are present) about possible changes.

We extend our supplementary data structure of labels: There are no longer only non-root and root labels, but root labels are divided in local ones (corresponding to local clusters) and global ones (corresponding to a part of a global cluster). Of course, each processor has its own local array of labels. A global label records the number of sites in that cluster within that strip, the left neighbour (that is, the global label in the left neighbour strip that corresponds to the same global cluster) and the right neighbour. Left neighbour or right neighbour can be void in a global label, but not both, because in that case it would describe a local cluster.

When adding a site or a whole local cluster to a global cluster during the local part, we simply record the number of added sites in the global label. But when two global clusters join, we have to inform the neighbours about this change. Let us call this process "pairing", as a pair of clusters is joined forever.

When the local part is finished, the borders of the strips have to be examined, in order to find out if there are interconnections between clusters in different strips. In such cases, local labels can be converted to global ones.

Even more, let us examine the following situation: In our strip, we have to different global labels that are connected to two different global labels in the left neighbour strip. Now these two clusters join during our local part. It is easy to achieve that these two different clusters are joined within our strip, but we also have to inform our left neighbour, because the two labels in that strip have to be joined, too. And what if they also have connections to the next left strip? We have to pass the information even further. In order to avoid such complex communication patterns, we once again use the method of deferred information exchange: we store the information that two global labels have to be paired in a special data structure and exchange these data with our nearest neighbours after the local part. When this triggers the pairing within the next-nearest neighbours, our nearest neighbour puts a note into its data structure and informs the next-nearest neighbour one local part later. Thus, the neccessary information for pairing large global clusters (that span several strips) trickles along the strips step by step. Of course, when we need to rely on the fact that all global labels are correctly paired, we have to do a lengthy relaxtion process: we repeat the nearest-neighbour pairing over and over again, until there is no longer any pairing information exchanged between any strips.

So, our algorithm now looks like this: Within the strip, do the normal Hoshen-Kopelman algorithm in our part of the hyperplane. Whenever two different global clusters join, put an entry into our pairing data structure. After the local part is finished, exhange the borders with our neighbour strips and find out if there are interconnections between the strips. In that case, convert local clusters to global ones (if they are not already global). Exchange pairing information with our neighbour and do the pairing. If new pairing information arises, simply record it; we will exchange it after the next hyperplane.

2.4 To make matters worse: Recycling

When simulating large lattices, we have to keep memory consumption low. Unfortunately, much memory is wasted for non-root labels. On sequential computers, we can recycle this memory easily (using Nakanishi recycling). On parallel computers, this becomes a difficult and complicated task.

After we have relabeled our current hyperplane with root labels only (both local and global ones), we can safely delete all non-root labels that point to local root labels, as these correspond to clusters within our strip that were never in touch with other strips (otherwise they would point to a global root label). On the other hand, we must not delete non-root labels that point to global root labels, as they were possibly in touch with labels of other strips, and those other strips could reference them still (avoid "dangling pointers").

So, one prerequisite for recycling "global" non-root labels is to replace all references to global non-root labels by references to global root labels. We do that as follows: we walk through the list of our global labels and put their pointers to left (resp. right) neighbours in a message, which we send to the left (resp. right) processor. This one reclassifies all that labels and sends the message back, so that we can replace the old references to neighbour labels by the reclassified ones. After this process, all global labels reference only root labels in other strips, which allows us to safely delete all non-root labels.

Local root labels can be easily recycled the same way as in the sequential Hoshen-Kopelman algorithm: all local root labels that are still present in the current hyperplane are marked, all non-marked local root labels can be deleted. Of course, we must not mix local and global root labels.

The number of global labels generated is roughly proportional to the size of the interface between the strips. For higher dimensions, this means that we need to recycle even global root labels (for two dimensions, this is not neccessary). We do this by *reduction* of global clusters: a global cluster extends over several strips. In the strip that contains the right end of the cluster, we investigate if the part of that cluster in that strip is still alive (present in the current hyperplane); if not, we recycle it and inform the left neighbour of that fact (we also send the number of sites present in the recycled part, so that it can be added to the still-alive part). When the left neighbour that receives the message has itself no left neighbour, it can be safely converted to a local cluster (it just has lost the right neighbour). During the next recycling, it can be discarded in the local recycling process.

2.5 Counting of clusters

After we have done the whole simulation, we have to count the clusters that we have detected, by examining the list of labels. Due to the parallelization, this is more complicated than in a sequential simulation, as some clusters are distributed over several strips, having root labels in each. These have to be joined, so that they can be correctly accounted. We do this by a *concentration* process: Each processor examines its global root labels. For each such label that has a left neighbour, it sends the number of sites of that label to the neighbour (together with the number of the corresponding label in the left strip) and records that the label no longer carries sites. It then receives the data from its right neighbour and adds the sites to the corresponding global label. By repeating this process, the number of sites for a global cluster are concentrated in the leftmost strip the cluster extends to. After this, clusters can be counted locally in each strip; the obtained data is added later by one single processor.

One exception is the infinite cluster, as this can extend over all strips and wrap around to itself. In that case, it cannot be concentrated. This allows us for an easy detection of connectivity: If we discover after the concentration that there is one cluster which has not been concentrated, then this is the infinite one. We sum it up by investigating the corresponding labels within all strips.

2.6 A step-by-step description of the algorithm

The following list is a semi-formal description of the algorithm. Local and communication part are repeated for each hyperplane the system consists of, recycling is done whenever neccessary after the local and communication part, and counting is done after the full system was examined.

1. *Initialization*: Occupy the zeroth plane for busbar, if desired; initialize all data structure; etc.

2. *Local*:

   (a) Examine the strip site by site. Do labeling.

   (b) When two different global clusters join at one site, generate pairing information for left and right neighbour, but defer communication until after the local part.

3. *Communication*:

   (a) Exchange borders of strip with neighbours.

   (b) When two clusters of both strips join, convert clusters to global. If they are already global, but not yet connected, generate pairing information.

   (c) Exchange pairing information. Pair global labels that belong together. During this, new pairing information can come up.

   (d) Check if recycling is neccessary due to tight memory conditions.

4. *Recycling* (if neccessary):

   (a) Reclassify the current hyperplane with root labels.

   (b) Delete all non-root labels that point to local root labels.

   (c) Reclassify the pointers to left and right of the global root labels by asking the neighbours for the corresponding root labels.

   (d) Delete all remaining non-root labels.

   (e) Mark all living local root labels and delete the non-marked ones.

   (f) Look for all global root labels that are not present in the current hyperplane and have no right neighbour; delete them and send the number of sites to the left neighbour.

   (g) When a global label is informed that its right neighbour was deleted, and it has no left neighbour, convert it to local.

5. *Counting*:

   (a) Count local clusters.

   (b) Concentrate global clusters.

   (c) Count global clusters.

   (d) Look for a global cluster which has not been concentrated. If it exists, we have connectivity. Sum up this cluster explicitly.

   (e) Do output.

2.7 Other ways of parallelizing Hoshen-Kopelman

There are, as mentioned above, certainly other ways of domain decomposition. Old work (like [15, 22, 29]) did parallel cluster counting for implementing Ising models with Swendsen-Wang dynamics, which cannot be compared directly with percolation (but is of course inspirative). A recent work of Teuler and Gimel [54] did investigate percolation, but the authors did store the full lattice instead of only one plane, which restricted them to rather small lattice sizes. A presumably still-in-progress work by MacIsaac and Jan (private communication) tries to use a domain decomposition in strips parallel to the hyperplane of investigation, a natural counterpart to the decomposition chosen within this thesis. Their approach should be easier to implement and more efficient in execution, as communication is needed only after the full Hoshen-Kopelman examination of the strip, and not after each investigated hyperplane. However, world record sizes for simulations will be possible only in two dimensions.

Chapter 3

Results of Monte Carlo studies

3.1 Typical errors in Monte Carlo data

When analysing Monte Carlo data, it is important to keep in mind that the results are influenced by several types of errors, namely:

- **Statistical errors**: Due to the stochastic nature of Monte Carlo methods, there are deviations from the "theoretically exact" values. These are completely normal. We can find out about these errors by doing many runs with different random numbers and then averaging over these generated values y_i, yielding $< y >$ as a good estimate for the value without statistical errors; $\Delta y = \sqrt{(< y^2 > - < y >^2)/(N-1)}$ is called the statistical error of $< y >$ and gives an estimate, how strong $< y >$ would change, if we add another statistically independent value y_i. The larger the system is, the smaller become the fluctuations. This makes simulations of huge lattices reasonable; even if we can do only few runs or even only a single run of that size, obtained values have high precision.

  We can estimate a probable statistical error by simulating smaller systems and extrapolating the errors to larger sizes. In many cases we will find that other sources of errors have greater influence than the statistical error.

- **Finite-size errors**: As all computers known to mankind have only finite memory and computing speed, we can only simulate finite systems. Such finite systems can show rather different behaviour than idealized infinite systems, especially if the systems are very small.

  Although these finite-size corrections can be very interesting in their own right (sometimes an infinite system is easier to handle with analytical methods than a finite system, but corresponding finite systems show a more complex and interesting behaviour), for studying percolation we are interested in the behaviour of an inifinite system. We can simulate finite systems of different sizes and extrapolate to infinity; but such extrapolations have to be done with caution. It is a good idea to try to estimate the finite size correction with other means, for example to compare the theoretically expected behaviour with the real one.

  When simulating large systems, finite-size corrections should become small. It is often very expensive to extrapolate the finite-size corrections to high precision, as this requires simulating lattices of various, very large sizes. A rough estimate should be enough for many purposes, to find out which type of error is the most important one.

- **Systematic errors**: These are the most problematic ones and they are very hard to deal with. Systematic errors arise when we do the simulations different than we really would like to do them, and in a systematic fashion. In some sort of sense, finite-size effects are also systematic errors: we try to obtain properties of infinite systems, but examine only finite ones. But as finite-size errors are easy to understand and rather easy to deal with, they have their own category.

  The most classical source of systematic errors stems from bugs in the program code. This is not the only reason why programs should be thoroughly tested after they were written.

  Another common source of systematic errors in Monte Carlo simulations are the pseudo-random number generators (PRNGs). In many cases, they are not as "random" as they should be. They can show short-length correlations (if one site is determined to be occupied, the next one has a higher probability to be occupied, too, thus favouring large clusters), long-range correlations (after $N >> 1$ random numbers, the sequence is simply reproduced, thus reducing effective system size), or medium-range correlations (every Nth site has an above-average probability to be occupied; when lattice size L is approx. N, unwanted structures are formed).

  In practice, all PRNGs show correlations of these kinds, but to a different degree. Choosing the right one depends on many factors: for example, the quality of random numbers depends also on lattice size (due to medium-range correlations). To make matters worse, some PRNGs are good, but very slow. In general, for small systems the most PRNGs are suitable, but for large systems, correlations can show devasting effects.

  It is sometimes helpful to use a "voting method": do the same simulation with different PRNGs and look if they all agree; if one significantly differs from the others, it is bad. This requires several runs and is thus not suitable for huge lattices. Unfortunately, especially huge systems show problems with random numbers.

  Systematic errors are so difficult to handle because they are hard to detect. They cannot simply be averaged out by doing several runs. There are many sources for systematic errors: in general, whenever we simplify a realistic systems in order to make it suitable for simulation, we generate systematic errors.

  It is important not to be overly optimistic and not to claim small error margins for a value, just because the statistical error is small: careful search for systematic errors is neccessary, for example by testing data against theoretical assumptions, by comparing with exact data where possible, or by other methods.

3.2 Cluster size distribution

We expect n_s, the number of clusters of size s, to follow a power-law: $n_s \propto s^{-\tau}$, with τ, the so-called Fisher exponent (cf. [14]) being a universal constant, only depending on dimensionality. To make handling of Monte Carlo data easier, we do not store all n_s for all s, but instead we gather these data in bins: the first bin stores n_1, the second $n_2 + n_3$, the third $n_4 + \ldots + n_7$, and so on. By growing these bins exponentially, we obtain an easily to handle amount of data even for very large simulations. Analysis of binned data is easy:

$$N_s = \sum_{s'=s}^{\infty} n_{s'} = \sum_{s'=s}^{\infty} (s')^{-\tau} \simeq \int_s^{\infty} (s')^{-\tau}\,\mathrm{d}s' = s^{-\tau+1}$$

By plotting the summed up cluster numbers, we can easily obtain all interesting information. When plotting the cluster size distribution double logarithmically, we expect from the power-law to see a straight line with slope $-\tau+1$. This is indeed the case, but it is not honest to judge from such a plot that the power-law is fulfilled well, as deviations from the law are hidden by the logarithmic scale. It is more honest to divide the real data by the expected behaviour and to plot the results on a linear scale (we still plot the x-axis representing s logarithmically, as our bins are growing exponentially in size, yielding equidistant points). In such a "honest" plot we see easily that our data are influenced by two effects: corrections to scaling for small s and finite-size effects for large s.

In two dimensions, the value of τ is supposed to be known exactly (cf. [38, 39, 40, 42]): $\tau = 187/91$. All our Monte Carlo data agree well with this value. In higher dimensions, there are no exact values known for τ, so we have to extract them from our data. Of course, if we have to extract more values from given data, the error margins for the values will increase. Due to this, our results for two dimensions are more precise than those for higher dimensions.

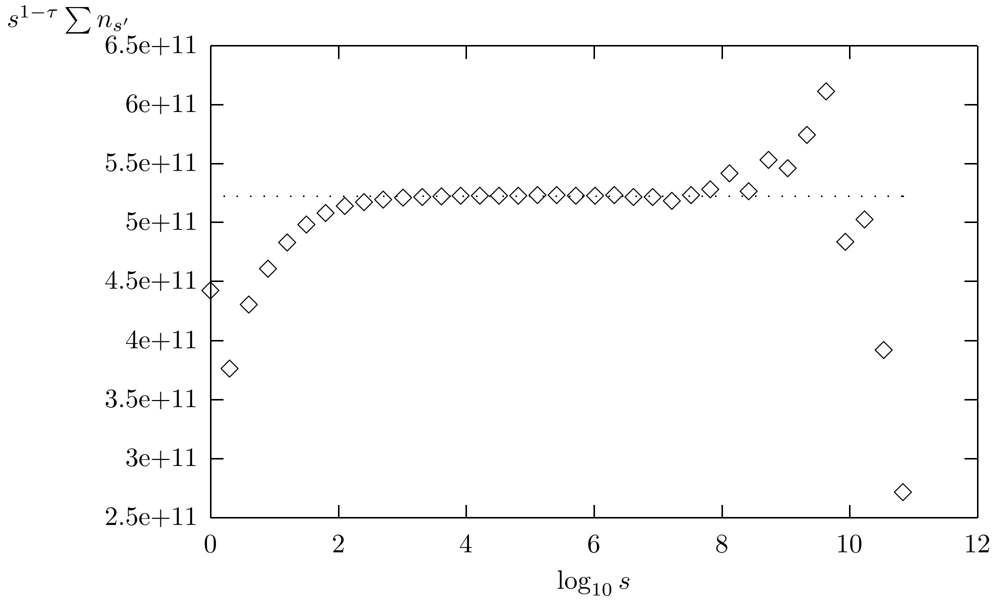


Figure 3.1: Cluster size distribution in two dimensions for world record size $L = 4000256$, using the Kirkpatrick-Stoll PRNG. The dotted line corresponds to the asymptotic behaviour.

We can extrapolate the asymptotic behaviour with higher precision, when we also take into account the corrections to scaling. By doing this (as described in the next section), we not only get good estimates for τ, but we can also better guess the error margins for τ.

For two dimensions, we find with high accuracy that our τ agrees well with the presumably exact $\tau = 187/91$.

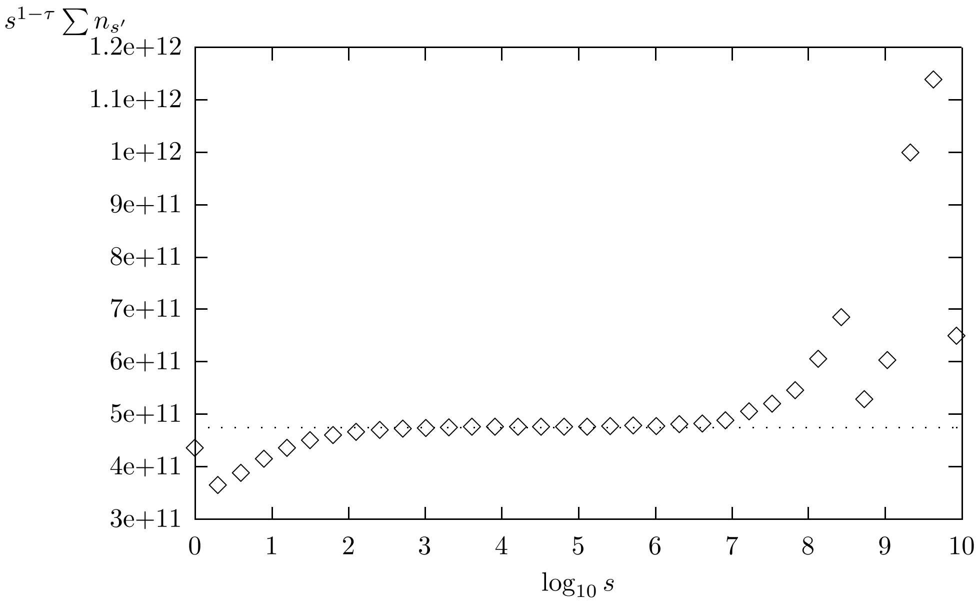


Figure 3.2: Cluster size distribution in three dimensions for world record size $L = 20224$, using the Kirkpatrick-Stoll PRNG. The dotted line corresponds to the asymptotic behaviour.

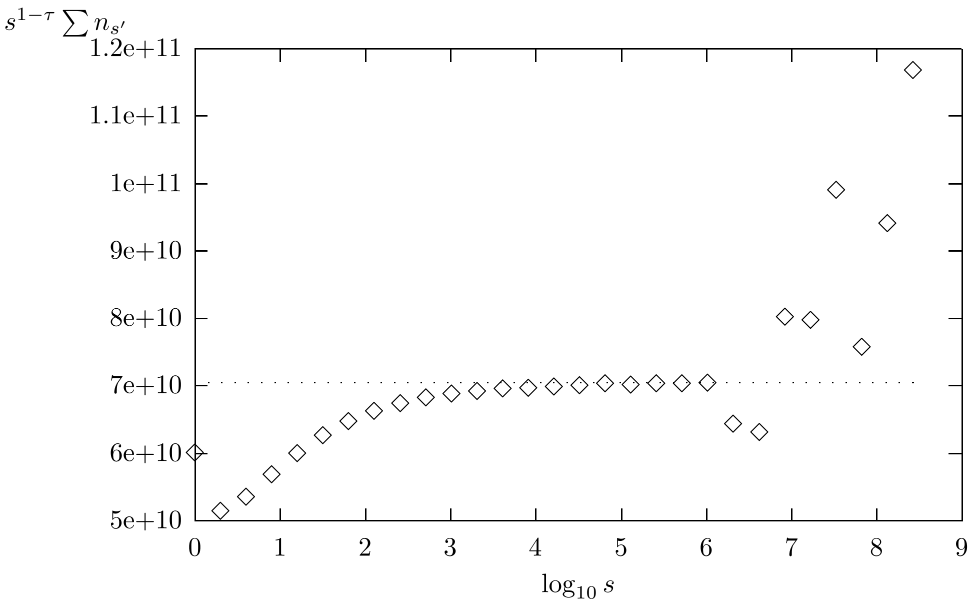


Figure 3.3: Cluster size distribution in four dimensions for world record size $L = 1036$, using the Kirkpatrick-Stoll PRNG. The dotted line corresponds to the asymptotic behaviour.

For three dimensions, we find $\tau = 2.190(2)$, which is roughly compatible with the old literature value $\tau = 2.186(2)$ found by Jan and Stauffer in [28], and $\tau = 2.189(2)$ from Lorenz and Ziff in [34] (they investigated bond percolation, but as τ is universal, their value is the same as for site percolation). Strangely, Gimel et al. took $\tau = 2.189$ as exact when analysing their $3d$ data, instead of trying to extract it from the data in [18].

In four dimensions, we find $\tau = 2.313(2)$, compatible with $\tau = 2.313(3)$ from Paul et al. [41], and $\tau = 2.3127(7)$ from Ballesteros et al. [5].

3.3 Corrections to scaling

The behaviour $n_s \propto s^{-\tau}$ is valid only for large s. The reason is simple: For this scaling behaviour to be exact, we need the condition that there are no inherent length scales, or in other words: when we renormalize our system, it should look the same (right at the critical point); if there is an inherent length, then it will be renormalized, too, and the system looks different.

One such length is the finite size of our system; this influence, which leads to finite-size corrections, will be covered in a seperate section.

Another length is the lattice spacing a (in this case $a = 1$, as we simulate not a real system, but an idealistic model). For small clusters, which are of size $s \simeq a$, renormalization would have a great effect (for example, a 1-cluster would vanish after renormalization); for large clusters, this effect gets smaller. A cluster, whose linear dimension is much larger than 1, should not be affected significantly by small-cell renormalization, or in other words: it does not "feel" the lattice spacing $a = 1$.

Small clusters should be heavily influenced by lattice spacing, thus we expect n_s for small s to deviate from the power-law $n_s \propto s^{-\tau}$.

Such deviations are expected to be non-universal, as they depend on microscopic details: i. e. the deviations should be different for triangular and square lattice.

The expected behaviour for small, but not too small clusters is (cf. [2])

$$n_s \propto s^{-\tau}(1 - k_1 s^{-\Delta_1})\,.$$

The correction term is called corrections to scaling, it could stem from an irrelevant operator or from a nonlinear scaling field (cf. [3, 7, 36]). If a nonlinear scaling field was the only cause, one would expect quantitatively $\Delta_1 = 55/91 \simeq 0.6044$ in two dimensions. This simple assumption is not compatible with Monte Carlo results.

To find good estimates for k_1 and Δ_1 (while we are mainly interested in Δ_1), huge lattices are very helpful, as finite-size effects make data analysis difficult (cf. fig. 3.4).

By taking into account the corrections to scaling, we also get a better estimate for τ. This is the case, because in the plot of the corrections to scaling we only get a straight line (for small s) when we choose k_0 and τ with high precision. Small deviations from the correct values will bend the straight line to one direction or the other. This is shown in fig. 3.5.

Of course, when we have to extract more parameters from our data, the error margins will become larger. As an example, Gimel et al. have taken $\tau = 2.189$ to be exact in three dimensions, instead of extracting it from the data. For that reason, they found $\Delta_1 = 0.65(2)$ with high precision, compared to our $\Delta_1 = 0.60(8)$; but for that reason, their error bars seem to be overly optimistic.

The results obtained from the world record simulations are: in two dimensions $\Delta_1 = 0.70(2)$, ruling out the simple nonlinear scaling fields assumption as the only source for corrections to scaling [3], as this would require $\Delta_1 = 55/91 \simeq 0.6044$.

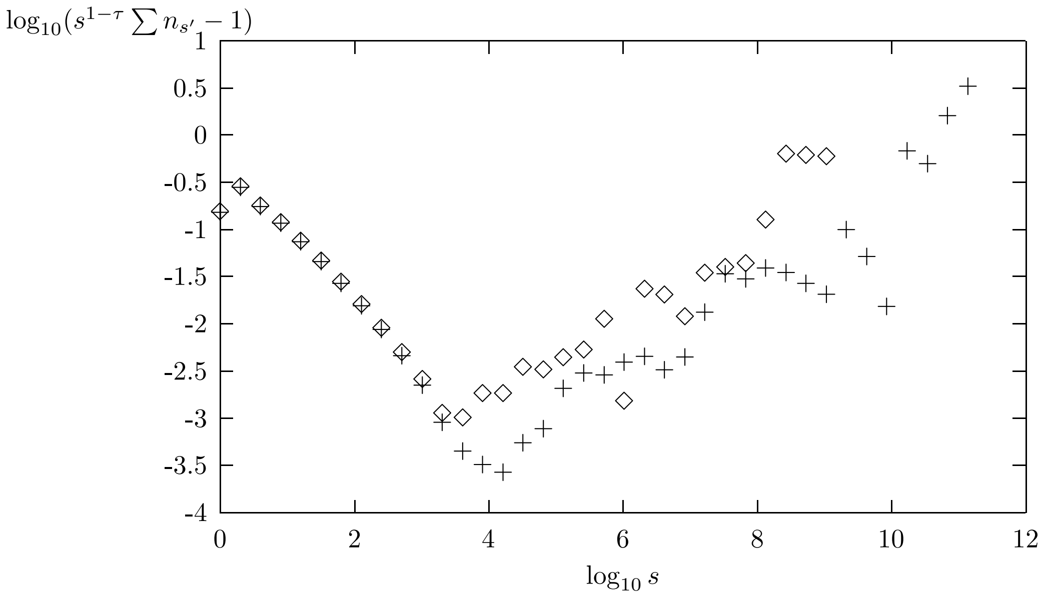


Figure 3.4: Corrections to scaling in two dimensions, $L = 500032$ ($\diamond$), $L = 4000256$ ($+$). Because of finite-size effects, the distribution does not follow a straight line, but at a given size of clusters, there are more than expected. For smaller L, this happens at smaller sizes s.

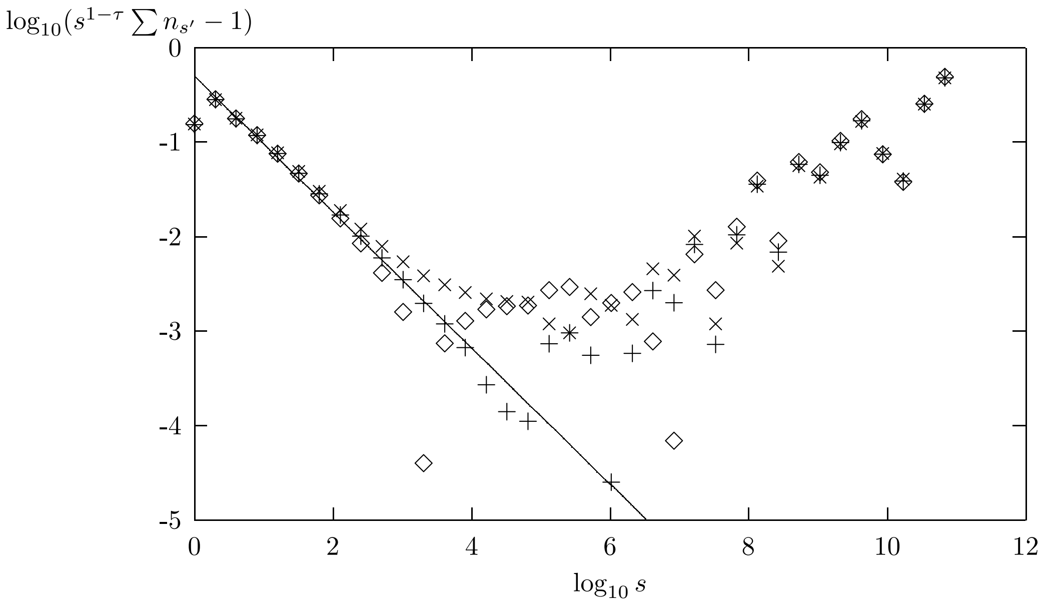


Figure 3.5: Corrections to scaling for $L = 4\text{M}$ in two dimensions. On the y-axis is plotted the binned data divided by the asymptotic behaviour $k_0 s^{-\tau}$. Three values were chosen for k_0: $5.21 \cdot 10^{11}$ ($\diamond$), $5.22 \cdot 10^{11}$ ($+$), and $5.23 \cdot 10^{11}$ ($\times$). The solid line represents the corrections to scaling power law $s^{0.72}$, which is a good approximation for the correctly chosen $k_0 = 5.22 \cdot 10^{11}$.

Another Monte Carlo value from literature is $\Delta_1 = 0.65(5)$ (MacLeod and Jan, [35]).

In three dimensions, we find $\Delta_1 = 0.60(8)$, agreeing roughly with $\Delta_1 = 0.70(5)$ found by Jan and Stauffer [28]. Gimel et al. found $\Delta_1 = 0.65(2)$, but the error seems to be overly optimistic.

In four dimensions, we find $\Delta_1 = 0.5(1)$.

3.4 Influence of boundary conditions on finite-size effects

A free surface (either by open boundary or by busbar) leads to modification of the asymptotic power law $n_s \propto s^{-\tau}$. This can be understood in terms of renormalization by the introduction of a new length-scale, the linear size of the system. Clusters of that size "feel" this length.

For open boundaries, it is easy to imagine the effect of such a surface on clusters: When a large cluster is placed near the surface, a part of it is cut off. Although the cluster would extend beyond the surface, we stop the counting of sites and thus get a too small cluster. We would expect an increase in n_s above the power-law.

The effect should be stronger for larger than for small clusters: When we shift around a small cluster on the lattice, it feels the influence of the surface only when it is very near to the surface. In the interior of the lattice, it does not feel the surface at all. A larger cluster does feel the surface earlier, at greater distance from the surface; there are not so many locations in the "interior" of the lattice. So we expect that for small clusters our power-law should not be influenced by finite-size effects (but by corrections-to-scaling, as explained above). The finite-size effects should become stronger the larger the clusters get. This can be seen in the data (cf. fig. 3.2: the n_s go up for large s. The drop at the end of the plot is caused by statistical fluctuations).

For busbar, the situation is different. Busbar means that the place above the uppermost plane is completely occupied. We assign the label 1 to the cluster formed by this, but we do not count the sites in the zeroth plane. This is just a trick to determine easily connectivity between uppermost and lowermost plane: If there is a reference to label 1 in the lowermost plane, we have connectivity.

This trick with busbar imposes peculiar finite-size effects different from open boundaries: Busbar, too, cuts clusters, but such cut clusters at the top of the system are joined by the zeroth plane, so they disappear and form a single, very huge cluster. Because of this disappearance, we expect that there are not as many large clusters as expected by the power-law. This can be seen very clearly in four dimensions (cf. fig. 3.6).

This makes busbar data rather problematic for analysis.

Open boundaries and busbar implement free surfaces in the system. As clusters can be placed anywhere within the volume of the lattice, but feel finite-size effects only when they touch the surface, the effects should be proportional to surface divided by volume, or for a L^d system: proportional to $1/L$.

This makes high-dimensional systems especially suitable for studying finite-size effects: statistical fluctuations are proportional to overall system size L^d. In two dimensions, these fluctuations dominate for small L, whereas in four dimensions even systems with small L have many sites and thus small statistical fluctuations.

When we want to avoid the $1/L$-behaviour, we have to avoid free surfaces. This can be done by fully periodic and/or helical boundary conditions. Even in this case we have finite-size effects: In a system of size L^d there can be no cluster larger than L^d. But these effects are dominated by the volume of the lattice and thus should

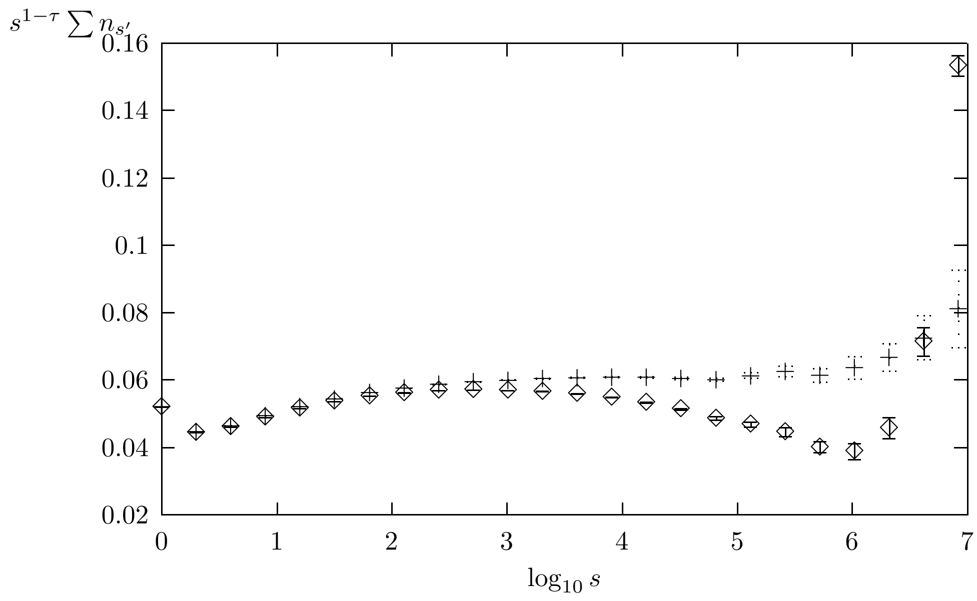


Figure 3.6: Cluster size distribution in four dimensions, using Ziff's four-tap PRNG, $L = 301$. Open boundaries (+) and busbar ($\diamond$). Error bars represent statistical error.

be proportional to $1/L^d$.

But fully periodic boundary conditions are expensive for the Hoshen-Kopelman algorithm, as we have to remember the first plane after the whole simulation and have to connect it to the last plane; even more, we must not discard labels associated with the first plane during recycling.

Obviously, data for finite systems are easier to analyse for fully periodic boudnary conditions, as the disturbing finite-size effects are less strong (but still present for small L). But even with the lower-quality data of open boundaries, we can find good estimates for the infinite system by plotting values for systems of different sizes against $1/L$. The intersect is the value for the infinite system.

Not only the boundary conditions, but also things like the aspect ratio of the investigated system (i. e. width divided by height in a $2d$ system) do have an impact on finite-size scaling. Several publications did investigate this in detail, cf. [1, 4, 12, 33, 34, 49, 56, 57, 59].

3.5 Number density

The total number of clusters divided by the number of sites in the system is the so called *number density* $n(p)$. It is independent from the system size, but as it depends on the microscopic details of the lattice, it is non-universal. Some number densities are known exactly for bond percolation (cf. [6, 53]). Unfortunately, for site percolation on square, cubic, and hypercubic lattices, such values have to be found numerically.

The total number density is a sum of the densities for all cluster sizes. This can be understood by looking at lattice animals in two dimensions (cf. [47, section 2.3]): To form a 1-cluster, we need one occupied site (probability p) surrounded by four free sites (probability $1 - p$), so the probability or density of 1-clusters is $p(1 - p)^4$. For a 2-cluster we need two occupied sites and six surrounding free sites;

but there are two different orientations for such a 2-cluster, so the corresponding density is $2p(1-p)^6$. These densities multiplied by L^d are the cluster numbers from above. But as the number of lattice animals grows exponentially with s, we can never calculate the exact values for large bins; and of course we cannot calculate the infinite sum to get the total number density.

Monte Carlo studies are a useful tool to get high precision estimates for the number density. We denote the density at the critical point by $n_c = n(p_c)$.

In two dimensions, our values for n_c are dominated by statistical fluctuations. This can be seen in fig. 3.7; for $L = 1\text{M}$, seven independent runs for each PRNG were done (in order to study the effects of random numbers; see below). The scattering of the points at $L = 1\text{M}$ is stronger than the variation of points for different L. From this, we can calculate a number density in two dimensions of $n_c = 0.02759791(5)$, agreeing well with the values for $L = 3.5\text{M}$ and $L = 4\text{M}$ (for larger lattices, fluctuations are significantly smaller). Ziff et al. have found a value of $n_c = 0.0275981(3)$ in [57], within errors in good agreement with the value found here.

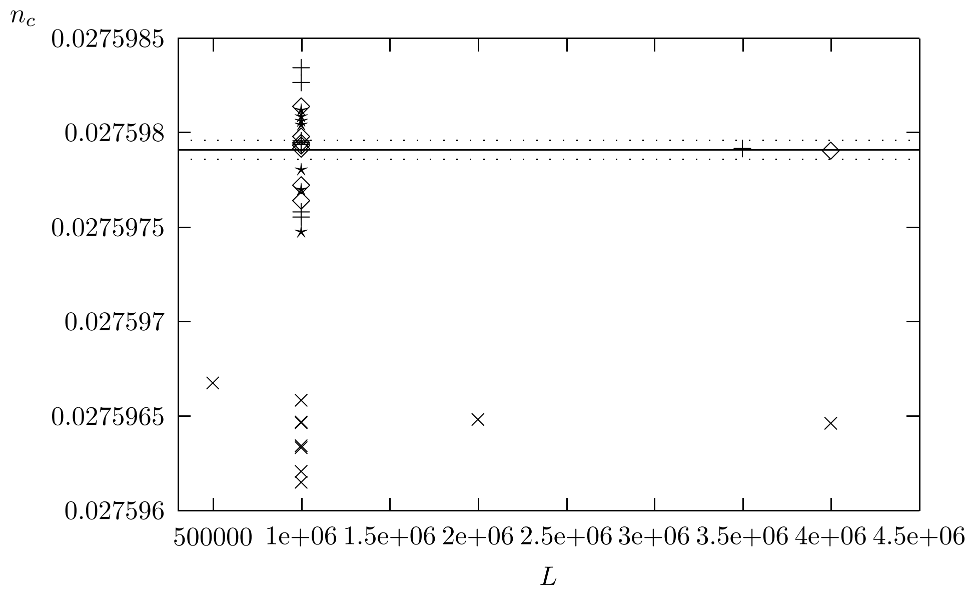


Figure 3.7: Number densities for various system sizes L, various PRNGs, and various runs with different random seeds. Used PRNGs: Kirkpatrick-Stoll ($\diamond$), Ziff's four-tap ($+$), ibm*16807 ($\times$), Ziff's six-tap ($\star$). The solid line corresponds to the average of the $L = 1\text{M}$ runs except the ibm*16807 ones, the dotted lines to the statistical error margins.

In four dimensions, the situation is different: Finite-size effects should go with $1/L$, too, but as the statistical fluctuations are propotional to the number of sites $1/L^d$, finite-size effects should dominate. This is indeed the case, as can be seen in fig. 3.8: The data points only slightly scatter around the regression line $\propto 1/L$. From this plot, we can extrapolate $n_c = 0.0519980(2)$.

3.6 The problem with not-so-random numbers

As can be seen in fig. 3.7, there is a problem with one of the four utilized random number generators, the linear congruential generator `ibm=ibm*16807`. While the

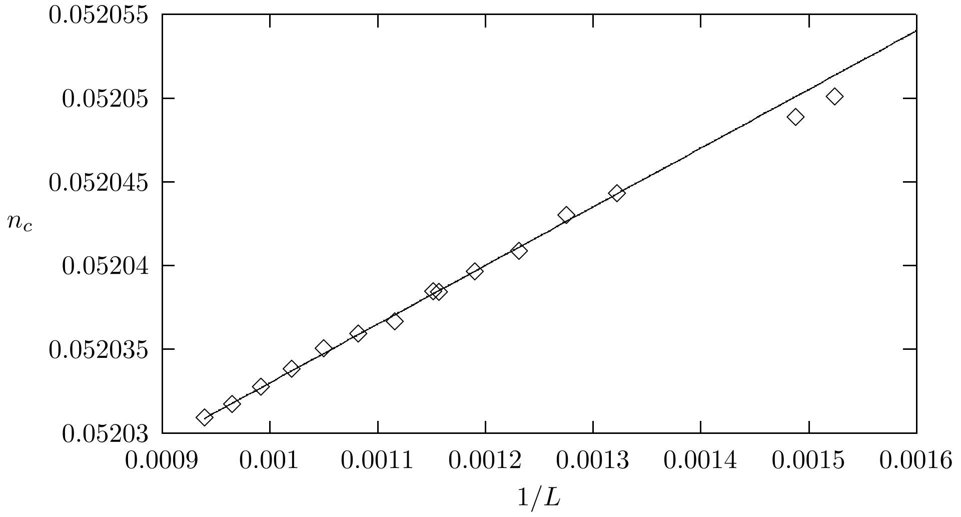


Figure 3.8: Number densities for various system sizes L. The solid lines corresponds to the extrapolation to infinity. From varying the line, we can estimate an error for n_c at ∞.

resulting number densities from the other generators agree well within statistical fluctuations, those from the `ibm*16807` deviate visibly. This is due to correlations in the produced random numbers. The `ibm*16807` is known to be problematic in literature (cf. [48, part II, chapter 1]), and here once again this is shown clearly. The other generators (which are all lagged fibonacci generators) seem to be compatible with each other, so it is wise to choose the fastest one (cf. section 4.3).

3.7 Summary of obtained results

The results below are for the world record simulations.

d	L	τ	Δ_1	n_c
2	4000256	187/91	0.70(2)	0.02759791(5)
3	20224	2.190(2)	0.60(8)	0.052442(2)
4	1036	2.313(2)	0.5(1)	0.0519980(2)

Chapter 4

Runtime and efficiency

When examining the speed of our program, we are interested in not only the absolute speed for a given set of parameters, but we also want to know how the speed scales with varying size L or varying number of processors N. And of course we expect a dependence on the parameters like the dimensionality d, the probility p, or the size of various data structures in memory.

A thorough study of the scaling of speed depending on all these parameters in all combinations would require hundreds of large runs; this would eat up our precious computing time budget and would give us no new insight into physics. From a physicist's point of view, it would be wasted effort.

Thus we are only interested in some rough estimates about how the program will perform. Especially we want to know, if it "performs well", if it does not waste too much computing time in communication.

As the parallel program was used only on the Cray T3E of the Research Center Jülich, all times are for that machine.

4.1 Speed per processor for the whole simulation

We are especially interested in the overall runtime for the whole simulation; especially when doing world record simulations, we must not occupy too much time on the computer, otherwise our job could be terminated before it has finished.

A nice and useful measure for speed is the number of sites that one processor can examine in one second. Of course, this will differ for varying lattice sizes and varying number of processors, but we can at least roughly extrapolate.

Dim.	L	N	Speed [MSite/s]
2	4000256	256	5.74423
2	2000064	64	6.32598
2	1000064	64	5.27195
3	20224	256	3.55194
3	12096	64	4.25586
4	1036	37	3.95107
4	924	33	4.16548
4	756	27	4.24066

Table 4.1: Overall speed per processor of the whole simulation. The PRNG used in these simulations is Kirkpatrick-Stoll

From the table, it becomes clear that the simulation of a larger lattice means less speed (this can be explained by the fact that working on larger data structures is slower due to machine architecture), and that for a given lattice size the simulation on less processors means higher speed (because there is less communication neccessary). At least, the speeds do not differ drastically, so the parallelization seems to be efficient (cf. in two dimensions using the `ibm*16807` generator, $L = 2000128$ on 128 processors with 5.99821 MSite/s and $L = 2000064$ on 64 processors with 6.49730 MSite/s, a difference of only eight percent).

It might be surprising that for higher dimensions the speed is comparable (albeit slightly lower) to two dimensions, because in higher dimensions we have to check more neighbours when investigating an occupied site. But as we do our simulations at the critical threshold p_c, and this is lower in higher dimensions, we have to investigate less occupied sites, which accounts for higher speed.

4.2 Runtime of different parts of the simulation

We are especially interested to know, if there is too much time wasted with communication, or if our program is parallelized well. Overall time for the local part should be proportional to L^d, whereas the communication part should require time proportional to the interface between strips, NL^{d-1}.

In two dimensions, using the `ibm*16807` generator, we can compare two runs with $L = 2000128$, $N = 128$, and $L = 2000064$, $N = 64$. When using twice as much processors, the local part was about one percent faster (which can be due to statistics), whereas the time for communication rose by 84 percent. A more precise study of these effects would require a lot more runs with differing L and N, of course.

In four dimensions, several runs were done with different L and N, but constant $L/N = 28$. These runs were done to study finite-size effects, but can also be used to investigate performance, cf. fig. 4.1.

The run times for local and communication parts scale as expected. But what is even more important: although for higher dimensions more communication is neccessary, even in four dimensions the local part clearly dominates the run time, although included in the time difference are parts like recycling and counting, which are also needed for sequential simulations (but are complicated by the parallel structure and also require some communication). It is thus reasonable to call the method of domain decomposition chosen here *parallel efficient.* Not too much time is wasted with communication.

4.3 Impact on speed of different PRNGs

The pseudo-random number generator used in the simulation can have an impact on runtime, too. A complex PRNG that does lengthy calculatons in order to generate a random number slows down the simulation. In higher dimensions, the impact is stronger; as we are mainly interested in investigating systems at the critical point p_c, and p_c is lower in higher dimensions, we would produce a higher fraction of random numbers just to find out that the site in investigation is not occupied and no work has to be done. Thus a slow PRNG means a higher penalty for smaller p.

As described in section 3.5, some PRNGs are not suitable for simulating huge lattices. The Kirkpatrick-Stoll or two-tap generator produces reasonable results and is rather fast. Other lagged Fibonacci generators, as Ziff's four-tap and six-tap, do not produce significantly different results, but are significantly slower, especially the four-tap one. It utilizes large taps and thus needs a large data structure to store its

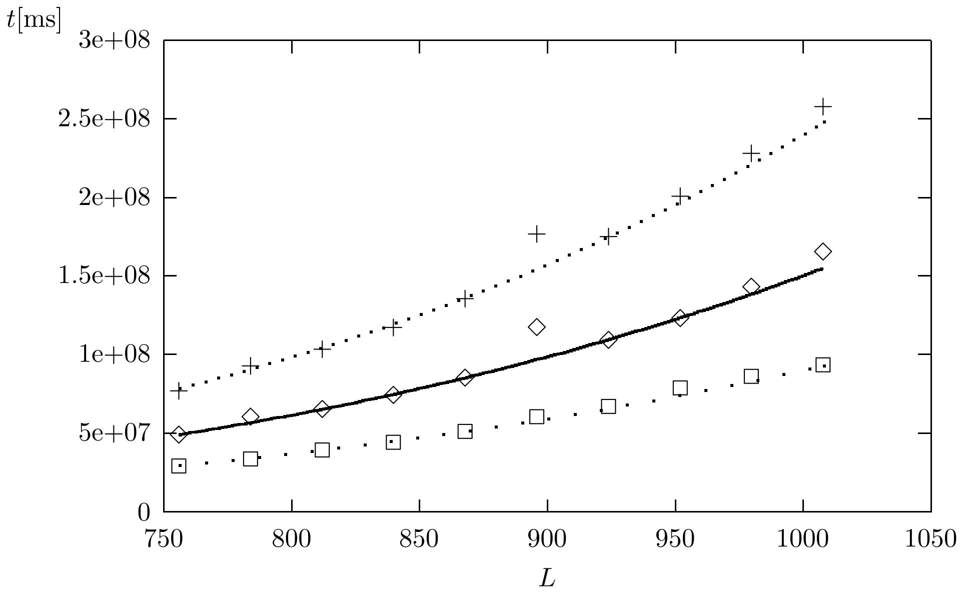


Figure 4.1: Total time (+), local time (◇), and total minus local time (□). The lines are regression curves proportional to L^4. The local time for $L = 896$, corresponding to $N = 32$, lies above the expected value due to effects of the machine architecture (thrashing of direct-mapped cache).

random numbers. On the `21164` microprocessor of the Cray T3E, this data cannot be kept in cache and has to be reloaded from memory frequently, which slows down computation. The six-tap generator uses a data structure that is even smaller than that of Kirkpatrick-Stoll, but it uses a more complex operation and is thus slower, too (but significantly faster than four-tap).

At the moment, there is no good reason to use the more complex and slower PRNGs, as Kirkpatrick-Stoll shows no visible systematic errors. This can change, of course, when we can simulate even larger systems (with even more powerful computers). The four-tap and six-tap promise us, due to their more complex nature, to be less affected by correlations in the random numbers. But this has to be checked in simulations.

Chapter 5

Summary and outlook

5.1 Summary

Within this diploma thesis, a novel approach to parallelizing the well-known Hoshen-Kopelman algorithm has been chosen, suitable for simulating huge lattices in high dimensions on massively-parallel computers with distributed memory and message passing. This method consisted of domain decomposition of the simulated lattice into strips perpendicular to the hyperplane of investigation that is used in the Hoshen-Kopelman algorithm. This approach is more complicated than others, but it allows for simulating huge lattices, even in dimensions above two.

Using the parallelized algorithm, it was possible to simulate random site percolation on the square (resp. cubic and hypercubic) lattice in two, three, and four dimensions, with maximum lattice sizes of $L = 4000256$ ($2d$), $L = 20224$ ($3d$), and $L = 1036$ ($4d$). These are the largest systems percolation was ever simulated on, thus yielding three world records. All simulations were done on the very fast Cray T3E at the Research Center Jülich (in the Top 500 list of the world's most powerful supercomputers from November 2000, it ranked 40th).

Using the data generated with the world record simulations, it was possible to investigate some properties of percolation with high precision, i. e. critical exponents like the Fisher exponent τ or the corrections to scaling exponent Δ_1, and the number density at the critical point n_c. Comparison with values obtained by other groups with other methods were in reasonable agreement.

During the simulation of very large lattices it became clear that some pseudo-random number generators produced wrong results due to their correlations. When going to even larger lattices, it could happen that even more generators show as unsuitable.

A limiting influence for the precision of estimating some critical exponents was found in the boundary conditions. By investigating larger and larger systems, this influence can be weakened. Another possibility could be implementing fully periodic boundaries.

When parallelizing algorithms that were introduced for sequential computers, it is important to try to implement them with high parallel efficiency, in other words: do not waste too much time with communication in comparison to the sequential implementation. One of the major results of this thesis was to prove that even for dimensions up to four most time was spent in local calculations, which means a high parallel efficiency. Together with the possibility of using many processors in parallel, this yielded a situation where it was possible to quickly simulate a large system with a given set of parameters and to receive results within hours, while the same simulation on a sequential computer could have taken a week to finish.

The capability of simulating very large lattices allows us to find some very precise results for critical properties of percolation, additionally to those investigated here, just by investing more computing time (and some effort on data analysis).

5.2 Outlook

There remain several things that could and should be done, but cannot be covered within this thesis due to limited time.

As the program was implemented for two, three, and four dimensions, it would be rather straightforward to go to even higher dimensions. But then the overhead of communication versus the local calculations would grow, so this cannot be stretched too far. It would be a good idea for higher dimensions to implement some memory-saving tricks (like 32 bit compound labels instead of 64 bit plain labels, a unification of the local and global label storages, and some more).

A significant improvement in quality of the obtained results should be possible by the implementation of fully periodic boundary conditions, instead of leaving the top and bottom plane open, as this should reduce the finite-size effects drastically. Especially better estimates for the corrections to scaling should be possible. On the other hand, such fully periodic boundaries consume more memory and time, but it should be worth the expense. In general, the investigation of things like boundary conditions, aspect ratio of finite systems, and other finite-size influences, could yield new insight even into the characteristics of infinite systems.

All simulations carried out for this thesis were done right at the critical threshold p_c. Of course, it is possible to move away from p_c, and thus examine additional critical exponents, and even the scaling function $f(z)$. This will require a lot of computing time, but no new code.

Research in percolation theory is far from the end, and using Monte Carlo techniques with new algorithms on parallel computers can help to clarify many still open questions.

Appendices

Appendix A

Acknowledgements

First and foremost I would like to thank Prof. Stauffer for suggesting the topic of my diploma thesis and for his care, which consisted of a combination of laissez-faire and impetus. Next I would like to thank Profs. Jan and Ziff for invaluable ideas and sportsmanlike competition. I also appreciated the Wednesday seminars with their fruitful discussions on lots of different topics, and would like to thank all participants. And last but not least, I would like to thank my parents for their love and support.

Appendix B

Bibliography

[1] M. Acharyya, D. Stauffer, *Effects of boundary conditions on the critical spanning probability*, Int. J. Mod. Phys. C **9**, 643 (1998).

[2] J. Adler, M. Moshe, V. Privman, *Corrections to scaling for percolation*, in: [13], pp. 397–423.

[3] A. Aharony, M. E. Fisher, *Nonlinear scaling fields and corrections to scaling near criticality*, Phys. Rev. B **27**, 4394 (1983).

[4] A. Aharony, D. Stauffer, *Test of universal finite-size scaling in two-dimensional site percolation*, J. Phys. A **30**, L301 (1997).

[5] H. G. Ballesteros, L. A. Fernández, V. Martín-Mayor, A. Muñoz Sudupe, G. Parisi, J. J. Ruis-Lorenzo, *Measures of critical exponents in the four-dimensional site percolation*, Phys. Lett. B **400**, 346 (1997).

[6] R. J. Baxter, H. N. V. Temperley, S. E. Ashley, *Triangular Potts model at its transition temperature, and related models*, Proc. R. Soc. London A **358**, 535 (1978).

[7] K. Binder, D. Stauffer, *Monte Carlo Studies of "Random" Systems*, in: K. Binder (Ed.), *Applications of the Monte Carlo Method*, 2nd ed. (Springer, Heidelberg, 1987), pp. 241–275.

[8] S. R. Broadbent, *Discussion on symposium on Monte Carlo methods*, J. Roy. Statist. Soc. B **16**, 68 (1954).

[9] S. R. Broadbent, J. M. Hammersley, *Percolation Processes. Crystals and mazes*, Proc. Cambridge Philos. Soc. **53**, 629 (1957).

[10] A. Bunde, S. Havlin (Eds.), *Proceedings of the International Conference on Percolation and Disordered Systems: Theory and Applications*, Physica A **266** (1999).

[11] A. Bunde, S. Havlin (Eds.), *Fractals and Disordered Systems*, (Springer, Heidelberg, 1991).

[12] J. L. Cardy, *Critical percolation in finite geometries*, J. Phys. A **25**, L201 (1992).

[13] G. Deutscher, R. Zallen, J. Adler, *Percolation structure and processes*, Annals of the Israel Physical Society **5**, (Adam Hilger, Bristol, 1983).

[14] M. E. Fisher, *The theory of condensation and the critical point*, Physics **3**, 255 (1967).

[15] M. Flanigan, P. Tamayo, *A Parallel Cluster Labeling Method for Monte Carlo Dynamics*, Int. J. Mod. Phys. C **3**, 1235 (1992).

[16] M. Flanigan, P. Tamayo, *Parallel cluster labeling for large-scale Monte Carlo simulations*, Physica A **215**, 461 (1995).

[17] P. J. Flory, *Molecular Size Distribution in Three Dimensional Polymers. I. Gelation*, pp. 3083–3090, *II. Trifunctional Branching Units*, pp. 3091–3096, *III. Tetrafunctional Branching Units*, pp. 3096–3100, J. Am. Chem. Soc. **63** (1941).

[18] J.-C. Gimel, T. Nicolai, D. Durand, *Size distribution of percolating clusters on cubic lattices*, J. Phys. A **33**, 7687 (2000).

[19] G. R. Grimmett, *Percolation*, in: J.-P. Pier (ed.), *Development of Mathematics 1950–2000*, (Birkhäuser, Basel, 2000), pp. 547–575.

[20] U. Gropengiesser, *Numerical Methods for the Determination of the Properties of Phase Transitions and Ground States of Ising and Ising Spin Glass Systems*, Inaugural-Dissertation, Universität zu Köln, 1995.

[21] F. Gutbrod, *New trends in pseudo-random number generation*, in: D. Stauffer (Ed.), Annual Reviews of Computational Physics **VI** (World Scientific, Singapore, 1999), pp. 203–257.

[22] R. Hackl, H.-G. Matuttis, J. M. Singer, T. Husslein, I. Morgenstern, *Parallelization of the 2D Swendsen-Wang Algorithm*, Int. J. Mod. Phys. C **4**, 1117 (1993).

[23] J. M. Hammersley, *Percolation Processes. The connectivity constant*, Proc. Cambridge Philos. Soc. **53**, 642 (1957).

[24] J. M. Hammersley, *Percolation Processes. Lower bounds for the critical probability*, Ann. Math. Statist. **28**, 790 (1957).

[25] J. M. Hammersley, D. C. Handscomb, *Monte Carlo Methods*, (Methuen, London, 1964).

[26] J. M. Hammersley, *Origins of percolation theory*, in: [13], pp. 47–57.

[27] J. Hoshen, R. Kopelman, *Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm*, Phys. Rev. B **14**, 3438 (1976).

[28] N. Jan, D. Stauffer, *Random Site Percolation in Three Dimensions*, Int. J. Mod. Phys. C **9**, 341 (1998).

[29] J. Kertész, D. Stauffer, *Swendsen-Wang Dynamics of Large 2D Critical Ising Models*, Int. J. Mod. Phys. C **3**, 1275 (1992).

[30] S. Kirkpatrick, E. P. Stoll, J. Comput. Phys. **40**, 517 (1981).

[31] P. L. Leath, *Cluster size and boundary distribution near percolation threshold*, Phys. Rev. B **14**, 5046 (1976).

[32] P. L. Leath, *Cluster shape and critical exponents near Percolation Threshold*, Phys. Rev. Lett. **36**, 921 (1976).

[33] C. D. Lorenz, R. M. Ziff, *Universality of the excess number of clusters and the crossing probability function in three-dimensional percolation*, J. Phys. A **31**, 8147 (1998).

[34] C. D. Lorenz, R. M. Ziff, *Precise determination of the bond percolation thresholds and finite-size scaling corrections for the sc, fcc, and bcc lattices*, Phys. Rev. E **57**, 230 (1998).

[35] S. MacLeod, N. Jan, *Large Lattice Simulation of Random Site Percolation*, Int. J. Mod. Phys. C **9**, 289 (1998).

[36] A. Margolina, Z. Djordjevic, H. E. Stanley, D. Stauffer, *Corrections to scaling for branched polymers and gels*, Phys. Rev. B **28**, 1652 (1983).

[37] H. Nakanishi, H. E. Stanley, *Scaling studies of percolation phenomena in systems of dimensionality two to seven: Cluster numbers*, Phys. Rev. B **22**, 2466 (1980).

[38] B. Nienhuis, E. K. Riedel, M. Schick, *Magnetic exponents of the two-dimensional q-state Potts model*, J. Phys. A **13**, L189 (1980).

[39] B. Nienhuis, *Analytical solution of the two leading exponents of the dilute Potts model*, J. Phys. A **15**, 199 (1982).

[40] M. P. M. den Nijs, *A relation between the temperature exponents of the eight-vertex and q-state Potts model*, J. Phys. A **12**, 1857 (1979).

[41] G. Paul, R. M. Ziff, H. E. Stanley, *Precolation threshold, Fisher exponent, and shortest path exponent for 4 and 5 dimensions*, `arXiv: cond-mat/0101136` (2001).

[42] R. P. Pearson, *Conjecture for the extended Potts model magnetic eigenvalue*, Phys. Rev. B **22**, 2579 (1980).

[43] M. Sahimi, *Applications of Percolation Theory*, (Taylor & Francis, London, 1994).

[44] H. E. Stanley, J. S. Andrade Jr., S. Havlin, H. A. Makse, B. Suki, *Percolation phenomena: a broad-brush introduction with some recent applications to porous media, liquid water, and city growth*, in: [10], pp. 5–16.

[45] D. Stauffer, *Violation of dynamical scaling for randomly dilute Ising ferromagnets near percolation threshold*, Phys. Rev. Lett. **35**, 394 (1975).

[46] D. Stauffer, *Scaling theory of percolation clusters*, Phys. Reports **54**, 1 (1979).

[47] D. Stauffer, A. Aharony, *Introduction to Percolation Theory*, 2nd ed., (Taylor & Francis, London, 1992).

[48] D. Stauffer, F. W. Hehl, N. Ito, V. Winkelmann, J. G. Zabolitzky, *Computer Simulation and Computer Algebra*, 3rd ed., (Springer, Heidelberg, 1993).

[49] D. Stauffer, *Finite-size effect in seven-dimensional percolation*, Physica A **210**, 317 (1994).

[50] D. Stauffer, *World records in the size of simulated Ising models*, Braz. J. Phys. **30**, 787 (2000).

[51] P. Tamayo, *Magnetization relaxation to equilibrium on large 2D Swendsen-Wang Ising models*, Physica A **201**, 543 (1993).

[52] R. C. Tausworthe, Math. Comput. **19**, 201 (1965).

[53] H. N. V. Temperley, E. H. Lieb, *Relations between the 'percolation' and 'colouring' problem and other graph-theoretical problems associated with regular planar lattices: some exact results for the 'percolation' problem*, Proc. R. Soc. London A **322**, 251 (1971).

[54] J. M. Teuler, J.-C. Gimel, *A direct parallel implementation of the Hoshen-Kopelman algorithm for distributed memory architectures*, Comp. Phys. Comm. **130**, 118 (2000).

[55] A. Ticona, D. Stauffer, *Percolation cluster numbers in seven dimensions*, Physica A **290**, 1 (2001).

[56] R. M. Ziff, *Effective boundary extrapolation length to account for finite-size effects in the percolation crossing function*, Phys. Rev. E **54**, 2547 (1996).

[57] R. M. Ziff, S. R. Finch, V. S. Adamchik, *Universality of finite-size corrections to the number of critical percolation clusters*, Phys. Rev. Lett. **79**, 3447 (1997).

[58] R. M. Ziff, *Four-tap shift-register-sequence random-number generators*, Comput. in Phys. **12**, 385 (1998).

[59] R. M. Ziff, C. D. Lorenz, P. Kleban, *Shape-dependent universality in percolation*, in: [10], pp. 17–26.

Appendix C

Code of programs

C.1 Parallel program

Implementing the Hoshen-Kopelman algorithm on massively-parallel computers is a difficult thing, so it is no surprise that the resulting source-code is long, complex, and hard to understand. Before reading this program, it is important to understand how the domain decomposition works. This is covered in detail in chapter 2.

The code is written in standard Fortran 90, with some Cray-specific system calls for communication: `shmem_put()` delivers data from one processor to another, `shmem_get()` fetches data; `barrier()` and `shmem_barrier_all()` are used to synchronize all processors, with `shmem_barrier_all()` also waiting for completion of all remote write operations (i. e. `shmem_put()`); `shmem_n_pes()` gives the number of processors that the job is running on, `shmem_my_pe()` gives the identity of the processor the local thread is running on; the return value of `shmem_my_pe()` is the only way for the program to distinguish the different processors, as the code and all initial data is the same for all threads in the job.

The source-code is in fixed format (the coloumns have special meaning). The small slanted numbers on the left side of the listing are simply a guide to the eye and are not part of the program; they must not be mistaken with labels, which are part of the program and are typeset in the same font as the rest of the code.

The code presented here is the implementation for four dimensions, a more general case than three or two dimensions. For this reason, the code for two and three dimensions can be derived easily by just commenting out that portions of the code that deal with the fourth (or third) dimension. But as no one would like to type in about 1600 lines, it would be easier to obtain the code from the author, along with ready-to-use two- and three-dimensional versions.

One last note: There seems to be a bug in the code that only strikes when the size of a strip (L/N) becomes too small, smaller than ca. 30 sites; in such a case, an infinite loop can happen during the simulation. But for larger strip sizes, this never happens; two and three dimensions were never affected, due to the large ratio L/N.

```
c $Id: pperc4d.f,v 1.1 2001/02/25 13:22:47 dt Exp $
c Parallel implementation of:
c Random site percolation in up to seven dimensions. Counts clusters and
c determines connectivity. Uses recycling of labels.
c This implementation is valid only for FOUR DIMENSIONS.

c Uses domain decomposition into vertical strips, with one processor per
c strip. We are looking for a cluster percolating from left to right.
```

```
c local(i) negative
c          and even: local rootlabel with -local(i)/2 sites in the cluster.
c local(i) negative
c           and odd: pointer to global rootlabel -local(i)/2
c local(i) positive: points to other label
c local(i) zero:     free label, can be reused

      parameter (ISEED=1, LSTRIP=14, NSTRIP=64, L=LSTRIP*NSTRIP,
     *           IDIM=4, NSYS=L**IDIM, LLINE=L**(IDIM-2),
     *           LPLANE=LLINE*LSTRIP,
     *           MAXLOC=38e6, MAXGLO=2e6, MAXPR=1e5,
     *           P=0.196889, MAXBIN=45, MBP1=MAXBIN+1,
     *           DIVLIMMAX=0.70, LIMIT=DIVLIMMAX*MAXLOC,
     *           LIMITGLO=DIVLIMMAX*MAXGLO, NRECFREQ=1)
      logical ALLOW_GLOBAL_RECYC
      parameter (ALLOW_GLOBAL_RECYC=.true.)
      dimension plane(1-LLINE:LPLANE), local(MAXLOC), ns(0:MAXBIN)
      dimension global(MAXGLO), globle(MAXGLO), globri(MAXGLO)
      dimension prtlda(MAXPR), prtrda(MAXPR), prrcda(MAXPR)
      dimension ks_rnd(0:255)
      dimension border_send(1:LLINE), border_recv(1:LLINE)
      data ns/MBP1*0/

      common /t3e/ prtlda, prtrda, prrcda, border_send, border_recv,
     *             ns,    prtlpt, prtrpt, prrcpt,
     *             nonloc, relax, conn, chi,
     *             comm_logvar, comm_intvar, comm_realvar

      integer plane, local, global, globle, globri, glolab
      integer locmin, locmax, glomin, glomax
      integer prtlda, prtrda, prtlpt, prtrpt, prrcda, prrcpt
      integer gleft, gright, gone, gtwo
      integer pe_le, pe_ri, pe_this
      integer shmem_n_pes, shmem_my_pe
      integer comm_intvar
      logical conn, left, top, back, nonloc, relax, comm_logvar
      logical fourth, accum_log
      real*8 fli, chi, chisum, comm_realvar
      integer ks_idx
      integer time, tim_loc, tim_prex, tim_nbex, tim_perc,
     *        tim_glob, tim_fullrelax, tim_detconn, tim_loccount,
     *        tim_concen, tim_globcount, tim_label1, tim_recyc
      integer nrecyc, rec_countdown
      logical want_recycling, once_more
      integer ks_rnd
      integer IP
      integer border_send, border_recv

      if(shmem_my_pe() .eq. 0) then
        print *, '# Using Kirkpatrick-Stoll PRNG'
        if(ALLOW_GLOBAL_RECYC) then
          print *, '# Global recycling on'
        end if
        print *, '# Size of system: ', L, ' ** ', IDIM
        print *, '# Number of strips: ', NSTRIP
        print *, '# Initial random seed: ', ISEED
        print *, '# Probability: ', P
      end if
      tim_glob=irtc()
      IP=2147483648.0d0*(4.0d0*P-2.0d0)*2147483648.0d0
      rcplog=1.0d0/dlog(2.0d0)
      conn=.true.
      maxclu=0
      ninfclu=0
      nocc = 0
      nrecyc = 0
      rec_countdown = NRECFREQ

      if(shmem_n_pes() .ne. NSTRIP) stop 3
      pe_this = shmem_my_pe()
      pe_le = pe_this - 1
      pe_ri = pe_this + 1
      if(pe_this .eq. 0)          pe_le = NSTRIP - 1
      if(pe_this .eq. NSTRIP - 1) pe_ri = 0

      call ks_warmup()

c     In the beginning, all labels are reusable
      do 10 i = 1, MAXLOC
        local(i) = 0
   10   continue
      locmin = 2
      do 11 i = 1, MAXGLO
        global(i) = 0
        globle(i) = 0
        globri(i) = 0
   11   continue
      glomin = 1

c     In the beginning, there is no pairing information
      prtlpt = 0
      prtrpt = 0

c     To determine connectivity, we use the easiest method: The zeroth
c     plane is set to be completely the cluster with number 1, which
c     means that if in the end there appears a cluster 1 in the last plane,
c     we have connectivity and cluster 1 is the infinite cluster.
c     By this, cluster 1 percolates through the whole system horizontically
c     in the beginning. We must thus look for vertcial connectivity only,
c     because otherwise we would get wrong results. During the investigation,
c     cluster 1 may cause trouble just because of this fact. Thus we should
c     think about handling cluster 1 in a special way during relaxation.
c     And of course we must never re-use label 1 after recycling. We achieve
c     this by keeping locmin always larger than 1.

c     Instead, we now look for connectivity in horizontically percolating
c     clusters and thus leave the zeroth plane free.
      do 15 i = 1, LPLANE
        plane(i) = MAXLOC    ! for busbar, set to 1
   15   continue
      glomin = 1
!     For busbar, set plane() to 1 and:
!     local(1) = -3
!     global(1) = -1
!     globle(1) = 1
!     globri(1) = 1
!     glomin = 2

      nonroo = 0     ! non-root-labels connected to global clusters
      numroo = 0     ! local root-labels
      numglo = 0     ! global root-labels in one strip
      numgloclu = 0  ! number of global clusters
      numsit = 0     ! number of occupied sites
      chi = 0.0d0

      tim_loc  = 0
      tim_nbex = 0
      tim_prex = 0
      tim_recyc = 0
      tim_perc = irtc()
      do 20 ivert = 1, L
        i = 1
c       Clear the borders, because we do not know yet, if there are
c       neighbouring clusters in the neighbouring strips.
        do j = 1-LLINE, 0
          plane(j) = MAXLOC
        end do
c       Do the local step
        time = irtc()
        do 30 ihoriz = 1, LPLANE
          ks_idx = iand(ks_idx + 1, 255)
          ks_rnd(ks_idx) = ieor(ks_rnd(iand(ks_idx-103, 255)),
     *                          ks_rnd(iand(ks_idx-250, 255)) )
          if(ks_rnd(ks_idx) .lt. IP) then
c           The site in investigation is occupied.
            nocc = nocc + 1
            top  = (plane(i  ) .lt. MAXLOC)
            back = (plane(i-1) .lt. MAXLOC)
            fourth=(plane(i-L) .lt. MAXLOC)
            left = (plane(i-LLINE) .lt. MAXLOC)
            accum_log = (left.or.top.or.back.or.fourth)
            if(.not. accum_log) then
c             All neighbours are free, so a new cluster starts.
c             First we have to find a free label.
   31         nrlab = locmin
              locmin = locmin + 1
              if(locmin .eq. MAXLOC) stop 1
              if(local(nrlab) .ne. 0) goto 31
              plane(i) = nrlab
              local(nrlab) = -2
            else
c             At least one of the neighbours are occupied, which means work.
c             First we determine the rootlabels for all neighbours.
              nleft = MAXLOC
              ntop  = MAXLOC
              nback = MAXLOC
              nfourth=MAXLOC
c             The label of the back neighbour (if occupied) is certainly a
c             rootlabel, either local or global.
              if(back) nback = plane(i-1)
c             With the other neighbours, that need not be the case.
              if(top) then
                ntop = plane(i)
                if(local(ntop) .gt. 0) then
```

```
 32        ntop = local(ntop)
           if(local(ntop) .gt. 0) goto 32
           local(plane(i)) = ntop
         endif
       endif
       if(fourth) then
         nfourth = plane(i-L)
         if(local(nfourth).gt.0) then
 34        nfourth = local(nfourth)
           if(local(nfourth) .gt. 0) goto 34
           local(plane(i-L)) = nfourth
         endif
       endif
       if(left) then
         nleft = plane(i-LLINE)
         if(local(nleft) .gt. 0) then
 33        nleft = local(nleft)
           if(local(nleft) .gt. 0) goto 33
           local(plane(i-LLINE)) = nleft
         end if
       end if


c      Now that we have the rootlabels for the neighbouring
c      sites, we have to do some examination. If both labels are
c      local root-labels, life is easy. If only one of the labels
c      is a global label, life becomes difficult. If both labels
c      are global, we have a lot of work to do.

c      First find the smallest rootlabel.
       new = min0(nleft, ntop, nback, nfourth)

c      We count the sites within the cluster negative in ici.
c      Do not forget to divide site numbers by two.
       ici = -1
       if(left) then
         nonloc=.false.
         nrlab = local(nleft)
         if(iand(nrlab, 1) .eq. 1) then
           nonloc = .true.
           nrlab = global(-nrlab/2)
         endif
         ici = ici + nrlab/2
         if(new .ne .nleft) then
c          The cluster coming from left does not have the smallest
c          label. Thus it will be transferred to a non-root-label.
c          But we have to differentiate between two cases: if it
c          is a local rootlabel, we just redirect the label.
c          If it is a global rootlabel, we have to look if the smallest
c          label is a global rootlabel. If yes, we have to add pairing
c          information. If no, we have to tranform the smallest label
c          to a global rootlabel.
           if(nonloc) then
             if(iand(local(new), 1) .eq. 1) then
c              Rootlabel "new" is global. Add pairing information.
c              Redirect "nleft" to "new".
c              Pairing info for left neighbour:
               gone  = globle(-local(new  )/2)
```

```
                      gtwo  = globle(-local(nleft)/2)
                      if(gone .ne. gtwo) then
                        gleft = min0(gone, gtwo)
                        gone  = max0(gone, gtwo)
                        globle(-local(new)/2) = gleft
                        if(gone .ne. MAXLOC) then
                          prtlpt = prtlpt + 2
                          if(prtlpt .ge. MAXPR) stop 2
                          prtlda(prtlpt-1) = gleft
                          prtlda(prtlpt  ) = gone
                        endif
                      endif
c                     Pairing info for right neighbour:
                      gone  = globri(-local(new  )/2)
                      gtwo  = globri(-local(nleft)/2)
                      if(gone .ne. gtwo) then
                        gright= min0(gone, gtwo)
                        gone  = max0(gone, gtwo)
                        globri(-local(new)/2) = gright
                        if(gone .ne. MAXLOC) then
                          prtrpt = prtrpt + 2
                          if(prtrpt .ge. MAXPR) stop 2
                          prtrda(prtrpt-1) = gright
                          prtrda(prtrpt  ) = gone
                        endif
                      endif
c                     After this pairing, redirect "nleft" to "new"
                      local(nleft) = new
                    else
c                     Rootlabel "new" is local. Transform it to global.
c                     We do this by pointing label "new" to the global label
c                     of "nleft". "nleft" then becomes a pointer to "new".
c                     Of course, we have to count the sites in local cluster
c                     "new". And we have to take care that the sites in the
c                     global cluster are not counted again.
                      ici = ici + local(new)/2
                      local(new)   = local(nleft)
                      local(nleft) = new
c                     We enter a one to show that this label must not be
c                     recycled, but it contains no sites, as 1/2 = 0.
                      global(-local(new)/2) = -1
                    endif
                  else
                    local(nleft) = new
                  endif
                endif
              endif
              if(top .and. (nleft .ne. ntop)) then
c               If ntop  =  nleft, then we already have treated the cluster.
c               There is no need to investigate it once again.
c               If ntop =/= nleft, we have to investigate that cluster.
                nonloc = .false.
                nrlab = local(ntop)
                if(iand(nrlab, 1) .eq. 1) then
                  nonloc = .true.
                  nrlab = global(-nrlab/2)
                endif
                ici = ici + nrlab/2
                if(new .ne. ntop) then
                  if(nonloc) then
                    if(iand(local(new), 1) .eq. 1) then
                      gone = globle(-local(new )/2)
                      gtwo = globle(-local(ntop)/2)
                      if(gone .ne .gtwo) then
                        gleft = min0(gone, gtwo)
                        gone  = max0(gone, gtwo)
                        globle(-local(new)/2) = gleft
                        if(gone .ne. MAXLOC) then
                          prtlpt = prtlpt + 2
                          if(prtlpt .ge. MAXPR) stop 2
                          prtlda(prtlpt-1) = gleft
                          prtlda(prtlpt  ) = gone
                        endif
                      endif
                      gone = globri(-local(new )/2)
                      gtwo = globri(-local(ntop)/2)
                      if(gone .ne. gtwo) then
                        gright = min0(gone, gtwo)
                        gone   = max0(gone, gtwo)
                        globri(-local(new)/2) = gright
                        if(gone .ne. MAXLOC) then
                          prtrpt = prtrpt + 2
                          if(prtrpt .ge. MAXPR) stop 2
                          prtrda(prtrpt-1) = gright
                          prtrda(prtrpt  ) = gone
                        endif
                      endif
                      local(ntop) = new
                    else
c                     If new=nleft, sites in new were already counted.
c                     If new=nback AND new=/=nleft, we have to count them now.
                      if(new .ne. nleft) ici = ici + local(new)/2
                      local(new ) = local(ntop)
                      local(ntop) = new
                      global(-local(new)/2) = -1
                    endif
                  else
                    local(ntop) = new
                  endif
                endif
              endif
              if(fourth.and.(nleft.ne.nfourth).and.
     *                     (ntop.ne.nfourth)) then
                nonloc = .false.
                nrlab = local(nfourth)
                if(iand(nrlab,1).eq.1) then
                  nonloc = .true.
                  nrlab = global(-nrlab/2)
                endif
                ici = ici + nrlab/2
                if(new.ne.nfourth) then
                  if(nonloc) then
                    if(iand(local(new),1).eq.1) then
                      gone = globle(-local(new)/2)
                      gtwo = globle(-local(nfourth)/2)
                      if(gone .ne. gtwo) then
```

```
                        gleft = min0(gone, gtwo)
                        gone  = max0(gone, gtwo)
                        globle(-local(new)/2) = gleft
                        if(gone .ne. MAXLOC) then
                          prtlpt = prtlpt + 2
                          if(prtlpt .ge. MAXPR) stop 2
                          prtlda(prtlpt-1) = gleft
                          prtlda(prtlpt  ) = gone
                        endif
                      endif
                      gone = globri(-local(new)/2)
                      gtwo = globri(-local(nfourth)/2)
                      if(gone .ne. gtwo) then
                        gright = min0(gone, gtwo)
                        gone   = max0(gone, gtwo)
                        globri(-local(new)/2) = gright
                        if(gone .ne. MAXLOC) then
                          prtrpt = prtrpt + 2
                          if(prtrpt .ge. MAXPR) stop 2
                          prtrda(prtrpt-1) = gright
                          prtrda(prtrpt  ) = gone
                        endif
                      endif
                      local(nfourth) = new
                    else
                      if((new.ne.nleft).and.(new.ne.ntop))
     *                  ici = ici + local(new)/2
                      local(new) = local(nfourth)
                      local(nfourth) = new
                      global(-local(new)/2) = -1
                    endif
                  else
                    local(nfourth) = new
                  endif
                endif
              endif
              if(back.and.(nleft.ne.nback).and.(ntop.ne.nback)
     *               .and.(nfourth.ne.nback)) then
                nonloc = .false.
                nrlab = local(nback)
                if(iand(nrlab, 1) .eq. 1) then
                  nonloc = .true.
                  nrlab = global(-nrlab/2)
                endif
                ici = ici + nrlab/2
                if(new .ne. nback) then
                  if(nonloc) then
                    if(iand(local(new), 1) .eq. 1) then
                      gone = globle(-local(new )/2)
                      gtwo = globle(-local(nback)/2)
                      if(gone .ne .gtwo) then
                        gleft = min0(gone, gtwo)
                        gone  = max0(gone, gtwo)
                        globle(-local(new)/2) = gleft
                        if(gone .ne. MAXLOC) then
                          prtlpt = prtlpt + 2
                          if(prtlpt .ge. MAXPR) stop 2
                          prtlda(prtlpt-1) = gleft
                          prtlda(prtlpt  ) = gone
                        endif
                      endif
                      gone = globri(-local(new )/2)
                      gtwo = globri(-local(nback)/2)
                      if(gone .ne. gtwo) then
                        gright = min0(gone, gtwo)
                        gone   = max0(gone, gtwo)
                        globri(-local(new)/2) = gright
                        if(gone .ne. MAXLOC) then
                          prtrpt = prtrpt + 2
                          if(prtrpt .ge. MAXPR) stop 2
                          prtrda(prtrpt-1) = gright
                          prtrda(prtrpt  ) = gone
                        endif
                      endif
                      local(nback) = new
                    else
                      local(new  ) = local(nback)
                      local(nback) = new
                      global(-local(new)/2) = -1
                    endif
                  else
                    local(nback) = new
                  endif
                endif
              endif
c             Now write back the number of sites in the cluster to label "new".
c             We have to distinguish the cases that "new" is local or global.
              ici = ici * 2
              if(iand(local(new), 1) .eq. 1) then
                global(-local(new)/2) = ici - 1
              else
                local(new) = ici
              endif
              plane(i) = new
            endif
          else
c           The site in investigation is not occupied.
            plane(i) = MAXLOC
          endif
          i = i + 1
   30     continue
        tim_loc = tim_loc + irtc() - time

c       After calculating the local part, we determine if recycling is
c       necessary. As recycling is an expensive process and the other
c       processors cannot continue with work while one processor is
c       recycling, we recycle in all strips simultaneously. This means
c       that we must recycle if at least one processor runs out of
c       memory; so we have to communicate. On the other hand, we don't
c       want to do this communication after every local step, thus we do
c       it only after all NRECFREQ steps. A processor sets want_recycling to
c       true if it runs out of memory (this is the case when
c       locmin .ge. LIMIT). If at least one processor sets this flag,
c       recycling occurs in all strips.

        time = irtc()
```

```
    rec_countdown = rec_countdown - 1
    if(rec_countdown .eq. 0) then
     rec_countdown = NRECFREQ
     want_recycling=((locmin.ge.LIMIT).or.(glomin.ge.LIMITGLO))
     call glob_or_logical(want_recycling)
     if(want_recycling) then
c     Do recycling.
      nrecyc = nrecyc + 1
      call full_relax()
      call reclass_plane()
c     Next tell our neighbours to which labels in their strips our
c     global rootlabels point and ask them, what the corresponding
c     rootlabels are. We then set our global labels to point to these
c     rootlabels in the neighbouring strips. We do that for all our
c     global labels and talk to our left and right neighbour. After this,
c     we can delete all non-root-labels in all strips.
      call prep_rec_glo()
c     According to theory, we should now be allowed to
c     DELETE ALL NON-ROOT-LABELS, as we have removed all local AND
c     global references to non-root-labels.
c     In preparation for recycling 'dead' local root-labels, we mark all
c     local root-labels by inverting their sign; we mark all global
c     labels, so that we can delete global labels that are no longer
c     referenced by local().

      do j = 1, MAXLOC
       li = local(j)
       if(li .ge. 0) then
        ! is non-root:
        local(j) = 0
       else if(iand(li, 1) .eq. 0) then
        ! is root and local:
        local(j) = -li
       else
        ! is root and global
        nrlab = -li/2
        li = global(nrlab)
        if(li .lt. 0) global(nrlab) = -li
       end if
      end do
c     Now walk through plane and flip back all root-labels that are
c     still in use (don't flip labels twice or flip global labels by
c     accident; first look if the value of local() is positive, then
c     make it negative).
      do j = 1, LPLANE
       nrlab = plane(j)
       if(local(nrlab).gt.0) local(nrlab) = -local(nrlab)
      end do
c     Throw away all root-labels that are still positive, as they are
c     no longer in use. Do not forget to put them into the bins.
      do j = 1, MAXLOC
       if(local(j).gt.0) then
        numroo = numroo + 1
        numsit = numsit + local(j)/2
        fli = local(j)/2
        ibin = dlog(fli)*rcplog+0.00001d0
        if(ibin .le. MAXBIN) ns(ibin) = ns(ibin) + 1
        chi = chi + fli * fli
```

```
                local(j) = 0
              end if
            end do
            locmin = 2
c           Now flip back global labels that are still referenced by local()
c           and delete all others.
            do j = 1, MAXGLO
              nrlab = global(j)
              if(nrlab .gt. 0) then
                global(j) = -nrlab
              else
                global(j) = 0
              end if
            end do
            glomin = 2
            if(ALLOW_GLOBAL_RECYC) then
              call barrier()
c             We walk through all our living global labels and if they
c             have left neighbours, we tell these neighbours that our label
c             points to them, so that they can update their globri().
              j = 0
              prtlpt = 0
              do
                do
                  j = j + 1
                  if(j.gt.MAXLOC) exit
                  if(local(j).ge.0.or.iand(local(j),1).eq.0) cycle
                  k = -local(j)/2
                  if(globle(k).ne.MAXLOC) then
                    prtlpt = prtlpt + 2
                    prtlda(prtlpt-1) = globle(k)
                    prtlda(prtlpt  ) = j
                    if(prtlpt.gt.MAXPR-5) exit
                  end if
                end do
                once_more = .false.
                if(j.lt.MAXLOC) once_more = .true.
                prtlpt = prtlpt + 1                ! add sentinel
                prtlda(prtlpt) = 0
                call shmem_put(prrcda(1),prtlda(1),prtlpt,pe_le)
                call barrier()
                prtlpt = 0
                prrcpt = 0
                do
                  prrcpt = prrcpt + 2
                  if(prrcpt.gt.MAXPR) STOP 8 !CANTHAPPEN
                  if(prrcda(prrcpt-1).eq.0) exit
                  nrlab = prrcda(prrcpt-1)
                  do
                    if(local(nrlab).le.0) exit
                    nrlab = local(nrlab)
                  end do
                  if(local(nrlab).eq.0) cycle
                  k = -local(nrlab)/2
                  if(globri(k).eq.MAXLOC) then
                    globri(k) = prrcda(prrcpt)
                  end if
                end do
            call glob_or_logical(once_more)
            if(.not. once_more) exit
          end do
          ! Do global recycling by reduction
          ! First we mark living global labels
          do j = 1, LPLANE
            nrlab = plane(j)
            if((nrlab .ne. MAXLOC)
     *       .and. (iand(local(nrlab),1).eq.1)) then
              nrlab = -local(nrlab)/2
              if(global(nrlab).lt.0) global(nrlab) = -global(nrlab)
            end if
          end do
          ! Now we have marked all living global labels by inverted sign.
          ! All global labels that are dead in our strip are not flipped.
          ! We can recycle only global labels, that are dead and have no
          ! right neigbour. In the next sweep, we flip roles: Dead labels
          ! without right neighbour become positive, others negative.
          do j = 1, MAXGLO
            if(global(j).eq.0) cycle
            if(global(j).gt.0) then
              global(j)=-global(j)
            else
              if((globle(j).ne.MAXLOC).and.
     *          (globri(j).eq.MAXLOC)) global(j)=-global(j)
            end if
          end do
          ! All labels that shall be recycled are now positive in global(),
          ! all others are negative (or zero if not occupied at all).
          j = 0
          icount = 0
          prtlpt = 0
          do
            icount = icount + 1
            do
              j = j + 1
              if(j .gt. MAXLOC) exit
              if( local(j).ge.0 .or. iand(local(j),1).eq.0 ) cycle
              k = -local(j)/2
              if( global(k).gt.0 ) then
                prtlpt = prtlpt + 2
                prtlda(prtlpt-1) = globle(k)
                prtlda(prtlpt  ) = -global(k)
                global(k) = 0
                local(j) = 0
                if( prtlpt.gt.MAXPR-5 ) exit
              end if
            end do
            once_more = .false.
            if( j.lt.MAXLOC ) once_more = .true.
            ! Add sentinel
            prtlpt = prtlpt + 1
            prtlda(prtlpt) = 0
            call barrier()
            call shmem_put(prrcda(1), prtlda(1), prtlpt, pe_le)
            call barrier()
            ! Walk through received data and put it into our global()
            prtlpt = 0
```

```
              prrcpt = 0
              do
                prrcpt = prrcpt + 2
                if(prrcpt.gt.MAXPR) STOP 8 !CANTHAPPEN
                if(prrcda(prrcpt-1).eq.0) exit
                nrlab = prrcda(prrcpt-1)
                do
                  if(nrlab.eq.0) exit
                  if(local(nrlab).lt.0) exit
                  nrlab = local(nrlab)
                end do
                if(nrlab.eq.0) cycle
                k = -local(nrlab)/2
                global(k) = (global(k)/2 + prrcda(prrcpt)/2) * 2 - 1
                globri(k) = MAXLOC
                if(globle(k).eq.MAXLOC) then
                  ! Convert our label back to local.
                  local(nrlab) = global(k) + 1
                  global(k) = 0
                end if
              end do
              call glob_or_logical(once_more)
              if(.not. once_more) exit
            end do
c           We walk through all our living global labels and if they
c           have left neighbours, we tell these neighbours that our label
c           points to them, so that they can update their globri().
            j = 0
            prtlpt = 0
            do
              do
                j = j + 1
                if(j.gt.MAXLOC) exit
                if(local(j).ge.0.or.iand(local(j),1).eq.0) cycle
                k = -local(j)/2
                if(globle(k).ne.MAXLOC) then
                  prtlpt = prtlpt + 2
                  prtlda(prtlpt-1) = globle(k)
                  prtlda(prtlpt  ) = j
                  if(prtlpt.gt.MAXPR-5) exit
                end if
              end do
              once_more = .false.
              if(j.lt.MAXLOC) once_more = .true.
              prtlpt = prtlpt + 1                ! add sentinel
              prtlda(prtlpt) = 0
              call shmem_put(prrcda(1),prtlda(1),prtlpt,pe_le)
              call barrier()
              prtlpt = 0
              prrcpt = 0
              do
                prrcpt = prrcpt + 2
                if(prrcpt.gt.MAXPR) STOP 8 !CANTHAPPEN
                if(prrcda(prrcpt-1).eq.0) exit
                nrlab = prrcda(prrcpt-1)
                do
                  if(local(nrlab).lt.0) exit
                  nrlab = local(nrlab)
```

```
                 end do
                 k = -local(nrlab)/2
                 if(globri(k).eq.MAXLOC) then
                   globri(k) = prrcda(prrcpt)
                 end if
               end do
               call glob_or_logical(once_more)
               if(.not. once_more) exit
             end do
           endif
         endif
       endif
       tim_recyc = tim_recyc + irtc() - time

c      We have calculated the local part in our strip. Now we have to
c      communicate with our neighbours to find out, which clusters are
c      interconnected between strips.

       time = irtc()
c      To make life easier, we reclassify our borders.
c      Left border:
       do j = 1, LLINE
         nrlab = plane(j)
         if(local(nrlab) .gt. 0) then
   41      nrlab = local(nrlab)
           if(local(nrlab) .gt. 0) goto 41
           local(plane(j)) = nrlab
           plane(j) = nrlab
         endif
       end do
c      Reclassify right border:
       do j = LPLANE-LLINE+1, LPLANE
         nrlab = plane(j)
         if(local(nrlab) .gt. 0) then
   42      nrlab = local(nrlab)
           if(local(nrlab) .gt. 0) goto 42
           local(plane(j)) = nrlab
           plane(j) = nrlab
         endif
       end do

c      First we receive the right border of our left neighbour and
c      store it left of our left border. We then examine it. If there
c      is no interconnection, we are happy, if not, we have to work.
c      Because PLANE() is to large to put it into a common block, we
c      have to use a (counter-intuitive) trick: We store the data-to-transmit
c      in border_send() and receive it in border_recv(). For transfers within
c      one PE, that ist transfer from plane() to border_send(), we can use
c      shmem_put(), because it does not conflict with local addresses.

       call shmem_put(border_send(1), plane(LPLANE-LLINE+1),
     *               LLINE, pe_this)
       call barrier()
       call shmem_get(border_recv(1), border_send(1), LLINE, pe_le)
       call barrier()
       do j = 1, LLINE
         if(max0(border_recv(j), plane(j)) .ne. MAXLOC) then
c          There is an interconnection.
```

```
            nrlab = plane(j)
            if(local(nrlab) .gt. 0) then
   53         nrlab = local(nrlab)
              if(local(nrlab) .gt. 0) goto 53
              local(plane(j)) = nrlab
              plane(j) = nrlab
            end if
            nrlab = local(nrlab)
            if(iand(nrlab, 1) .eq. 0) then
c             Our label is a local label and has to be transferred to a
c             global label.
c             First find a free global label.
   51         glolab = glomin
              glomin = glomin + 1
              if(glomin .ge. MAXGLO) stop 4
              if(global(glolab).ne.0) goto 51
c             "glolab" is our new free global label.
              global(glolab) = nrlab - 1
              globle(glolab) = border_recv(j)
              globri(glolab) = MAXLOC
              local(plane(j)) = -(2 * glolab + 1)
            else
c             Our label is a global label, so we have to generate
c             pairing information for our left neighbour.
c             If the labels in border_recv(j) and in globle(-nrlab/2) are
c             different, we have to tell our left neigbour to join them, and
c             store only the smaller one. If they are the same, we don't have
c             to do anything.
              gone = border_recv(j)
              gtwo = globle(-nrlab/2)
              if(gone .ne. gtwo) then
                gleft = min0(gone, gtwo)
                gone  = max0(gone, gtwo)
                if(gone .ne. MAXLOC) then
                  globle(-nrlab/2) = gleft
                  prtlpt = prtlpt + 2
                  if(prtlpt .ge. MAXPR) stop 2
                  prtlda(prtlpt-1) = gleft
                  prtlda(prtlpt  ) = gone
                endif
              endif
            endif
          endif
        end do

c       Left border:
        do j = 1, LLINE
          nrlab = plane(j)
          if(local(nrlab) .gt. 0) then
   43       nrlab = local(nrlab)
            if(local(nrlab) .gt. 0) goto 43
            local(plane(j)) = nrlab
            plane(j) = nrlab
          endif
        end do
c       Reclassify right border:
        do j = LPLANE-LLINE+1, LPLANE
          nrlab = plane(j)
          if(local(nrlab) .gt. 0) then
   44       nrlab = local(nrlab)
            if(local(nrlab) .gt. 0) goto 44
            local(plane(j)) = nrlab
            plane(j) = nrlab
          endif
        end do
c       Next we receive the left border of our right neighbour and store
c       it right of our right border.
        call shmem_put(border_send(1), plane(1), LLINE, pe_this)
        call barrier()
        call shmem_get(border_recv(1), border_send(1), LLINE, pe_ri)
        call barrier()
        do j = 1, LLINE
          if(max0(border_recv(j), plane(j+LPLANE-LLINE))
     *              .ne. MAXLOC) then
            nrlab = plane(j+LPLANE-LLINE)
            if(local(nrlab) .gt. 0) then
   54         nrlab = local(nrlab)
              if(local(nrlab) .gt. 0) goto 54
              local(plane(j+LPLANE-LLINE)) = nrlab
              plane(j+LPLANE-LLINE) = nrlab
            end if
            nrlab = local(nrlab)
            if(iand(nrlab, 1) .eq. 0) then
   52         glolab = glomin
              glomin = glomin + 1
              if(glomin .ge. MAXGLO) stop 4
              if(global(glolab) .ne. 0) goto 52
              global(glolab) = nrlab - 1
              globri(glolab) = border_recv(j)
              globle(glolab) = MAXLOC
              local(plane(j+LPLANE-LLINE)) = -(2 * glolab + 1)
            else
              gone = globri(-nrlab/2)
              gtwo = border_recv(j)
              if(gone .ne. gtwo) then
                gright = min0(gone, gtwo)
                gone   = max0(gone, gtwo)
                if(gone .ne. MAXLOC) then
                  globri(-nrlab/2) = gright
                  prtrpt = prtrpt + 2
                  if(prtrpt .ge. MAXPR) stop 2
                  prtrda(prtrpt-1) = gright
                  prtrda(prtrpt  ) = gone
                endif
              endif
            endif
          endif
        end do

        tim_nbex = tim_nbex + irtc() - time

c       Now we have to exchange the pairing information and actually do
c       the pairing. It is important to remember that during this pairing,
c       new pairing information can come up (i. e., two non-local clusters
c       with different left neighbours are being paired, so we have to inform
c       our left neighbour strip that also these two clusters have to be
```

```
c       paired. We could do this using a relaxation process: we repeat
c       the pairing process until there are no more pairing information to
c       transmit in any of the nodes, so the relaxation is at end.
c       Luckily, we do not have to do such an expensive process: because after
c       each local percolation step pairing is done, we just do the next
c       relaxation step after the next local step. Only after all local steps
c       are over, which means the whole system has been examined, we have to
c       do the expensive relaxation until there are no more changes detected.

c       First we transmit pairing information to our left neighbour.

c       Add a sentinel at the end of the pairing list; we do not transmit
c       the number of elements in the pairing list, the sentinel denotes
c       the end.
        time = irtc()

        prtlpt = prtlpt + 1
        if(prtlpt .ge. MAXPR) stop 2
        prtlda(prtlpt) = 0

        call barrier()
        call shmem_put(prrcda(1), prtlda(1), prtlpt, pe_le)
        call barrier()
        prtlpt = 0

c       Now walk through the list of pairing information.
        prrcpt = 0
   60   continue
        prrcpt = prrcpt + 2
        if(prrcpt .ge. MAXPR) stop 2   ! CANTHAPPEN
        none = prrcda(prrcpt-1)
        ntwo = prrcda(prrcpt  )
c       When we reach the sentinel, we leave the loop.
        if(none .eq. 0) goto 69
c       Reclassify the labels.
        if(local(none) .gt. 0) then
          nold = none
   61     none = local(none)
          if(local(none) .gt. 0) goto 61
          local(nold) = none
        endif
        if(local(ntwo) .gt. 0) then
          nold = ntwo
   62     ntwo = local(ntwo)
          if(local(ntwo) .gt. 0) goto 62
          local(nold) = ntwo
        endif
c       At this point, we already know that both labels are global, otherwise
c       they would not have been put into the pairing list. But it can happen
c       that both labels belong to the same cluster after reclassification.
        if(none .ne. ntwo) then
c         Redirect "ntwo" to "none". Count number of sites. Generate pairing
c         info, but only for the LEFT neighbour, as we received the note for
c         pairing these labels from the right.
          gone = globle(-local(none)/2)
          gtwo = globle(-local(ntwo)/2)
          if(gone .ne. gtwo) then
            gleft = min0(gone, gtwo)
```

```
              gone  = max0(gone, gtwo)
              globle(-local(none)/2) = gleft
              if(gone .ne. MAXLOC) then
                prtlpt = prtlpt + 2
                if(prtlpt .ge. MAXPR) stop 2 !CANTHAPPEN
                prtlda(prtlpt-1) = gleft
                prtlda(prtlpt  ) = gone
              endif
            endif
            ici =       global(-local(none)/2)/2
            ici = ici + global(-local(ntwo)/2)/2
            global(-local(none)/2) = 2 * ici - 1
            local(ntwo) = none
          endif
          goto 60
   69     continue

c         Next transmit pairing information to our right neighbour.
c         For detailed comments on the process, see above.

c         Add sentinel.
          prtrpt = prtrpt + 1
          if(prtrpt .ge. MAXPR) stop 2
          prtrda(prtrpt) = 0

c         Transmit.
          call barrier()
          call shmem_put(prrcda(1), prtrda(1), prtrpt, pe_ri)
          call barrier()
          prtrpt = 0

c         Walk through pairing information.
          prrcpt = 0
   70     continue
          prrcpt = prrcpt + 2
          if(prrcpt .ge. MAXPR) stop 2 !CANTHAPPEN
          none = prrcda(prrcpt-1)
          ntwo = prrcda(prrcpt  )
c         Quit on sentinel reached.
          if(none .eq. 0) goto 79
c         Reclassify labels.
          if(local(none) .gt. 0) then
            nold = none
   71       none = local(none)
            if(local(none) .gt. 0) goto 71
            local(nold) = none
          endif
          if(local(ntwo) .gt. 0) then
            nold = ntwo
   72       ntwo = local(ntwo)
            if(local(ntwo) .gt. 0) goto 72
            local(nold) = ntwo
          endif

c         Join them if they are not the same.
          if(none .ne. ntwo) then
c           Redirect and count; generate pairing info only for RIGHT neighbour.
            gone = globri(-local(none)/2)
```

```
     gtwo = globri(-local(ntwo)/2)
     if(gone .ne. gtwo) then
      gright = min0(gone, gtwo)
      gone   = max0(gone, gtwo)
      globri(-local(none)/2) = gright
      if(gone .ne. MAXLOC) then
       prtrpt = prtrpt + 2
       if(prtrpt .ge. MAXPR) stop 2 !CANTHAPPEN
       prtrda(prtrpt-1) = gright
       prtrda(prtrpt  ) = gone
      endif
     endif
     ici =       global(-local(none)/2)/2
     ici = ici + global(-local(ntwo)/2)/2
     global(-local(none)/2) = 2 * ici - 1
     local(ntwo) = none
    endif
    goto 70
 79 continue

    tim_prex = tim_prex + irtc() - time

 20 continue

   tim_perc = irtc() - tim_perc

c  Now the global percolation process has ended. We just have two more
c  tasks to do: full relaxation of all pairing information and accounting
c  for clusters and connectivity.

c  Full relaxation of paring information consists of the following
c  processes: nearest neighbour interaction (transmit pairing info
c  to left and right and walk through received pairing info); tell the
c  master node if there has been any pairing info transmitted; the master
c  node decides if there were transmissions and thus more relaxation has
c  to be done; the master node tells all slaves if they have to repeat
c  the process or not; repeat everything until fixpoint reached.

   tim_fullrelax = irtc()
   call full_relax()
   tim_fullrelax = irtc() - tim_fullrelax


c  Now we also have done the full relaxation. After this hell of work,
c  we can now get to the results: we will do the final accounting.
c  But on parallel computers, even this is a difficult task.

   tim_loccount = irtc()

c  First we count the local clusters. During this process, we throw away
c  everything we have counted. Of course, we must not throw away nonroot
c  labels that are connected to a nonlocal cluster.
   do 220 i = 1, MAXLOC
    nrlab = i
    if(local(nrlab) .gt. 0) then
 221 nrlab = local(nrlab)
     if(local(nrlab) .gt. 0) goto 221
    endif
```

```
        nrlab = local(nrlab)
        if(nrlab .eq. 0) goto 220
        if(iand(nrlab, 1) .eq. 0) then
c         It is local, we can take it into account.
          if(local(i) .lt. 0) then
            numroo = numroo + 1
            numsit = numsit - local(i)/2
            fli = -local(i)/2
            ibin = dlog(fli)*rcplog+0.00001d0
            if(ibin .le. MAXBIN) ns(ibin) = ns(ibin) + 1
            chi = chi + fli*fli
          endif
          local(i) = 0
        endif
  220   continue
      tim_loccount = irtc() - tim_loccount

c     In order to be able to put also global clusters into bins, we have to
c     concentrate the number of sites in that cluster, which means that one
c     processor can report the number of all sites in that global cluster.

      tim_concen = irtc()
      do 300 istep = 1, 2*NSTRIP
        i = 1
  301   continue
c         We remember the necessity of transmitting one more round in this
c         step by setting nonloc to true.
          nonloc = .false.
          prtlpt = 0
c         All labels are either global root-labels or non-root-labels pointing
c         to global root-labels. We just transmit the site-numbers of the
c         root-labels to the left neighbour (if there is one).
          if(i .ge. MAXLOC) goto 303
  302     continue
            nrlab = local(i)
            i = i + 1
            if(nrlab .lt. 0) then
c             Is a root-label.
              nleft = globle(-nrlab/2)
              if(nleft .ne. MAXLOC) then
c               Has a left neighbour.
                nonloc = .true.
                prtlpt = prtlpt + 2
                if(prtlpt .ge. MAXPR) stop 2 !CANTHAPPEN
                prtlda(prtlpt-1) = nleft
                prtlda(prtlpt  ) = global(-nrlab/2)
                global(-nrlab/2) = 0
              endif
            endif
            if( (prtlpt .gt. MAXPR-5) .or.
     *          (i .ge. MAXLOC) ) goto 303
            goto 302
c         Now transmit the information.
  303     continue
          prtlpt = prtlpt + 1
          if(prtlpt .ge. MAXPR) stop 2 !CANTHAPPEN
c         Add sentinel.
          prtlda(prtlpt) = 0
          call barrier()
          call shmem_get(prrcda(1), prtlda(1), MAXPR, pe_ri)
          call barrier()
          prtlpt = 0
          prrcpt = 0
c         Put the received info into our data.
  304     continue
            prrcpt = prrcpt + 2
            if(prrcpt .ge. MAXPR) stop 2 !CANTHAPPEN
            nrlab = prrcda(prrcpt-1)
            if(nrlab .eq. 0) goto 306
            if(local(nrlab) .gt. 0) then
  305         nrlab = local(nrlab)
              if(local(nrlab) .gt. 0) goto 305
            endif
            nrlab = local(nrlab)
            global(-nrlab/2) = global(-nrlab/2)
     *                       + (prrcda(prrcpt)/2)*2
            goto 304
  306     continue

          if(i.ge.MAXLOC) nonloc = .false.
          call glob_or_logical(nonloc)
          if(nonloc) goto 301
  300   continue
        tim_concen = irtc() - tim_concen

        tim_globcount = irtc()
        numofinf=0
c       Now we have to put the labels into the bins.
        conn = .false.
        do 310 i = 1, MAXLOC
          nrlab = local(i)
          if(nrlab .eq. 0) goto 310
          if(nrlab .gt. 0) then
c           All non-root-labels that are still present, are connected to
c           a global cluster, otherwise they would have been discarded
c           after the local counting.
            nonroo = nonroo + 1
          else
            numglo = numglo + 1
            nsize = global(-nrlab/2)
            numsit = numsit - nsize/2
c           Count this global cluster only if it is concentrated; do not
c           count part of a horizontically percolating cluster.
            if((-nsize/2.gt.0).and.(globle(-nrlab/2).eq.MAXLOC)) then
              numgloclu = numgloclu + 1
              fli = -nsize/2
              ibin = dlog(fli)*rcplog+0.0001d0
              if(ibin .le. MAXBIN) ns(ibin) = ns(ibin) + 1
              chi = chi + fli*fli
c             print *, pe_this, 'global', i, -nsize/2
            else
              if(nsize/2.ne.0) then
                conn = .true.
                ninfclu = ninfclu - nsize/2
                numofinf = numofinf + 1
              endif
          endif
        endif
  310 continue
      call glob_or_logical(conn)

c     We have to sum up all the bins. And all the other interesting data.
      do j = 0, MAXBIN
        call glob_sum_integer(ns(j))
      end do

      call glob_sum_integer(numofinf)
      call glob_sum_integer(nocc)
      call glob_sum_integer(ninfclu)
      call glob_sum_integer(nonroo)
      call glob_sum_integer(numroo)
      call glob_max_integer(numglo)
      call glob_sum_integer(numgloclu)
      call glob_sum_integer(numsit)
      call glob_sum_real(chi)
      tim_globcount = irtc() - tim_globcount

      tim_label1 = irtc()

c     Now we have to take care of the horizontically percolating cluster.
      fli = ninfclu

      if(fli.gt.0) then
      ibin = dlog(fli)*rcplog+0.0001d0
      if(ibin .le. MAXBIN) ns(ibin) = ns(ibin) + 1
      endif

      tim_label1 = irtc() - tim_label1

      tim_glob = irtc() - tim_glob


      call glob_sum_integer(tim_loc)
      call glob_sum_integer(tim_prex)
      call glob_sum_integer(tim_nbex)
      call glob_sum_integer(tim_perc)
      call glob_sum_integer(tim_fullrelax)
      call glob_sum_integer(tim_detconn)
      call glob_sum_integer(tim_loccount)
      call glob_sum_integer(tim_concen)
      call glob_sum_integer(tim_globcount)
      call glob_sum_integer(tim_label1)
      call glob_sum_integer(tim_glob)
      call glob_sum_integer(tim_recyc)
c     Do the output. All times in milliseconds.
      call barrier()
      if(pe_this .eq. 0) then
        print *, '# Using Kirkpatrick-Stoll PRNG'
        if(ALLOW_GLOBAL_RECYC) then
          print *, '# Global recycling on'
        end if
        print *, '# Local time: ', tim_loc*1.333d-8*1000
        print *, '# Nb. exch. time: ', tim_nbex*1.333d-5
        print *, '# Pr. exch. time: ', tim_prex*1.333d-5
```

```
          print *, '# Recycling time: ', tim_recyc*1.333d-5
          print *, '# Pure percolation time: ', tim_perc*1.333d-5
          print *, '# Full relaxation time: ', tim_fullrelax*1.333d-5
          print *, '# Local counting time: ', tim_loccount*1.333d-5
          print *, '# Concentration time: ', tim_concen*1.333d-5
          print *, '# Global counting time: ', tim_globcount*1.333d-5
          print *, '# Glob.-perc. time: ', (tim_glob-tim_perc)*1.333d-5
          print *, '# Global total time: ', tim_glob*1.333d-5
          print *, '# Size of system: ', L, ' ** ', IDIM
          print *, '# Number of strips: ', NSTRIP
          print *, '# Initial random seed: ', ISEED
          print *, '# Probability: ', P
          print *, '# Integer Probabilty: ', IP
          print *, '# Number of occupied sites: ', numsit + ninfclu
          print *, '# nocc: ', nocc
          print *, '# Local rootlabels: ', numroo
          print *, '# Global clusters: ', numgloclu
          print *, '# Total number of clusters: ', numroo+numgloclu
          print *, '# Number density: ', (1.0*(numroo+numgloclu)/NSYS)
          print *, '# Max. of global roots: ', numglo
          print *, '# Nonroots pointing to Globals: ', nonroo
          print *, '# MAXLOC = ', MAXLOC
          print *, '# LIMIT = ', DIVLIMMAX, ' * MAXLOC'
          print *, '# Number of garbage collections: ', nrecyc
          if(conn) then
            print *, '# Size of infinite cluster: ', ninfclu
          else
            print *, '# No infinite cluster'
          endif
          print *, '# Number of infinite cluster labels', numofinf
          print *, '# Second moment: ', chi/NSYS

          do ibin = 0, MAXBIN
            if(ns(ibin) .ne. 0) print *, 2**ibin, ns(ibin)
          enddo
        endif

        contains

        subroutine prep_rec_glo()
          logical once_more
          integer nrbegin, nrend, nrlab, k
          ! to left:
          nrbegin = 1
          do
            nrend = nrbegin - 1
            prtlpt = 0
            once_more = .false.
            do
              nrend = nrend + 1
              if((global(nrend).ne.0)
     *             .and.(globle(nrend).ne.MAXLOC)) then
                prtlpt = prtlpt + 1
                prtlda(prtlpt) = globle(nrend)
              end if
              if((prtlpt.ge.MAXPR-5).or.(nrend.ge.MAXGLO)) exit
            end do
            if(prtlpt.gt.1) once_more = .true.
          prtlpt = prtlpt + 1
          prtlda(prtlpt) = 0
          call barrier()
          call shmem_put(prrcda(1), prtlda(1), prtlpt, pe_le)
          call barrier()
          prrcpt = 0
          prtlpt = 0
          do
            prrcpt = prrcpt + 1
            nrlab = prrcda(prrcpt)
            if(nrlab .eq. 0) exit
            do
              if(local(nrlab).le.0) exit
              nrlab = local(nrlab)
            end do
            prrcda(prrcpt) = nrlab
          end do
          call barrier()
          call shmem_put(prtlda(1), prrcda(1), prrcpt, pe_ri)
          call barrier()
          ! in prtlda is now the reclassified info. We have to put it back
          ! to globle()
          nrend = nrbegin - 1
          prtlpt = 0
          do
            nrend = nrend + 1
            if((global(nrend).ne.0)
     *             .and.(globle(nrend).ne.MAXLOC)) then
              prtlpt = prtlpt + 1
              globle(nrend) = prtlda(prtlpt)
            end if
            if(prtlda(prtlpt+1).eq.0) exit
          end do
          nrbegin = nrend
          call glob_or_logical(once_more)
          if(.not. once_more) exit
        end do
        ! to right:
        nrbegin = 1
        do
          nrend = nrbegin - 1
          prtrpt = 0
          once_more = .false.
          do
            nrend = nrend + 1
            if((global(nrend).ne.0)
     *             .and.(globri(nrend).ne.MAXLOC)) then
              prtrpt = prtrpt + 1
              prtrda(prtrpt) = globri(nrend)
            end if
            if((prtrpt.ge.MAXPR-5).or.(nrend.ge.MAXGLO)) exit
          end do
          if(prtrpt.gt.1) once_more = .true.
          prtrpt = prtrpt + 1
          prtrda(prtrpt) = 0
          call barrier()
          call shmem_put(prrcda(1), prtrda(1), prtrpt, pe_ri)
          call barrier()
        prrcpt = 0
        prtrpt = 0
        do
          prrcpt = prrcpt + 1
          nrlab = prrcda(prrcpt)
          if(nrlab .eq. 0) exit
          do
            if(local(nrlab).le.0) exit
            nrlab = local(nrlab)
          end do
          prrcda(prrcpt) = nrlab
        end do
        call barrier()
        call shmem_put(prtrda(1), prrcda(1), prrcpt, pe_le)
        call barrier()
        ! in prtrda is now the reclassified info. We have to put it back
        ! to globri()
        nrend = nrbegin - 1
        prtrpt = 0
        do
          nrend = nrend + 1
          if((global(nrend).ne.0)
     *             .and.(globri(nrend).ne.MAXLOC)) then
            prtrpt = prtrpt + 1
            globri(nrend) = prtrda(prtrpt)
          end if
          if(prtrda(prtrpt+1).eq.0) exit
        end do
        nrbegin = nrend
        call glob_or_logical(once_more)
        if(.not. once_more) exit
      end do
      prtlpt = 0
      prtrpt = 0
    end subroutine

    subroutine klass(nrlab)
      integer nrlab, nold
      if(local(nrlab) .gt. 0) then
        nold = nrlab
        do
          nrlab = local(nrlab)
          if(local(nrlab) .le. 0) exit
        end do
        local(nold) = nrlab
      end if
    end subroutine

    subroutine reclass_plane()
      integer k, nold, nrlab
      do k = 1, LPLANE
        nrlab = plane(k)
        if(local(nrlab).gt.0) then
          nold = nrlab
          do
            nrlab = local(nrlab)
            if(local(nrlab).le.0) exit
          end do
        local(nold) = nrlab
        plane(k) = nrlab
      end if
    end do
  end subroutine

  subroutine full_relax()
    logical once_more
    do
      once_more = .false.
      ! to left:
      if(prtlpt .ne. 0) once_more = .true.
      prtlpt = prtlpt + 1            ! add sentinel
      if(prtlpt .ge. MAXPR) stop 2
      prtlda(prtlpt) = 0
      call barrier()
      call shmem_put(prrcda(1), prtlda(1), prtlpt, pe_le)
      call barrier()
      prtlpt = 0
      prrcpt = 0
      do
        prrcpt = prrcpt + 2
       !if(prrcpt .ge. MAXPR) stop 2  ! CANTHAPPEN
        none = prrcda(prrcpt-1)
        ntwo = prrcda(prrcpt  )
        if(none .eq. 0) exit       ! reached sentinel
        call pair_to_left(none, ntwo)
      end do
      ! to right:
      if(prtrpt .ne. 0) once_more = .true.
      prtrpt = prtrpt + 1            ! add sentinel
      if(prtrpt .ge. MAXPR) stop 2
      prtrda(prtrpt) = 0
      call barrier()
      call shmem_put(prrcda(1), prtrda(1), prtrpt, pe_ri)
      call barrier()
      prtrpt = 0
      prrcpt = 0
      do
        prrcpt = prrcpt + 2
       !if(prrcpt .ge. MAXPR) stop 2    ! CANTHAPPEN
        none = prrcda(prrcpt-1)
        ntwo = prrcda(prrcpt  )
        if(none .eq. 0) exit       ! reached sentinel
        call pair_to_right(none, ntwo)
      end do
      ! one more time?
      call glob_or_logical(once_more)
      if(.not. once_more) exit
    end do
  end subroutine

  subroutine pair_to_left(none, ntwo)
    integer none, ntwo, gone, gtwo, gleft
    call klass(none)
    call klass(ntwo)
    if(none .ne. ntwo) then
      gone = globle(-local(none)/2)
```

```
            gtwo = globle(-local(ntwo)/2)
            if(gone .ne. gtwo) then
              gleft = min0(gone, gtwo)
              gone  = max0(gone, gtwo)
              globle(-local(none)/2) = gleft
              if(gone .ne. MAXLOC) then
                prtlpt = prtlpt + 2
                if(prtlpt .ge. MAXPR) stop 2     ! can it happen?
                prtlda(prtlpt-1) = gleft
                prtlda(prtlpt  ) = gone
              end if
            end if
            global(-local(none)/2) = 2 * (global(-local(none)/2)/2
     *       + global(-local(ntwo)/2)/2) - 1
            local(ntwo) = none
          end if
        end subroutine

        subroutine pair_to_right(none, ntwo)
          integer none, ntwo, gone, gtwo, gright
          call klass(none)
          call klass(ntwo)
          if(none .ne. ntwo) then
            gone = globri(-local(none)/2)
            gtwo = globri(-local(ntwo)/2)
            if(gone .ne. gtwo) then
              gright = min0(gone, gtwo)
              gone   = max0(gone, gtwo)
              globri(-local(none)/2) = gright
              if(gone .ne. MAXLOC) then
                prtrpt = prtrpt + 2
                if(prtrpt .ge. MAXPR) stop 2     ! can it happen?
                prtrda(prtrpt-1) = gright
                prtrda(prtrpt  ) = gone
              end if
            end if
            global(-local(none)/2) = 2 * (global(-local(none)/2)/2
     *       + global(-local(ntwo)/2)/2) - 1
            local(ntwo) = none
          end if
        end subroutine

c     The following subroutine synchronises a logical variable over
c     all pe’s. That means, it takes the value of the variable on all
c     pe’s, does a logical or on these values and distributes the
c     result back to the variables. The variable does not need to be
c     in a common-block, as the special variable comm_logvar is used
c     for data-transfer.
        subroutine glob_or_logical(logvar)
          logical logvar
          integer k
          comm_logvar = logvar
          call barrier()
          if(pe_this.eq.0) then
            do k = 1, NSTRIP-1
              call shmem_logical_get(comm_logvar, comm_logvar, 1, k)
              logvar = logvar .or. comm_logvar
            end do
            comm_logvar = logvar
            do k = 1, NSTRIP-1
              call shmem_logical_put(comm_logvar, comm_logvar, 1, k)
            end do
          end if
          call barrier()
          logvar = comm_logvar
        end subroutine

c       Do a global sum of integers and redistribute the result back to
c       the variables.
        subroutine glob_sum_integer(intvar)
          integer intvar, k
          comm_intvar = intvar
          call barrier()
          if(pe_this.eq.0) then
            do k = 1, NSTRIP-1
              call shmem_integer_get(comm_intvar, comm_intvar, 1, k)
              intvar = intvar + comm_intvar
            end do
            comm_intvar = intvar
            do k = 1, NSTRIP-1
              call shmem_integer_put(comm_intvar, comm_intvar, 1, k)
            end do
          end if
          call barrier()
          intvar = comm_intvar
        end subroutine

c       Find the minimum integer.
        subroutine glob_min_integer(intvar)
          integer intvar, k
          comm_intvar = intvar
          call barrier()
          if(pe_this .eq. 0) then
            do k = 1, NSTRIP - 1
              call shmem_integer_get(comm_intvar, comm_intvar, 1, k)
              intvar = min0(intvar, comm_intvar)
            end do
            comm_intvar = intvar
            do k = 1, NSTRIP - 1
              call shmem_integer_put(comm_intvar, comm_intvar, 1, k)
            end do
          end if
          call barrier()
          intvar = comm_intvar
        end subroutine

c       Find the maximum integer.
        subroutine glob_max_integer(intvar)
          integer intvar
          intvar = -intvar
          call glob_min_integer(intvar)
          intvar = -intvar
        end subroutine

c       Do a global sum of reals and redistribute the result back to
c       the variables.
```

```
      subroutine glob_sum_real(realvar)
        real realvar
        integer k
        comm_realvar = realvar
        call barrier()
        if(pe_this.eq.0) then
          do k = 1, NSTRIP-1
            call shmem_real_get(comm_realvar, comm_realvar, 1, k)
            realvar = realvar + comm_realvar
          end do
          comm_realvar = realvar
          do k = 1, NSTRIP-1
            call shmem_real_put(comm_realvar, comm_realvar, 1, k)
          end do
        end if
        call barrier()
        realvar = comm_realvar
      end subroutine

      subroutine ks_warmup()
        integer i, ii, ibm
        integer ici, one
        integer k, up
        one = 1
        ibm = 2 * (ISEED + pe_this) - 1
        do i = 0, 255
          ici = 0
          do ii = 1, 64
            ici = ishft(ici, 1)
            ibm = ibm * 16807
            if(ibm .lt. 0) ici = ior(ici, one)
          end do
          ks_rnd(i) = ici
        end do
        ks_idx = 0
        do i = 1, 8*256
          ks_idx = iand(ks_idx + 1, 255)
          ks_rnd(ks_idx) = ieor(ks_rnd(iand(ks_idx-103, 255)),
     *                          ks_rnd(iand(ks_idx-250, 255)) )
        end do
      end subroutine

      end
```

C.2 Sequential program with averaging

I used the program below on sequential computers mainly to study finite-size effects and statistical fluctuations. It also served as a test for the parallel program. Of course, it is not possible to simulate huge lattices on sequential computers.

The program supports both open boundaries and busbar, is suitable for two to seven dimensions, and can utilize different PRNGs. It does several runs for a given system size and averages over them.

```
c Random site percolation in up to seven dimensions. Counts clusters.
c Uses recycling of labels.
c Do several runs and find out not only average values, but also statistical
c errors.
```

```
c label(i) negative: rootlabel with -label(i) sites in the cluster.
c label(i) positive: points to other label
c label(i) zero:     free label, can be reused

c2345 7
      parameter (ISEED=1, L=301, IDIM=4, NSYS=L**IDIM,
     *           LPLANE=L**(IDIM-1), MAX=100e6,
     *           P=0.196889, MAXBIN=45, MBP1=MAXBIN+1,
     *           LIMIT=0.95*MAX,
     *           NREPEAT=50)
      dimension plane(LPLANE), label(MAX), ns(0:MAXBIN)
      data ns/MBP1*0/
      integer plane
      logical conn, left, top, three, four, five, six, seven
      real*8 fli, chi
      integer*4 IP, ks(0:16383), idx
      integer*8 sum
      real*8 ftemp, fsum
      real*8 fbin1(0:MAXBIN), fbin2(0:MAXBIN)
      real*8 fnc1, fnc2, fchi1, fchi2, flargest1, flargest2
      real*4 etime, tstart(2), tstop(2)

c     First of all, we will say that the program has started and what
c     parameters we are using.
      print *, '# Koelle Alaaf!'
      print *, '# Size of system: ', L, ' ** ', IDIM
      print *, '# Using Ziff PRNG'
      print *, '# Boundaries: open'
      print *, '# Initial random seed: ', ISEED
      print *, '# Probability: ', P
      print *, '# Number of independent runs: ', NREPEAT

      dummy = etime(tstart)

      call ks_warmup()

      IP=(2.0d0*P-1.0d0)*2147483648.0d0
      rcplog=1.0d0/dlog(2.0d0)
      do i = 0, MAXBIN
        fbin1(i) = 0.0d0
        fbin2(i) = 0.0d0
      end do
      fnc1 = 0.0d0
      fnc2 = 0.0d0
      fchi1 = 0.0d0
      fchi2 = 0.0d0
      flargest1 = 0.0d0
      flargest2 = 0.0d0

      do irepeat = 1, NREPEAT
      conn=.true.
      nrecyc=0
      numsit=0
      numroo=0
      maxclu=0
      chi=0.0d0
      left =.false.
      top  =.false.
      three=.false.
      four =.false.
      five =.false.
      six  =.false.
      seven=.false.

      do i = 0, MAXBIN
        ns(i) = 0
      end do

c     In the beginning, all labels are (re-)usable.
      do 10 i = 1, MAX
   10   label(i) = 0
      nrmin = 2

!     Open b.c . instead of busbar
c     To determine connectivity, we use the easiest method: The zeroth
c     plane is set to be completely the cluster with number 1, which
c     means that if in the end there appears a cluster 1 in the last plane,
c     we have connectivity and cluster 1 is the infinite cluster.
c     Of course, this influences statistics, but it makes our life easier.
c     But this also means, that label 1 must never be recycled. We achieve
c     this by keeping nrmin always larger than one.
      do 11 i=1, LPLANE
   11   plane(i)=MAX
      !label(1)=-1

      i   = 1
      im1 = L**(IDIM-1)
      if(IDIM.eq.2) goto 15
      im2 = im1-L**(IDIM-2)
      if(IDIM.eq.3) goto 15
      im3 = im2-L**(IDIM-3)
      if(IDIM.eq.4) goto 15
      im4 = im3-L**(IDIM-4)
      if(IDIM.eq.5) goto 15
      im5 = im4-L**(IDIM-5)
      if(IDIM.eq.6) goto 15
      im6 = im5-L**(IDIM-6)
   15 continue

      do 20 istep=1, NSYS
        idx = iand(idx + 1, 16383)
        ks(idx) = ieor(ieor(ks(iand(idx-471, 16383)),
     *                     ks(iand(idx-1586, 16383)) ),
     *                ieor(ks(iand(idx-6988, 16383)),
     *                     ks(iand(idx-9689, 16383))))
        if(ks(idx).lt.IP) then
c         The site in investigation is occupied.
          top  =(plane(i  ).lt.MAX)
          left =(plane(im1).lt.MAX)
          if(IDIM.eq.2) goto 17
          three=(plane(im2).lt.MAX)
          if(IDIM.eq.3) goto 17
          four =(plane(im3).lt.MAX)
          if(IDIM.eq.4) goto 17
          five =(plane(im4).lt.MAX)
          if(IDIM.eq.5) goto 17
          six  =(plane(im5).lt.MAX)
          if(IDIM.eq.6) goto 17
          seven=(plane(im6).lt.MAX)
   17     continue
          if(.not.(left.or.top.or.three.or.four
     *             .or.five.or.six.or.seven)) then
c           Alle neighbours are free, so a new cluster starts.
c           Fist we have to find a new free label.
   30       nrlab = nrmin
            nrmin = nrmin + 1
            if(nrmin.eq.MAX) stop 1
            if(label(nrlab).ne.0) goto 30
            plane(i)=nrlab
            label(nrlab)=-1
          else
c           At least one of the neighbours is occupied, which means work.
c           First we determine the rootlabels for all neighbours.
            nleft =MAX
            ntop  =MAX
            nthree=MAX
            nfour =MAX
            nfive =MAX
            nsix  =MAX
            nseven=MAX
c           The label of the left neihgbour (if occupied) is certainly a
c           rootlabel.
            if(left) then
              nleft=plane(im1)
            endif

c           With the other heighbours, this need not be the case.
            if(top) then
              ntop =plane(i)
              if(label(ntop).gt.0) then
   51           ntop=label(ntop)
                if(label(ntop).gt.0) goto 51
                label(plane(i))=ntop
              endif
            endif
            if(IDIM.eq.2) goto 31
            if(three) then
              nthree=plane(im2)
              if(label(nthree).gt.0) then
   52           nthree=label(nthree)
                if(label(nthree).gt.0) goto 52
                label(plane(im2))=nthree
              endif
            endif
            if(IDIM.eq.3) goto 31
            if(four) then
              nfour=plane(im3)
              if(label(nfour).gt.0) then
   53           nfour=label(nfour)
                if(label(nfour).gt.0) goto 53
                label(plane(im3))=nfour
              endif
            endif
            if(IDIM.eq.4) goto 31
          if(five) then
            nfive=plane(im4)
            if(label(nfive).gt.0) then
 54           nfive=label(nfive)
              if(label(nfive).gt.0) goto 54
              label(plane(im4))=nfive
            endif
          endif
          if(IDIM.eq.5) goto 31
          if(six) then
            nsix=plane(im5)
            if(label(nsix).gt.0) then
 55           nsix=label(nsix)
              if(label(nsix).gt.0) goto 55
              label(plane(im5))=nsix
            endif
          endif
          if(IDIM.eq.6) goto 31
          if(seven) then
            nseven=plane(im6)
            if(label(nseven).gt.0) then
 56           nseven=label(nseven)
              if(label(nseven).gt.0) goto 56
              label(plane(im6))=nseven
            endif
          endif

 31       continue

c         Now find the smallest rootlabel.
          new=nleft
          if(ntop.lt.new)   new=ntop
          if(IDIM.eq.2) goto 33
          if(nthree.lt.new) new=nthree
          if(IDIM.eq.3) goto 33
          if(nfour.lt.new)  new=nfour
          if(IDIM.eq.4) goto 33
          if(nfive.lt.new)  new=nfive
          if(IDIM.eq.5) goto 33
          if(nsix.lt.new)   new=nsix
          if(IDIM.eq.6) goto 33
          if(nseven.lt.new) new=nseven
 33       continue

c         We count the sites within the cluster positive in ici.
          ici=1
          if(left ) then
            ici=ici-label(nleft)
            if(nleft.ne.new) label(nleft)=new
          endif
          if(top  ) then
            if(ntop.ne.nleft) ici=ici-label(ntop)
            if(ntop.ne.new) label(ntop)=new
          endif
          if(IDIM.eq.2) goto 32
          if(three) then
            if(nthree.ne.nleft.and.nthree.ne.ntop)
     *        ici=ici-label(nthree)
```

```
              if(nthree.ne.new) label(nthree)=new
            endif
            if(IDIM.eq.3) goto 32
            if(four) then
              if(nfour.ne.nleft.and.nfour.ne.ntop.and.
     *           nfour.ne.nthree) ici=ici-label(nfour)
              if(nfour.ne.new) label(nfour)=new
            endif
            if(IDIM.eq.4) goto 32
            if(five) then
              if(nfive.ne.nleft.and.nfive.ne.ntop.and.
     *           nfive.ne.nthree.and.nfive.ne.nfour)
     *           ici=ici-label(nfive)
              if(nfive.ne.new) label(nfive)=new
            endif
            if(IDIM.eq.5) goto 32
            if(six) then
              if(nsix.ne.nleft.and.nsix.ne.ntop.and.
     *           nsix.ne.nthree.and.nsix.ne.nfour.and.
     *           nsix.ne.nfive) ici=ici-label(nsix)
              if(nsix.ne.new) label(nsix)=new
            endif
            if(IDIM.eq.6) goto 32
            if(seven) then
              if(nseven.ne.nleft.and.nseven.ne.ntop.and.
     *           nseven.ne.nthree.and.nseven.ne.nfour.and.
     *           nseven.ne.nfive.and.nseven.ne.nsix)
     *           ici=ici-label(nseven)
              if(nseven.ne.new) label(nseven)=new
            endif

   32       label(new)=-ici
            plane(i)=new
          endif
        else
c         The site in investigation is not occupied.
          plane(i)=MAX
        endif

c       Now comes the Garbage Collector.
        if(nrmin.ge.LIMIT) then
c         Do recycling. First we reclassify all sites in plane(), which means
c         that we give them their rootlabels.
          nrecyc=nrecyc+1
          do 400 j = 1, LPLANE
            nrlab = plane(j)
            if(label(nrlab).gt.0) then
  401         nrlab = label(nrlab)
              if(label(nrlab).gt.0) goto 401
              label(plane(j)) = nrlab
              plane(j) = nrlab
            endif
  400       continue
c         Now we can delete safely all non-root-labels.
          do 402 j = 1, MAX
  402       if(label(j).gt.0) label(j) = 0
c         Now we have to find out, which rootlabels are still in use.
c         We make label(plane(j)) positive, so that alle non-marked
```

```
c        rootlabels are negative thereafter.
         do 403 j = 1, LPLANE
           nrlab = plane(j)
           if(label(nrlab).lt.0) label(nrlab) = -label(nrlab)
  403      continue
c        Now we can throw away all non-marked labels. Of course, we
c        have to put them into the bins.
         do 404 j = 1, MAX
           li = label(j)
           if(li.lt.0) then
             maxclu=min0(maxclu,li)
             numroo=numroo+1
             numsit=numsit+li
             fli=-li
             ibin=dlog(fli)*rcplog+0.00001d0
             if(ibin.le.MAXBIN) ns(ibin)=ns(ibin)+1
             chi=chi+fli*fli
             label(j) = 0
           endif
  404      label(j) = -label(j)
c        Now everything should be fine. If there is no infinite cluster,
c        label 1 is now zero, but it must not be reused.
         nrmin=2
       endif
c      End of Garbage Collector.

       i=i+1
       if(i.gt.LPLANE) i=1
       im1=im1+1
       if(im1.gt.LPLANE) im1=1
       if(IDIM.eq.2) goto 20
       im2=im2+1
       if(im2.gt.LPLANE) im2=1
       if(IDIM.eq.3) goto 20
       im3=im3+1
       if(im3.gt.LPLANE) im3=1
       if(IDIM.eq.4) goto 20
       im4=im4+1
       if(im4.gt.LPLANE) im4=1
       if(IDIM.eq.5) goto 20
       im5=im5+1
       if(im5.gt.LPLANE) im5=1
       if(IDIM.eq.6) goto 20
       im6=im6+1
       if(im6.gt.LPLANE) im6=1
   20  continue

c     We have examined the whole system now, so we can output all
c     interesting data.

c     To find out if there is connectivity, we have to walk through
c     the bottom plane and look for references to label 1.
      conn = .false.
      do 110 j = 1, LPLANE
       nrlab = plane(j)
       if(label(nrlab).gt.0) then
  111    nrlab = label(nrlab)
         if(label(nrlab).gt.0) goto 111
```

```
          endif
          if(nrlab.eq.1) conn = .true.
  110   continue

c       Statistical account of labels.
        nrlab = 0
        do 100 i=1, MAX
          li=label(i)
          if(li.ne.0) nrlab = nrlab + 1
          if(li.ge.0) goto 100
          maxclu=min0(maxclu,li)
          numroo=numroo+1
          numsit=numsit+li
          fli=-li
          ibin=dlog(fli)*rcplog+0.00001d0
          if(ibin.le.MAXBIN) ns(ibin)=ns(ibin)+1
          chi=chi+fli*fli
  100   continue
        fmax=-maxclu
        chi=(chi-fmax*fmax)/NSYS

c       Sum up ns(i), so that ns(i) = sum_{s'>s} ns'
        sum = 0
        do i = MAXBIN, 0, -1
          sum = sum + ns(i)
          ns(i) = sum
        end do

c       Now we add values to average and error values.
        do i = 0, MAXBIN
          ftemp = 1.0d0*ns(i)/NSYS
          fbin1(i) = fbin1(i) + ftemp/NREPEAT
          fbin2(i) = fbin2(i) + ftemp*ftemp/NREPEAT
        end do
        !ftemp = 1.0d0*/NSYS
        !fnc1 = fnc1 + ftemp/NREPEAT
        !fnc2 = fnc2 + ftemp*ftemp/NREPEAT
        fchi1 = fchi1 + chi/NREPEAT
        fchi2 = fchi2 + chi*chi/NREPEAT
        ftemp = -1.0d0*maxclu/NSYS
        flargest1 = flargest1 + ftemp/NREPEAT
        flargest2 = flargest2 + ftemp*ftemp/NREPEAT

        end do !irepeat

        dummy = etime(tstop)

c       Do the output.
        print *, '# Required runtime: ', (tstop(1) - tstart(1)), 'seconds'
        print *, '# Number density: ', fbin1(0), ' +- ' ,
     *  sqrt((fbin2(0)-fbin1(0)*fbin1(0))/(NREPEAT-1))
        print *, '# Size of largest cluster: ', flargest1, ' +- ',
     *  sqrt((flargest2-flargest1*flargest1)/(NREPEAT-1))
        print *, '# Second moment: ', fchi1, ' +- ',
     *  sqrt((fchi2-fchi1*fchi1)/(NREPEAT-1))
        print *, ''

        do i=0, MAXBIN
```

```
          if((fbin1(i).ne.0.0d0).and.(fbin2(i).gt.(fbin1(i)*fbin1(i))))
     *    print *, 2**i, fbin1(i),
     *    sqrt((fbin2(i)-fbin1(i)*fbin1(i))/(NREPEAT-1))
        end do

        contains

        subroutine ks_warmup()
          integer i, ii, ibm
          integer ici
          ibm = 2 * ISEED - 1
          do i = 0, 16383
            ici = 0
            do ii = 1, 32
              ici = ishft(ici, 1)
              ibm = ibm * 16807
              if(ibm .lt. 0) ici = ior(ici, 1)
            end do
            ks(i) = ici
          end do
          idx = 0
          do i = 1, 8*16384
            idx = iand(idx + 1, 16383)
            ks(idx) = ieor(ieor(ks(iand(idx-471, 16383)),
     *                          ks(iand(idx-1586, 16383)) ),
     *                     ieor(ks(iand(idx-6988, 16383)),
     *                          ks(iand(idx-9689, 16383))))
          end do
        end subroutine

        end
```

Appendix D

Details of pseudo-random number generators

When doing Monte Carlo simulations, we need random numbers, but not "really" random ones. When we change small details in our programs and want to check if we have introduced errors in the code, we want to be able to reproduce a simulation *exactly*; in that case, we need exactly the same sequence of random numbers. To achieve this, we do not use real random numbers, but pseudo-random numbers. An overview over generating pseudo-random numbers in general can be found in [21].

D.1 Linear congruential generators

The simplest method of producing a sequence of pseudo-random numbers is a rule $x_n = M \cdot x_{n-1} \bmod c$. For implementation on computers, we use a c of 2^{31} or 2^{63}, in this case x_n is just a 32-bit or 64-bit signed integer, and the integer multiplication itself cuts off the leading bits. Choosing the right multiplier M is essential: Well known values are 65539, $16807 = 7^5$, or 13^{13} for 64-bit integers only. Of course, M must be odd, otherwise we would receive only zeros for x_n after a short time.

These generators are known to be problematic (cf. [48, part II, chapter 1]), and in this diploma thesis, they showed wrong behaviour, too (cf. section 3.5). But they are easy to implement and fast.

D.2 Lagged Fibonacci generators

When we use two or more pseudo-random numbers and combine them to a new one, it should be random, too. This is the principle of LFGs. We do not combine the last two numbers to form the next one, because this would mean to introduce strong correlations; instead, we use large taps between the numbers that we combine. There are several ways of combining the numbers, i. e. adding or multiplying, but the standard method is to use the bitwise *exclusive-or* operation. An overview over different LFGs (also called shift-register-sequence random-number generators) can be found in [58].

We can produce large numbers of different LFGs by choosing different amounts of the numbers that we combine and by different taps within the sequence for the numbers. A well-known standard LFG is the one named after KIRKPATRCIK and STOLL (cf. [30]), despite the fact that mathematicians prefer to call it after TAUSWORTHE (cf. [52]). It combines two numbers and chooses them with taps 103 and 250 ($x_n = x_{n-103} \oplus x_{n-250}$), which accounts for the third name: `R(103,250)`.

This generator is known to have weaknesses due to its three-point correlations, but for the simulations done for this diploma thesis, such problems did not occur.

Generators with higher quality can be obtained using more and/or larger taps. Two of them were used within this diploma thesis: Ziff's four-tap `R(471,1586,6988, 9689)` and Ziff's six-tap `R(18,36,37,71,89,124)`. Both are slower than Kirkpatrick-Stoll, and their better quality did not show up significantly in the simulations carried out here (for other applications, this can differ drastically; cf. [58] for a list of such applications).

One problem still remains: in order to use a LFG which largest tap is n, we first have to produce n random numbers through other means, before we can use the LFG-rule. We can use a LCG to determine the initial values bit by bit, but then we have to do a relaxation on these random numbers: we produce some thousand of them by the LFG-rule without using them, only after this warm up we start using the random numbers.

Appendix E

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und alle benutzten Quellen und Hilfsmittel vollständig angegeben habe.

Köln, den 10. Juni 2001