Numerical methods for the determination of the properties and critical behaviour of percolation and the Ising model

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

vorgelegt von
Daniel Tiggemann
aus Bad Pyrmont

Köln 2007

Berichterstatter: Prof. Dr. D. Stauffer
Prof. Dr. E. Speckenmeyer

Tag der mündlichen Prüfung: 29. November 2006

Contents

Abstract **5**

Zusammenfassung **7**

1 Introduction **9**

1.1 Goal of this thesis . . . 9
1.2 What is percolation? . . . 10
1.3 Phenomena of percolation . . . 11
1.4 Computational approaches—algorithms . . . 14
1.4.1 Exact enumeration . . . 14
1.4.2 Justification for Monte Carlo algorithms . . . 14
1.4.3 Leath algorithm . . . 15
1.4.4 Newman-Ziff algorithm . . . 15
1.4.5 Hoshen-Kopelman algorithm . . . 16

2 Fluctuations of Cluster Numbers **19**

2.1 Introduction . . . 19
2.2 Statistical measures . . . 19
2.3 Simulations . . . 20
2.3.1 Distribution of Cluster Numbers . . . 20
2.3.2 Variance of cluster numbers . . . 21
2.3.3 Skewness and kurtosis of distributions . . . 21
2.4 Summary and outlook . . . 23

3 Parallelizing the Hoshen-Kopelman algorithm using domain decomposition **25**

3.1 Devising a parallelized version of the Hoshen-Kopelman algorithm . 25
3.1.1 Domain decomposition . . . 26
3.1.2 Clusters extending over several sub-domains . . . 26
3.1.3 Recycling of redundant labels . . . 29
3.1.4 Counting of clusters . . . 30
3.1.5 Fully periodic boundary conditions . . . 30
3.1.6 Step-by-step description of the algorithm . . . 31
3.2 Other ways of parallelizing Hoshen-Kopelman . . . 32
3.3 Results of Monte Carlo simulations . . . 32
3.3.1 Cluster size distribution . . . 33
3.3.2 Corrections to scaling . . . 36
3.3.3 Influence of boundary conditions on finite-size effects . . . 38
3.3.4 Number density . . . 40
3.3.5 Quality of pseudo-random number generators . . . 40
3.3.6 Speed of simulations and parallel efficiency . . . 41
3.4 Summary and outlook . . . 43

4 Growing Lattices **47**
4.1 Motivation 47
4.2 Computational method 48
4.2.1 A modified Hoshen-Kopelman algorithm 48
4.2.2 Fully periodic boundary conditions 49
4.2.3 Recycling of labels 50
4.3 Results for three dimensions 51
4.3.1 Cluster size distribution 51
4.3.2 Number density 52
4.3.3 Number of spanning clusters 52
4.3.4 Size of largest cluster 54
4.4 Results for two dimensions 59
4.5 Speed of simulations 60
4.6 Summary and outlook 61

5 Critical Behaviour of the Ising Model **63**
5.1 Rationale 63
5.2 Introduction 63
5.3 Computational method 64
5.3.1 Parallelization 66
5.3.2 Multi-spin coding 67
5.4 Simulations 69
5.5 Results 70
5.6 Speed of simulation and parallel efficiency 72
5.7 Summary and outlook 72

6 Summary and Outlook **75**
6.1 Summary 75
6.2 Outlook 76

A Acknowledgements **81**

B Bibliography **83**

C Typical errors in Monte Carlo data **89**

D Pseudo-random number generators **91**
D.1 Linear congruential generators 91
D.2 Lagged Fibonacci generators 91
D.3 Hashing generators 92
D.4 Speed of different random number generators 93

E Amdahl's law and measuring parallel efficency on real-life computers **95**

F Code of programs **97**
F.1 General remarks 97
F.2 Parallelized Hoshen-Kopelman 97
F.3 Percolation on growing lattices 116
F.4 Ising model 126

G Erklärung **131**

Abstract

For this thesis, numerical methods have been developed, based on Monte Carlo methods, which allow for investigating percolation and the Ising model with high precision. Emphasis is on methods to use modern parallel computers with high efficiency. Two basic approaches for parallelization were chosen: replication and domain decomposition, in conjunction with suitable algorithms.

For percolation, the Hoshen-Kopelman algorithm for cluster counting was adapted to different needs. For studying fluctuations of cluster numbers, its traditional version (i. e. which is already published in literature) was used with simple replication. For simulating huge lattices, the Hoshen-Kopelman algorithm was adapted to domain decomposition, by dividing the hyperplane of investigation into strips that were assigned to different processors. By using this way of domain decomposition, it is viable to simulate huge lattices (with world record sizes) even for dimensions $d > 2$ on massively-parallel computers with distributed memory and message passing. For studying properties of percolation in dependence of system size, the Hoshen-Kopelman algorithm was modified to work on changing domains, i. e. growing lattices. By using this method, it is possible to simulate a lattice of linear size $L_{\max}$ and investigate lattices of size $L_i < L_{\max}$ for free. Here again, replication is a viable parallelization strategy.

For the Ising model, the standard Monte Carlo method of importance sampling with Glauber kinetics and multi-spin coding is adapted to parallel computers by domain decomposition of the lattice into strips. Using this parallelization method, it is possible to use massively-parallel computers with distributed memory and message passing in order to study huge lattices (again world record sizes) over many Monte Carlo steps, in order to investigate the dynamical critical behaviour in two dimensions.

The following table summarizes the most important numerical results found in this thesis for percolation at the critical threshold. d is the dimensionality, p_c is the probability chosen for the critical threshold, taken from literature; L is the size of the largest lattice that has been simulated with periodic boundary conditions for this thesis; d, p_c, and L are parameters. The results are τ, the Fisher exponent for the cluster size distribution, known exactly in two dimensions, Δ_1, the exponent for the corrections to scaling for small clusters, n_c, the number density (number of clusters per lattice site), n_{sp}, the number of spanning clusters, and D, the fractal dimension of the largest cluster (also known exactly for $d = 2$).

d	p_c	L	τ	Δ_1	n_c	n_{sp}	D
2	0.5927464	7000000	187/91	0.73(2)	0.027597857(2)	0.52(1)	91/48
3	0.311608	25024	2.190(1)	0.65(5)	0.05243812(9)	0.15(3)	2.52(1)
4	0.196889	1305	2.315(2)	0.48(8)	0.0519995(2)		
5	0.1407966	225	2.41(1)	0.30(10)	0.0460321(2)		

Simulations for $d = 3, 4, 5$ are world records in simulated system size. Results for $d = 2$ are for the largest published simulation. By simulating huge lattices, very precise values for τ, Δ_1, and n_c could be found.

By investigating growing lattices for $d = 2$ and $d = 3$, the fractal dimension D of the largest cluster (which is the spanning cluster, if it forms) could be examined with high precision. Furthermore, the number of spanning clusters n_{sp} in dependence on p and L has been investigated. At the critical threshold, n_{sp} reaches a constant value for large enough L for $d = 2$ and $d = 3$. Slightly above or below p_{c}, n_{sp} shows a delicate finite-size behaviour.

For fluctuations in percolation, the distribution of cluster numbers n_s for fixed size s is Gaussian for small s and large L. The position of the maximum (i. e. $\langle n_s \rangle$) is in compliance to the power law $n_s \propto s^{-\tau}$. Variance of cluster numbers shows the same behaviour as the mean cluster numbers, with deviations for small s, i. e. $(\langle n_s^2 \rangle - \langle n_s \rangle^2)/\langle n_s \rangle = 1 + k_2 s^{-\Delta_2}$, with $k_2 = 0.25(5)$ and $\Delta_2 = 1.2(2)$.

For the Ising model, simulations of huge lattices (up to $L = 2 \cdot 10^6$) for many Monte Carlo steps showed that the dynamical critical exponent in two dimensions is *not* $z = 2$ with logarithmic corrections, but with high probability $z > 2$ with simple power-law behaviour, with the current best estimate $z = 2.167(3)$.

This thesis has shown, like many other works in the recent years, that supercomputing is a valuable tool for physics, giving rise to the branch of computational physics, sometimes considered to be the third branch after experimental and theoretical physics. Due to ever increasing computer power, problems can now be investigated with numerical methods, which seemed to be completely inaccessible only one or few decades ago. Although purely analytical solutions are preferable to numerical work, it is now possible to decide questions that cannot be answered by paper and pencil alone. The numerical methods presented in this thesis allow for investigating percolation and the Ising model with high precision, utilizing modern parallel computers.

Zusammenfassung

Im Rahmen dieser Arbeit wurden auf der Basis von Monte-Carlo-Verfahren numerische Methoden entwickelt, die es erlauben, Perkolation und das Ising-Modell mit hoher Präzision zu untersuchen. Dabei wurde darauf geachtet, dass diese Methoden moderne Parallelrechner effizient nutzen. Zwei verschiedene Ansätze zur Parallelisierung wurden gewählt: Replikation und Gebietszerlegung, in Verbindung mit geeigneten bzw. modifizierten Algorithmen.

Für Perkolation wurde der Hoshen-Kopelman-Algorithmus, der zum Cluster-Zählen dient, modifiziert. Um Fluktuationen von Cluster-Zahlen zu untersuchen, wurde die traditionelle Version des Algorithmus (die in der Fachliteratur veröffentlicht ist) zusammen mit einfacher Replikation benutzt. Um große Gitter zu simulieren, wurde der Hoshen-Kopelman-Algorithmus angepasst auf Gebietszerlegung, wobei die akutelle zu untersuchende Hyperebene in Streifen zerteilt wird, die verschiedenen Prozessoren zugeordnet werden. Durch diese Art der Gebietszerlegung ist es möglich, sehr große Gitter (mit Weltrekordgrößen) auf massiv-parallelen Rechnern mit verteiltem Speicher und Message Passing zu simulieren, selbst für Dimensionen $d > 2$. Um Eigenschaften von Perkolation in Abhängigkeit der simulierten Systemgröße zu untersuchen, wurde der Hoshen-Kopelman-Algorithmus angepasst auf sich ändernde Gebiete, d. h. wachsende Gitter. Durch diese Methode ist es möglich, Gitter der linearen Größe $L_{\max}$ zu untersuchen und Resultate für beliebige Gittergrößen $L_i < L_{\max}$ ohne weiteren Rechenaufwand zu erhalten. Hierbei ist Replikation wieder ein erfolgreiches Parallelisierungsverfahren.

Für das Ising-Modell wurde die Standard-Monte-Carlo-Methode des Importance Sampling mit Glauber-Kinetik und Multi-Spin Coding angepasst auf Parallelrechner mit Hilfe von Gebietszerlegung des zu simulierenden Gitters in Streifen. Mit dieser Parallelisierungsmethode ist es möglich, massiv-parallele Rechner mit verteiltem Speicher und Message Passing nutzen, um große Gitter (wiederum Weltrekordgrößen) über viele Monte-Carlo-Zeitschritte zu simulieren. Damit konnte das dynamisch-kritische Verhalten des Ising-Modells in zwei Dimensionen untersucht werden.

Die folgende Tabelle führt die wichtigsten numerischen Resultate auf, die im Rahmen dieser Arbeit für Perkolation am kritischen Punkt gefunden wurden. d ist die Dimensionalität, p_{c} der Wert, der als kritische Wahrscheinlichkeit gewählt wurde (der Literatur entnommen); L ist die lineare Größe des größten für diese Arbeit mit periodischen Randbedingungen simulierten Gitters; d, p_{c} und L sind Parameter. Die Resultate sind τ, der Fisher-Exponent der Clustergrößenverteilung (der exakte Wert ist für $d = 2$ bekannt); Δ_1 ist der Exponent für die Korrekturen zum Skalenverhalten (corrections to scaling) für kleine Cluster, n_{c} ist die Clusterzahlendichte (number density; die Zahl der Cluster geteilt durch die Zahl der Gitterplätze), n_{sp} die Zahl der systemdurchspannenden Cluster (spanning clusters), und D ist die fraktale Dimension des größten Clusters.

d	p_{c}	L	τ	Δ_1	n_{c}	n_{sp}	D
2	0.5927464	7000000	187/91	0.73(2)	0.027597857(2)	0.52(1)	91/48
3	0.311608	25024	2.190(1)	0.65(5)	0.05243812(9)	0.15(3)	2.52(1)
4	0.196889	1305	2.315(2)	0.48(8)	0.0519995(2)		
5	0.1407966	225	2.41(1)	0.30(10)	0.0460321(2)		

Die Simulationen für $d = 3, 4, 5$ sind Weltrekorde in der Größe des simulierten Systems. Die Resultate für $d = 2$ sind für die größte publizierte Simulation. Durch die Simulation sehr großer Gitter konnten sehr genaue Werte für τ, Δ_1 und n_{c} gefunden werden.

Mit der Methode, Perkolation auf wachsenden Gittern zu simulieren, konnte die fraktale Dimension D des größten Clusters (welcher auch der systemdurchspannende Cluster ist, wenn er sich bildet) mit hoher Präzision untersucht werden. Ferner wurde auch die Zahl der systemdurchspannenden Cluster n_{sp} untersucht, in Abhängigkeit von p und L. Am kritischen Punkt nimmt n_{sp} einen konstanten Wert an für hinreichend große L, für $d = 2$ und $d = 3$. Nahe oberhalb oder unterhalb p_{c} zeigt n_{sp} ein komplexes Finite-Größe-Verhalten (finite size behaviour).

Für Fluktuationen bei Perkolation nimmt die Verteilung der Cluster-Zahlen n_s für ein festes s eine Gauß-Verteilung an für kleine s und große L. Die Position des Maximums (d. h. $\langle n_s \rangle$) folgt dem Potenzgesetz $n_s \propto s^{-\tau}$. Die Varianzen der Cluster-Zahlen zeigt das gleiche Verhalten wie die Mittelwerte, mit kleinen Abweichungen für kleine s, d. h. $(\langle n_s^2 \rangle - \langle n_s \rangle^2)/\langle n_s \rangle = 1 + k_2 s^{-\Delta_2}$, mit $k_2 = 0.25(5)$ und $\Delta_2 = 1.2(2)$.

Für das Ising-Modell haben Simulationen von großen Gittern (bis zu $L = 2 \cdot 10^6$) über viele Monte-Carlo-Zeitschritte gezeigt, dass der dynamisch-kritische Exponent in zwei Dimensionen *nicht* $z = 2$ mit logarithmischen Korrekturen ist, sondern mit großer Wahrscheinlichkeit $z > 2$ mit einfachem Potenzgesetzverhalten, wobei die beste Abschätzung $z = 2.167(3)$ ist.

Diese Arbeit zeigt, wie schon viele Arbeiten der letzten Jahre, dass Supercomputing ein wertvolles Werkzeug der Physik ist, und dass der Zweig der Computerphysik (computational physics) sich als dritter Zweig der Physik nach Experimental- und Theoretischer Physik heruszukristallisieren beginnt. Durch die ständig wachsende Rechenleistung können nun physikalische Probleme mit numerischen Methoden untersucht werden, die noch vor einem oder wenigen Jahrzehnten völlig unzugänglich erschienen. Obwohl rein analytische Lösungen wünschenswerter sind als numerische Arbeiten, können nun Fragen entschieden werden, die nicht mit Bleistift und Papier allein beantwortet werden können. Die numerischen Methoden, die in dieser Arbeit präsentiert werden, erlauben es, Perkolation und das Ising-Modell mit hoher Präzision zu untersuchen, unter Verwendung moderner Parallelrechner.

Chapter 1

Introduction

1.1 Goal of this thesis

Goal of this thesis is to develop numerical methods, i. e. algorithms, for investigating simple models of statistical physics, namely percolation and the Ising model, while emphasis is placed on percolation. These models are simple in comparison to more elaborate models, which are investigated heavily in physics, but their simplicity nonewithstanding, they offer a wealth of phenomena comparable to that encountered in other models. The development of numerical methods for percolation and the Ising model is driven by two motivations: the methods can be used (either directly or analogously) for other models, too, and secondly, insight won into simple models may be applicable also to complex models. For example, percolation is a model of disordered media, which become ever more important.

Another goal is to show that numerical methods play an important role in modern theoretical physics, and allow to get new insight into problems that elude purely analytical investigation. This growing importance is due to ever increasing computer power and the invention of ever more advanced algorithms. The combination of both make numerical investigations possible that were deemed impossible only decades ago.

Statistical physics investigates the complicated behaviour of simple constituents, i. e. the emergence of collective behaviour of a large number of constituents, which differs qualitatively from the behaviour of the simple constituents.

One remarkable fact is that this collective behaviour is separable from the individual behaviour, and sometimes the same collective behaviour can be seen in systems that differ in microscopic details. This gives rise to the idea of universality: Some quantities that can be observed behave in a qualitatively same way. For example, right at a phase transition, some observables diverge with power-laws, giving rise to critical exponents. When critical exponents of two systems are the same, they belong to the same universality class.

Using numerical methods, it is possible to obtain very precise estimates for quantities that are of interest in many models. Using these estimates makes it possible to test theories that were developed using analytical methods. These tests are very important for advancing theories; sometimes the lack of precise estimates can discourage progress, while precise values can encourage it. In this way, computer simulations are comparable to experiments; but while simulations can never be regarded as true nature (and thus never make experiments obsolete; ultimately, understanding nature is the goal of all physics), by using simulations it is possible to investigate models, parameter regions, or characteristics of systems, which are not accessible to any conceivable real experiment.

In this thesis, three different "modifications" of the traditional Hoshen-Kopelman algorithm will be presented, along with results for these: simple replication in order to study fluctuations (not really a modification, rather a different mode of usage), parallelization using domain decomposition in order to study huge lattices, and growing lattices (i. e. changing domains) in order to study the dependence of percolation observables on system size L (and in this respect, also to study finite size effects). For the Ising model, an efficient parallelization, again using domain decomposition, will be presented, in order to settle a long dispute over the dynamical critical exponent z.

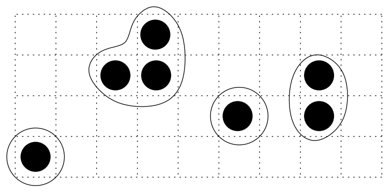
A unified theme for this thesis can be formulated like this: Use powerful computers and adapted algorithms to investigate simple and standard models of statistical physics.

1.2 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 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; the sites that are connected in this way are called wetted sites. Within this thesis, only site percolation will be investigated; bond percolation differs only gradually, not principally. A good introduction to percolation theory can be found in [StAh94], a short overview in [Bund91], [SAHM99]. Some remarks on the history of percolation theory can be found in [Grim00], [Hamm83].

Research in percolation started 1941 with Paul Flory investigating the gelation of polymers [Flor41], although the term *percolation* was first coined by Broadbent and Hammersley in 1957 [BrHa57], [Hamm57a], [Hamm57b]; Broadbent was investigating gas masks, which sheds a first light on the diversity of applications for percolation theory (for more on applications, cf. [Sahi94]).

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" [HaHa64, 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 [Lea76a], [Lea76b] 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 [HoKo76] 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 has been possible to get results of high precision. This thesis will present computational investigations of percolation and the Ising model, combining powerful computers and new algorithmic approaches.

1.3 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. [SAHM99]: "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. Clusters can have a fractal structure (cf. fig. 1.1 for an example), many properties are governed by power-laws. Even more, the concept of renormalization can be easily demonstrated on percolation.

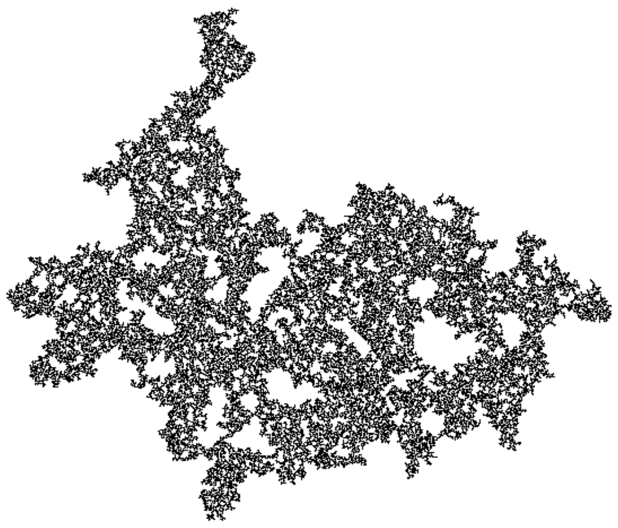


Figure 1.1: A single cluster, simulated right at the critical threshold p_c. Its fractal structure is visible, notably it contains holes and has a ragged perimeter.

When there is a cluster that goes from top to bottom, we call this cluster "spanning" or "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, below p_c we speak of the sub-critical, above of the super-critical phase.

Lots of effort has been put into finding exact values of p_c, but so far only the value for the one-dimensional lattice ($p_c = 1$), and some values for two-dimensional lattices ($p_c = 1/2$ for site percolation on the triangular lattice, $p_c = 1/2$ for bond percolation on the square lattice, $p_c = 2\sin(\pi/18)$ for bond triangular, $p_c = 1 - 2\sin(\pi/18)$ for bond hexagonal, cf. [Grim99]) have been found. For other lattices, especially for all lattices with $d > 2$, only numerical approximations are known (with the exception of the so-called Bethe lattice, also known as Cayley tree, which is often described as a lattice with $d = \infty$; for $d \to \infty$, asymptotical behaviour is known, cf. [Kest90]). Much computational effort was put into finding good approximations for these p_c (for example, [NeZi00], [Gras03], [PZS01], to cite only a few), as a precise value is necessary to do simulations right at the critical threshold, were most of the interesting physics happens. One remarkable fact is that the value of p_c depends heavily on microscopic details (the exact type of the lattice, e. g. square vs. triangular in two dimensions), while other properties do not depend on these details, but only on the dimensionality (i. e. differ for two and three dimensions).

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 (cf. fig. 1.2; the idea for that figure was taken from [SAHM99]).

Due to this self-similarity, we expect a power-law for the distribution of cluster sizes right at the critical point (power-laws are self-similar under rescaling; for $P(x) = x^a$, $P(c \cdot x) = (cx)^a = c^a x^a \propto P(x)$, with c and a constants):

$$n_s(p_c) \propto s^{-\tau} \tag{1.1}$$

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. [Stau75], [StAh94])

$$n_s(p_c) = k_0 s^{-\tau} \cdot \underbrace{(1 - k_1 s^{-\Delta_1} + \ldots)}_{\text{for small } s} \cdot \underbrace{f((p - p_c)s^\sigma)}_{\text{away from } p_c}, \tag{1.2}$$

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; e. g. it is the same for a square lattice and a triangular lattice. In eq. 1.2 τ is universal, while k_0 is not. When two different physical systems share the same set of critical exponents, these belong to the same universality class. Non-universal properties may still differ, of course.

For percolation, scaling arguments relate several critical exponents with each other (cf. [Stau79]): $1/(\tau - 2) = 1 + \gamma/\beta$, 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. The Newman-Ziff algorithm (cf. section 1.4.4) allows for efficiently investigating percolation observables depending on p, but is not further investigated in this thesis.

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, $d_u = 6$ is called the upper critical dimension). Additionally, $f(z)$ is expected to be a Gaussian.

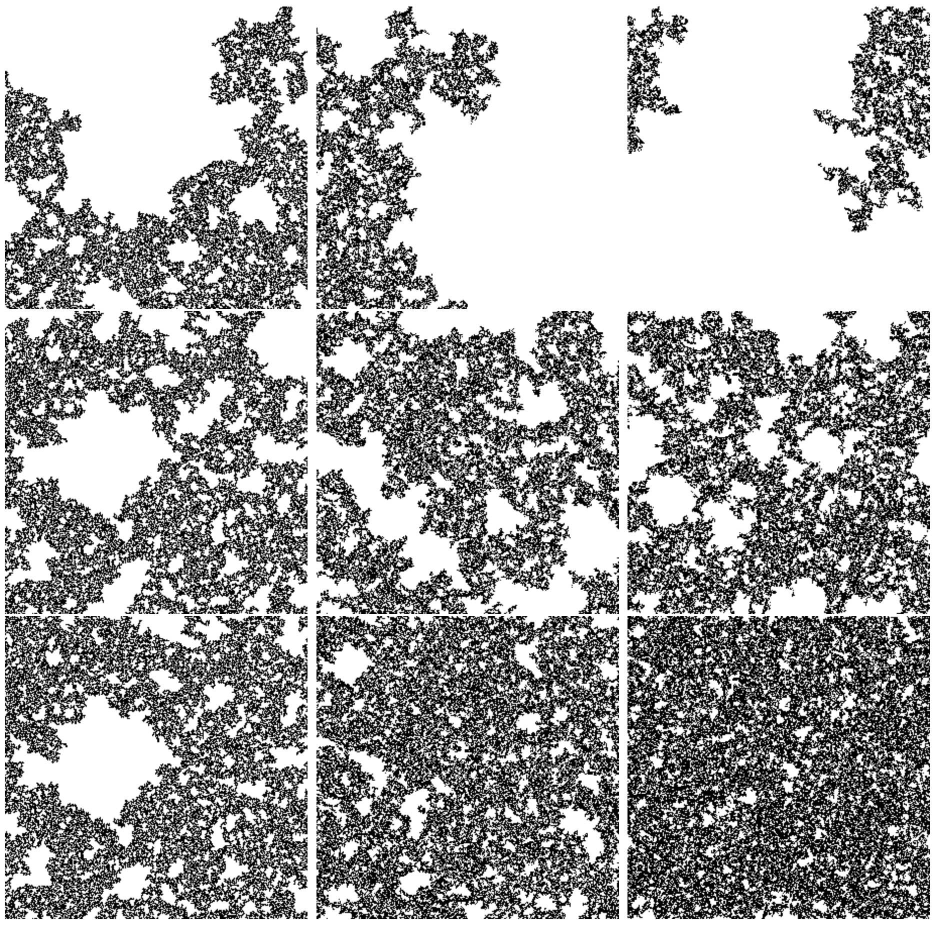


Figure 1.2: Renormalization of an infinite percolation cluster by averaging over sites, creating super-sites. Top row is for $p = 0.585 < p_\mathrm{c}$ (in this case, the largest cluster is plotted, as no infinite cluster forms), middle row for $p = 0.5927464 \simeq p_\mathrm{c}$, bottom row for $p = 0.6 > p_\mathrm{c}$. For each renormalization step, a patch of 3×3 sites is averaged to one super-site, i. e. if there are more free than occupied sites in the patch, the resulting super-site is free, and vice versa. Columns from left to right are zero, one, and two renormalizations. For $p = p_\mathrm{c}$, the system stays self similar, e. g. the holes have roughly the same size and geometric features stay the same. For $p < p_\mathrm{c}$, the holes grow, and after enough steps the cluster vanishes, as it is no longer infinite (does not fill the whole lattice). For $p > p_\mathrm{c}$, the holes shrink, the infinite cluster starts to obtain bulk properties; this effect increases for increasing p.

1.4 Computational approaches—algorithms

As only for special cases exact solutions of percolation are available, using a computer to study the problem is a reasonable approach, especially as computers nowadays offer huge memory and high computing speed. The fastest machines on earth offer now hundreds of TFlops (TFlops: 10^{12} floating point operations per second), with PFlops possible even in this decade. The Top 500 list of the fastest supercomputers of the world, published semi-annually at www.top500.org since 1993, shows exponential growth of computing power over time. There is deceleration expected for the near future. A major fuel for growth is the invention of massively parallel computers. These require adapted algorithms in order to fully exploit their power; while processes in physics have inherent parallelism, many algorithms devised for simulating these are sequential in nature (as they were written for sequential computers in the first place) and have to be adapted to parallelism.

In order to study percolation, several algorithms have been developed; some of these shall be presented in this section, and one, the Hoshen-Kopelman algorithm with some variants, will be covered in detail in the rest of this thesis.

Due to the stochastic nature of percolation, the model is well suited to be studied with Monte Carlo methods, but also exact enumeration is used. Both approaches have advantages and disadvantages: Exact enumeration gives exact results (within its domain) and is directly understandable, but only suitable to very small systems (which show qualitatively different behaviour from larger systems, as will be shown e. g. in chapter 4). Monte Carlo algorithms allow for studying very large systems (cf. chapter 3), but introduce errors into the results, which make careful analysis necessary (for more on errors in Monte Carlo simulation, cf. appendix C).

1.4.1 Exact enumeration

Although it would be nice to do a full enumeration of all possible configurations for a lattice of size $N = L^d$, this would be a Herculean task for all but the smallest L, as each site of the lattice can be occupied or free, resulting in 2^N configurations, an exponentially growing number. Even a very small 8×8-lattice would require examination of 2^{64} configurations; even the number of configurations could not be stored in the 64-bit word of modern computers. In Monte Carlo simulations not *all* configurations are realized, but only a very small subset of considerably probable configurations which are deemed representative. Therefore, exact enumeration of lattices is nowhere used.

Another way of exact enumeration is to enumerate clusters, not whole lattices. This is useful for investigating the properties of clusters. The number of possible configurations for a cluster grows exponentially, so the enumeration of clusters seems to suffer from the same disadvantages as the enumeration of lattices. But given all possible clusters of size s, it is possible to calculate the perimeter polynomial $n_s(p)$, which gives the probability that a specific site belongs to a s-cluster at occupation probability p (or in other words, the concentration of s-clusters in a lattice). Using these polynomials, it is possible to determine $p_{\max}(s)$ (where $n_s(p)$ becomes maximal), and by extrapolating to $s \to \infty$, to find a value for p_{c}, with increasing precision for perimeter polynomials for increasing s (cf. [Syke86], [Mert90], [MeLa92], [Mart90]).

1.4.2 Justification for Monte Carlo algorithms

While exact enumeration of lattices or clusters is futile for large entities, it is also not necessary. As an example, an $L^d = N$ lattice with only one occupied site, all others free, has a probability of formation of $Np(1-p)^{(N-1)}$. For intermediate p

and large enough lattices, this probability is low enough to be neglegible, meaning that we can omit enumerating these configurations. Even more, all N possible configurations are equivalent, so counting only one would be enough.

Instead of generating all possible configurations, investigating them, and then averaging the observables by statistical weight, it is more reasonable to generate configurations only with a probability proportional to their statistical weight. This is in essence what is done with Monte Carlo algorithms. With a fixed probability p, we occupy sites or leave them free $(1-p)$, and generate configurations which are probable for the chosen p. By repeating this, we obtain several representative configurations, and can then average over these.

The Leath algorithm generates clusters by randomly occupying sites, while the Hoshen-Kopelman algorithm generates lattices. The Newman-Ziff algorithm is special, as it occupies from 0 to N sites in a lattice, sweeping all possible values of p, but it does so in a random order.

In essence, all these Monte Carlo algorithms choose representative samples out of the huge number of possible configurations, by generating these samples with a random process.

1.4.3 Leath algorithm

The Leath algorithm was published in 1976 by Leath, cf. [Lea76a], [Lea76b], and is used to generate single clusters. It uses two data structures, a lattice of sites' status (which can be 'occupied', 'free', or 'unexamined') and a list of sites that need to be examined. At the start, one site in the lattice is marked as occupied, its neighbours are added to the list. Then for all sites in the list, a random number is generated and the site is marked as occupied with probability p or free with $1-p$. All neighbours of that site which are marked as unexamined, are added to the list. The process is repeated until no more sites are left on the list. After that, the lattice contains one cluster that can be investigated further. Some observables can be calculated on the fly during the cluster generation (e. g. number of sites in the cluster).

It is possible to generate lattice statistics using this method, e. g. the size distribution of clusters n_s. When using lattices large enough that no cluster reaches the borders, i. e. it grows until it stops naturally (due to the random numbers), then no finite size effects occur. By simulating many clusters and accounting them, we get a reasonable statistic for n_s, which can always be improved by just simulating more clusters.

The lattice can become quite large for large clusters. There are several tricks available in literature that cut down on this memory requirement, by storing only parts of the lattice (as the cluster does not grow homogeneously), cf. [Gras03] for an example and a literature list, or by not storing the lattice at all, cf. [PZS01].

1.4.4 Newman-Ziff algorithm

When simulating percolation, we are in many cases not only interested in results for one fixed p, but for several different p_i. The traditional method was to do several independent simulations using e. g. the Hoshen-Kopelman algorithm, each with a different p_i. When we use exactly the same sequence of pseudo-random numbers, this means that for small p some sites are left free which would be occupied for higher p. Another way to achieve this would be to fill the lattice in random order, starting with no occupied site and ending with all sites occupied, and counting clusters for each occupied site that is added.

This is the basic idea of the Newman-Ziff algorithm, which was published in 2000 by Newman and Ziff [NeZi00]. They start with an empty lattice and occupy one site after the next in a random order. This is achieved by generating a list of all lattice sites, fully permuting this list, and then walking this list. For each added site, clusters are counted and statistics stored. In order to obtain results in dependence on p, not on n (number of occupied sites in the lattice), a convolution is necessary. This is an expensive operation, but has to be done only once for each observable we are interested in. It is possible to generate many lattices and average them (using different random orders, in order to produce better statistics), only afterwards doing the convolution.

This algorithm is naturally well suited for studying observables in dependence on p. Probably the most important of these is the binary observable, if a infinite cluster forms or not. The probability where this step occurs is the critical probability p_c of the phase transition. Newman and Ziff used this to measure p_c with very high precision.

De Freitas, Lucena and Roux published a similar algorithm, but for the context of time dependent percolation [FLR99, FrLu00].

1.4.5 Hoshen-Kopelman algorithm

The Hoshen-Kopelman algorithm, published in 1976 by Hoshen and Kopelman [HoKo76], 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 an L^d lattice, we only have to store L^{d-1} sites, which yields a significant advantage for memory consumption in low dimensions.

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

●			●
●		●	●

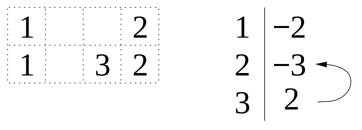
We now go through the lattice line by line, and within each line from left to right. Whenever we encounter an occupied site that is not connected to another occupied site to the left or to the top, we say that this site starts a new cluster and assign it a new number as cluster label, starting from 1. Additionally, we put this 1 (number of sites in cluster) into a label array, under the label of the newly created cluster (for technical reasons, we give it a negative sign). On the other hand, when the site under investigation has an occupied neighbour to the left or top, it inherits its cluster label from that neighbour. Another site is added for this label in the label array. Thus, after seven examined sites our lattice and the label array look like

1			2
1		3	?

1	−2
2	−1
3	−1

The eighth 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 separate 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. Of course, these root labels need to be accounted before discarding them. This method is known as Nakanishi recycling (cf. [NaSt80]).

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 [Broa54] "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 and now have nearly fully replaced the once ubiquitous vector supercomputers. They offer unrivaled performance for a rather low price (compared to traditional vector machines), 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, rather cheap compute clusters, built from commodity components, offer parallel computing power without the price premium of custom-made MPPs, and thus the use of replication is justifiable (often contemptuously called "poor man's parallelization"). This is especially important in the field of Monte Carlo simulations, as averaging over several runs for the same parameters, only using different random numbers, is important to improve statistics and reduce errors.

While parallelization of serial algorithms is generally challenging, sometimes it is reasonable to put 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 [FlTa92, FlTa95, Grop95, HMS93, KeSt92, Tama93, TeGi00]. Here (cf. chapter 3), 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. In the meantime, the world record for two dimensions presented here was challenged by Moloney and Pruessner [MoPr03], but they never published the results for their world record run (not even on direct request), thus it is hard to accept their claim for a new world record.

Another variation of the Hoshen-Kopelman algorithm, which allows simulation of growing percolation lattices, and thus making size-dependent properties directly available, is presented in chapter 4.

Chapter 2

Fluctuations of Cluster Numbers

2.1 Introduction

Percolation is a thoroughly studied model in statistical physics. Normally, only the mean numbers $\langle n_s \rangle$ of clusters are studied, but their fluctuations are ignored. In order to study these fluctuations, it is necessary to investigate distributions, i. e. many independently generated lattices.

This need leads to the method of replication: instead of parallelizing a single run (by domain decomposition or other means), a conventional, sequential program is used, but with many runs in parallel, each with different parameters (here, the inital seed for the random number generator is changed for each run). Due to the simplicity, this method is also called "poor man's parallelization"; nevertheless, it can be reasonable science (as it is here) and can offer valuable insight, with very little effort.

Well suited to replication are problems that can be simulated on single processor computers (meaning that memory consumption or wall clock time are no restrictions), and it is necessary to sweep a parameter region.

2.2 Statistical measures

When investigating fluctuations of n_s, we are interested in how these n_s are distributed for many independent simulations. One method to describe such distributions are histograms, i. e. counting how many times a simulation result n_s fell into the interval $[n_{s,i}, n_{s,i+1}[$; another method is to characterize a distribution by its statistical measures mean, variance, skewness, and kurtosis.

An important distribution is the so-called Gaussian; a Gaussian distribution of a variable x is described by

$$f(x) \propto \mathrm{e}^{-\frac{(x-\bar{x})^2}{2\sigma^2}},$$

with mean $\bar{x} = 1/N \sum x_i$, often denoted $\langle x \rangle$, and variance $\sigma^2 = 1/(N-1)\sum(x_i - \bar{x})^2$, often written as $\langle x^2 \rangle - \langle x \rangle^2$. The mean describes the position of the maximum, while the variance describes the width of the Gaussian. The square root of the variance is the standard deviation, which is often taken as a measure for the statistical error. Further measures are the skewness $\mathrm{Skew}(x) = 1/N \sum[(x_i - \bar{x})/\sigma]^3$, which describes if and how the distribution is skewed with an asymmetric tail to the left or right, and the kurtosis $\mathrm{Kurt}(x) = 1/N \sum[(x_i - \bar{x})/\sigma]^4 - 3$, which describes if the distribution is more peaked or more flattened than a Gaussian. Both skewness and kurtosis vanish for a true Gaussian (that is the reason why the term -3 appears in the definition of the kurtosis), i. e. they measure the non-Gaussianness of a distribution. For more details and an example implementation for calculating these measures, cf. [PTVF92, chapter 14].

2.3 Simulations

To study fluctuations of cluster numbers, lattices of different sizes have been investigated thousands of times with different random numbers, using the Hoshen-Kopelman algorithm. Only two-dimensional site percolation on square lattices was studied, at the critical threshold $p_c = 0.5927464$ (cf. [NeZi00]; this value was chosen for all simulations of percolation in two dimensions throughout this thesis, in order to obtain comparable data); fully periodic boundary conditions were used in order to mitigate perturbation of open boundaries, which would overshadow all other finite-size effects (see section 3.3.3 for a more detailed discussion).

Investigated lattice sizes were $L = 10^3$, $2 \cdot 10^3$, $3 \cdot 10^3$, $4 \cdot 10^3$, $7 \cdot 10^3$, $1.5 \cdot 10^4$, $3 \cdot 10^4$ (5000 runs each), $5 \cdot 10^3$, 10^4 (20000 runs each), $2 \cdot 10^4$ (40000), 10^5 (1500). The simulations were done on fast workstations; the implementation of the Hoshen-Kopelman algorithm was sequential, but of course independent runs could be done in parallel, using replication. Although this is the most primitive way of parallelization, it is well suited for this type of study.

For random numbers the six-tap generator R(18,36,37,71,89,124) was used (cf. appendix D). Total required CPU time was about 2500 hours.

2.3.1 Distribution of Cluster Numbers

Plotting the distribution of cluster numbers for clusters of fixed size s yields Gaussians for small s, cf. figure 2.1; the position of the maximum (corresponding to $\langle n_s \rangle$) becomes smaller for larger s, in compliance with the well-known power-law $n_s \propto s^{-\tau}$ (see section 1.3 and subsection 3.3.1 for a discussion of this power-law); the width of the Gaussian (corresponding to the variance $\langle n_s^2 \rangle - \langle n_s \rangle^2$) also decreases for increasing s.

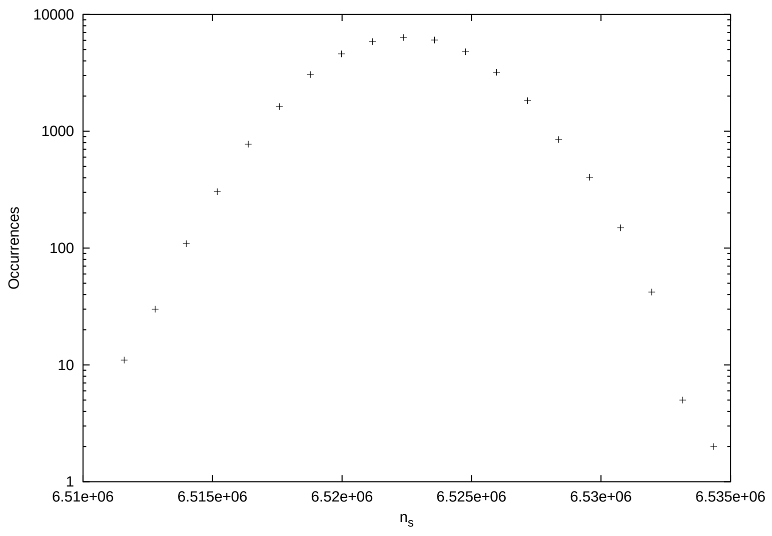


Figure 2.1: Histogram of occurences of n_s for $s = 1$, $L = 2 \cdot 10^4$, and 40000 runs.

When $\langle n_s \rangle$ becomes small enough with increasing s, the left-hand side of the distribution gets distorted and finally vanishes, as cluster numbers have to be positive integers. This is a finite-size effect, as for large L this effect takes place for larger s, cf. figure 2.2.

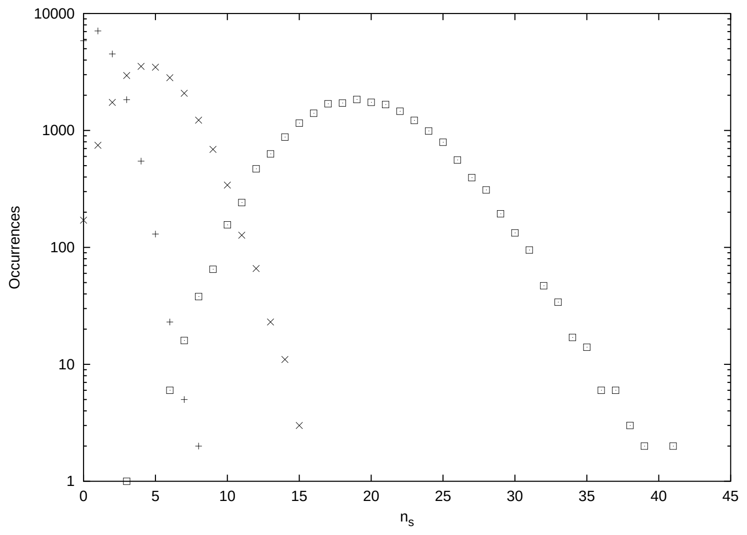


Figure 2.2: Distribution of n_s for $s = 700$, for $L = 5 \cdot 10^3$ (+), $L = 10^4$ ($\times$), and $L = 2 \cdot 10^4$ ($\square$); 20000 runs each.

The right-hand side remains quasi-Gaussian, it fits $\exp(-\text{const} \cdot s^{\zeta})$, with $\zeta = 2$ for small s and ζ decaying slowly for increasing s; this is consistent with results for skewness and kurtosis, see below.

2.3.2 Variance of cluster numbers

The mean $\langle n_s \rangle$ shows the expected power-law $n_s \propto s^{-\tau}$, with corrections to scaling $(1 - k_1 s^{-\Delta_1})$ ($\tau = 187/91$ is known exactly, $\Delta_1 = 0.70(2)$ can be extracted from data here; Δ_1 is discussed in more detail in subsection 3.3.2); the same is true for the variance $\langle n_s^2 \rangle - \langle n_s \rangle^2$, including the same corrections to scaling. Thus we expect $(\langle n_s^2 \rangle - \langle n_s \rangle^2)/\langle n_s \rangle = 1$, as Coniglio et al. claimed a long time ago [CSS79]. However, for small s, we see a power-law dependence $(\langle n_s^2 \rangle - \langle n_s \rangle^2)/\langle n_s \rangle - 1 = k_2 s^{-\Delta_2}$ (Coniglio et al. saw only a deviation for $s = 1$, as they had far less computing capacity). A graphical estimate yields $\Delta_2 = 1.2(2)$ and $k_2 = 0.25(5)$, cf. figure 2.3.

2.3.3 Skewness and kurtosis of distributions

It is interesting not only to investigate the mean and variance of n_s, but also higher moments like skewness and kurtosis. While mean and variance are enough to describe a true Gaussian, the parameters skewness and kurtosis can describe deviations of such a Gaussian.

The skewness grows linearly with s, cf. figure 2.4; this means that the skewness becomes too large to be useful for increasing s; this is due to the distortion of the distributions.

The kurtosis shows no power-law dependence, cf. figure 2.5; for intermediate s it grows exponentially, for large s the situation is unclear; far more extensive simulations need to be done to clarify this. But here, too, the kurtosis is no good measure for large s.

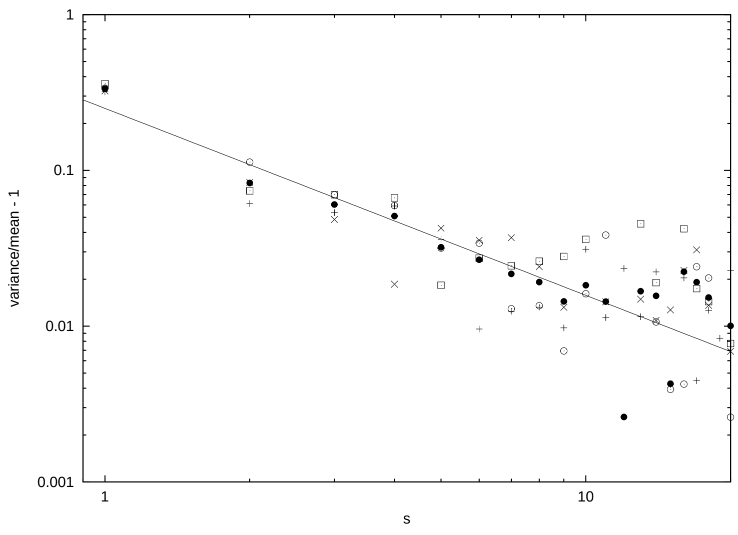


Figure 2.3: Variance divided by mean for $L = 2 \cdot 10^4$ and 40000 runs. The black bullets correspond to the full set of 40000 runs; the other signs correspond to four partitions of 10000 runs each. This partitioning was done to look for systematic deviations for specific s; none are visible. The solid line corresponds to the power-law $k_2 s^{-\Delta_2}$, with $k_2 = 0.25$ and $\Delta_2 = 1.2$ chosen.

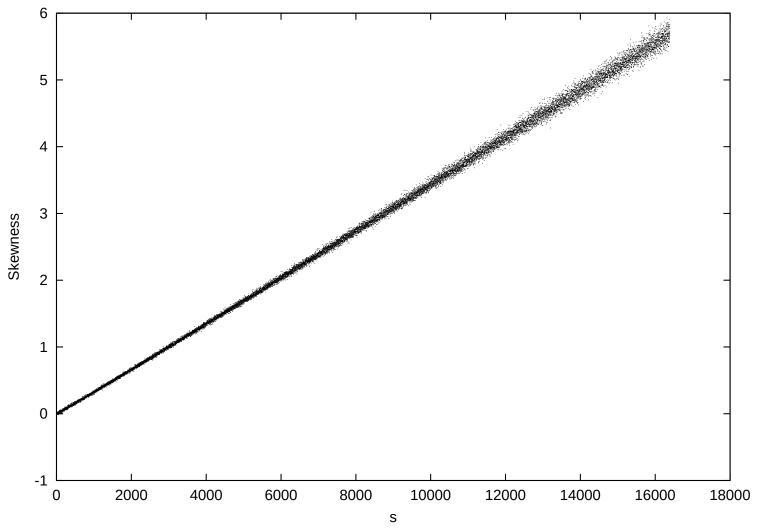


Figure 2.4: Skewness for $L = 2 \cdot 10^4$, 40000 runs.

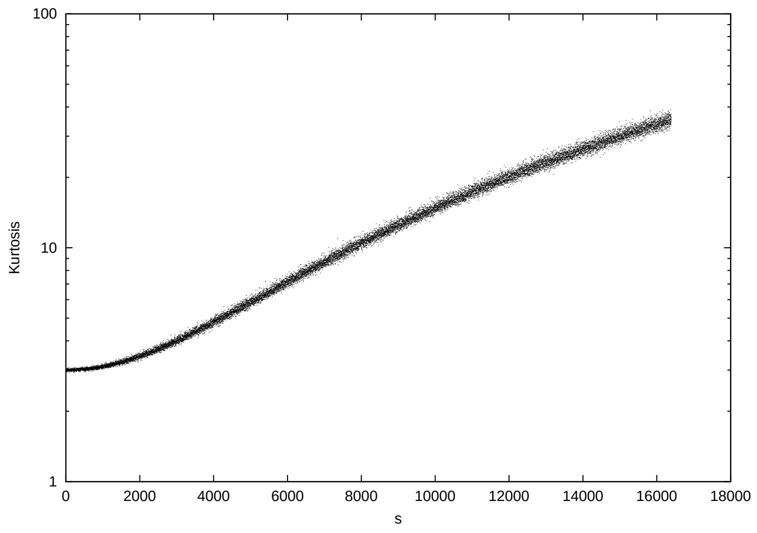


Figure 2.5: Kurtosis for $L = 2 \cdot 10^4$, 40000 runs.

2.4 Summary and outlook

The variance of the cluster numbers is equal to their mean, with power-law corrections for small cluster sizes s: $\langle n_s^2 \rangle - \langle n_s \rangle^2 = \langle n_s \rangle (1 + k_2 s^{-\Delta_2})$, $\Delta_2 = 1.2(2)$, $k_2 = 0.25(5)$.

The distributions of cluster numbers are Gaussians for small s and become distorted with increasing s, yielding growing values for skewness and kurtosis. The skewness grows linearly with s, the kurtosis grows exponentially for intermediate s.

Further studies of fluctuations can be done while studying other properties of percolation; as Monte Carlo algorithms demand averaging over several independent runs for obtaining better statistics, distributions of cluster numbers come for free.

Chapter 3

Parallelizing the Hoshen-Kopelman algorithm using domain decomposition

3.1 Devising a parallelized version of the Hoshen-Kopelman algorithm

In the previous chapter, we have seen that replication can be a viable approach to parallelization for some scientific problems. Precondition is that each of the seperate runs fits into the memory of a single-processor machine. If this is not possible, we have to divide the data we use in our program into parts and distribute these over the memory of several single-processor machines, normally by means of domain decomposition. This way, we do not only acquire more memory, but also more processing power; ideally, N processors should solve our problems N times faster. This makes very large simulations feasible.

Sometimes large simulations are needed to investigate effects that are not visible in small systems, or in order to test hypotheses over several orders of magnitude in system size; power-law or logarithmic corrections to intrinsic behaviour can produce qualitatively different results for small and large systems. Even more, larger systems promise better precision for Monte Carlo data (cf. appendix C). In short: size does matter in science, too.

The advantage of parallelization (more memory, more processing power) comes with two disadvantages: parallel overhead for computation (N processors are less than N times faster), affecting the computer, and increased difficulty of programming, affecting the human programmer. In this chapter, the parallelization of the Hoshen-Kopelman algorithm by domain decomposition into vertical strips is described, a fruitful approach which produced world records in simulated system size. This method was originally published in the diploma thesis [Tigg01]; here it is refined, e. g. by adding fully periodic boundary conditions. Furthermore, while [Tigg01] published source code parallelized with shmem-directives, which are available only for a few parallel computers, appendix F of this thesis presents a program parallelized with the MPI library, which is available on nearly every parallel computer in use today. Hopefully other scientists can use this program for their research. Some comments on how to work with the code and how to extend it can be found in appendix F.2.

3.1.1 Domain decomposition

One of the major limitations for the Hoshen-Kopelman algorithm in higher dimensions is memory, or lack thereof. An old world record size for a simulation in four dimensions was 611^4 (cf. [Stau00]), which required approx. 1.5 GByte only for storing the hyperplane of investigation, aside from more memory needed for supplementary data structures. Pushing this world record further requires huge amounts of memory not available in standard sequential computers.

Fortunately, massively parallel computers offer the necessary 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 has been investigated, communication between the domains resp. processors is necessary. 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. Theoretically, if a single processor can handle a L^{d-1} hyperplane in memory, N processors can handle lattices of linear size $L' = N^{1/(d-1)}L$. 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.

3.1.2 Clusters extending over several sub-domains

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 necessary. 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.

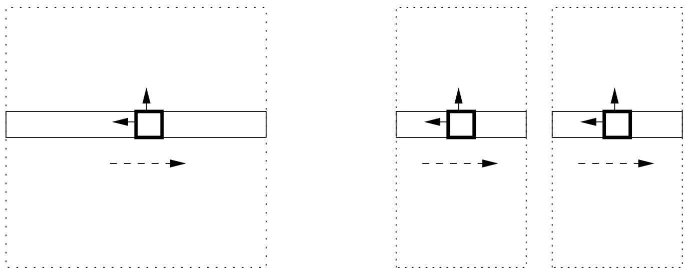


Figure 3.1: A sketch of domain decomposition for two-dimensional percolation; on the left side, a sequential implementation, on the right side, a parallel implementation decomposed into two sub-domains. The site with a strong black border is the site currently in investigation. The full lattices are not stored, but exist only virtually (thus the dotted lines). The line which is in investigation is stored (solid lines). The current site is connected to left and top neighbours (arrows), left was investigated one step before, top is stored in the line and will be overwritten with results from the current site, afterwards the site to the right will be investigated (flow is indicated by dashed arrow).

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 necessary 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. Either 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.

We introduce a local label array (analogous to the traditional label array in Hoshen-Kopelman) and a global label array. For simplicity (one of the most desirable properties when parallelizing programs), both local and global labels have entries in the local array; these are distinguished by a single bit; just as the sign determines if a label is an indirect one, pointing to another, or a direct one, meaning the array contains the number of sites in the cluster, we use the least significant bit in that array entry to determine if it is a local label (and thus contains the number of sites in the cluster) or a global one, in which case it points to the corresponding global entry in the array of global labels.

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.

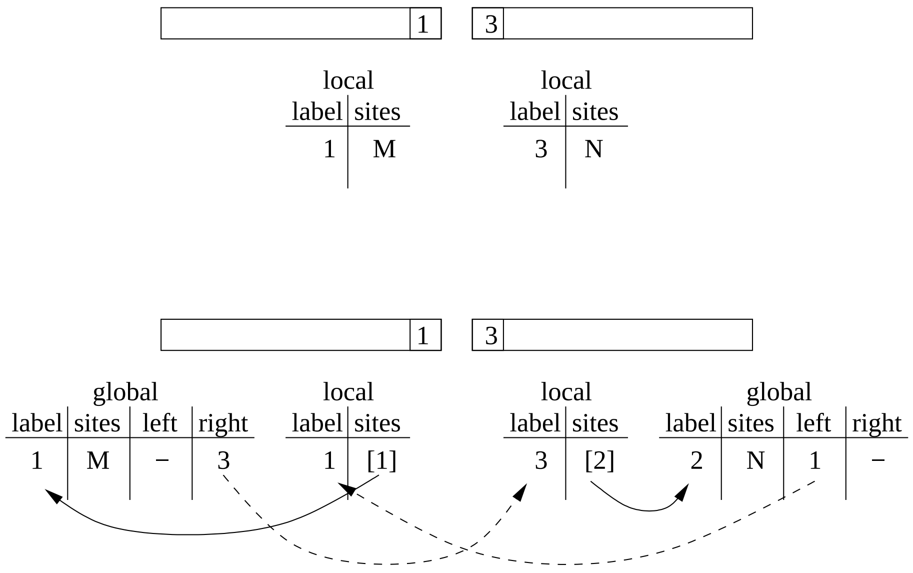


Figure 3.2: Suppose we have finished one local examination of the divided line of investigation. The left processor has a local cluster labelled ‘1’, with M sites in it, the right processor a cluster ‘3’ with N sites (top part of diagram). When exchanging boundaries, both clusters need to be joined, as they touch each other (bottom part). Both need to be converted from local to global status. Entries in the global arrays are allocated, the number of sites for the cluster parts are put into it. Furthermore, the left processor puts the label of the former local cluster in the right processor into its global array, and vice versa. This is done to keep these cluster in contact (they may even grow to other processors). In the local arrays, the numbers of entries in the global array are entered for the labels; these are marked by setting the least significant bit. Solid lines show the connection from local label to global one within a processor; dashed lines show how the parts of the global cluster reference each other. It is important to note that even global labels are always referenced through local ones, both within and across processors. This avoids relabeling of sites in the hyperplane of investigation; even more, it ensures that local and global labels can be treated equally for reclassification and other parts of the program.

Even more, let us examine the following situation: In our strip, we have two 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 necessary 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 relaxation 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.

3.1.3 Recycling of redundant labels

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 still could reference them (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 these 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 necessary). 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.

This reduction of global clusters is remarkably similar to the concentration process needed for accounting global clusters, described in the following subsection. The difference is that for recycling global labels, only “dead” labels are reduced, while for full cluster accounting *all* global labels are reduced. Due to this similarity, both can be done using the same subroutines, cf. appendix F.2.

3.1.4 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 is 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.

3.1.5 Fully periodic boundary conditions

When simulating a d-dimensional lattice, boundary conditions determine how the hyperfaces of the L^d hypercube are interconnected (e. g. the left and right face of a cube). Open boundaries mean that these are not connected at all, while periodic boundaries mean that a given site in the left face has as left neighbour the site with the same coordinates (within the face) in the right face, and vice versa. A square lattice with fully periodic boundaries in both dimensions becomes a torus (cf. fig. 3.3), a cubic lattice becoms a hypertorus, etc.

Due to the domain decomposition in strips, periodic boundary conditions come for free in one dimension, in the direction perpendicular to the strips. In $d > 2$ dimensions, for $d - 2$ directions we can easily implement helical boundary conditions, by storing the hyperplane in a linear array and accessing neighbouring sites using offsets of ± 1, $\pm L$, $\pm L^2$, $\ldots$; helical boundaries are essentially like periodic boundaries, but with a twist of one site. They behave essntially the same, i. e. they do not introduce open borders into the system. Therefore periodic and helical boundary conditions are normally summarized as periodic.

In the direction in which the hyperplane of investigation moves during the HK algorithm, the lattice has open boundaries. These can be made periodic by storing the first (uppermost) hyperplane after it has been simulated, and then connecting it to the lowermost hyperplane after the whole lattice has been simulated. Additionally, it is important for recycling to consider not only labels present in the current hyperplane, but also those in the first.

Combining these techniques, we can simulate lattices with fully periodic boundary conditions, meaning that no open borders are present in the system, preventing systematic errors due to these. The disadvantage of this method is that twice the amount of sites need to be stored for the hyperplanes; for high dimensions, this can limit the size of lattices that can be simulated. But in many cases, smaller lattices with fully periodic b. c. yield more precise results than larger lattices with open b. c. Section 3.3.3 shows examples.

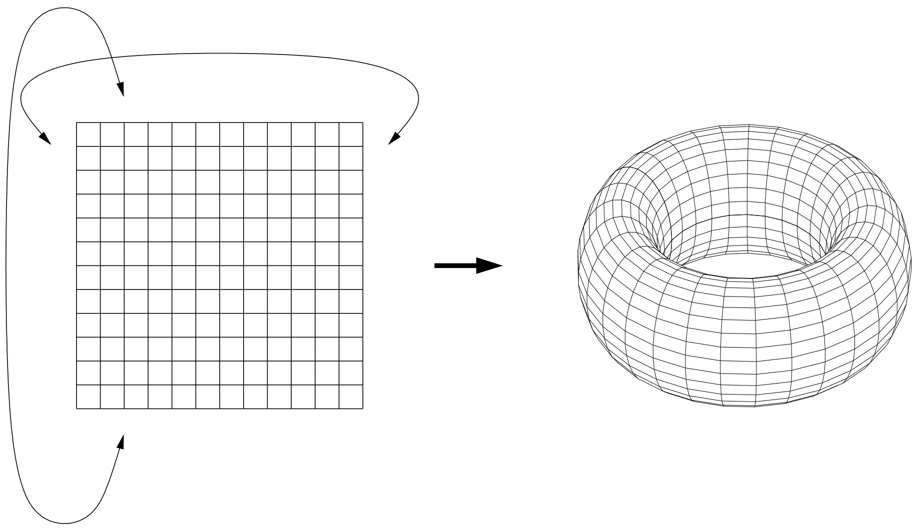


Figure 3.3: When connecting the left and right resp. the top and bottom sides of a square lattice, the result is a torus, without any open border.

3.1.6 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 necessary 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; for periodic b. c., copy the first simulated plane into buffer; initialize all data structures; 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 necessary due to tight memory conditions.

4. *Recycling* (if necessary):

   (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.

3.2 Other ways of parallelizing Hoshen-Kopelman

There are, as mentioned above, certainly other ways of domain decomposition. Old work (like [FlTa92], [HMS93], [KeSt92]) did parallel cluster counting for implementing Ising models with Swendsen-Wang dynamics, which cannot be compared directly with percolation (but is of course inspirational). Teuler and Gimel [TeGi00] did investigate percolation, but the authors did store the full lattice instead of only one plane, which restricted them to rather small lattice sizes. In unpublished work by MacIsaac and Jan (private communication), they tried 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 have been 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 would have been possible only in two dimensions.

Moloney and Pruessner devised a way to parallelize the Hoshen-Kopelman algorithm in an asynchronous way [MoPr03], i. e. small patches of a larger lattice are simulated independently and stored (only the border of the patch needs to be stored, the interior is irrelevant after simulation); after enough patches have been simulated, these are joined to form a larger lattice. This method is well suited for low dimensions; for higher dimensions, the ratio of surface vs. volume grows and makes this scheme inefficient, i. e. too much data have to be stored.

3.3 Results of Monte Carlo simulations

Some of the results presented here were already presented in the diploma thesis [Tigg01]. The old results were generated using the Cray T3E of the NIC Jülich, while new results were generated using the JUMP of NIC Jülich and the compute-cluster Clio of the Computing Centre of the University of Cologne.

Generally, the old results were produced with open boundaries or busbar in one direction, and periodic resp. helical boundary conditions in other directions. The new results were done without open boundaries, using fully periodic or helical b. c. This reduces systematic errors inevitably caused by the open boundaries (see discussion below), but this means that twice the number of lattice sites have to be stored, reducing the maximal system size that can be simulated. For comparison, some of the new results have been generated using open b. c. in one direction, e. g. for five dimensions.

In the rest of this chapter, old and new data will be labelled accordingly. New data with open boundaries are marked as such; when not explicitly marked, new data are for fully periodic b. c.

3.3.1 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. [Fish67]) 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 in a log-log-plot, 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 an “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 known exactly (cf. [NRS80], [Nien82], [Nijs79], [Pear80]): $\tau = 187/91$. 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, results for two dimensions are more precise than those for higher dimensions.

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 subsection), 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 exact $\tau = 187/91$.

For three dimensions, we find $\tau = 2.190(1)$, which is roughly compatible with the old literature value $\tau = 2.186(2)$ found by Jan and Stauffer in [JaSt98], and $\tau = 2.189(2)$ from Lorenz and Ziff in [LoZi98b] (they investigated bond percolation, but as τ is universal, their value is the same as for site percolation). Gimel et al. took $\tau = 2.189$ as exact when analysing their $d = 3$ data, instead of trying to extract it from the data in [GND00]. Old data from [Tigg01] showed $\tau = 2.190(2)$.

In four dimensions, we find $\tau = 2.315(2)$, compatible with $\tau = 2.313(3)$ from Paul et al. [PZS01], and $\tau = 2.3127(7)$ from Ballesteros et al. [Ball97]. Old data from [Tigg01] showed $\tau = 2.313(2)$.

In five dimensions, we find $\tau = 2.41(1)$, in agreement with $\tau = 2.412(4)$ from [PZS01]. [Tigg01] did not investigate $d = 5$.

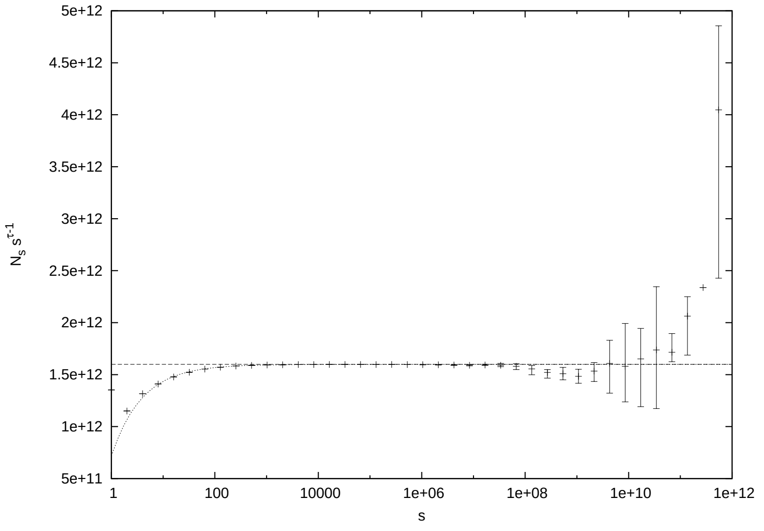


Figure 3.4: Cluster size distribution in two dimensions for $L = 7 \cdot 10^6$, averaged over three runs (new data, fully periodic b. c.). The error bars correspond to minimum and maximum values of the runs (standard deviation is not a reasonable error indicator for three samples). For large s, only few clusters occur and thus fluctuations are strong (cf. chapter 2 for more on fluctuations). The dashed line corresponds to the asymptotic behaviour for large clusters. The dotted line shows corrections to scaling for small s (see below), here it corresponds to $\Delta_1 = 0.73$ and $k_1 = 0.55$, i. e. a correction term $(1 - 0.55 \cdot s^{-0.73})$.

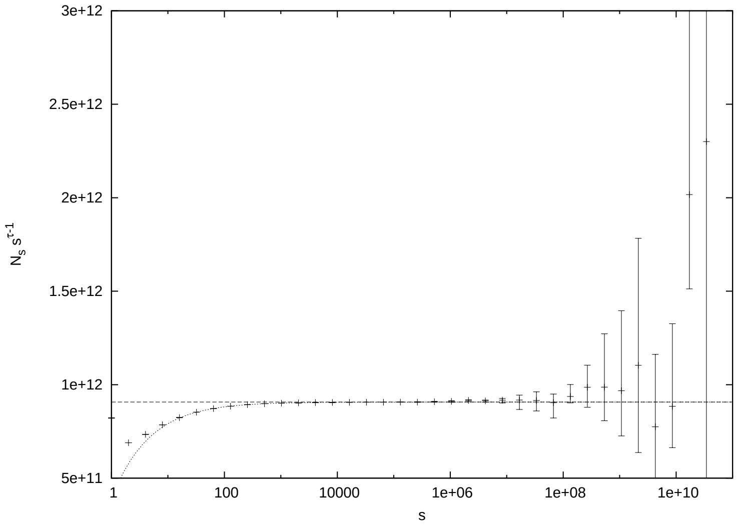


Figure 3.5: Cluster size distribution in three dimensions for world record size $L = 25024$, averaged over three runs (new data, fully periodic b. c.). Error bars as above. The dashed line corresponds to the asymptotic behaviour, the dotted line corresponds to corrections to scaling with $\Delta_1 = 0.65$ and $k_1 = 0.57$.

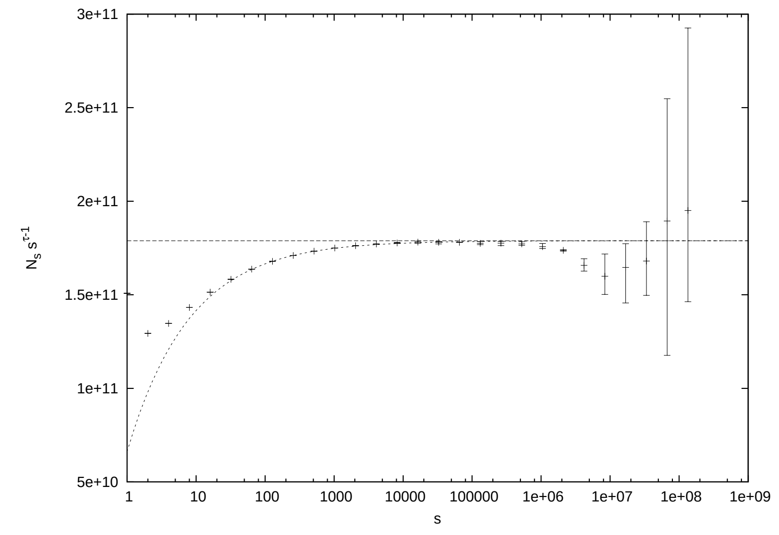


Figure 3.6: Cluster size distribution in four dimensions for world record size $L = 1305$, averaged over three runs (new data, fully periodic b. c.). Error bars as above. The dashed line corresponds to the asymptotic behaviour, the dotted line corresponds to corrections to scaling with $\Delta_1 = 0.48$ and $k_1 = 0.63$.

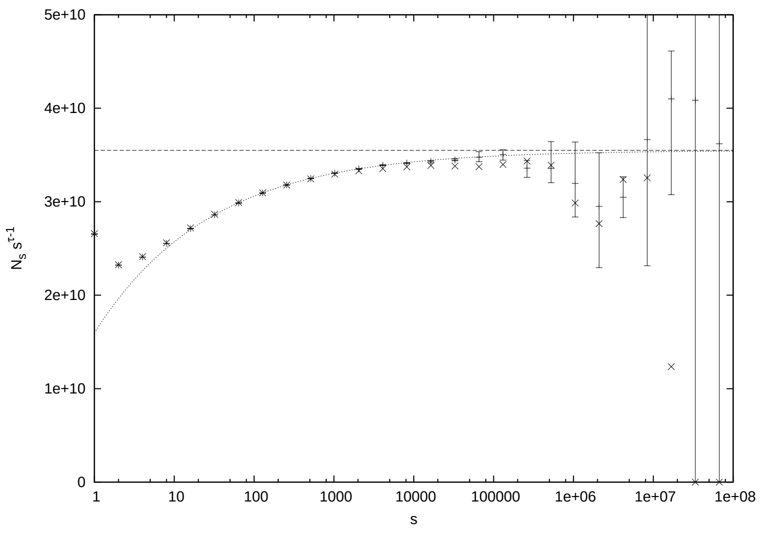


Figure 3.7: Cluster size distribution in five dimensions for world record size $L = 225$, averaged over three runs (+ with error bars, new data, fully periodic b. c.), and for one run with world record size $L = 270$, scaled by $(225/270)^5$ ($\times$ without error bars, new data, open b. c. in one direction). Error bars are determined as above. The dashed line corresponds to the asymptotic behaviour, the dotted line corresponds to corrections to scaling with $\Delta_1 = 0.30$ and $k_1 = 0.55$.

3.3.2 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. [Adle83])

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

The correction term is called corrections to scaling, it could stem from an irrelevant operator or from a nonlinear scaling field (cf. [Ahar83], [BiSt87], [MDSS83]). 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. 3.8).

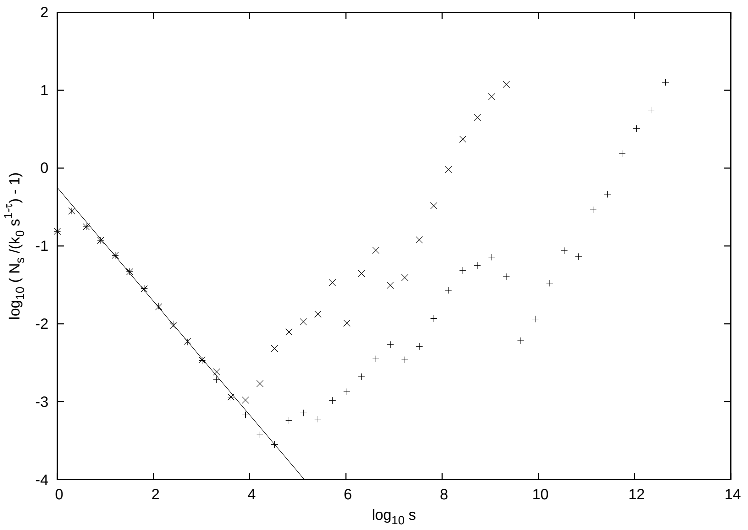


Figure 3.8: Corrections to scaling in two dimensions, $L = 7 \cdot 10^6$ (+, averaged over 3 runs), $L = 10^5$ (×, averaged over 10 runs); both new data. Because of finite-size effects, the distribution does not follow a straight line, but at a given size of clusters, there are more than expected. For smaller L, this happens at smaller sizes s. The solid line corresponds to $s^{0.73}$.

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. 3.9.

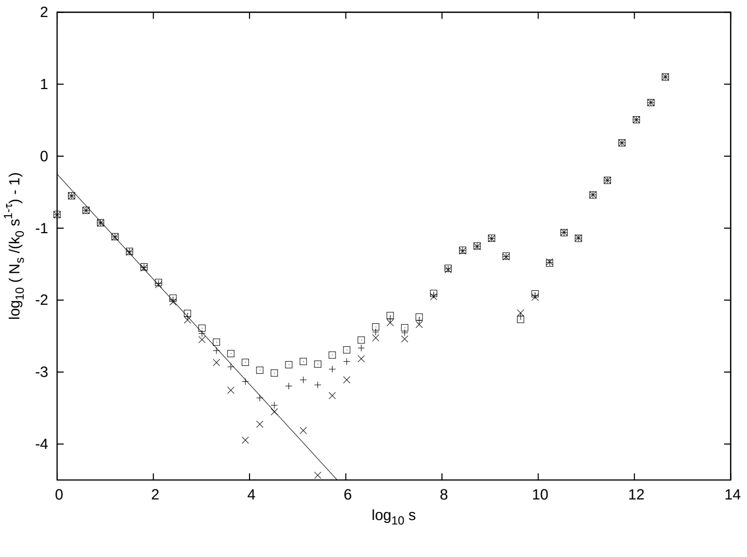


Figure 3.9: Corrections to scaling for $L = 7 \cdot 10^6$ in two dimensions (new data). On the y-axis is plotted the binned data divided by the asymptotic behaviour $k_0 s^{-\tau}$. Three values were chosen for k_0: $1.5983 \cdot 10^{12}$ (+, optimal), $1.5973 \cdot 10^{12}$ ($\times$, slightly too small), and $1.5993 \cdot 10^{12}$ ($\square$, slightly too large). The solid line represents the corrections to scaling power law $s^{0.73}$, which is a good approximation for small s for the chosen optimal $k_0 = 1.5983 \cdot 10^{12}$.

In order to verify if our corrections to scaling are correct, we can use the same method as for verifying the power-law for the cluster size distribution: we plot the N_s divided by the expected behaviour, in order to see discrepancies. Figure 3.10 shows the result.

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, which is the same as our $\Delta_1 = 0.65(5)$, but with smaller error margins; but for the mentioned reason, their error bars seem to be overly optimistic.

The results obtained from the parallelized simulations are: in two dimensions $\Delta_1 = 0.73(2)$, ruling out the simple nonlinear scaling fields assumption as the only source for corrections to scaling [Ahar83], as this would require $\Delta_1 = 55/91 \simeq 0.6044$. Another Monte Carlo value from literature is $\Delta_1 = 0.65(5)$ (MacLeod and Jan, [MLJ98]). The old result from [Tigg01] was $\Delta_1 = 0.70(2)$.

In three dimensions, we find $\Delta_1 = 0.65(5)$, agreeing roughly with $\Delta_1 = 0.70(5)$ found by Jan and Stauffer [JaSt98]. Gimel et al. found $\Delta_1 = 0.65(2)$ [GND00], but the error seems to be overly optimistic. Old result from [Tigg01] was $\Delta_1 = 0.60(8)$.

In four dimensions, we find $\Delta_1 = 0.48(8)$, old result was $\Delta_1 = 0.5(1)$. In five dimensions, we find $\Delta_1 = 0.30(10)$; [Tigg01] did not investigate $d = 5$.

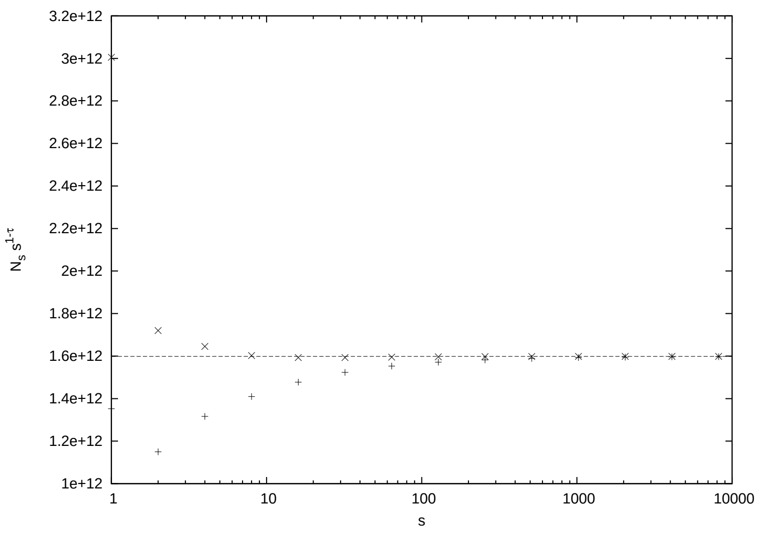


Figure 3.10: Binned cluster sizes N_s divided by expected behaviour without corrections to scaling $N_s = s^{-\tau+1}$ $(+)$, and with corrections to scaling $N_s = s^{-\tau+1}(1 - k_1 s^{-\Delta_1})$ $(\times)$. Data are averaged over three runs with $L = 7 \cdot 10^6$ (new data). The dashed line shows asymptotic behaviour for large clusters. Corrections to scaling show a good approximation to the data for small cluster sizes s, with the exception of the very smallest clusters.

3.3.3 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 touches 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 of its center 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. 3.11: the n_s go up for large s).

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. 3.11).

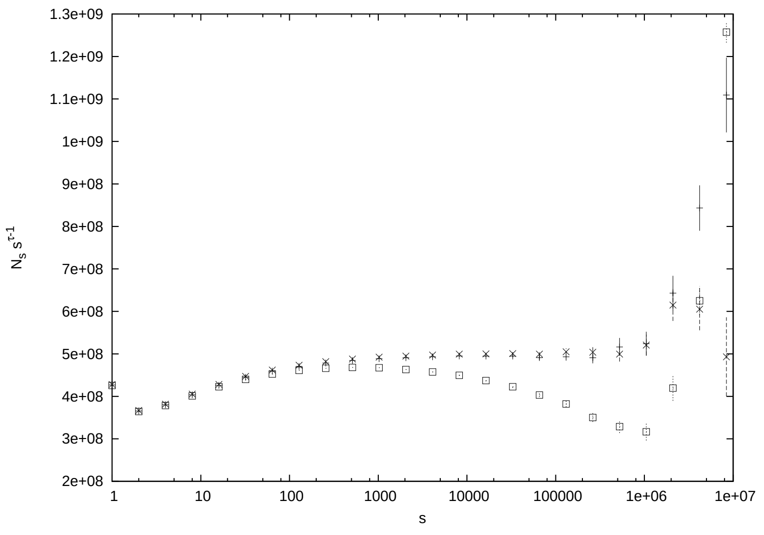


Figure 3.11: Cluster size distribution in four dimensions, using Ziff's four-tap PRNG, $L = 301$ (new data). Periodic boundaries ($+$), open boundaries ($\times$), and busbar ($\square$). Largest cluster for busbar would lie at $7.5 \cdot 10^9$ on the y-axis, outside of this plot. Scale was chosen to demonstrate the behaviour for medium to large clusters. The results here are averages over 50 runs each; error bars are statistical error of mean, i. e. standard deviation divided by square root of number of samples (cf. appendix C). For large clusters, the distribution of n_s is no longer Gaussian (cf. chapter 2), thus the error bars derived from standard deviation for large s are problematic.

This makes busbar data very 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 effects of boundary conditions: 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 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 analyze for fully periodic boundary 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 details like the aspect ratio of the investigated system (i. e. width divided by height in a $d = 2$ system) do have an impact on finite-size scaling. Several publications did investigate this in detail, cf. [AcSt98], [AhSt97], [Card92], [LoZi98a], [LoZi98b], [Stau94], [Ziff96], [ZFA97], [ZLK99].

3.3.4 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. [Baxt78], [TeLi71]). 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. [StAh94, 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. 3.12 (old data from [Tigg01]); for $L = 10^6$, 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 $L = 10^6$ 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 $L = 3.5 \cdot 10^6$ and $L = 4 \cdot 10^6$ (for larger lattices, fluctuations are significantly smaller). A re-investigation of $d = 2$ with new data shows $n_c = 0.027597857(2)$, which is compatible, but significantly more precise (due to larger system size of $L = 7 \cdot 10^6$, averaging over three runs, and periodic b. c.). Ziff et al. have found a value of $n_c = 0.0275981(3)$ in [ZFA97], in agreement with the values found here.

In four dimensions, the situation is different: When using open boundaries, 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. 3.13 ($\times$ correspond to open b. c.): The data points only slightly scatter around the regression line $\propto 1/L$. From this plot, we can extrapolate $n_c = 0.051998(2)$ (the old value $n_c = 0.0519980(2)$ reported in [Tigg01] was overly optimistic). With new data, we get $n_c = 0.0519995(2)$, which has higher precision due to larger L, more runs for averaging, and periodic b. c.

3.3.5 Quality of pseudo-random number generators

As can be seen in fig. 3.12, there is a problem with one of the four utilized random number generators, the linear congruential generator `ibm=ibm*16807`. While the resulting number densities from the other generators agree well within statistical fluctuations, those from the `ibm*16807` deviate visibly. This is due to correlations

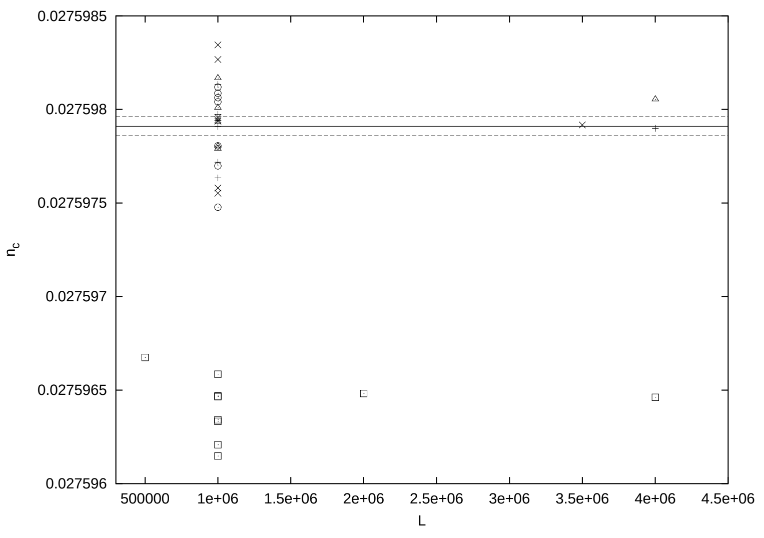


Figure 3.12: Number densities for various system sizes L, various PRNGs, and various runs with different random seeds (old data). Used PRNGs: Kirkpatrick-Stoll (+), Ziff's four-tap (×), ibm*16807 (□), Ziff's six-tap (○), ibm*13^{13} (△). The solid line corresponds to the average of the $L = 10^6$ runs except the ibm*16807 ones, the dashed lines to the statistical error margins (standard error of mean).

in the produced random numbers. The `ibm*16807` is known to be problematic in literature (cf. [SHIW93, part II, chapter 1]), and here once again this is shown clearly. The other generators (which are lagged fibonacci generators) seem to be compatible with each other, so it is wise to choose the fastest one (cf. appendix D).

3.3.6 Speed of simulations and parallel efficiency

When parallelizing a formerly sequential algorithm, we are interested in the efficiency of the result, i. e. how much faster N processors are than one processor. This is the so-called speedup $S(N) = T(1)/T(N)$, with $T(N)$ runtime for N processors; of course, this calculation is valid only if all other parameters besides N that affect speed are kept the same. Ideally, we strive for $S(N) = N$, meaning that N processors are N times faster than a single processor. In reality, this is rarely the case; e. g. synchronization and communication between processors introduce parallel overhead into the runtime (which would not be necessary for sequential programs), meaning that the speedup is reduced.

In order to estimate how well a program is parallelized, we need to understand in what parts of the parallel algorithm how much of the runtime is spent, and in what way the processors communicate with each other. For the algorithm presented in this chapter, we can make the following partitioning: investigating the lattice sites is a local part (in would be done by the sequential algorithm, too); each processor investigates its part of the hyperplane. Afterwards, exchange with neighbouring processors is necessary, introducing parallel overhead (which would be unnecessary for a sequential program): the boundaries are swapped in order to detect clusters that extend over several strips, and pairing information is exchanged. Generally, the amount of communication per processor is $\propto L^{d-2}$ for a single hyperplane, while the amount of local investigation is $\propto L^{d-1}/N$ per processor (number of sites per strip). In order to obtain high parallel efficiency, we want $L^{d-1}/N \gg L^{d-2}$, or in other words, $L \gg N$. Other parts of the program are recycling and counting clusters at

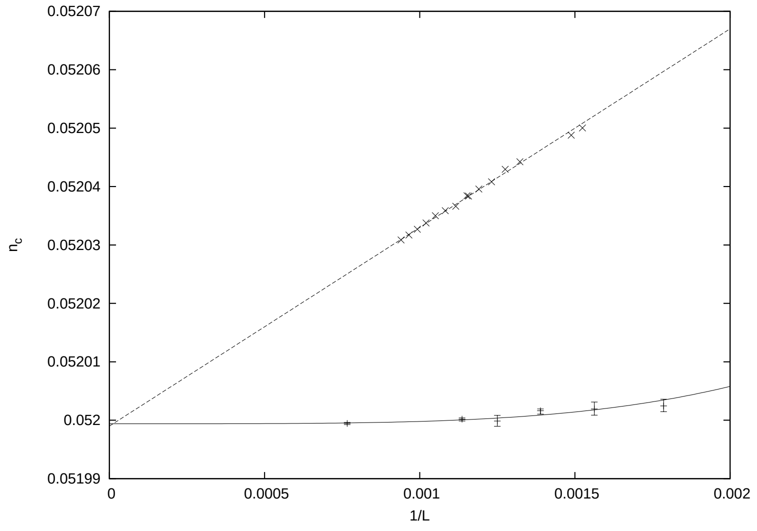


Figure 3.13: Number densities for various system sizes L for $d = 4$, using open b. c. in one direction ($\times$, old data), and using fully periodic b. c. ($+$, new data, averaged over three runs each; error bars correspond to minimum and maximum of three runs). The lines correspond to the extrapolations to infinity: the dashed line is $\propto 1/L$, while the solid line is $\propto 1/L^4$. Thus it becomes clear that for finite systems, fully periodic b. c. allow a more accurate estimate for number density, while open and periodic b. c. converge for $L \to \infty$. The leftmost data point $+$ corresponds to world record size $L = 1305$, where the finite size corrections to $L = \infty$ are smaller than the statistical fluctuations (not visible at this scale); thus large systems with fully periodic b. c. are practically "infinitely large". We obtain $n_{c,\text{periodic}}(L = \infty) = 0.0519995(2)$. For open boundaries, we can estimate the value $n_{c,\text{open}}(L = \infty) = 0.0519980(20)$ by varying the line, in order to estimate the finite-size influences. The old value reported in [Tigg01], $n_{c,\text{open}}(L = \infty) = 0.0519980(2)$, was overly optimistic, highlighting that estimating finite-size effects can be problematic.

the end. These have both local and global components, but for correctly chosen parameters, they should be negligible in comparison to the pure investigation of the lattice.

Measuring parallel efficiency by instrumenting the code with timers for different sections (local and global) is useful, but offers only limited insight. For example, recycling is both a local and a global process; breaking it down to smallest pieces is tedious and problematic, as timers have a finite resolution. Even more, as the decomposed domains depend on the number of processors, we can have super-linear speedup advantages: when the number of processors increases for a fixed problem size (also called "strong scaling"), the domains per processor become smaller, and thus recycling is necessary less frequently, possibly offsetting the disadvantage we obtain from more communication, necessary because there are more processors that need to be synchronized. Even the purely local step within a sub-domain is influenced by the parallelization, as we need to distinguish between local and global labels, and treat these accordingly.

The only possibility to get reliable conclusion of parallel efficiency is to measure runtimes for different numbers of processors, and then to use Amdahl's law [Amda67] to estimate the sequential portion s of the program; cf. appendix E for more on Amdahl's law. Figure 3.14 shows results for $d = 2$, $d = 3$, and $d = 4$. From the demanded $L \gg N$ we expect lower dimensions to be more efficient than higher ones, as for smaller d we can simulate larger L, while the number of processors N is dictated by available hardware. Nevertheless no clear trend is visible in dependence on dimensionality. This can be explained by machine architecture: the time consumption of communication depends not only on the message size, but also includes a fixed overhead for every single message, the so-called latency. For small messages, latency can dominate the overall time, meaning that $d = 2$ (with messages consisting of single words) is not faster than $d = 3$. Another noteworthy effect on runtime can come from the processor caches. Data structures fitting into the cache can be accessed faster than those that need to be fetched from main memory every time. When dividing the hyperplane of investigation over more processors, a larger part of it can be kept in cache, giving super-linear speedup (i. e. $S(N) > N$, although this seems counter-intuitive).

When measuring the program on real computers, we find a parallel portion of 0.985 from fig. 3.14, with strong scaling (i. e. the problem size is kept constant, regardless of the number of processors). In reality, we would simulate larger lattices when using many processors, meaning less parallel overhead in comparison to the investigation of the lattice. Therefore, it is safe to conclude that the modified Hoshen-Kopelman algorithm presented in this chapter is efficiently parallelized, and is suitable for large numbers of processors.

3.4 Summary and outlook

Parallelizing the Hoshen-Kopelman algorithm by decomposing the hyperplane of investigation into strips is an efficient method, which allows for simulating huge lattices.

The results below are for world record simulations. Those marked with "(old)" were generated on a Cray T3E with open b. c. in one direction and were previously published in [Tigg01], those with "(new)" on a compute cluster with Opteron processors and a fast interconnect, with fully periodic b. c. The entry for $d = 2$ marked with "(MoPr)" is from [MoPr03] and currently sets the world record mark for $d = 2$, although that paper only reports that the simulation was done, but does not present any results for it. Therefore $d = 2$, $L = 7 \cdot 10^6$ presented here is the largest published simulation. For $d = 5$, a single run for $L = 270$ with open b. c. was done

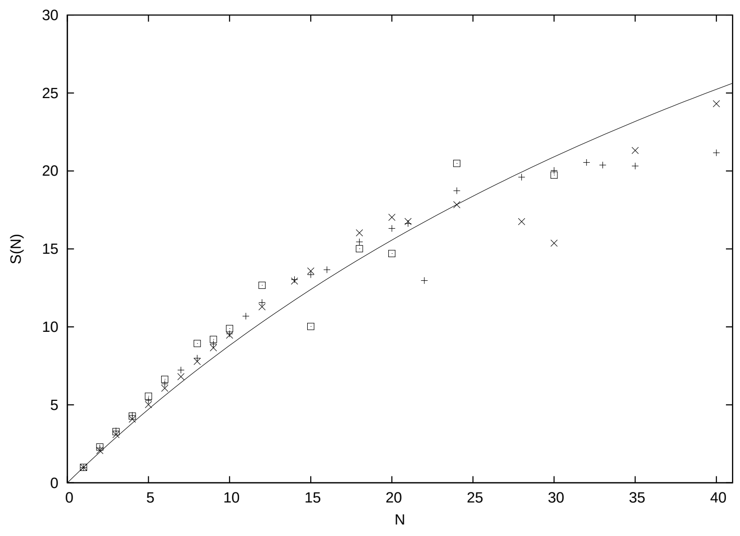


Figure 3.14: Speedup $S(N)$ versus number of processors N, for two-dimensional $(+)$, three-dimensional $(\times)$, and four-dimensional $(\square)$ percolation. The solid line corresponds to Amdahl's law with a parallel portion of 0.985. Total lattice size was kept constant (strong scaling), at $L = 110880$ $(d = 2)$, $L = 2520$ $(d = 3)$, and $L = 360$ $(d = 4)$. Higher speedup $S(N)$ than predicted by Amdahl's law for small N can be explained by super-linear speedup due to cache effects (N processors have N times the amount of cache). The scattering of measurements is due to influences of the operating system, which become worse for larger processor numbers, as all other processors need to wait for a single processor that gets distracted by an operating system task.

(cf. fig. 3.7), but as open b. c. introduce strong finite-size effects, it is not included in the table below.

d		L	τ	Δ_1	n_c
2	(old)	4000256	187/91	0.70(2)	0.02759791(5)
	(new)	7000000	187/91	0.73(2)	0.027597857(2)
	(MoPr)	22200000	187/91	unpublished	unpublished
3	(old)	20224	2.190(2)	0.60(8)	0.052442(2)
	(new)	25024	2.190(1)	0.65(5)	0.05243812(9)
4	(old)	1036	2.313(2)	0.5(1)	0.0519980(2)
	(new)	1305	2.315(2)	0.48(8)	0.0519995(2)
5	(new)	225	2.41(1)	0.30(10)	0.0460321(2)

Although the previous world records marked by "old" are indeed rather old (approx. two computer generations lie between these and this thesis), with the exception for two dimensions they have not been challenged, until now. The only challenge in two dimensions came from Moloney and Pruessner, who invented an innovative algorithm, but they only worked in two dimensions and never above that. It is important to note that in two dimensions, memory is not really a limit, but only computing time. Given a fast workstation with lots of memory and several years of computing time, it is possible to challenge the two-dimensional world record. The reason is that we need to store only a line of investigation, L sites in size, for an L^2 system. In higher dimensions, this significant memory advantage of the Hoshen-Kopelman algorithm shrinks to L^{d-1} for an L^d system. Thus, the real challenge are high dimensions.

The only way to simulate huge lattices for high dimensions is to choose the way of domain decomposition presented in this chapter, as it is necessary to divide the hyperplane of investigation. This complicates programming considerably; the fact that the old world records for $d = 3$ and $d = 4$ have not been broken for over six years, despite serious efforts by some scientists (e. g. MacIsaac), shows how difficult this approach is. Hopefully, the source code presented in appendix F allows other scientists to use this method for studying percolation on large lattices.

Chapter 4

Growing Lattices

4.1 Motivation

In the previous chapters we have seen that replication is a suitable strategy for investigating fluctuations, while domain decomposition allows for simulating huge lattices. In this chapter, we will modify the Hoshen-Kopelman algorithm to work on *changing* domains, i. e. *growing* lattices.

The HK algorithm allows for examining large percolation lattices using Monte Carlo methods, as for simulating an L^d lattice only a hyperplane of L^{d-1} sites has to be stored. The traditional way to do a simulation using the HK algorithm is to choose an L and an occupation probability p, and then walk through the lattice in a linear fashion, one hyperplane after the other, calculating interesting observables (like cluster numbers) on the fly or at the end. When one is interested in the dependence of these observables on either p or L, one has to repeat these simulations with varying values of these parameters.

In many cases, we are interested in observables in dependence on L or p, meaning that we would have to do many simulations with only slightly changing values of these parameters. In would be nice to have algorithms which allow direct investigation, i. e. which produce results for sweeps in p or L in one run.

In 2000 Newman and Ziff published an algorithm which allows to simulate percolation for all $0 \leq p \leq 1$ in one run, making it very easy to study percolation properties in dependence on p ([NeZi00], see also [FrLu00]; section 1.4.4 of this thesis has a short description). Unfortunately, their algorithm lacks a desirable property of the HK algorithm: they need to store the full lattice of L^d sites, whereas for HK only memory for L^{d-1} sites is needed.

A modification of the HK algorithm presented here allows to simulate percolation for many $1 \leq L_i \leq L_{\max}$ in one run, with only a small performance penalty (depending on the number of intermediate L_i which are chosen for investigation). The important advantage of the HK algorithm, small memory consumption, is preserved. In three dimensions, the modified algorithm needs to store three times as much sites as the original one (a constant factor), but this is still $\propto L^{d-1}$, meaning a considerable advantage for low dimensions.

This modified algorithm shall be presented in this chapter, along with simulation results for site percolation on the cubic and the square lattice, i. e. $d = 3$ and $d = 2$. The algorithm can be adapted to any other dimensionality.

While the changing domains make a further domain decomposition for parallelization particularly difficult, it is possible to use the method of replication for this technique, too. In fact, this is highly advisable, as runs for different random numbers are necessary to ensure statistical independence.

4.2 Computational method

The traditional HK algorithm works by dividing the L^d lattice into L hyperplanes of L^{d-1} sites, then investigating one hyperplane after the other. One advantage of this simple scheme is that it allows for domain decomposition and thus parallelization (cf. chapter 3).

Another approach is to work recursively on growing lattices: first simulate a lattice of size L^d, then advance to $(L+1)^d$ by adding a shell with a thickness of one site onto the old lattice. This way, when simulating an $L^d_{\max}$ lattice, all sub-lattices of sizes $L_i < L_{\max}$ can be investigated; the investigation of intermediate lattices takes some additional time (for counting clusters, etc.), but the time spent on generating the full lattice minus time for investigation is the same, because the same number of sites is generated as with the traditional approach.

4.2.1 A modified Hoshen-Kopelman algorithm

This recursive approach was chosen here. It works as follows: imagine a cube of size L^3. It has six faces, labelled A, B, C, X, Y, Z (cf. fig. 4.1). In order to increase the size L of the cube by one, three new faces A', B', C' with a thickness of one are slapped onto the old faces A, B, C; additionally, edges AB, BC, and CA, and a corner ABC are added, forming a full $(L+1)^3$ cube.

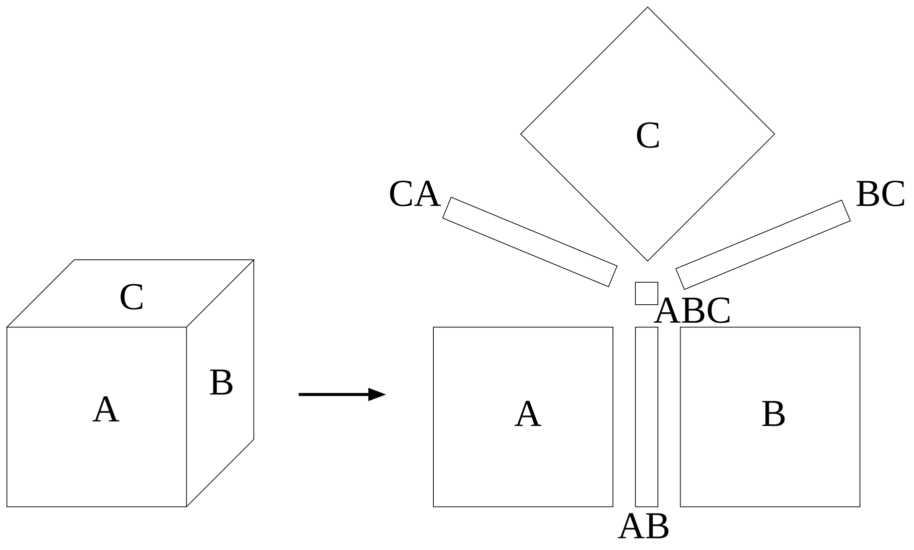


Figure 4.1: Decomposition of the surface of the cubic lattice. Rear faces X, Y, and Z are not shown. Due to the Hoshen-Kopelman algorithm, in d dimensions, only a $(d-1)$-dimensional hyperplane needs to be stored, here for the growing cube only its surface. When going from an L-cube to an $(L+1)$-cube, the surface grows. As edges and corners have to be treated specially, they are stored separately. For an $L \times L \times L$ cube, the faces (A, B, C) are $(L-1) \times (L-1)$, the edges (AB, BC, CA) $(L-1) \times 1$, and the corner (ABC) is 1×1.

Using the HK algorithm (both the original and the modified version), each newly generated site has d old neighbours. For $d = 3$, we have top, left, and back neighbours. For the sake of simplicity, we assume here that top and left neighbours are within a new face A' (resp. B', C'), while back neighbours are in the old face A. Due to geometry, site numbering is shifted by one: the site A'$(i > 0, j > 0)$ has a back neighbour of A($i-1, j-1$). A'($0, j > 0$) has a back neighbour CA($j-1$), A'($i > 0, 0$) has a back neighbour AB($i-1$), and A'($0,0$) has ABC. Thus edges and the corner need to be treated specially (cf. fig. 4.2 for more clarity).

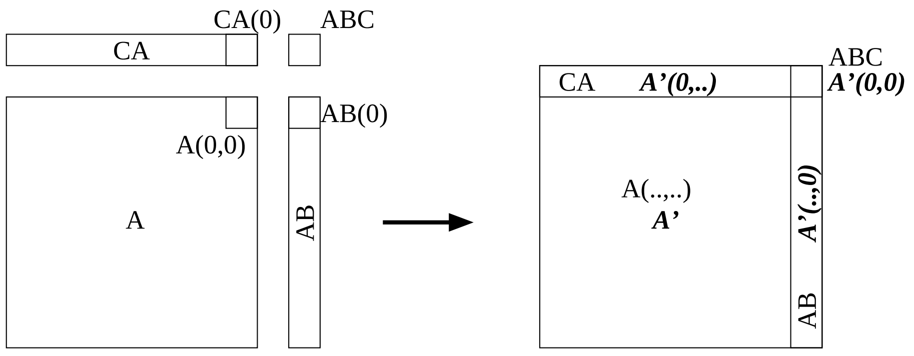


Figure 4.2: When growing the cube from L to $L+1$, the surface which needs to be stored for the Hoshen-Kopelman algorithm, needs to grow, too. The faces of the cube, here A, grow from $(L-1)\times(L-1)$ (A, plain font in figure) to $L \times L$ (A', bold italic in figure). For the largest part of A', A'($i > 0, j > 0$), the back neighbours are sites of A, but for the inner rim of A', A'($i = 0, j > 0$) resp. A'($i > 0, j = 0$), the back neighbours are CA resp. AB. For A'($0,0$), it is ABC.

The site A'(i,j) has the left neighbour A'($i+1, j$) and the top neighbour A'($i, j+1$), when we start working inwards from A'(L, L). This way, we do not need to store A and A', but can overwrite A with the values of A'. The same trick is used in the original HK algorithm for storing only one hyperplane instead of two.

This way we need three faces of size $(L-1)^2$, three edges of $L-1$ and one corner of size 1, for a total of $3L^2 - 3L + 1$ sites, in order to simulate a system of size L^3. For the traditional approach, we need L^2 sites, meaning one third, a fixed factor. But we need an additional label array for both methods, so that memory consumption does not differ drastically.

As the new approach has about the same speed and only moderately higher memory consumption than the traditional one, it is a viable alternative.

One important thing to keep in mind is that for a single run, results for L_1 and $L_2 > L_1$ are *not* statistically independent, e. g. results for L and $L+1$ are naturally highly correlated. This problem can be alleviated by averaging over many runs with different random numbers, because results for different random numbers are suitably statistically independent, as long as the used random number generator has a sufficiently high quality.

4.2.2 Fully periodic boundary conditions

For small lattices, open boundaries would mean a strong distortion of results, in the most cases proportional to $1/L$ (surface divided by volume). In order to get rid of these influences, it is necessary to use periodic boundary conditions, i. e. sites on an open border are connected to corresponding sites on the opposite border.

This is also possible for the modified HK algorithm. It is necessary to store not only the front faces A, B, C, but also the rear faces X, Y, Z. When we choose to obtain observables for an intermediate lattice size L, we connect sites on A to Z, on B to Y, and on C to X.

When during periodicization a cluster on one face connects to the same cluster on the opposite face, that cluster spans the whole system and wraps around, and thus can be identified as the spanning (or "infinite") cluster. When such a cluster is detected, the system percolates. It is possible that more than one such cluster forms; the number of different spanning clusters is counted as n_{sp}.

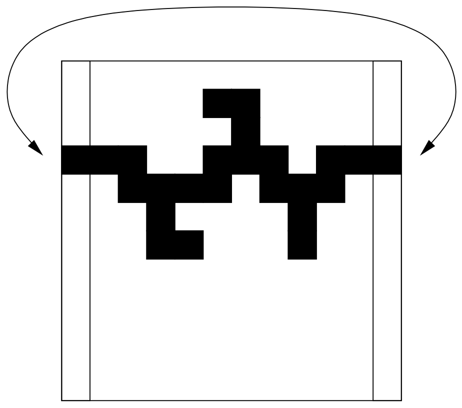


Figure 4.3: A sketch of periodicization, here in two dimensions for clarity. Black squares denote occupied sites, only one cluster is shown. The left and the right border of the lattice are connected to each other, in order to close the system periodically. When a specific cluster connects to itself during this process, this cluster spans the whole system; it is identified as the spanning cluster; in principle, more than one such cluster can form. Another way of defining an spanning cluster were to demand that it is present in both the left and right border, without touching itself; this would be suitable for open boundaries, as then periodicization is not necessary.

As the label array is modified by periodicization, we need to save a copy first, because we need the original array to continue the simulation for larger L afterwards. After the periodicization, the modified label array contains all cluster properties; these can be easily extracted and written out. The modified array can be discarded afterwards.

Using this method, it is possible to use fully periodic boundary conditions for intermediate L, although the lattice grows afterwards. Periodic b. c. mean a performance penalty and a doubling of memory consumption (as six faces instead of three need to be stored). But the same is also true of the traditional HK algorithm for periodic b. c., were two instead of one hyperplane need to be stored, still giving a factor of three in memory consumption.

4.2.3 Recycling of labels

Sites in the plane of investigation (for $d = 3$, the surface of the cube) point to labels in the auxilliary label structure, in order to indicate to which clusters these sites belong. Due to the way with which the HK algorithm works (both original and modified version), lots of these labels are indirect ones, which point to other labels. Other labels correspond to clusters that are hidden in the bulk of the lattice and no longer touch a surface. It is possible to get rid of these labels and thus free up precious space by using the method known as Nakinishi recycling [NaSt80].

In order to support fully periodic b. c., for each recycling all labels represented in A, B, C, X, Y, Z, AB, BC, CA, ABC, have to be taken into account. The method is fully analogous to that used for the traditional HK algorithm.

4.3 Results for three dimensions

All simulations were done using the R(471, 1586, 6988, 9689) pseudo-random number generator (cf. appendix D and [Ziff98]). The value of p_c is not known exactly for site percolation on the cubic lattice. Here it was chosen as $p_c = 0.311608$ [LoZi98a, Ball99]. Simulations were done for $p = 0.25, 0.3, p_c, 0.35$, in order to investigate behaviour below, at, and above the critical threshold. For each value of p, 1000 runs with different random numbers were made and used for averaging. Total required CPU time was 10000 hours on 2.2 GHz Opteron processors.

4.3.1 Cluster size distribution

As discussed above in 3.3.1, for the number n_s of clusters of size s in a lattice, we expect a distribution $n_s \propto s^{-\tau}$ right at p_c. To make analysis easier, we look at N_s, the number of clusters with at least s sites, as we did before:

$$N_s = \sum_{s'=s}^{\infty} n_{s'} \propto s^{-\tau+1} \tag{4.1}$$

We can determine an effective τ_{eff} by numerical differentiation of the data: $-\tau_{\text{eff}} + 1 = d(\log n_s)/d(\log s)$, for any L we choose. As for small s corrections to scaling and for large s finite-size effects produce deviations from the expected power-law, τ_{eff} is not constant, cf. fig. 4.4. For large L a plateau forms and we can determine $\tau = 2.1895(10)$; from the figure it becomes clear that larger L mean a more precise estimate for τ, showing once more the benefit of large lattices.

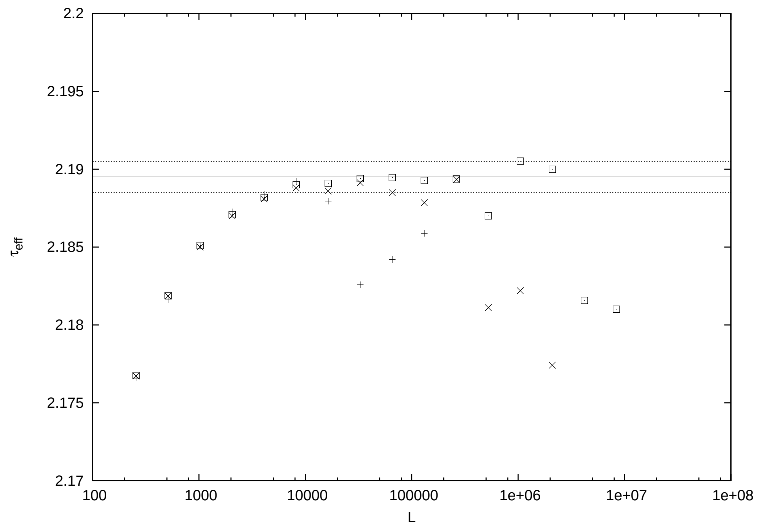


Figure 4.4: Effective exponent τ_{eff} determined from numerical differentiation of cluster size distribution. + are for $L = 1000$, $\times$ for $L = 2500$, and $\Box$ for $L = 5000$. Solid line is for $\tau = 2.1895$, dotted lines denote estimated error of ± 0.001. For smaller L, the plateau with τ_{eff}-values undisturbed by finite size effects is smaller. Due to the corrections to scaling for small s, a minimum size of $L \simeq 2500$ is necessary to estimate a reasonably precise value of τ.

When plotting $N_s \propto s^{-\tau+1}$ in a log-log-plot, we would expect a straight line with slope $-\tau + 1$. Deviations from the power law would be hard to detect due to the logarithmic scale, thus we plot $s^{\tau-1} N_s$ linearly on the y-axis. By varying τ until a flat plateau forms, we can confirm the previous estimate for $\tau = 2.1895(10)$. We can clearly see in fig. 4.5 the corrections to scaling for small s, which are the same for different L, as these are caused by the identical lattice spacing, and for large s, which differ for various L, as these are caused by the finite system size.

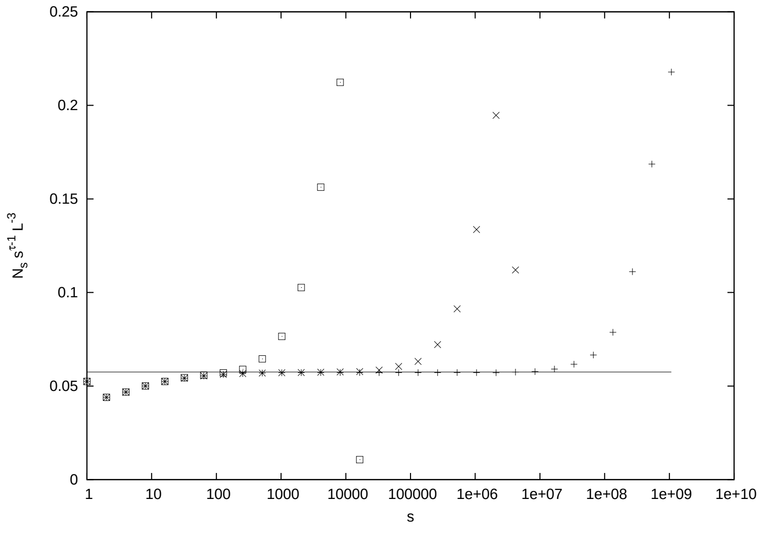


Figure 4.5: Binned cluster size distribution N_s for $L = 50$ (□), $L = 500$ (×), and $L = 5000$ (+). By plotting $N_s s^{\tau-1} L^{-3}$, all points should fall on the same line. For small s, we can see corrections to scaling, for large s the effects of the finite system size, even though the boundary conditions are fully periodic. For larger L, the finite size effects occur for larger s, as expected.

For $p < p_c$, n_s drops off sharply for small s, with s slowly growing for growing L (not plotted, as a plot would not convey useful information). The same is true for $p > p_c$, but there does an additional spanning cluster form (not plotted, too).

4.3.2 Number density

In subsection 3.3.4 we have studied number density $n(p, L)$ for large lattices, while here we can easily investigate the dependence of n on L. As all simulations for this chapter were done using fully periodic boundary conditions, we expect finite-size effects to be inverse proportional to the number of lattice sites in the bulk, i. e. $\propto L^{-3}$. This is visible in fig. 4.6, which yields a value of $n(p = p_c, L \to \infty) = 0.0524381(1)$, in agreement with $n_c = 0.05243812(9)$ from chapter 3.

4.3.3 Number of spanning clusters

A "spanning" cluster is a cluster that spans the whole system. When using periodic b. c., this cluster needs to wrap around, i. e. span the whole system and then periodically connect to itself. Such a cluster is often called "infinite", even for finite system sizes, as any cluster that spans the whole lattice would become infinitely large for $L \to \infty$; but there are also non-spanning clusters that grow $\propto L^D$, with $D > 0$, and therefore become infinitely large for $L \to \infty$ (cf. [JSA98], which uses the term "effectively infinite" for these clusters). We denote the number of spanning clusters at $p = p_c$ with n_{sp}.

According to theory, for $p < p_c$ no spanning cluster is expected, for $p = p_c$ about one with fractal properties (i. e. mass $\propto L^D$, with $0 < D < d$), and for

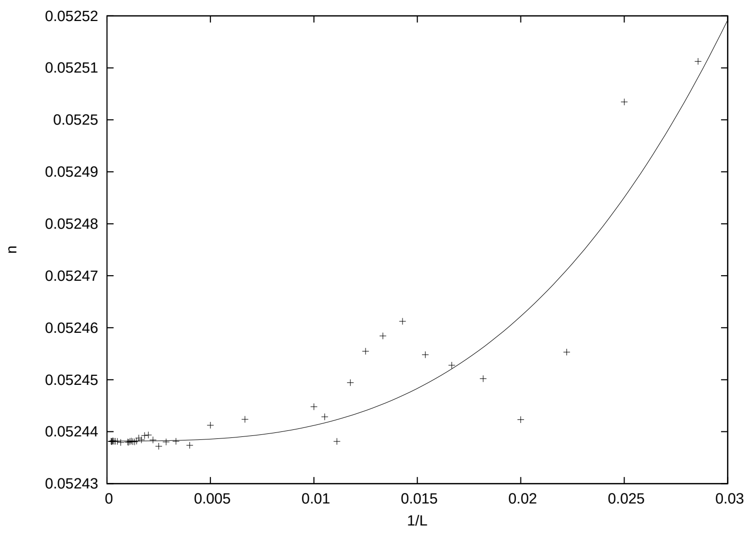


Figure 4.6: Number density $n(p, L)$ of clusters per lattice site, plotted against $1/L$. The intercept is the value for $L \to \infty$, $n(p_\mathrm{c}, \infty) = 0.0524381(1)$. The solid line is $\propto L^{-3}$; finite-size effects here are bulk effects, as for periodic b. c. no open boundaries exist in the system. Compare this figure to figure 3.13, where open boundaries lead to finite-size effects $\propto 1/L$. Although all data points are for averages over 1000 runs, statistical fluctuations are still visible. These become smaller for larger L, as the number density is a self-averaging property (for larger L, there are more clusters that can be averaged).

$p > p_\mathrm{c}$ one with bulk properties (i. e. mass $\propto L^d$, with d the Euclidean dimension of the lattice). The critical threshold p_c is well-defined only for infinite lattices, where the number of spanning clusters (when these clusters are really infinite) n_sp is exactly zero below p_c and exactly one above p_c. At $p = p_\mathrm{c}$, we expect $n_\mathrm{sp} = \mathrm{const}$, i. e. not depending on L, for $d < d_\mathrm{u}$ ($d_\mathrm{u} = 6$ being the upper critical dimension); for $d > d_\mathrm{u}$, we expect $n_\mathrm{sp} \propto L^{d-6}$, i. e. an infinite number of spanning clusters for $L \to \infty$, possibly depending on boundary conditions (cf. [FASC04], [Aize97]). In this chapter, we will only investigate $d = 2$ and $d = 3$, i. e. we expect a finite number of spanning clusters at the critical probability p_c. Below p_c, n_sp drops sharply to zero, above p_c it goes up sharply to one. p_c depends on the lattice size.

For simulations at p_c, here a value of $p = 0.311608$ was chosen. This value is near the true value, as can be seen in fig. 4.7, but a rather complex finite size behaviour is visible: For a fixed value p very near to p_c, the average number of spanning clusters decays with growing system size as $n_\mathrm{sp} \propto (\log L)^{-3/4}$; at first impression, one would conclude that this value p was chosen to small. By increasing p we can investigate this in detail, cf. fig. 4.8. In that figure, the change in p is quite small, at most $\Delta p = 0.9 \cdot 10^{-4}$ from the $+$ to the $\triangle$. Exactly the same random numbers were used for the different p-runs; because of this, some fluctuations in the data are correlated between various p.

Basically, for small lattice sizes (i. e. $L < 100$), small variations in p have no visible effect. For $L = 2$, n_sp starts at about 0.5, but drops to about 0.2 for $L = 100$, while small changes in concentration have no strong effects. For $L > 1000$, such small changes alter the behaviour qualitatively: For the rather large $p = 0.311700$, n_sp grows rapidly (roughly linear to L) to the expected $n_\mathrm{sp} = 1$, it even overshoots slightly above this value, meaning that on average more than one spanning cluster forms in a single lattice. For $p = 0.311660$, the same behaviour is visible, but the growth of n_{sp} is slower. For smaller p, only for very large lattices a slight increase is visible. Thus, finite system sizes have a very strong qualitative influence on the critical probabilites of percolation. For a fixed p, a change in L can make a difference between percolating and non-percolating, and the same is true for a fixed L and slight changes in p. The latter is well known (in fact, that is how phase-transition is defined for percolation), the former has been used for finite-size scaling. But the interdependence of p and L has to be investigated in more detail, as it shows a complex behaviour.

From figure 4.8 we can estimate $n_{\mathrm{sp}}(p \to p_{\mathrm{c}}, L \to \infty) = 0.15(3)$. [FSC03] finds a value of $n_{\mathrm{sp}}(p_{\mathrm{c}}, \infty) = 0.4$, but uses different boundary conditions, which are expected to change the result. For small L, deviations from the expected constant value are visible. These corrections could be logarithmic as shown in fig. 4.7, meaning that the true value for n_{sp} would be lower. For very small $p - p_{\mathrm{c}}$, only for huge lattices n_{sp} goes up to unity, as expected for $p > p_{\mathrm{c}}$. Because for this need of huge lattices to determine if p is below or above the critical value, the method of growing lattices seems not to be well suited for determing p_{c}; other methods, especially Newman-Ziff, are more promising in this regard.

For $p > p_{\mathrm{c}}$, n_{sp} is expected to be exactly equal one for infinite lattices. For finite lattices, n_{sp} becomes one when p is sufficiently larger than p_{c}, or the lattice becomes sufficiently large for a p near p_{c}.

When keeping p constant above p_{c}, an interesting behaviour can be seen with varying L, with three domains: for small L, n_{sp} is well below one, for intermediate L, n_{sp} is above one, and for large L, n_{sp} is exactly one. The domain for $n_{\mathrm{sp}} > 1$ is interesting, as theory predicts only one spanning cluster per lattice, but for finite system size more than one can form. This can be seen in fig. 4.9, where $p = 0.35$ (well above p_{c}) and many runs up to $L = 50$ were averaged. The anomalous surplus of spanning clusters decays exponentially after a certain L_{max}, where L_{max} depends on $\Delta p = p - p_{\mathrm{c}}$, growing with decreasing Δp. [Sen96] showed similar results, although there p was not kept constant, but was allowed to slightly exceed p_{c} until for a given set of random numbers one or more spanning clusters formed. The number of spanning clusters formed in that way was found to decline exponentially.

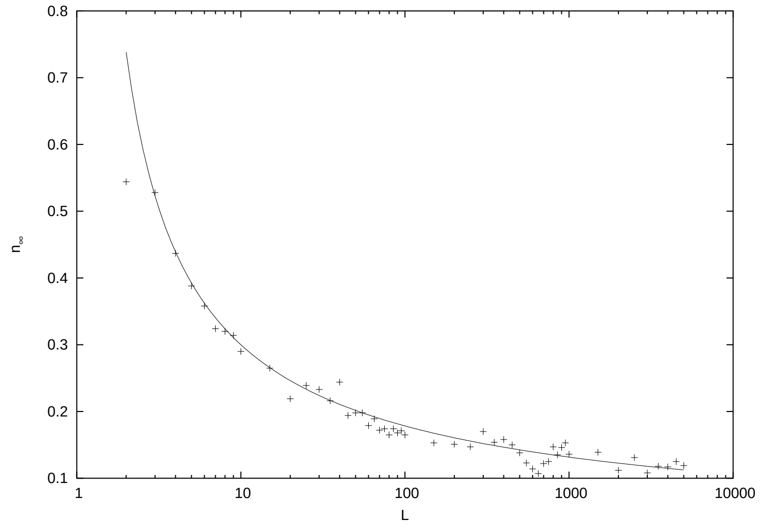
For $p < p_{\mathrm{c}}$, no spanning cluster forms for even small L. Only for very small L, sometimes an spanning cluster appears. For larger p this L becomes larger (not plotted).

4.3.4 Size of largest cluster

As mentioned above, the spanning cluster has fractal properties at $p = p_{\mathrm{c}}$, meaning that it grows $\propto L^D$, with D a non-integer value. From fig. 4.10 we can extract a value of $D = 2.52(1)$.

For $p > p_{\mathrm{c}}$, the spanning cluster obtains bulk properties, i. e. it grows $\propto L^{D_{\mathrm{b}}}$ with $D_{\mathrm{b}} = d = 3$; it no longer has fractal properties (not plotted).

For $p < p_{\mathrm{c}}$, we expect the largest cluster (which does not span the whole lattice) to grow $\propto \log L$ (cf. [MaHe84]). This is true for $p = 0.25$ (fig. 4.11) and $p = 0.3$ (not plotted), only slope and intercept are different. From fig. 4.11 another important effect can be seen: although it seems natural to determine the largest cluster s_{max} by taking the largest cluster of all independent runs, this would mean that no averaging happens, and thus the values of s_{max} are not statistically independent for different L: a large cluster in one single run would dominate the results. Thus, even here averaging is necessary to obtain meaningful results.



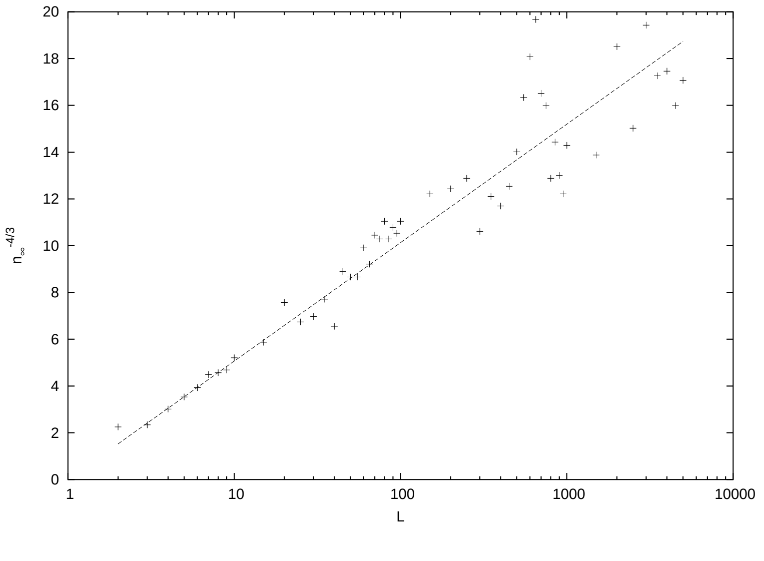
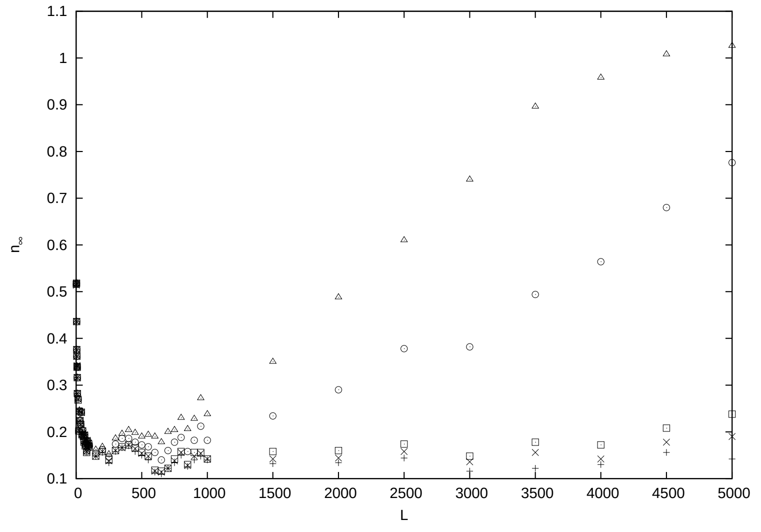


Figure 4.7: Number $n_{\rm sp}$ of spanning clusters, at $p = 0.311608 \simeq p_{\rm c}$, averaged over 1000 runs, depending on lattice size L. Right at $p_{\rm c}$ we expect $n_{\rm sp} = $ const, i. e. to be independent of L. As here $n_{\rm sp}$ depends on L, the value $p = 0.311608$ chosen for this thesis is slightly to small; sufficiently large lattices have to be simulated in order to see this, even when using periodic b. c. For the number of spanning clusters we get $n_{\rm sp} = 0.3(\log_{10} L)^{-3/4}$.



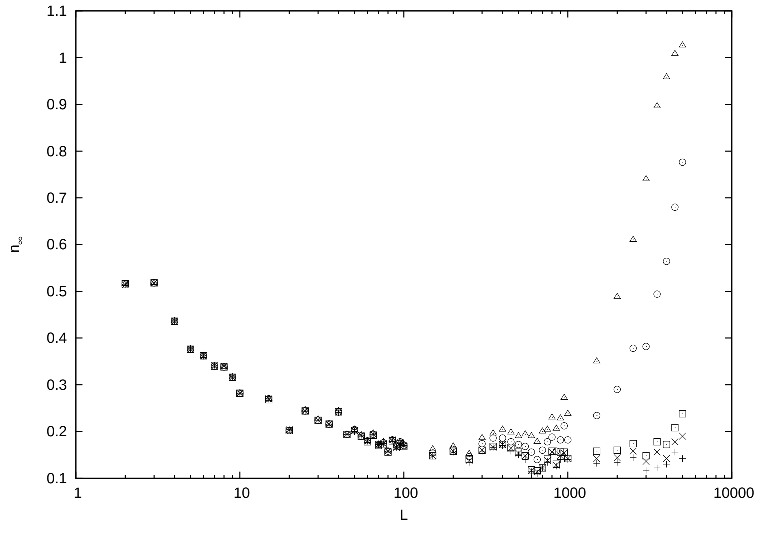


Figure 4.8: Number n_{sp} of spanning clusters, averaged over 500 runs, depending on lattice size L, at $p = 0.311610$ ($+$), $p = 0.311615$ ($\times$), $p = 0.311620$ ($\square$), $p = 0.311660$ ($\circ$), $p = 0.311700$ ($\triangle$). Data for upper and lower plot are the same, the difference is that the lower has logarithmic x-axis, while the upper has a linear one.

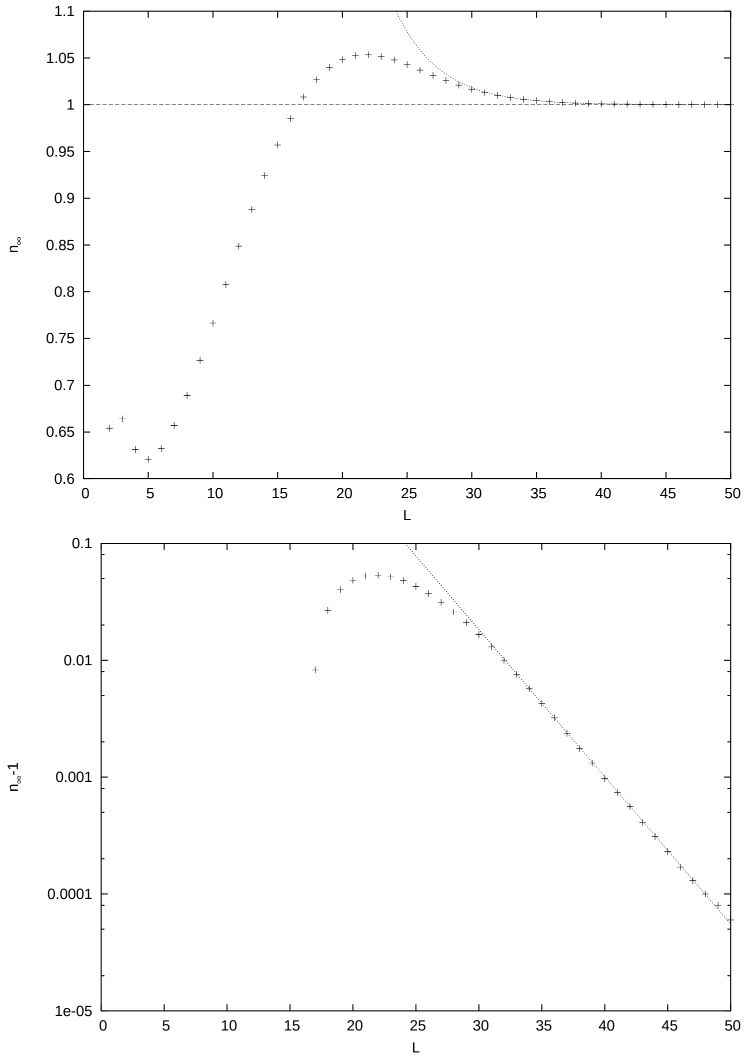


Figure 4.9: Number n_{sp} of spanning clusters (i. e. that wrap around the system) for $p = 0.35$, depending on lattice size. Theory predicts that only one such cluster forms. However, for small lattice sizes a complex finite size behaviour can be seen: for $L < 17$, less than one spanning cluster forms on average, for $L > 17$ more than one cluster forms, with a maximum at $L = 22$, afterwards n_{sp} drops to 1 exponentially. The solid line corresponds to $1 + e^{-0.29(L-16.2)}$. For this plot, 10^7 runs of up to $L = 50$ have been averaged.

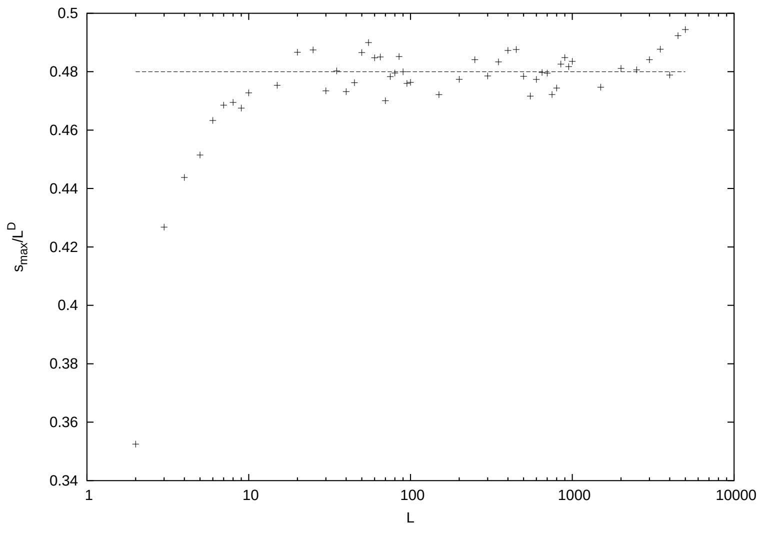


Figure 4.10: Size $s_{\max}$ of the largest cluster (which is the spanning cluster, if it forms), at $p = p_c$, scaled by L^D, depending on simulated lattice size. The cluster grows $\propto L^D$, with a fractal dimension of $D = 2.52(1)$. Thus, by scaling, we get a straight line (the line visible in the plot serves as guide to the eye). For very small lattice sizes L, corrections to scaling can be seen.

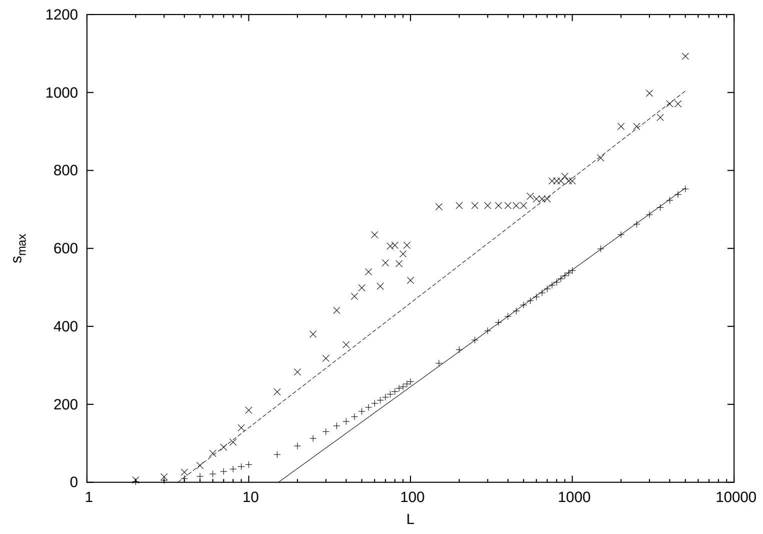


Figure 4.11: Size $s_{\max}$ of the largest cluster, at p=0.25, depending on simulated lattice size. $+$ are for $s_{\max}$ averaged over 1000 runs, $\times$ for the maximal $s_{\max}$ of 1000 runs, which has stronger fluctuations (as the maximum is determined by single clusters; in fact, as no averaging is done, the results for L and $L + 1$ are *not* statistically independent). The solid line corresponds to $300 \log_{10}(L) - 355$, the dashed line to $320 \log_{10}(L) - 180$. Thus, for $p < p_c$, the largest cluster grows $\propto \log(L)$.

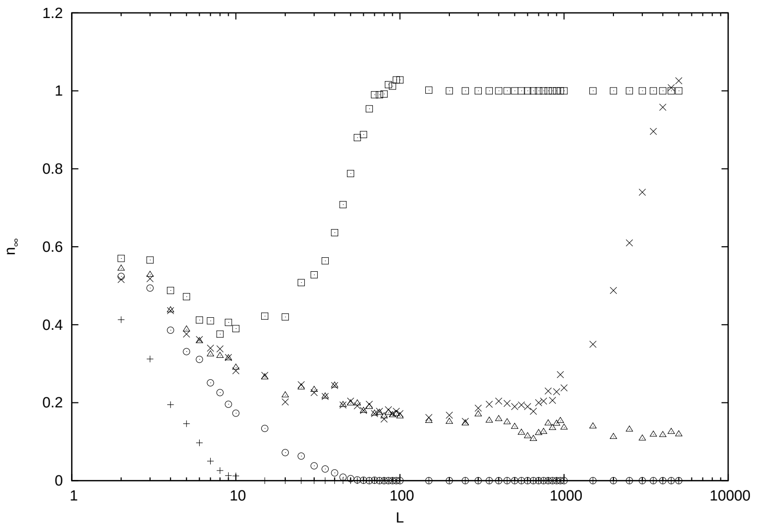


Figure 4.12: Number of spanning clusters n_{sp}, in three dimensions, for $p = 0.25$ ($+$), 0.3 ($\circ$), $0.311608 \simeq p_{\mathrm{c}}$ ($\triangle$), 0.3117 ($\times$), 0.32 ($\square$). The real value of p_{c} seems to be very near the literature value of 0.311608, as there no bending upwards is visible for the L range presented here; the value $p = 0.3117$ is clearly too large. We can extrapolate a value of $n_{\mathrm{sp}}(p \to p_{\mathrm{c}}, L \to \infty) = 0.15(3)$.

4.4 Results for two dimensions

The algorithm was also adapted to $d = 2$. Simulations were done for $p = 0.5$, $p = 0.58$, $p = 0.5927464 \simeq p_{\mathrm{c}}$ [NeZi00], $p = 0.60$, and $p = 0.65$, with 1000 runs each. Simulated system size was $L_{\max} = 2.5 \cdot 10^5$. Used random number generator was the same as for three dimensions, $\mathrm{R}(471, 1586, 6988, 9689)$.

For $d = 2$, several critical exponents are known exactly, e. g. the exponent $\tau = 187/91$ for the cluster size distribution $n_s \propto s^{-\tau}$, or the fractal dimension of the spanning cluster at $p = p_{\mathrm{c}}$, $D = 91/48$. The numerical results for simulations done at p_{c} show these expected values, which further supports that the exact solutions are correct, and which shows that the used random number generator has a sufficiently high quality.

The behaviour of the number density is qualitatively the same as for three dimensions, only the finite-size corrections are $\propto L^{-2}$ (instead of $\propto L^{-3}$ for $d = 3$, as is expected for bulk properties; compare figs. 4.6 and 4.13). The value for $n(p_{\mathrm{c}}, L \to \infty) = 0.0275979(4)$ is compatible with the value $n_{\mathrm{c}} = 0.027597857(2)$ obtained in chapter 3.

For the number n_{sp} of spanning clusters, we expect it to drop off sharply with increasing L below p_{c}, to increase sharply to $n_{\mathrm{sp}} = 1$ with L above p_{c}, and to reach a constant value for $p = p_{\mathrm{c}}$. Figure 4.14 displays the results from the simulations, which verify our expectations. For $p = p_{\mathrm{c}}$, we get a value of $n_{\mathrm{sp}}(p_{\mathrm{c}}, \infty) = 0.52(1)$. Here, in $d = 2$, n_{sp} approaching a constant value can be seen more easily than in $d = 3$, as we can simulate lattices with much larger linear dimension L.

Above p_{c}, there is a region were $n_{\mathrm{sp}} > 1$, but the effect is weaker than in three dimensions, in accordance with [Sen96]. While in two dimensions any cluster which percolates both horizontally and vertically would block any other cluster from percolating, in three dimensions several clusters could coexist which percolate in all three dimensions. Thus $n_{\mathrm{sp}} > 1$ for $d = 2$ means that sometimes two clusters form which percolate both either horizontally or vertically, but not in both directions at

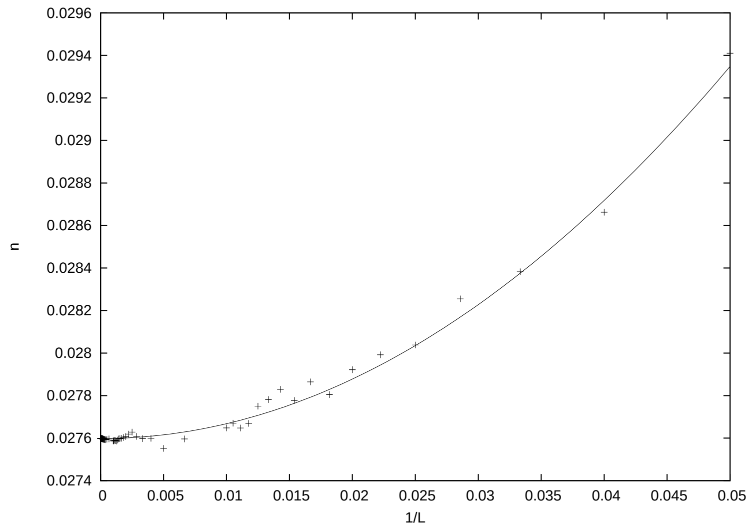


Figure 4.13: Number density $n(p, L)$ of clusters per lattice site, plotted against 1/L. The intercept is the value for $L \to \infty$, $n(p_\text{c}, \infty) = n_\text{c} = 0.0275979(4)$. The solid line is $\propto L^{-2}$.

the same time. This is a finite-size effect, as it vanishes for growing L.

4.5 Speed of simulations

The following table shows the minimal runtimes of five runs for two- and three-dimensional percolation, for different algorithms. Fixed L means the traditional HK algorithm, growing L the changed algorithm presented in this chapter. Simulated system sizes were $L = 2.5 \cdot 10^5$ for $d = 2$ and $L = 5 \cdot 10^3$ for $d = 3$, simulations were done right at the critical threshold p_c (values for two and three dimensions, respectively). For the traditional HK approach, the cluster statistics are accounted only once, after simulating the full lattice. For the growing lattices, cluster statistics are accounted multiple times, for each L_i we are interested in. For comparison, growing lattices were simulated with only one accounting of clusters after simulating the full lattice, for estimating the impact of multiple accountings. The times are in seconds for 2.2 GHz Opteron processors.

	$d = 2$	$d = 3$
fixed L, single accounting	2234 s	4402 s
growing L, single accounting	2194 s	4460 s
growing L, multiple accountings	2161 s	4531 s

For both two and three dimensions, the speed of the traditional and the new approach are roughly equal. For $d = 2$, the new method is even faster, by 3 percent, while for $d = 3$, it is slower by 3 percent. The impact of multiple cluster accountings is marginal: for $d = 3$, it increases runtime by less than 2 percent, while for $d = 2$ it even lowers the runtime by less than 2 percent.

While we would expect both approaches to be of the same speed, as the same number of sites L^d are investigated, using the same core part for investigating a single site, the small variations of runtime can be explained by the machine architecture. For $d = 3$, growing lattices mean a small performance penalty, as up to three times the number of sites have to be stored for the new approach to hold the

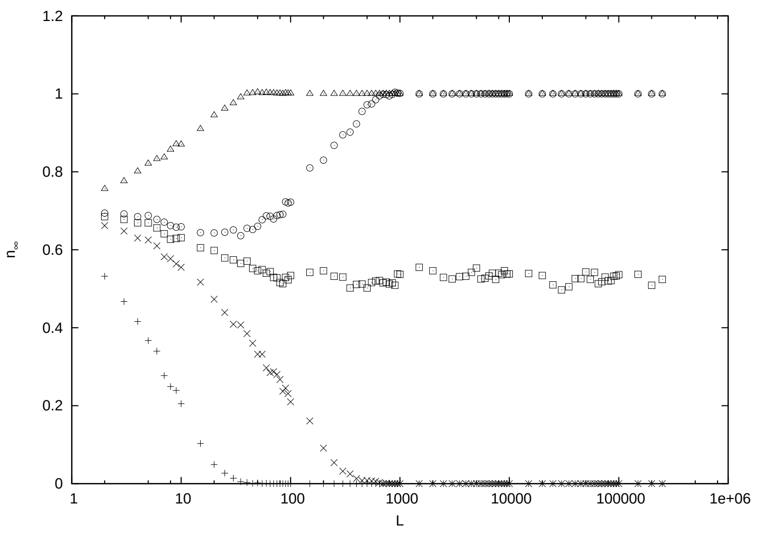


Figure 4.14: Number of spanning clusters n_{sp}, in two dimensions, for $p = 0.5$ ($+$), 0.58 ($\times$), $p = 0.5927464 \simeq p_{\mathrm{c}}$ ($\square$), 0.6 ($\circ$), 0.65 ($\triangle$). $p = 0.5927464$ is a very good value for p_{c}, giving a flat line for $n_{\mathrm{sp}}(L)$, with $n_{\mathrm{sp}}(p_{\mathrm{c}}, L \to \infty) = 0.51(2)$.

current plane of investigation; for $L = 5000$, these cannot be hold in the cache of the processor, and must be fetched from main memory. This higher memory traffic slows down the computation. The same is true for the multiple accountings: these are additional computations that take up time; when this effort is small compared to simulating the whole lattice, the performance penalty is correspondingly small.

For $d = 2$, the situation is more complex: the new approach is slightly faster than the traditional one, although this seems counter-intuitive at first. Again, this can be explained by machine architecture. While ultimately the new approach needs to store two times as much sites for the line of investigation, it starts with less sites for small L (as opposed to the traditional way, where always L sites have to be stored). For small L, this line fits entirely into the cache of the processor, meaning significantly higher speed. This offsets the performance penalty for larger L, where the new approach means more memory traffic. The accountings also have a small positive effect on runtime: While they mean more amount of computation, for each accounting all labels in the plane of investigation are reclassified (afterwards pointing directly to root labels, no longer through indirect labels); then, looking up labels becomes faster. Even more, the label array is compacted, meaning more labels fit into the cache.

It is probable that for higher dimensions, i. e. $d > 3$, the performance penalty of the new approach grows, as up to d times the amount of sites have to be stored for the hyperplane of investigation and generally, the hyperplane of investigation becomes larger in comparison to overall system size (L^{d-1} to L^d), meaning more memory accesses and weakening the effects of the processor cache.

4.6 Summary and outlook

It is possible to modify the Hoshen-Kopelman algorithm in order to obtain not only results for an $L_{\max}^d$ lattice in one simulation run, but also results for intermediate L_i with $L_i < L_{\max}$. Compared to the original HK algorithm, there is a higher memory consumption (a constant factor of d for storing the lattices sites) and at most a small performance penalty (for $d = 2$, even a performance gain is possible). The modified algorithm is very useful in order to study the dependence of percolation properties on the lattice size.

Following results were obtained for two and three dimensions (τ and D for $d = 2$ are known exactly):

	$d = 2$	$d = 3$
τ	187/91	2.1895(10)
$n_{\rm sp}$	0.52(1)	0.15(3)
$n_{\rm c}$	0.0275979(4)	0.0524381(1)
D	91/48	2.52(1)

Due to fully periodic boundary conditions, the size dependence of the number density is $[n_{\rm c}(L) - n_{\rm c}(\infty)] \propto L^{-d}$ in d dimensions, as expected. For the number of spanning clusters $n_{\rm sp}$, the finite-size behaviour depends delicately on the value chosen for p, even small deviations from $p_{\rm c}$ can have strong influences. Slightly below $p_{\rm c}$, $n_{\rm sp}$ decays slowly to zero for growing L, slightly above $p_{\rm c}$, $n_{\rm sp}$ goes up, even above unity for some intermediate L-values (meaning that more than one spanning cluster forms on average), and then decays exponentially to unity. For a p-value chosen very near the true $p_{\rm c}$, $n_{\rm sp}$ reaches a fixed value $0 < n_{\rm sp} < 1$, confirming the theory for $d < 6$.

A natural next step for research would be to move away from two and three dimensions and to adapt the algorithm to higher d. This means dissecting a d-dimensional hypercube into $(d-1)$-dimensional hyperfaces, $(d-2)$-dimensional hyperedges, and so on.

Parallelization by using domain decomposition is apparently very hard, as the size of the domains would be changing. This would only be necessary in order to simulate huge lattices, where the L-dependence might not be very interesting. As several runs always need to be averaged, in order to obtain statistically independent data, replication is always a viable parallelization strategy.

Chapter 5

Critical Behaviour of the Ising Model

5.1 Rationale

Despite most of this thesis being occupied with percolation theory, many of the computational methods have equivalents in other fields of statistical physics. One of the most important models in statistical physics is the Ising Model. Even more, there is an interesting connection between percolation and the Ising model: both are special cases of the q-state Potts model (cf. [KaFo69]). Furthermore, both cases have been investigated using Monte Carlo methods for a long time. Although a computational implementation of the Ising model is quite different from percolation, the method of domain decomposition for parallelization is fruitful for both. Because of that, in this chapter a parallel implementation of the Ising model will be presented, along with results from world record simulations.

5.2 Introduction

The Ising model is a simple model for ferromagnetism, initally suggested by W. Lenz [Lenz20] and later investigated in detail for the one-dimensional case by E. Ising [Isin25]. In one dimension, it shows no phase transition, but in $d > 1$ dimensions there is a second-order phase transition with interesting critical bevaviour.

The Hamiltonian for an Ising magnet is in its broadest form

$$\hat{H} = -\frac{1}{2}\sum_{i,j} J_{i,j} s_{i,\mathrm{z}} s_{j,\mathrm{z}} - \sum_i H_\mathrm{z} s_{i,\mathrm{z}}$$

. The $J_{i,j}$ are exchange integrals which describe the coupling between two spins s_i, s_j, and each spin can have values of $s = \pm 1$; H is an external magnetic field.

For simulations carried out here, we neglect any external field and simplify the Hamiltonian, yielding an energy for the lattice of

$$E = -J \sum_{\langle i,j \rangle} s_i s_j \tag{5.1}$$

The summation is done only over nearest neighbours, i. e. only these interact with each other. J is the coupling constant, which is $J > 0$ for a ferromagnet, favouring parallel spins over anti-parallel ones (an anti-parallel pair has an energy deficit of $2J$ over a parallel one). Spins are called up and down, depending on their sign.

In $d > 1$ dimensions, the Ising model shows a phase transition. The low-temperature phase is ordered, with a non-vanishing spontaneous magnetization $M = \sum_i s_i$. The high-temperature phase is disordered, and the magnetization without an external field vanishes, $M = 0$. Figure 5.1 shows the two phases, and the system right at the phase transition (which will be the main topic of this chapter).

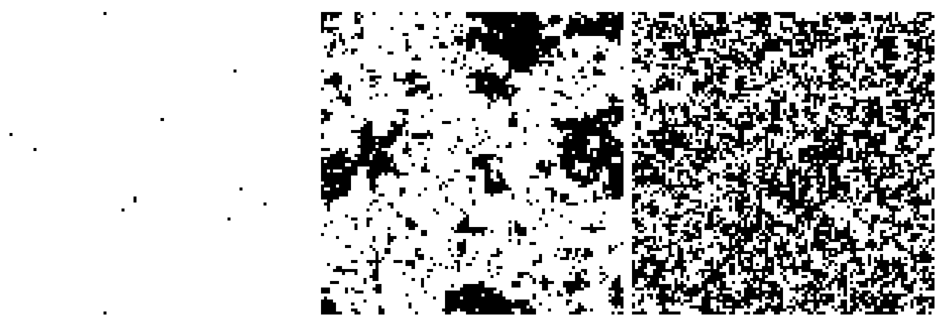


Figure 5.1: An 100×100 lattice for different temperatures, each 1000 timesteps after initialization (initialization is all spins up). White pixels correspond to up spins, black pixels to down spins. Left is for $T = \frac{1}{2}T_\mathrm{c}$ (low-temperature phase), middle for $T = T_\mathrm{c}$ (exactly at the phase transition), right for $T = 2T_\mathrm{c}$ (high-temperature phase). For $T < T_\mathrm{c}$, an ordered phase is visible (spontaneous magnetization) with only small fluctuations of spins. These fluctuations grow stronger with increasing temperature. For $T > T_\mathrm{c}$, the phase is strongly disordered. For $T = T_\mathrm{c}$, there is disorder (no single domain) and order (groups of parallel spins tend to form clusters; these are self-similiar, i. e. fractal).

Although the two-dimensional Ising model was solved exactly by Onsager in 1944 [Onsa44] and much work has been done in this field over the last decades, some questions still remain open. One is the dynamical critical behaviour, cf. figure 5.2.

When we take a lattice with initially all spins up, right at the Curie point T_c, with $J/k_\mathrm{B}T_\mathrm{c} = \frac{1}{2}\ln(1+\sqrt{2})$, the magnetization M decays with time as $M \propto t^{-\beta/\nu z}$, where $\beta = 1/8$ is the exponent for the spontaneous magnetization and $\nu = 1$ is the exponent for the correlation length; both are known exactly.

Two main suggestions have been made for the value of z in Glauber kinetics (explained below): one is $z \simeq 2.167$ asymptotically, with simple power law behaviour ([Kall84], [Stau97], [Stau99]), the other is $z = 2$ with logarithmic corrections to the power law behaviour. The latter was suggested by Domany [Doma84] and later by Swendsen [Swen99]. Lots of work has been done in order to rule out one of these assumptions (e. g. [NiBl96], [NiBl00], [Arju03]), but with no final result.

We want to test these suggestions here using numerical data obtained by using the supercomputer JUMP at the Research Center Jülich. We compare these with older data.

5.3 Computational method

For both percolation and the Ising model Monte Carlo methods are used. For percolation simple sampling is utilized: configurations are produced by occupying sites randomly with a fixed probability, and not taking any Boltzmann factors into account. Even more, there is no time evolution; after generating a configuration, it is accounted.

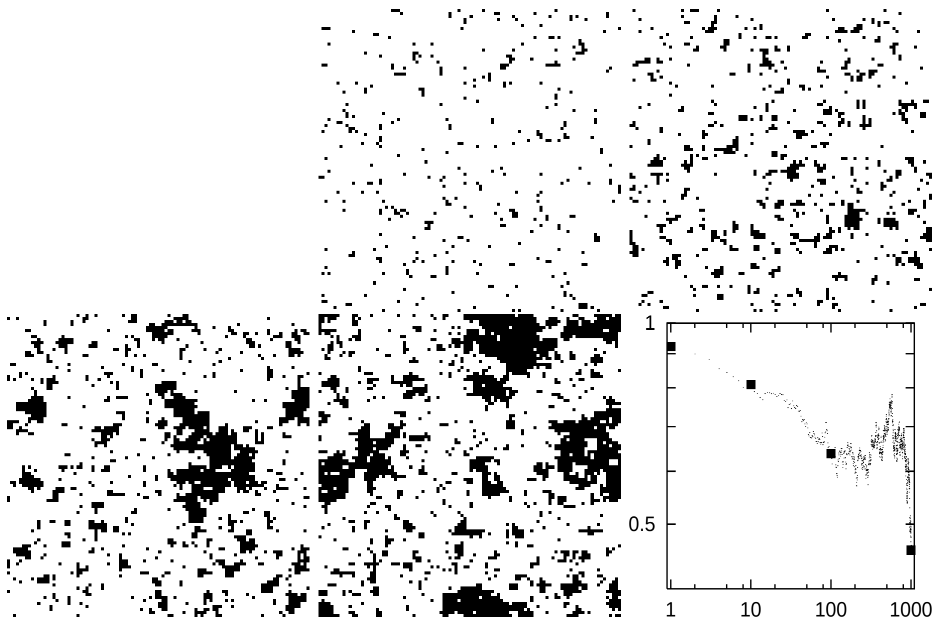


Figure 5.2: An 100×100 lattice for different timesteps (from left to right and top to bottom) $t = 0$, 1, 10, 100, 1000, right at the critical temperature T_{c}. White pixels represent spin up, black pixels spin down. The lattice is initialized with all spins up, then the decay of magnetization is simulated. Initially isolated down spins form larger clusters. After very long times the resulting magnetization tends towards $M = 0$ (meaning same number of up and down spins). The sixth panel shows the time-dependence of magnetization (in a log-log-plot), with black boxes marking the data points corresponding to the five other panels. Visible are strong fluctuations in the magnetization; these are caused by the finite (and rather small) size of the system.

For the Ising model importance sampling is used: configurations with a higher Boltzmann weight are realized with higher probability. This is achieved using the Monte Carlo method. Starting from a configuration, a small change in it is contemplated, e. g. flipping a single spin. The energy change ΔE caused by this is calculated and then the spin flip is committed with a probability W that depends on $\mathrm{e}^{-\beta\Delta E}$ (here β is the inverse temperature, not to be mistaken with the exponent β for spontaneous magnetization; unfortunately, nomenclature in physics is sometimes ambigous). Several ways of calculating this transition probability are known; frequently used are Metropolis kinetics, $W_{\mathrm{M}} = \min(1, \mathrm{e}^{-\beta\Delta E})$ (cf. [MRRTT53], a groundbreaking paper which pioneered the use of Monte Carlo simulations in physics) and Glauber kinetics $W_{\mathrm{G}} = \mathrm{e}^{-\beta\Delta E}/(1 + \mathrm{e}^{-\beta\Delta E})$ (cf. [Glau63]). The difference between these can be seen in figure 5.3. Although both obey ergodicity and detailed balance, for non-equilibrium states the dynamic behaviour can differ; to obtain results comparable to literature, we use Glauber kinetics here. See [LaBi05, subsection 4.2.1] for a detailed discussion of Metropolis vs. Glauber kinetics.

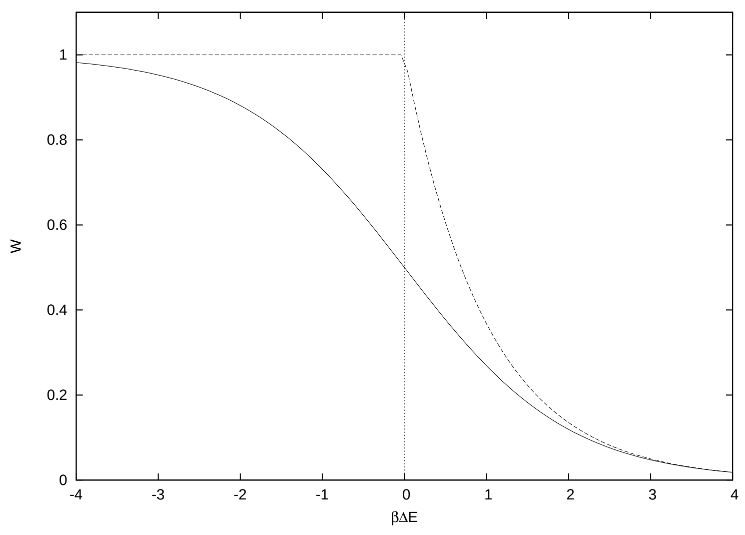


Figure 5.3: Transition probability W for a spin flip with associated energy change ΔE, at inverse temperature β, for Metropolis kinetics (dashed line) and Glauber kinetics (solid line). For $|\beta\Delta E| \gg 1$, $W_{\mathrm{G}} \simeq W_{\mathrm{M}}$.

By using the importance sampling Monte Carlo method, we count configurations which have thermodynamically a high probability of realization, thus resulting observables are representative.

These spin flips can be interpreted as a dynamical process: spin flips in a real lattice caused by thermal fluctuations. This is done in this chapter for simulating the decay of magnetization right at the phase transition.

5.3.1 Parallelization

In order to simulate huge lattices for many timesteps, we need to parallelize the program to obtain higher speed and more memory. Parallelizing the Ising model is easier than for percolation, as it has a higher locality. Ising-spins interact only with nearest neighbours, whereas clusters in percolation can span the entire system; when decomposing the system into domains, a single cluster can span several domains, being highly non-local.

Figure 5.4 shows a sketch of how the algorithm works: In order to determine if a single spin should be flipped or not, the spin itself and its four neighbors (or more generally $2d$ neighbours in a d-dimensional hypercubic lattice) are examined, giving rise to the stencil operator in the figure. The number of anti-parallel spins is counted, as this determines the energy change of the system if the spin were flipped. Using this number as an index, we can lookup the resulting Boltzmann-weight in a table (pre-calculated for speed reasons), and by using a random number we determine if we flip the spin or not.

In the left half of figure 5.4 a system with open boundaries is shown. When examing spins on the border of the lattice, we miss one or two neighbouring spins. This problem can be solved by introducing periodic boundary conditions: When the stencil operator reaches the border of the lattice, it is continued on the opposite side. This way, we have no longer open borders in the system, which would seriously influence our results.

Technically, periodic b. c. are implemented by adding buffer planes around the lattice, and copying the results of one border (real spins) to the opposite buffer (virtual spins), cf. the right half of figure 5.4.

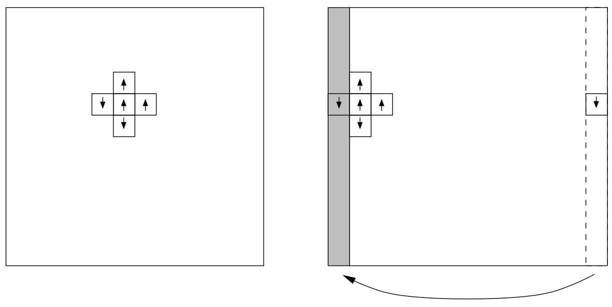


Figure 5.4: Stencil operator and periodic boundary conditions. When calculating a spin at the left border of the lattice, the stencil operator would extend one site beyond the lattice. For implementing periodic boundary conditions, the lattice is extended by buffer planes, and here the rightmost column (separated by a dashed line) is copied into the left buffer column (shaded with grey). For updating to be physically correct, it is important always to use newest values of spins, i. e. after re-calculating a spin in the rightmost column, it needs to be copied to the buffer column directly.

These buffer planes are the first step for parallelizing the Ising-model. By decomposing the lattice into several domains, e. g. strips, as shown in figure 5.5, we can distribute the lattice over several processors. The interior spins in the lattice (i. e. not touching the border of the domains) can be calculated locally by each processor, independently of each other. This is possible due to the high locality of the Ising model. Only spins on the border need to know the values of the neighbouring spins from another processor. We use the buffer planes originally introduced for the periodic b. c. for this purpose, and copy the borders of one domain to the buffer of the neighbouring domain using suitable MPI-calls.

5.3.2 Multi-spin coding

Each spin can be in one of two states: $s_i = +1$ or $s_i = -1$ (up or down respectively). Storing a single spin in an entire computer word would be a waste of precious memory, a single bit would suffice. As we want to simulate huge lattices, we store 64 spins in a single 64-bit word in memory. By putting several spins in one computer

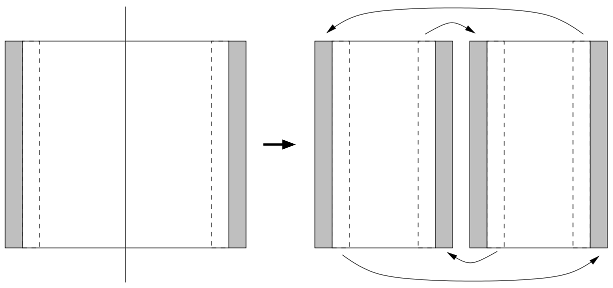


Figure 5.5: Periodic boundary conditions lead naturally to domain decomposition into strips. When dividing the lattice into N strips, we add buffer columns to each strip. The strips are arranged in a circular fashion; the left buffer column of any strip gets its values from the rightmost column of the left neighbour strip, and vice versa. The difference to a sequential implementation is that copying these buffer columns is done via message passing directives.

word, we can simulate several spins in parallel by using simple integer operations. For example, if up spins are represented by `1` and down spins by `0` (or vice versa), we can determine anti-parallel spins by the simple machine-instruction `xor` (exclusive or). If we occupy four bits per spin, we can handle 16 spins in parallel in one 64-bit word.

Even more, we can sum up the number of anti-parallel spins with a simple integer-add, if we take care that the sum never overflows into a neighbouring field. This is shown in figure 5.6. In d dimensions, on a hypercubic lattice, each spin has $2d$ neighbours, meaning it can have at most $2d$ anti-parallel neighbouring spins. For up to $d = 7$, this number fits into four bits without overflow, thus by occupying four bits per spin, we can handle 16 operations in parallel with simple integer machine-instructions.

Unfortunately, for determining if we flip a spin or not, we have to extract the fields from the machine-word serially, doing a table lookup for each (to determine the Boltzmann-weight associated with a spin flip), and generating a random number for the final decision (to flip or not to flip, that is the question). This serial part reduces the gain of the multi-spin technique, but nearly an order of magnitude in speed gain is possible.

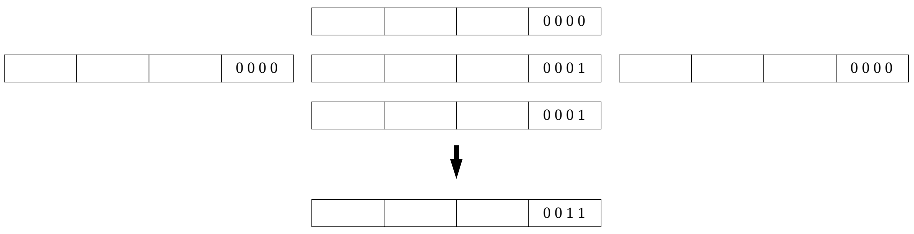


Figure 5.6: Counting the number of anti-parallel spins in a machine-word in parallel. By doing four `xor`s and three additions, we can handle 16 spins in parallel when using a 64-bit machine-word.

Another important factor for increasing speed is compression of spins in memory. Instead of occupying a full word per spin, or even four bit per spin, we want to store each spin in one bit only in main memory, and extract it to four bits just for the calculation. This is done by decompression before and compression after calculating spin flips. Decompression is achieved by shifts and a bitwise `and` with suitable bit masks, compression by shifting and `or`ing the values together; both very efficient operations. This is shown in figure 5.7. This compression not only uses memory in a very efficient way, but also reduces consumption of memory bandwidth (between processor and main memory), alleviating a bottleneck of most modern microprocessors.

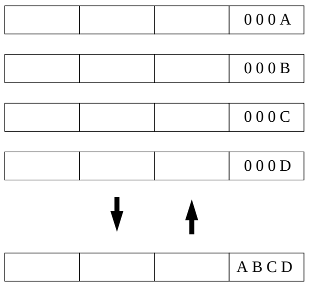


Figure 5.7: Compression is done by shifting one word by three bits, one by two, one by one, and shifting one not at all. These shifted words are `or`-ed together, giving one compressed word for four uncompressed ones. The reverse operation is achieved by `and`-ing the compressed word with four different bitmasks (each has one of four bits in the bitfield set), and then shifting the resulting words by three, two, one, or zero positions.

By using these methods, we have a second layer of parallelization in our program: we utilize single arithmetic and logical units (ALU) of the processor in a parallel fashion, in an SIMD-approach (single instruction, multiple data). Combining this with MIMD-parallelization (multiple instructions, multiple data) via domain decomposition yields a very efficient algorithm.

Simulating the Ising model by manipulating several spins at once in a machine word was first invented by Friedberg and Cameron for very small lattices (cf. [FrCa70]). Later Rebbi re-invented this method (cf. [CJR79]) and popularized it as “multi-spin coding”. It has proven to be a very useful technique.

5.4 Simulations

Data was generated by initializing a lattice with all spins up and then doing a Monte Carlo simulation of the Ising model with Glauber kinetics. To obtain higher speed, multi spin coding and parallelization (via MPI) were used. Speed on one 1.7 GHz Power4+ processor was roughly 160 million sites per second. Up to 512 processors were used in parallel for the largest simulations.

Although the supercomputer JUMP is rather large, there are some restrictions on the size of lattices that can be simulated. Here, $L = 1.5 \cdot 10^5$ and $L = 2 \cdot 10^6$ were chosen (the old world record for the two-dimensional Ising model was $L = 10^6$ [Link95], [Stau99]), with periodic boundary conditions. Thus finite size effects should be negligibly small. Several independent runs were done for $L = 1.5 \cdot 10^5$ for averaging, 50 runs each for the random number generators $x_{n+1} = 13^{13} \cdot x_n \bmod 2^{63}$ (called LCG(13^{13})) and the 64-bit implementation of Ziff’s four-tap generator R(471,1586,6988,9689) (cf. appendix D and [Ziff98]). For $L = 1.5 \cdot 10^5$ the simulations were done up to 6000 timesteps (full sweeps through the lattice). For the larger lattices, only considerably smaller times were possible, due to restrictions in computing time. For investigating finite-size effects, lattices with $L = 5 \cdot 10^4$ were simulated with LCG(16807), again averaging over 50 runs.

The effective exponent z can be determined from the $M(t)$ data by numerical differentiation: $-1/8z = d(\log M)/d(\log t)$.

5.5 Results

Even when averaging $M(t)$ over several independent runs, fluctuations are visible when calculating the effective $z(t)$. Thus each point in Figs. 5.8–5.11 represents many $z(t)$; these points were generated by dividing the data for $z(1 \ldots 6000)$ into several intervals and then doing a least squares fit in each. Each point is the central point of the fit in the interval. The errorbars for z (not shown in the plots for better legibility) are of the order of the symbol size for short times and grow to up to ± 0.03 for long times. The new data, especially for the large systems, allows for a better fit of the Swendsen suggestion. The new fitted curve has a maximum at about $t \simeq 1700$ ($1/t \simeq 6 \cdot 10^{-4}$).

The last point for $L = 1.5 \cdot 10^5$, corresponding to the interval $t = 3000 \ldots 6000$, is subject to strong fluctuations and thus doubtful. Unfortunately, this is the most interesting data point. Nevertheless, a trend is visible: for larger times, the critical exponent seems to go up, not down, thus being in contradiction to the Domany-Swendsen suggestion.

This could also be due to finite-size effects: for $L = 5 \cdot 10^4$, the effect of increasing z seems to be stronger (cf. Fig. 5.11), but more simulations would be needed for confirmation.

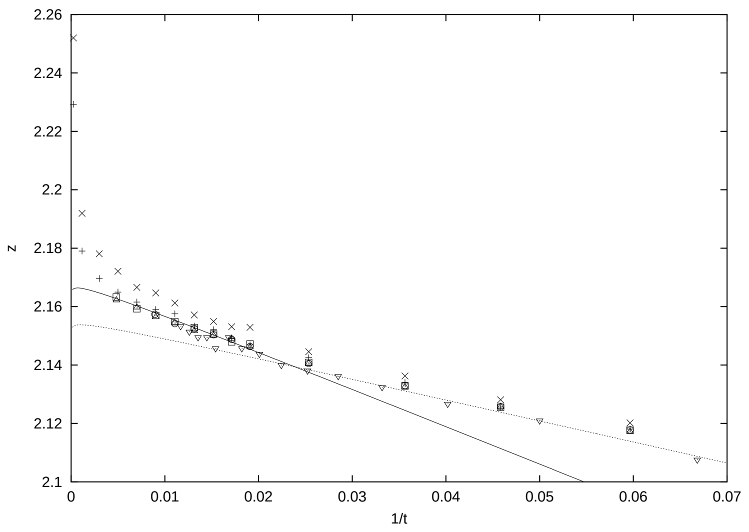


Figure 5.8: Monte Carlo data for the two-dimensional Ising model at the Curie point. $+$ are for $L = 1.5 \cdot 10^5$ with LCG(13^{13}) generator, averaged over 50 runs, $\times$ for $L = 1.5 \cdot 10^5$ with R(471,1586,6988,9689) generator, averaged over 50 runs, $\square$ for $L = 2 \cdot 10^6$ with LCG(13^{13}), $\circ$ for $L = 2 \cdot 10^6$ with R(471,1586,6988,9689), $\triangle$ for $L = 2 \cdot 10^6$ with LCG(16807), $\bigtriangledown$ for $L = 10^6$ with LCG(16807), data by Stauffer [Stau99] (large systems one run each). The lines represent the Swendsen suggestion for a fit, $\beta / \left(\frac{t}{t - t_0} \cdot \left(\frac{1}{2\beta} - \frac{c}{1 + c \cdot \log(t - t_0)} \right) \right)$, with $c = 0.005$ and $t_0 = 0.6$ for the solid line (new fit parameters) and $c = 0.004625$ and $t_0 = 0.34$ for the dotted line (Swendsen's original parameters [Swen99]).

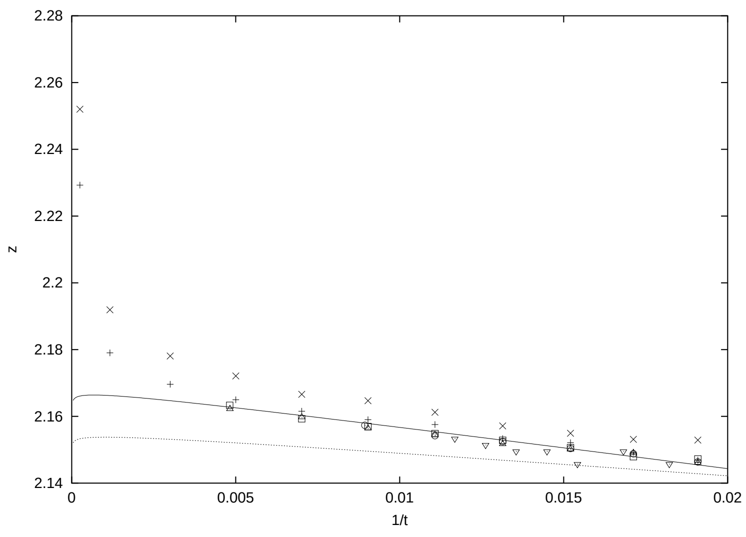


Figure 5.9: Same plot as in fig. 5.8, but with expanded $1/t$-axis.

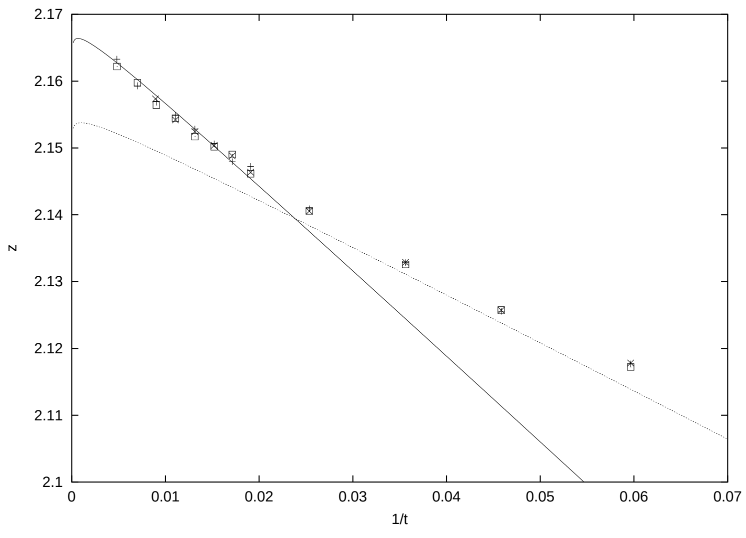


Figure 5.10: Monte Carlo data for three runs with $L = 2\cdot 10^6$. $+$ are for LCG(13^{13}), $\times$ for R(471,1586,6988,9689), $\square$ for LCG(16807); data is the same and the lines represent the same fits as in figs. 5.8 and 5.9.

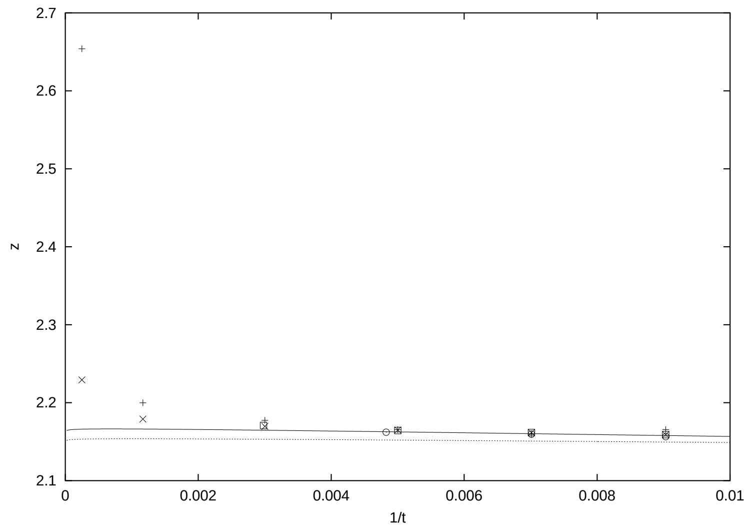


Figure 5.11: Monte Carlo data for $L = 5 \cdot 10^4$ with LCG(16807), averaged over 50 runs (+), $L = 1.5 \cdot 10^5$ with LCG(13^{13}), averaged over 50 runs ($\times$), $L = 10^6$ with R(471,1586,6988,9689), averaged over four runs ($\Box$), and $L = 2 \cdot 10^6$ with LCG(16807), one run ($\circ$).

5.6 Speed of simulation and parallel efficiency

By using both parallelization via domain decomposition and via multi-spin coding, we can achieve very high speeds for simulating the Ising-model, up to 160 million sites per second on a Power-processor of the JUMP; with a clock frequency of 1.7 GHz, this translates to roughly ten cycles per spin. On an Opteron processor with 2.2 Ghz, speed is 90 million sites per second (25 cycles per spin).

When using a parallelized version on N processors, we are interested in the speedup $S(N)$ over a single processor. This speedup is often governed by Amdahl's law [Amda67], $S(N) = 1/(s + \frac{1-s}{N})$, with s signifying the purely sequential portion of the program, and $1 - s$ the purely parallel one. When keeping the overall lattice size L constant, we have so-called "strong scaling", cf. figure 5.12; when using larger L for higher N, we have "weak scaling" (i. e. we adapt problem size to our computational capability).

One sequential part of the program is the communication for exchanging the border planes; as each processor needs to communicate with two neighbouring processors, the wall clock time spent on communicating does not depend on the number of processors, but on the amount of data that needs to be transferred (two times L sites for decomposition into strips of $L \times L/N$).

5.7 Summary and outlook

Although it is possible to argue that the numerical data for very long times is doubtful, as the influence of fluctuations on the value of z increases, the current precision data seems bad for the Domany-Swendsen assumption. It would be possible to modify the fit to the data, but the trend for long times rather contradicts the value of $z = 2$. It is thus safe to say that the dynamical critical exponent is $z > 2$ with simple power-law behaviour, with current best estimate of $z = 2.167(3)$.

Nevertheless, there is still work to do: the influence of various random number

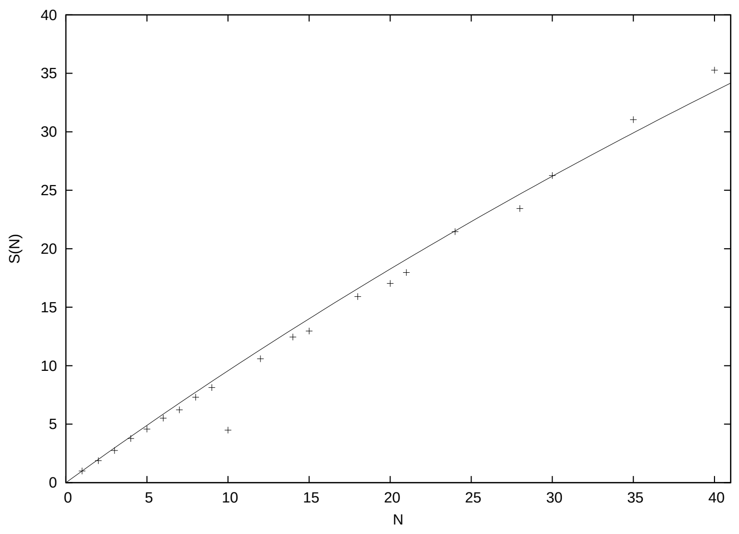


Figure 5.12: Speedup for two-dimensional Ising model, + show median value of five runs. Solid line is Amdahl's law with a sequential portion of $s = 0.005$. Total lattice size is kept constant at $L = 10080$.

generators is important, in order to find one which allows precise data, but is still fast enough for large-scale simulations: for $L = 1.5 \cdot 10^5$, averaged over 50 runs, Ziff's four-tap generator produces results which differ systematically from other generators and system sizes. This is not the case for a single run with $L = 2 \cdot 10^6$ and four-tap generator. Data for $L = 5 \cdot 10^4$, averaged over 25 runs, showed the same behavior as for $L = 1.5 \cdot 10^5$. For these lattice sizes, R(471,1586,6988,9689) seems not to be suited well.

Furthermore, the influence of finite-size effects for long times should be investigated in more detail, and the effects of fluctuations in the magnetization in general.

Chapter 6

Summary and Outlook

6.1 Summary

In the previous chapters, we have examined several numerical methods for investigating the properties and the critical behaviour of percolation and the Ising model. In chapter 2, replication of the traditional Hoshen-Kopelman algorithm was presented as a viable method for investigating fluctuations of cluster numbers. In chapter 3, a parallelized version of the Hoshen-Kopelman algorithm was presented, which uses domain decomposition of the hyperplane of investigation into strips, therefore reducing the amount of memory that needs to be stored per processor. This method allows for implementation on massively-parallel computers with distributed memory, which utilize message-passing. Due to the chosen method of domain decomposition, it is possible to simulate huge lattices also for dimensions $d > 2$, while other approaches to domain decomposition published in literature are severely limited for $d > 2$. In chapter 4, a modification of the Hoshen-Kopelman algorithm was presented, which allows to simulate percolation on growing lattices, i. e. to simulate a lattice of size L_1 and then extend it to size $L_2 > L_1$ without re-investigating the whole lattice; only newly added sites have to be investigated. This approach is very suitable for studying finite-size effects and generally all properties in dependence on system size. Chapter 5 deals with the Ising model. A method already used for chapter 3, i. e. domain decomposition, is also applied to this model, together with the method of multi-spin coding. This allowed for simulating huge lattices over many timesteps in order to investigate critical behaviour with high accuracy.

For all simulations of percolation, follwing values for p_c were chosen:

d	p_c
2	0.5927464
3	0.311608
4	0.196889
5	0.1407966

When investigating fluctuations in percolation, we found that the distribution of cluster numbers n_s for fixed size s is Gaussian for small s and large L. The position of the maximum (i. e. $\langle n_s \rangle$) is in compliance to the power law $n_s \propto s^{-\tau}$. For large s or small L, the left-hand side of the distribution gets distorted and vanished, while the right-hand side stays quasi-Gaussian, i. e. it fits $\exp(-\mathrm{const} \cdot s^{\zeta})$, with $\zeta = 2$ for small s, and ζ decaying for increasing s. Variance of cluster numbers shows the same behaviour as the mean cluster numbers, with deviations for small s, i. e. $(\langle n_s^2 \rangle - \langle n_s \rangle^2)/\langle n_s \rangle = 1 + k_2 s^{-\Delta_2}$, with $k_2 = 0.25(5)$ and $\Delta_2 = 1.2(2)$. The skewness of the distribution grows linearly with s, while the kurtosis grows, but without clear power-law behaviour. Both skewness and kurtosis are no good measures for large s, due to the strong distortion of the distributions.

By simulating huge lattices, we found (τ is the Fisher exponent, known exactly for $d = 2$; Δ_1 is the exponent for corrections to scaling; n_c is the number density):

d	L	τ	Δ_1	n_c
2	7000000	187/91	0.73(2)	0.027597857(2)
3	25024	2.190(1)	0.65(5)	0.05243812(9)
4	1305	2.315(2)	0.48(8)	0.0519995(2)
5	225	2.41(1)	0.30(10)	0.0460321(2)

Simulations for $d = 3, 4, 5$ are world records. Results for $d = 2$ are for the largest published simulation.

By investigating growing lattices, we found (τ and n_c as above; n_{sp} is the number of spanning clusters; D is the fractal dimension of the largest cluster, known exactly for $d = 2$):

d	τ	n_{sp}	n_c	D
2	187/91	0.52(1)	0.0275979(4)	91/48
3	2.1895(10)	0.15(3)	0.0524381(1)	2.52(1)

n_{sp} shows a delicate finite-size behaviour.

For the Ising model in two dimensions, we found with world record simulations that the dynamical critical exponent is *not* $z = 2$ with logarithmic corrections, but with high probability $z > 2$ with simple power-law behaviour, with the current best estimate $z = 2.167(3)$.

This thesis has shown, like many other works in the recent years, that supercomputing is a valuable tool for physics, giving rise to the branch of computational physics, sometimes considered to be the third branch after experimental and theoretical physics. Due to ever increasing computer power, problems can now be investigated with numerical methods, which seemed to be completely inaccessible only one or few decades ago. Although purely analytical solutions are preferable to numerical work, it is now possible to decide questions that cannot be answered by paper and pencil alone.

Modern supercomputers are always massively-parallel; in order to use these efficiently, it is necessary to choose parallel approaches. Two very different methods have been demonstrated in this thesis: replication and domain decomposition. While domain decomposition is regarded to be the real thing and replication is generally frowned upon (some even call it "poor man's parallelization"), replication can be good science, too, and is suited very well for many problems. Domain decomposition enables us to simulate huge lattices, proved by world records obtained for this thesis.

6.2 Outlook

The source code for all simulation programs used in this thesis is published in the appendix F, has already been published in the diploma thesis [Tigg01], or is readily available in literature (see also remarks in appendix F). Hopefully, other scientists can use and extend these programs in order to further investigate percolation and the Ising model.

There are still lots of improvements possible: the program for parallelized Hoshen-Kopelman could be extended to higher dimensions, or refined to investigate more quantities. The program for percolation on growing lattices could be extended to higher dimensions, suitable for studying the number of spanning clusters around the upper critical dimension, where qualitative changes are expected. The parallelized program for the Ising model could be used to study the power-law corrections for z in more detail and to estimate a more precise value for z. It could also be used to study other quantities.

Both the field of percolation and the Ising model have still many questions left open. This thesis provides numerical methods that can be used to further investigate these open questions.

Appendices

Appendix A

Acknowledgements

First and foremost I would like to thank Prof. Stauffer for taking up with my slacking, but finally forcing me to finish. Also helpful in this regard was Prof. Lang of the Computing Centre of the University of Cologne. In agreement with hydrodynamics, increasing pressure makes the work flow faster; even reaching a state of turbulence fits into this picture.

During the pursuit of this thesis, several scientist have provided valuable input; to name but a few, Prof. Ziff, Prof. MacIsaac, Prof. Swendsen, and others, whose names I do not recall. Doing science in an intellectual vacuum is impossible: input by others, either directly or by papers, makes progress possible. Thus, I would like to thank especially Prof. Stauffer for his invaluable advice.

Computing time was kindly provided by NIC of FZ Jülich on their Cray T3E and IBM Regatta JUMP, and by the Computing Centre of the University of Cologne on their compute cluster Clio. Obviously, without the support of these Centres, this thesis would have been impossible.

Appendix B

Bibliography

[AcSt98] M. Acharyya, D. Stauffer, *Effects of boundary conditions on the critical spanning probability*, Int. J. Mod. Phys. C **9**, 643 (1998).

[Adle83] J. Adler, M. Moshe, V. Privman, *Corrections to scaling for percolation*, in: [DZJ83], pp. 397–423.

[Ahar83] A. Aharony, M. E. Fisher, *Nonlinear scaling fields and corrections to scaling near criticality*, Phys. Rev. B **27**, 4394 (1983).

[AhSt97] A. Aharony, D. Stauffer, *Test of universal finite-size scaling in two-dimensional site percolation*, J. Phys. A **30**, L301 (1997).

[Aize97] M. Aizenmann, *On the number of incipient spanning clusters*, Nucl. Phys. B **485**, 551 (1997).

[Amda67] G. Amdahl, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings **30**, 483 (1967).

[Arju03] M. Arjunwadkar, M. Fasnacht, J. B. Kadane, R. H. Swendsen, *A Bayesian analysis of Monte Carlo correlation times for the tow-dimensional Ising model*, Physica A **323**, 487 (2003).

[Ball97] 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).

[Ball99] H. G. Ballesteros, L. A. Fernández, V. Martín-Mayor, A. Muñoz Sudupe, G. Parisi, J. J. Ruis-Lorenzo, *Scaling corrections: site percolation and Ising model in three dimensions*, J. Phys. A **32**, 1 (1999).

[Baxt78] 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).

[BiSt87] 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.

[BrHa57] S. R. Broadbent, J. M. Hammersley, *Percolation Processes. Crystals and mazes*, Proc. Cambridge Philos. Soc. **53**, 629 (1957).

[Broa54] S. R. Broadbent, *Discussion on symposium on Monte Carlo methods*, J. Roy. Statist. Soc. B **16**, 68 (1954).

[Bru67] S. G. BRUSH, *History of the Lenz-Ising model*, Rev. Mod. Phys. **39**, 883 (1967).

[Bund91] A. BUNDE, S. HAVLIN (Eds.), *Fractals and Disordered Systems*, (Springer, Heidelberg, 1991).

[Bund99] A. BUNDE, S. HAVLIN (Eds.), *Proceedings of the International Conference on Percolation and Disordered Systems: Theory and Applications*, Physica A **266** (1999).

[Card92] J. L. CARDY, *Critical percolation in finite geometries*, J. Phys. A **25**, L201 (1992).

[CJR79] M. CREUTZ, L. JACOBS, C. REBBI, *Experiments with a gauge-invariant Ising system*, Phys. Rev. Lett. **42**, 1390 (1979).

[CSS79] A. CONIGLIO, H. E. STANLEY, D. STAUFFER, *Fluctuations in the number of percolation clusters*, J. Phys. A **12**, L323 (1979).

[Doma84] E. DOMANY, *Exact results for two- and three-dimensional Ising and Potts models*, Phys. Rev. Lett. **52**, 871 (1984).

[DZJ83] G. DEUTSCHER, R. ZALLEN, J. ADLER, *Percolation structure and processes*, Annals of the Israel Physical Society **5**, (Adam Hilger, Bristol, 1983).

[FASC04] S. FORTUNATO, A. AHARONY, A. CONIGLIO, D. STAUFFER, *Number of spanning clusters at the high-dimensional percolation thresholds*, Phys. Rev. E **70**, 056116 (2004).

[Fish67] M. E. FISHER, *The theory of condensation and the critical point*, Physics **3**, 255 (1967).

[Flor41] 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).

[FLR99] J. E. DE FREITAS, L. S. LUCENA, S. ROUX, *Percolation as a dynamical phenomenon*, Physica A **266**, 81 (1999).

[FlTa92] M. FLANIGAN, P. TAMAYO, *A Parallel Cluster Labeling Method for Monte Carlo Dynamics*, Int. J. Mod. Phys. C **3**, 1235 (1992).

[FlTa95] M. FLANIGAN, P. TAMAYO, *Parallel cluster labeling for large-scale Monte Carlo simulations*, Physica A **215**, 461 (1995).

[FrCa70] R. FRIEDBERG, J. E. CAMERON, *Test of the Monte Carlo method: Fast simulation of a small Ising lattice*, J. Chem. Phys. **52**, 6049 (1970).

[FrLu00] J. E. DE FREITAS, L. S. LUCENA, *Equivalence between the FLR time dependent percolation model and the Newman-Ziff algorithm*, Int. J. Mod. Phys. C **11**, 1581 (2000).

[FSC03] S. FORTUNATO, D. STAUFFER, A. CONIGLIO, *Percolation in high dimensions is not understood*, Physica A **334**, 307 (2003).

[Glau63] R. J. GLAUBER, *Time-dependent statistics of the Ising model*, J. Math. Phys. **4**, 294 (1963).

[GND00] J.-C. Gimel, T. Nicolai, D. Durand, *Size distribution of percolating clusters on cubic lattices*, J. Phys. A **33**, 7687 (2000).

[Gras03] P. Grassberger, *Critical percolation in high dimensions*, Phys. Rev. E **67**, 036101 (2003).

[Grim99] G. R. Grimmett, *Percolation*, 2nd ed., (Spring, Berlin, 1999).

[Grim00] G. R. Grimmett, *Percolation*, in: J.-P. Pier (ed.), *Development of Mathematics 1950–2000*, (Birkhäuser, Basel, 2000), pp. 547–575.

[Grop95] 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.

[Gutb99] 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.

[HaHa64] J. M. Hammersley, D. C. Handscomb, *Monte Carlo Methods*, (Methuen, London, 1964).

[Hamm57a] J. M. Hammersley, *Percolation Processes. The connectivity constant*, Proc. Cambridge Philos. Soc. **53**, 642 (1957).

[Hamm57b] J. M. Hammersley, *Percolation Processes. Lower bounds for the critical probability*, Ann. Math. Statist. **28**, 790 (1957).

[Hamm83] J. M. Hammersley, *Origins of percolation theory*, in: [DZJ83], pp. 47–57.

[HMS93] 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).

[HoKo76] J. Hoshen, R. Kopelman, *Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm*, Phys. Rev. B **14**, 3438 (1976).

[Isin25] E. Ising, *Beitrag zur Theorie des Ferromagnetismus*, Zeitschr. f. Physik **31**, 253 (1925).

[JaSt98] N. Jan, D. Stauffer, *Random Site Percolation in Three Dimensions*, Int. J. Mod. Phys. C **9**, 341 (1998).

[JSA98] N. Jan, D. Stauffer, A. Aharony, *An infinite number of effectively infinite clusters in critical percolation*, J. Stat. Phys. **92**, 325 (1998).

[KaFo69] P. W. Kasteleyn, C. M. Fortuin, *Phase transitions in lattice systems with random local properties*, J. Phys. Soc. Japan Suppl. **26**, 11 (1969).

[Kall84] C. Kalle, *Vectorised dynamic Monte Carlo renormalization group for the Ising model*, J. Phys. A **17**, L801 (1984).

[Kest90] H. Kesten, *Asymptotics in high dimensions for percolation*, in: G. R. Grimmett, D. J. A. Walsh, *Disorder in physical systems*, (Clarendon Press, Oxford, 1990), pp. 219–240.

[KeSt92] J. Kertész, D. Stauffer, *Swendsen-Wang Dynamics of Large 2D Critical Ising Models*, Int. J. Mod. Phys. C **3**, 1275 (1992).

[KiSt81] S. Kirkpatrick, E. P. Stoll, J. Comput. Phys. **40**, 517 (1981).

[KrWa41] H. A. Kramers, G. H. Wannier, *Statistics of the two-dimensional ferromagnet. Part I*, Phys. Rev. **60**, 252 (1941).

[LaBi05] D. P. Landau, K. Binder, *A guide to simulations in statistical physics, 2nd ed.*, (Cambridge University Press, Cambridge, 2005).

[Lea76a] P. L. Leath, *Cluster size and boundary distribution near percolation threshold*, Phys. Rev. B **14**, 5046 (1976).

[Lea76b] P. L. Leath, *Cluster shape and critical exponents near Percolation Threshold*, Phys. Rev. Lett. **36**, 921 (1976).

[Lenz20] W. Lenz, *Beitrag zum Verständnis der magnetischen Eigenschaften in festen Körpern*, Phys. Zeitschr. **21**, 613 (1920).

[Link95] A. Linke, D. W. Heermann, P. Altevogt, M. Siegert, *Large-scale simulation of the two-dimensional kinetic Ising model*, Physica A **222**, 205 (1995).

[LoZi98a] 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).

[LoZi98b] 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).

[MaHe84] A. Margolina, H. J. Herrmann, *On finite-size scaling of the order parameter in percolation*, Phys. Lett. A **104**, 295 (1984).

[Mart90] J. L. Martin, *The impact of large-scale computing on lattice statistics*, J. Stat. Phys. **58**, 749 (1990).

[MDSS83] A. Margolina, Z. Djordjevic, H. E. Stanley, D. Stauffer, *Corrections to scaling for branched polymers and gels*, Phys. Rev. B **28**, 1652 (1983).

[Mert90] S. Mertens, *Lattice animals: A fast enumeration algorithm and new perimeter polynomials*, J. Stat. Phys. **58**, 1095 (1990).

[MeLa92] S. Mertens, M. E. Lautenbacher, *Counting lattice animals – a parallel attack*, J. Stat. Phys. **66**, 669 (1992).

[MLJ98] S. MacLeod, N. Jan, *Large Lattice Simulation of Random Site Percolation*, Int. J. Mod. Phys. C **9**, 289 (1998).

[MoPr03] N. R. Moloney, G. Pruessner, *Asynchronously Parallelized Percolation on Distributed Machines*, Phys. Rev. E **67** 037701 (2003).

[MRRTT53] N. Metropolis, A. W. Rosenbluth, M .N .Rosenbluth, A. H. Teller, E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys. **21**, 1087 (1953).

[NaSt80] 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).

[NeZi00] M. E. J. Newman, R. M. Ziff, *Efficient Monte Carlo algorithm and high-precision results for percolation*, Phys. Rev. Lett. **85**, 4104 (2000).

[NiBl96] M. P. Nightingale, H. W. J. Blöte, *Dynamic exponent of the two-dimensional Ising model and Monte Carlo computation of the subdominant eigenvalue of the stochastic matrix*, Phys. Rev. Lett. **76**, 4548 (1996).

[NiBl00] M. P. Nightingale, H. W. J. Blöte, *Monte Carlo computation of correlation times of independent relaxation modes at criticality*, Phys. Rev. B **62**, 1089 (2000).

[Nien82] B. Nienhuis, *Analytical solution of the two leading exponents of the dilute Potts model*, J. Phys. A **15**, 199 (1982).

[Nijs79] 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).

[NRS80] B. Nienhuis, E. K. Riedel, M. Schick, *Magnetic exponents of the two-dimensional q-state Potts model*, J. Phys. A **13**, L189 (1980).

[Onsa44] L. Onsager, *Crystal statistics. I. A two-dimensional model with an order-disorder transition*, Phys. Rev. **65**, 117 (1944).

[Pear80] R. P. Pearson, *Conjecture for the extended Potts model magnetic eigenvalue*, Phys. Rev. B **22**, 2579 (1980).

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical recipes in Fortran, 2nd ed.*, (Cambridge University Press, Cambridge, 1992).

[PZS01] G. Paul, R. M. Ziff, H. E. Stanley, *Precolation threshold, Fisher exponent, and shortest path exponent for four and five dimensions*, Phys. Rev. E **64**, 026115 (2001).

[Sahi94] M. Sahimi, *Applications of Percolation Theory*, (Taylor & Francis, London, 1994).

[SAHM99] 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: [Bund99], pp. 5–16.

[Sen96] P. Sen, *Non-uniqueness of spanning clusters in two to five dimensions*, Int. J. Mod. Phys. C **7**, 603 (1996).

[Sen97] P. Sen, *Probability distribution and sizes of spanning clusters at the percolation thresholds*, Int. J. Mod. Phys. C **8**, 229 (1997).

[SHIW93] D. Stauffer, F. W. Hehl, N. Ito, V. Winkelmann, J. G. Zabolitzky, *Computer Simulation and Computer Algebra*, 3rd ed., (Springer, Heidelberg, 1993).

[StAh94] D. Stauffer, A. Aharony, *Introduction to Percolation Theory*, (Taylor & Francis, London, 1994).

[Stau75] D. Stauffer, *Violation of dynamical scaling for randomly dilute Ising ferromagnets near percolation threshold*, Phys. Rev. Lett. **35**, 394 (1975).

[Stau79] D. Stauffer, *Scaling theory of percolation clusters*, Phys. Reports **54**, 1 (1979).

[Stau94] D. Stauffer, *Finite-size effect in seven-dimensional percolation*, Physica A **210**, 317 (1994).

[Stau97] D. Stauffer, *Relaxation of Ising models near and away from criticality*, Physica A **244**, 344 (1997).

[Stau99] D. Stauffer, *Test of Domany-Swendsen hyporthesis for 2d kinetic Ising model*, Int. J. Mod. Phys. C **10**, 931 (1999).

[Stau00] D. Stauffer, *World records in the size of simulated Ising models*, Braz. J. Phys. **30**, 787 (2000).

[Swen99] R. H. Swendsen, conference talk at "Monte Carlo and Structure Optimization Methods for Biology, Chemistry, and Physics", Tallahassee (1999); and private communication.

[Syke86] M. F. Sykes, *Generating functions for connected embeddings in a lattice*, J. Phys. A **19**, 1007–1025, 1027–1032, 2425–2429, 2431–2438, 3407–3414 (coauthored by M. K. Wilkinson) (1986).

[Tama93] P. Tamayo, *Magnetization relaxation to equilibrium on large 2D Swendsen-Wang Ising models*, Physica A **201**, 543 (1993).

[Taus65] R. C. Tausworthe, Math. Comput. **19**, 201 (1965).

[TeGi00] 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).

[TeLi71] 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).

[Tigg01] D. Tiggemann, *Simulation of percolation on parallel computers*, Diploma thesis, Institute for Theoretical Physics, University of Cologne (2001).

[TiSt01] A. Ticona, D. Stauffer, *Percolation cluster numbers in seven dimensions*, Physica A **290**, 1 (2001).

[ZFA97] 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).

[Ziff96] R. M. Ziff, *Effective boundary extrapolation length to account for finite-size effects in the percolation crossing function*, Phys. Rev. E **54**, 2547 (1996).

[Ziff98] R. M. Ziff, *Four-tap shift-register-sequence random-number generators*, Comput. in Phys. **12**, 385 (1998).

[ZLK99] R. M. Ziff, C. D. Lorenz, P. Kleban, *Shape-dependent universality in percolation*, in: [Bund99], pp. 17–26.

Appendix C

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 $\bar{y}$ as a good estimate for the value without statistical errors; the standard deviation $\sigma = \sqrt{1/(N-1)\sum(x_i - \bar{x})^2}$ is a measure for the typical error of a single value y_i, while $\Delta y = \sigma/\sqrt{N}$ is called the statistical error of $\bar{y}$ and gives an estimate, how strong $\bar{y}$ would change, if we added another statistically independent value y_i; i. e. it is the error of the mean. The larger the system is, the smaller the fluctuations become. 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 infinite 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 differently 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 $N \gg 1$ 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 devastating 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 necessary, for example by testing data against theoretical assumptions, by comparing with exact data where possible, or by other methods.

Appendix D

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 [Gutb99].

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. [SHIW93, part II, chapter 1]), and in this thesis, they showed wrong behaviour, too (cf. section 3.3.4). 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 [Ziff98].

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 Kirkpatrick and Stoll (cf. [KiSt81]), despite the fact that mathematicians prefer to call it after Tausworthe (cf. [Taus65]). 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 large-scale percolation, such problems did not occur. Nevertheless, mostly higher-quality generators were used, in order to be on the safe side (there is no much use in simulating larger systems, if the results become worse due to bad random numbers).

Generators with higher quality can be obtained using more and/or larger taps. Two of them were used within this 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. [Ziff98] for a list of such applications). Interestingly, for the Ising model (with world record sizes), the multi-tap generators showed worse results than the rather primitive LCGs, proving once more that choosing the right PRNG for Monte Carlo simulations is more a black art than an exact science.

One problem with LFG 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.

D.3 Hashing generators

Hashing generators differ from conventional PRNGs in that they do not produce a sequence of random numbers after being initalized with a seed value, but instead produce a single random number from a single seed. This way, it is possible to generate a random number based on the number of a lattice site and a time index (or any other modifier). This is a very desirable advantage, as the produced order of random numbers is not dependent on domain decomposition, if the sites in the lattice are labeled in a globally consistent way. No matter how the lattice is partitioned over processors, the random number produced for every site is the same. Thus runs with different number of processors should produce the same result. If not, there is a bug in the program—unless we use floating point numbers, which are not real *real numbers*, as operations on these are neither associative nor commutative; changing the order of summation of floating point numbers can change the results, without any bugs in the program.

When using traditional PRNGs that produce sequences of random numbers, the domain decomposition dictates in what order the random numbers are distributed over the lattice sites. It is not possible to produce the same distribution of random numbers for N_1 and N_2 processors, unless we utilize $N_1 N_2$ PRNGs and assign these to fittingly decomposed domains. Generally, only N_i processors are usable, with N_i divisors of N, the total number of pre-planned domains. This reduces versatility.

While it would be possible to use traditional PRNGs in the same way, i. e. use the lattice site number as seed for the sequence and then producing only a single number, these numbers would be strongly correlated for neighbours, when using linear congruential generators. These correlations would strongly disturb physical effects; even much weaker correlations can have very negative influence on simulations. Lagged fibonacci generators probably would not show these strong correlations, but as they are expensive to set up, their use in this way is prohibitive.

The advantage of hashing generators is due to the strong non-linearity of the hashing function; for neighbouring lattice sites i and $i + 1$, $f(i)$ and $f(i + 1)$ are not correlated in a conceivable way (for large scale simulations, subtle correlations could show up).

Hashing generators are derived from cryptographic hashing functions. These functions are designed to produce strongly different hash values even for very similar inputs, a property that we want to exploit. But while cryptographic hashing functions need to be cryptographically strong, i. e. hard to break, this property is not needed for our purposes. Thus the algorithm can be simplified in order to obtain higher speed. One suitable algorithm (pseudo-DES) is published in [PTVF92].

The disadvantage of these generators is their reduced speed in comparison to sequence generators. For this reason, hashing generators are not used for traditional Monte Carlo methods, with the exception of using them for tests. Paul et al. [PZS01] describe one non-traditional use for a modified, high-dimensional Leath algorithm, where they use the reproducability of random numbers, so that they do not need to store state of lattice sites; they can regenerate the state by reproducing the original random number from the number of the lattice site.

D.4 Speed of different random number generators

Given two PRNGs of sufficient quality (i. e. their correlations do not influence the result), the faster PRNG is always preferrable, as with a fixed budget of computing time, the faster generator allows for simulating larger or more lattices, thus reducing statistical errors. High-performance computing means haggling for every processor cycle and every bit of memory.

The following table shows runtimes for simulations with different random number generators, averaged over five runs (each run with different inital seed). All times are in seconds, for full simulations. For each run, 10^{10} random numbers were generated. S (solo) means only random numbers were generated, without using them (pure overhead of PRNG), P means two-dimensional percolation on an $L = 10^5$ lattice, and I the two-dimensional Ising model on an $L = 10^4$ lattice with 100 Monte Carlo sweeps through the lattice. $P - S$ and $I - S$ mean times for percolation resp. Ising minus time for the solo run of the PRNG. 32 bit and 64 bit correspond to the size of the generated random word. All runs were done on fast Opteron processors.

	S	P	$P-S$	I	$I-S$
LCG(16807), 32 bit	18.34	301.58	283.24	61.87	43.52
LCG(16807), 64 bit	21.75	288.24	266.49	62.95	41.19
R(103,250), 32 bit	20.49	312.74	292.25	83.39	62.89
R(103,250), 64 bit	21.74	314.29	292.54	83.82	62.08
R(471,1586,6988,9689), 32 bit	38.95	345.35	306.39	91.70	52.74
R(471,1586,6988,9689), 64 bit	40.73	346.43	305.69	96.34	55.60
R(18,36,37,71,89,124), 32 bit	45.81	352.28	306.46	106.23	60.41
R(18,36,37,71,89,124), 64 bit	48.11	343.56	295.44	104.42	56.30
Pseudo-DES, 32 bit	279.48	576.92	297.46	314.38	34.92

Generally, the LCG generator is the fastest, while the pseudo-DES generator is the slowest, with a very significant performance penalty; therefore, its use for large scale simulations is not feasible. The lagged-fibonacci generators are slower than LCGs, but with an acceptable performance penalty.

As for all runs exactly the same number of random numbers is generated, it is instructive to compare the amount of simulation work per random number. For the Ising model, generating random numbers takes 1/3 of the runtime for the fastest generator. Due to multi-spin coding, examining spins is a very cheap operation, meaning that slow PRNGs have a very large impact on runtime. In contrast, for percolation examining a site is rather expensive, as whole words need to be examined, not just bits in parallel. Therefore, the impact of slow PRNGs is less pronounced.

For the lagged-fibonacci generators, an array is needed to hold intermediate values. This array competes for cache memory with other data structures needed for simulation, therefore these generators slow down the rest of the simulation. This is the reason why $P - S$ and $I - S$ are not constant for the different generators. It is therefore important to test not only the generator alone, but also in conjunction with the real simulation, in order to assess its speed.

The values for P and S with 32 and 64 bits for the six-tap generator were not swapped mistakenly, as one would believe; 64 bits are faster than 32 bits, although that is counter-intuitive (more memory is needed for 64 bits). One possible explanation is the optimizing compiler, which could be able to pipeline machine instructions better when all these instructions are in 64 bits (as the rest of the simulation is for both percolation and the Ising model). This emphasizes once again the importance of really measuring speed, and not trying to deduce it from assumptions.

In summary, the LCG is the fastest random number generator currently in use; where its quality is sufficient, it should be preferred. LFGs are slower, but for many apllications still fast enough. Pseudo-DES is too slow for any large-scale simulations.

Appendix E

Amdahl's law and measuring parallel efficency on real-life computers

Amdahl's law, named after Gene Amdahl, postulates a theoretical limit for parallel processing (cf. [Amda67]). Given a program that separates into two fractions, a strictly sequential part, and a fully parallel part, we can write its required runtime on a single processor as $T(1) = sT(1) + (1-s)T(1)$, where s signifies the sequential fraction, and $1-s$ the parallel. On N processors, we have $T(N) = sT(1) + \frac{1-s}{N}T(1)$, as the parallel part is supposed to be fully parallel, i. e. it can be broken down in arbitrary little pieces (as opposed to reality). We can then derive the speedup $S(N) = T(1)/T(N)$, i. e. how much faster the program is on N processors. This is Amdahl's law:

$$S(N) = \frac{1}{s + \frac{1-s}{N}} \tag{E.1}$$

Some observations are:

- For large N, i. e. $Ns \gg 1-s$, the sequential part of the program dominates runtime and limits possible speedup. It is hard to achieve efficient speedup with many processors.
- Even for $N \to \infty$, the maximum speedup is $1/s$. If the program already is near this limit, it is useless to try to run on even more processors. The return would be marginal.
- If $s \ll 1$ and N is small, then $S(N) \propto N$. It is easy to achieve efficient speedup with few processors.
- The sequential part s determines the success of parallelization. It is important to reduce s as much as possible.

Although the assumptions leading to Amdahl's law are oversimplified, it still offers high quality in predicting runtimes of parallel programs.

It is important to note that the sequential fraction s in many cases depends on the simulated system size, i. e. a larger system L^d can lead to a smaller s (strong scaling vs. weak scaling), cf. fig. E.1.

A larger number of processors for a given problem size means that a larger fraction of the data fits into the caches of the processors, giving rise to superlinear

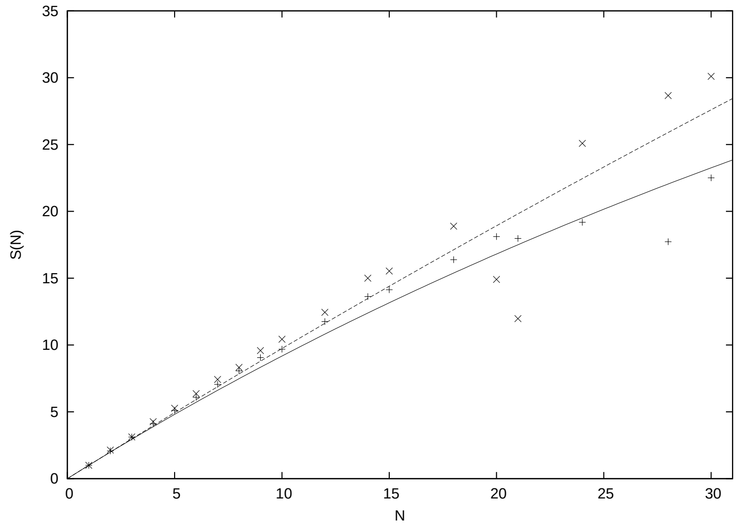


Figure E.1: Speedup $S(N)$ for three-dimensional percolation on N processors, $+$ are for $L = 2520$, $\times$ for $L = 5040$. The solid line is Amdahl's law for $s = 0.01$, the dashed line for $s = 0.003$. Outliers are probably caused by operating system influence (which gets worse for higher processor numbers). The high speedup for the $\times$ is probably caused by super-linear cache-effects, therefore the measured speedup seems to be very optimistic.

speedup, i. e. $S(N) > N$. This can complicate estimating the sequential portion of a program, again cf. fig. E.1.

When the operating system of a parallel computer is not specifically optimized for highly parallel computing, it sometimes can happen that operating system tasks interrupt parts of a parallel program on one processor, but not on others. In that case, the rest of the parallel program has to wait, wasting wall clock time. This explains the outliers visible in fig. E.1.

In summary, estimating parallel efficiency from the runtime of real-world simulations is problematic, but still worthwhile to establish how well a program is parallelized.

Appendix F

Code of programs

F.1 General remarks

The source code of all programs that were used for simulations published in this thesis, including those printed in this appendix, can be obtained by email from the author. In case the author can no longer be reached, an electronic version of this thesis will be published by the Library of the University of Cologne, available at kups.ub.uni-koeln.de; this way, it is possible to extract electronically the code of the programs printed in this appendix.

Programs shown here are written in Fortran 90, as Fortran compilers produce often faster code than C compilers. They are written in free format (as opposed to fixed format, where some columns of the source code have special meaning), as the age of keypunching is long over. These programs can be compiled with any standard-adherent Fortran 90 compiler (tested with IBM-, Sun-, Portland Group-, and Pathscale-Compilers).

The small slanted numbers on the left side of the listing serve as a guide to the eye and make pointing to parts of the program easier. They are not a part of the program and should not be mistaken with labels that are used in old-fashioned Fortran programs.

A sequential implementation of the traditional Hoshen-Kopelman algorithm is not presented here, as it is readily available in literature (cf. [StAh94, A.3] for a short version with detailed explanation, or [Tigg01, C.2] for a longer version).

F.2 Parallelized Hoshen-Kopelman

The source code for a version of parallelized Hoshen-Kopelman was already published in [Tigg01], but programmed using shmem-directives for the Cray T3E, and probably containg a programming error. The version presented here is ported to MPI (version 1.1) and should be usable on any MPI-capable computer.

Parameters for the simulation can be given as constants in the code. Remarkable are two boolean constants: `fully_periodic` switches fully periodic b. c. on or off; when off, only one instead of two hyperplanes are allocated. `sanity_checks` switches internal consistency checks on or off. Whenever modifying the program, it is wise to switch these on, in order to catch at least some possible errors introduced by new code. After regaining confidence in the program, one can switch off these checks for higher speed of simulations.

```
PROGRAM Verti

implicit none
```

```
include 'mpif.h'

! Parameters
!real*8, parameter    :: p = 0.5927464d0 ! 2d
!real*8, parameter    :: p = 0.311608d0 ! 3d
!real*8, parameter    :: p = 0.196889d0 ! 4d
real*8, parameter    :: p = 0.1407966d0 ! 5d
integer, parameter   :: D=5            ! 2 <= D <= 5
integer, parameter   :: L=225
integer, parameter   :: NPROCS = 45
logical, parameter   :: fully_periodic = .true.
integer, parameter   :: MAXLOC=45e6
integer, parameter   :: MAXGLO=15e6
integer, parameter   :: MAXBIN=60
integer, parameter   :: LOCLIM=0.7*MAXLOC
integer, parameter   :: GLOLIM=0.7*MAXGLO
integer, parameter   :: MAXPAIR=262143
integer, parameter   :: MAXPREP=65535
integer, parameter   :: MAXBUF=65536
logical, parameter   :: sanity_checks = .true.
integer*8, parameter :: one = 1
integer*8, parameter :: Lsystem= (one*L)**D
integer, parameter   :: Lline  = L**(D-2)
integer, parameter   :: Lplane = L/NPROCS*Lline
integer, parameter   :: idx_pred = 0
integer, parameter   :: idx_succ = 1

! Variables
integer*4 :: tag, status(MPI_STATUS_SIZE), err
integer*4 :: pe_me, pe_pred, pe_succ, mpi_nprocs
real*8    :: rcplog
integer, parameter :: RNDMAX = 16383
integer*4 :: IP, rnd_data(0:RNDMAX), rnd_idx, current_seed     ! 32-bit version
integer*4 :: ISEED
integer*8,allocatable :: plane(:), upper(:)
integer*8,allocatable :: local(:)
integer*8,allocatable :: globsites(:)
integer*8,allocatable :: globneigh(:,:)
integer*8,allocatable :: border_recv(:)
integer*8 :: ns(0:MAXBIN), nsglobal(0:MAXBIN)
integer*8,save :: preprec_send(0:MAXPREP), preprec_recv(0:MAXPREP)
integer*8,save :: sendbuf(0:MAXBUF), recvbuf(0:MAXBUF)
integer*8,save :: pairdata(0:MAXPAIR, 0:1)
integer   :: pairptr(0:1)
integer*8,save :: pairrecv(0:MAXPAIR)
integer*8 :: largest
integer   :: locptr, gloptr
integer   :: nrecyc
real*8    :: t_total, t_init, t_perc, t_anal, t_local, t_nbex, t_prex,&
             t_recyc, t_relax, t_connect, t_concen, t_facct
real*8    :: t_total_s, t_init_s, t_perc_s, t_anal_s, t_local_s, &
             t_nbex_s, t_prex_s, t_recyc_s, t_relax_s, t_connect_s, &
             t_concen_s, t_facct_s
integer*8 :: nfree, nocc, nfreeglobal, noccglobal
real*8    :: chi, chiglobal
integer*8 :: num_acc_sites=0,num_acc_sites_global
integer*8 :: num_leftover=0,num_leftover_global
integer*8 :: num_glo_labels=0, num_glo_labels_global ! how many global labels do we need to keep?

integer   :: glob_line, glob_site

! Main program
allocate(plane(-Lline:Lplane-1))
if(fully_periodic) allocate(upper(0:Lplane-1))
allocate(local(MAXLOC))
allocate(globsites(0:MAXGLO))
allocate(globneigh(0:1,0:MAXGLO))
allocate(border_recv(0:Lline-1))
call mpi_init(err)
t_total_s = mpi_wtime()
t_local = 0.0d0
t_nbex = 0.0d0
t_prex = 0.0d0
t_recyc = 0.0d0
IP = 2147483648.0d0*(2.0d0*p-1.0d0)
rcplog = 1.0d0/dlog(2.0d0)
if(((L/NPROCS)*NPROCS).ne.L) then
  print *, 'L/NPROCS must be a natural number!'
  call mpi_abort(MPI_COMM_WORLD, 0, 0)
end if

call mpi_comm_size(MPI_COMM_WORLD, mpi_nprocs, err)
if(mpi_nprocs.ne.NPROCS) then
  print *, 'Wrong number of processors, is: ',mpi_nprocs,', should be: ',NPROCS
  call mpi_abort(MPI_COMM_WORLD, 0, 0)
endif

tag = 1
call mpi_comm_rank(MPI_COMM_WORLD, pe_me, err)
pe_pred = pe_me - 1
if(pe_pred .eq. -1) pe_pred = NPROCS - 1
pe_succ = pe_me + 1
if(pe_succ .eq. NPROCS) pe_succ = 0

if(pe_me .eq. 0) then
  read *, ISEED
  print *, '# p = ', p
  print *, '# IP = ', IP
  print *, '# L = ', L
  print *, '# MAXLOC = ', MAXLOC, ', LOCLIM = ', LOCLIM
  print *, '# MAXGLO = ', MAXGLO, ', GLOLIM = ', GLOLIM
  print *, '# MAXBIN = ', MAXBIN
  print *, '# MAXPAIR=',MAXPAIR,', MAXPREP=',MAXPREP,' MAXBUF=',MAXBUF
  print *, '# Used PRNG: Fourtap, ISEED = ', ISEED
  print *, '# Lsystem = ', Lsystem,', Lplane = ',Lplane,', Lline = ',Lline
  print *, '# Nprocs = ', NPROCS
endif
call mpi_bcast(iseed, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
call rnd_randomize(ISEED, pe_me)
rcplog = 1.0d0/dlog(2.0d0)
nrecyc = 0
nfree = 0
chi = 0.0d0
pairptr(idx_pred) = 0
pairptr(idx_succ) = 0

t_init_s = mpi_wtime()
call initialize()
t_init = mpi_wtime() - t_init_s
t_perc_s = mpi_wtime()
call percolate()
t_perc = mpi_wtime() - t_perc_s
t_anal_s = mpi_wtime()
call analyze()
t_anal = mpi_wtime() - t_anal_s
call output()
t_total = mpi_wtime() - t_total_s
call mpi_reduce(t_total,t_total_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_init,t_init_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_perc,t_perc_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_anal,t_anal_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_local,t_local_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_nbex,t_nbex_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_prex,t_prex_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_recyc,t_recyc_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_relax,t_relax_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_connect,t_connect_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_concen,t_concen_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
call mpi_reduce(t_facct,t_facct_s,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
if(pe_me.eq.0) then
  print *, '# Required runtime: ', t_total_s/NPROCS, ' seconds'
  print *, '# Speed: ', 1d-6*Lsystem/t_total_s, ' MSites/s'
  print *, '# T total: ', t_total_s/NPROCS
  print *, '#   T init: ', t_init_s/NPROCS
  print *, '#   T perc: ', t_perc_s/NPROCS
  print *, '#     T local: ', t_local_s/NPROCS
  print *, '#     T neighb. exch.: ', t_nbex_s/NPROCS
  print *, '#     T pair. exch.: ', t_prex_s/NPROCS
  print *, '#     T recycling: ', t_recyc_s/NPROCS
  print *, '#   T analysis: ', t_anal_s/NPROCS
  print *, '#     T full relax.: ', t_relax_s/NPROCS
  print *, '#     T connect: ', t_connect_s/NPROCS
  print *, '#     T concentration: ', t_concen_s/NPROCS
  print *, '#     T final accnt.: ', t_facct_s/NPROCS
end if

call mpi_finalize(err)

contains

subroutine initialize()
  integer i

  ns(0:MAXBIN) = 0
  local(1:MAXLOC) = 0
  globsites(1:MAXGLO) = 0
  plane(-Lline:Lplane-1) = MAXLOC
  if(fully_periodic) upper(0:Lplane-1) = MAXLOC
  locptr = 1
  gloptr = 1
end subroutine

subroutine percolate()
  integer i, level

  glob_line = 1
  call perc_one_plane()      ! Simulate uppermost plane
  ! Copy plane for later reference (periodic b.c.)
  if(fully_periodic) upper(0:Lplane-1) = plane(0:Lplane-1)
  do level = 2, L            ! Simulate rest of system
    glob_line = level
    call perc_one_plane()
  end do
end subroutine

subroutine perc_one_plane()
  integer i
  logical want_recycling, any_wants_recycling
  integer   :: tag, status(MPI_STATUS_SIZE), err


  t_local_s = mpi_wtime()
  do i = 0, Lplane-1
    glob_site = i
    call site_Ndim(i)
  end do
  t_local = t_local + mpi_wtime() - t_local_s
  want_recycling = .false.
  if((locptr.ge.LOCLIM).or.(gloptr.ge.GLOLIM)) &
    want_recycling = .true.
  call mpi_allreduce(want_recycling, any_wants_recycling, 1, MPI_LOGICAL, &
                     MPI_LOR, MPI_COMM_WORLD, err)
  if(any_wants_recycling) then
    t_recyc_s = mpi_wtime()
    if(sanity_checks) call sanity_check()
    call garbage_collection()
    if(sanity_checks) call sanity_check()
    t_recyc = t_recyc + mpi_wtime() - t_recyc_s
  end if
  t_nbex_s = mpi_wtime()
  call neighbour_exchange(           0, idx_pred, pe_pred, pe_succ)
  call neighbour_exchange(Lplane-Lline, idx_succ, pe_succ, pe_pred)
  t_nbex = t_nbex + mpi_wtime() - t_nbex_s
  t_prex_s = mpi_wtime()
  call pairing_exchange(idx_pred, pe_pred, pe_succ)
  call pairing_exchange(idx_succ, pe_succ, pe_pred)
  t_prex = t_prex + mpi_wtime() - t_prex_s
end subroutine

subroutine analyze()
  integer i

  t_relax_s = mpi_wtime()
  call full_relaxation()
  t_relax = mpi_wtime() - t_relax_s
  t_connect_s = mpi_wtime()
  if(fully_periodic) call connect_top_and_bottom()
  t_connect = mpi_wtime() - t_connect_s
  t_relax_s = mpi_wtime()
  call full_relaxation()
  t_relax = t_relax + mpi_wtime() - t_relax_s
  t_concen_s = mpi_wtime()
  call concentrate()
  t_concen = mpi_wtime() - t_concen_s
  t_facct_s = mpi_wtime()
  call local_accounting()
  call mpi_reduce(num_leftover,num_leftover_global,1,MPI_INTEGER8,MPI_SUM,0,MPI_COMM_WORLD,err)
  if((pe_me.eq.0).and.(num_leftover_global.gt.0)) call account_bin(num_leftover_global)
  call mpi_reduce(ns(0),nsglobal(0),MAXBIN+1,MPI_INTEGER8,MPI_SUM,0,MPI_COMM_WORLD,err)
  if(pe_me.eq.0) then
    do i = MAXBIN-1, 0, -1
      nsglobal(i) = nsglobal(i) + nsglobal(i+1)
    end do
  end if
  call mpi_reduce(chi,chiglobal,1,MPI_REAL8,MPI_SUM,0,MPI_COMM_WORLD,err)
  call mpi_reduce(nfree,nfreeglobal,1,MPI_INTEGER8,MPI_SUM,0,MPI_COMM_WORLD,err)
  call mpi_reduce(nocc,noccglobal,1,MPI_INTEGER8,MPI_SUM,0,MPI_COMM_WORLD,err)
  call mpi_reduce(num_acc_sites,num_acc_sites_global,1,MPI_INTEGER8,MPI_SUM,0,MPI_COMM_WORLD,err)
  call mpi_reduce(num_glo_labels,num_glo_labels_global,1,MPI_INTEGER8,MPI_MAX,0,MPI_COMM_WORLD,err)
  t_facct = mpi_wtime() - t_facct_s
end subroutine

subroutine output()
  integer :: i

  if(pe_me.eq.0) then
    print *, '# Num. recycl.: ', nrecyc
    print *, '# Max num. global labels after recyc: ', num_glo_labels_global
    print *, '# Occupied: ', 1.0d0*noccglobal/Lsystem
    print *, '# Free: ', 1.0d0*nfreeglobal/Lsystem
    print *, '# Chi = ', chiglobal/Lsystem
    print *, '# Number density: ', 1.0d0*nsglobal(0)/Lsystem
    print *, '# n_occ = ', noccglobal
    print *, '# n_acc = ', num_acc_sites_global + num_leftover_global
    do i = 0, MAXBIN
      print *, (one*2)**i, nsglobal(i)
    end do
  end if
end subroutine

subroutine site_Ndim(here)
  integer, intent(in) :: here
  integer*8 :: new, ici
  integer   :: left, top, third, fourth, fifth
  integer*8 :: nleft, ntop, nthird, nfourth, nfifth
  logical occupied, left_occ, top_occ, third_occ, fourth_occ, fifth_occ
  logical global_present

  if( rnd_gen() .ge. IP ) then
    ! This site is not occupied
    nfree = nfree + 1
    plane(here) = MAXLOC
  else
    ! This site is occupied
    nocc = nocc + 1
    ici = 1
    top = here
    left = here - 1
    if(D .ge. 3) third = here - L
    if(D .ge. 4) fourth = here - L**2
    if(D .ge. 5) fifth = here - L**3
    top_occ  = (plane(top ) .lt. MAXLOC)
    left_occ = (plane(left) .lt. MAXLOC)
    occupied = top_occ .or. left_occ
    if(D .ge. 3) then
      third_occ = (plane(third) .lt. MAXLOC)
      occupied = occupied .or. third_occ
    end if
    if(D .ge. 4) then
      fourth_occ = (plane(fourth) .lt. MAXLOC)
      occupied = occupied .or. fourth_occ
    end if
    if(D .ge. 5) then
      fifth_occ = (plane(fifth) .lt. MAXLOC)
      occupied = occupied .or. fifth_occ
    end if
    if( .not. occupied ) then
      ! Neighbours are not occupied, start a new cluster
      new = locptr
      call advance_locptr()
      plane(here) = new
      local(new) = -2*ici
    else
      ! At least one neighbour is occupied
      ! If there is only one cluster adjacent, simply connect to it.
      ! If there are several different clusters, there are several cases:
      !   1. all are local -> keep smallest label, redirect all others;
      !   2. all but one are local, one is global -> keep global label,
      !      redirect all others;
      !   3. more than one is global -> keep smallest (i.e. min local())
      !      global label, generate pairing info for other globals, redirect
      !      all labels.
      global_present = .false.
      new   = MAXLOC
      nleft = MAXLOC
      ntop  = MAXLOC
      if(D .ge. 3) nthird   = MAXLOC
      if(D .ge. 4) nfourth  = MAXLOC
      if(D .ge. 5) nfifth   = MAXLOC
      ! Left is certainly a root label, top not necessarily
      if( left_occ ) then
        nleft = root(plane(left))
        new = nleft
        if(iand(local(nleft),one).ne.0) global_present = .true.
      end if
      if( top_occ  ) then
        ntop  = root(plane(top))
        if(global_present) then
          if(iand(local(ntop),one).ne.0) then
            new = min(new, ntop)
          end if
        else
          if(iand(local(ntop),one).ne.0) then
            new = ntop
            global_present = .true.
          else
            new = min(new, ntop)
          end if
        end if
      end if
      if(D .ge. 3) then
        if( third_occ  ) then
          nthird  = root(plane(third))
          if(global_present) then
            if(iand(local(nthird),one).ne.0) then
              new = min(new, nthird)
            end if
          else
            if(iand(local(nthird),one).ne.0) then
              new = nthird
              global_present = .true.
            else
              new = min(new, nthird)
            end if
          end if
        end if
      end if
      if(D .ge. 4) then
        if( fourth_occ  ) then
          nfourth  = root(plane(fourth))
          if(global_present) then
            if(iand(local(nfourth),one).ne.0) then
              new = min(new, nfourth)
            end if
          else
            if(iand(local(nfourth),one).ne.0) then
              new = nfourth
              global_present = .true.
            else
              new = min(new, nfourth)
            end if
          end if
        end if
      end if
      if(D .ge. 5) then
        if( fifth_occ  ) then
          nfifth  = root(plane(fifth))
          if(global_present) then
            if(iand(local(nfifth),one).ne.0) then
              new = min(new, nfifth)
            end if
          else
            if(iand(local(nfifth),one).ne.0) then
              new = nfifth
              global_present = .true.
            else
              new = min(new, nfifth)
            end if
          end if
        end if
      end if
```

```
      ! End of sanity check.
      ! Local clusters which are not equal new can simply be redirected to
      ! new, regardless if new is local or global.
      ! For global clusters, the situation is different: pairing info needs
      ! to be generated.
      if( left_occ ) then
        if(iand(local(nleft),one).eq.0) then
          ici = ici - local(nleft)/2
        else
          ici = ici - globsites(-local(nleft)/2)/2
        end if
        if( nleft .ne. new ) then
          if(iand(local(nleft),one).ne.0) then
            ! new and nleft are different, and both are global and have to
            ! be joined.
            call gen_pair(new, nleft)
            globsites(-local(nleft)/2) = 0
            ! Try:
            globneigh(0:1,-local(nleft)/2) = MAXLOC
          end if
          local(nleft) = new
        end if
      end if
      if( top_occ ) then
        if( ntop .ne. nleft ) then
          if(iand(local(ntop),one).eq.0) then
            ici = ici - local(ntop)/2
          else
            ici = ici - globsites(-local(ntop)/2)/2
          end if
          if(ntop.ne.new) then
            if(iand(local(ntop),one).ne.0) then
              call gen_pair(new, ntop)
              globsites(-local(ntop)/2) = 0
              ! Try:
              globneigh(0:1,-local(ntop)/2) = MAXLOC
            end if
            local(ntop) = new
          end if
        end if
      end if
      if(D .ge. 3) then
        if( third_occ ) then
          if((nthird.ne.nleft).and.(nthird.ne.ntop)) then
            if(iand(local(nthird),one).eq.0) then
              ici = ici - local(nthird)/2
            else
              ici = ici - globsites(-local(nthird)/2)/2
            end if
            if(nthird.ne.new) then
              if(iand(local(nthird),one).ne.0) then
                call gen_pair(new, nthird)
                globsites(-local(nthird)/2) = 0
              end if
              local(nthird) = new
            end if
          end if
        end if
      end if
      if(D .ge. 4) then
        if( fourth_occ ) then
          if((nfourth.ne.nleft).and.(nfourth.ne.ntop).and.(nfourth.ne.nthird))&
          then
            if(iand(local(nfourth),one).eq.0) then
              ici = ici - local(nfourth)/2
            else
              ici = ici - globsites(-local(nfourth)/2)/2
            end if
            if(nfourth.ne.new) then
              if(iand(local(nfourth),one).ne.0) then
                call gen_pair(new, nfourth)
                globsites(-local(nfourth)/2) = 0
              end if
              local(nfourth) = new
            end if
          end if
        end if
      end if
      if(D .ge. 5) then
        if( fifth_occ ) then
          if((nfifth.ne.nleft).and.(nfifth.ne.ntop).and.(nfifth.ne.nthird)&
             .and.(nfifth.ne.nfourth)) then
            if(iand(local(nfifth),one).eq.0) then
              ici = ici - local(nfifth)/2
            else
              ici = ici - globsites(-local(nfifth)/2)/2
            end if
            if(nfifth.ne.new) then
              if(iand(local(nfifth),one).ne.0) then
                call gen_pair(new, nfifth)
                globsites(-local(nfifth)/2) = 0
              end if
              local(nfifth) = new
            end if
          end if
        end if
      end if
      ! Write back number of sites. Differentiate between local and global
      ! labels.
      plane(here) = new
      if(global_present) then
        globsites(-local(new)/2) = -2*ici-1
      else
        local(new) = -2*ici
      end if
    end if
  end if
end subroutine

! root(MAXLOC) gives MAXLOC, to make life easier
integer*8 recursive function root(lab) result(rootlabel)
  integer*8, intent(in) :: lab

  if(lab .eq. 0) print *, 'Error: root(0)!'
  if(lab .eq. MAXLOC) then
    rootlabel = MAXLOC
  else if(local(lab) .lt. 0) then
    rootlabel = lab
  else if (local(lab) .eq. 0) then
    print *, 'ERROR: Dead end! lab = ', lab
    print *, 'pe = ', pe_me,', glob_line = ', glob_line, ', glob_site = ', glob_site
    call mpi_abort(MPI_COMM_WORLD, 0, 0)
  else
    local(lab) = root(local(lab))
    rootlabel = local(lab)
  end if
end function

subroutine gen_pair(lone, ltwo)
  integer*8, intent(in) :: lone, ltwo
  integer*8  :: gone, gtwo

  if(sanity_checks) then
    ! Are lone, ltwo root labels?
```

```
      if(local(lone).ge.0) print *, &
      'Error: in gen_pair, lone is not root label, pe=', pe_me,&
      ', line=',glob_line,', site=',glob_site
      if(local(ltwo).ge.0) print *, &
      'Error: in gen_pair, ltwo is not root label, pe=', pe_me,&
      ', line=',glob_line,', site=',glob_site
      ! Are lone, ltwo global labels?
      if(iand(local(lone),one).eq.0) print *, &
      'Error: local label lone fed to gen_pair. pe=',pe_me,', line=',&
      glob_line,', site=',glob_site,'lone=',lone
      if(iand(local(ltwo),one).eq.0) print *, &
      'Error: local label ltwo fed to gen_pair. pe=',pe_me,', line=',&
      glob_line,', site=',glob_site,'ltwo=',ltwo
    end if
    gone = -local(lone)/2
    gtwo = -local(ltwo)/2
    if(sanity_checks) then
      ! Are gone, gtwo valid?
      if((gone.lt.1).or.(gone.gt.MAXGLO-1)) print *, &
      'Error: in gen_pair, gone is invalid, pe=', pe_me,', line=',glob_line,&
      ', site=',glob_site,'gone=',gone
      if((gtwo.lt.1).or.(gtwo.gt.MAXGLO-1)) print *, &
      'Error: in gen_pair, gtwo is invalid, pe=', pe_me,', line=',glob_line,&
      ', site=',glob_site,'gtwo=',gtwo
      if(globsites(gone).eq.0) print *, &
      'Error: in gen_pair, globsites(gone)=0, pe=', pe_me,', line=',glob_line,&
      ', site=',glob_site,'gone=',gone
      if(globsites(gtwo).eq.0) print *, &
      'Error: in gen_pair, globsites(gtwo)=0, pe=', pe_me,', line=',glob_line,&
      ', site=',glob_site,'gtwo=',gtwo
      ! Are gone, gtwo different?
      if(gone.eq.gtwo) print *, &
      'Error: in gen_pair, gone=gtwo, pe=', pe_me,', line=',glob_line,&
      ', site=',glob_site,'gone=',gone
    end if
    call half_pair(gone, gtwo, idx_pred)
    call half_pair(gone, gtwo, idx_succ)
  end subroutine

  subroutine half_pair(gone, gtwo, idx)
    integer*8, intent(in) :: gone, gtwo
    integer, intent(in) :: idx
    integer*8 :: pone, ptwo, smaller, larger

    pone = globneigh(idx, gone)
    ptwo = globneigh(idx, gtwo)
    smaller = min(pone, ptwo)
    larger  = max(pone, ptwo)
    if(sanity_checks.and.(smaller.eq.0)) print *, &
    'Error: label 0 half_pair. pe = ',pe_me,', glob_line = ',glob_line,', &
    glob_site = ',glob_site,', idx = ', idx
    globneigh(idx, gone) = smaller
    if(larger.lt.MAXLOC) then
      pairdata(pairptr(idx)  , idx) = smaller
      pairdata(pairptr(idx)+1, idx) = larger
      call advance_pairptr(idx)
    endif
  end subroutine

  ! pred-succ-direction:            receive      send
  !   offset = 0             pred --------> me --------> succ
  !   idx = idx_pred
  !   pe_recv = pe_pred
  !   pe_send = pe_succ
  ! succ-pred-direction:             send        receive
  !   offset = Lplane-Lline   pred <-------- me <-------- succ
  !   idx = idx_succ
```

```
!   pe_recv = pe_succ
!   pe_send = pe_pred
subroutine neighbour_exchange(offset, idx, pe_recv, pe_send)
  integer offset
  integer idx         ! idx_pred, idx_succ
  integer*4, intent(in) :: pe_recv, pe_send

  integer   i
  integer*8 nrlab, first, second, smaller, larger
  integer   :: tag, status(MPI_STATUS_SIZE), err

  call reclassify_borders()
  call mpi_sendrecv(&
         plane(Lplane-Lline-offset), Lline, MPI_INTEGER8, pe_send, 5,&
               border_recv(0),       Lline, MPI_INTEGER8, pe_recv, 5,&
               MPI_COMM_WORLD, status, err)
  do i = 0, Lline-1
    if((border_recv(i).ne.MAXLOC).and.(plane(offset+i).ne.MAXLOC)) then
      ! There is an interconnection
      nrlab = root(plane(offset+i))
      if(iand(local(nrlab),one).eq.0) then
        ! Is a local label, transfer it to global
        globsites(gloptr) = local(nrlab) - 1
        if(sanity_checks.and.(border_recv(i).eq.0)) print *, &
        'Error: label 0 in neighbour_exchange'
        globneigh(idx,gloptr) = border_recv(i)
        globneigh(1-idx,gloptr) = MAXLOC
        local(nrlab) = -2*gloptr-1
        call advance_gloptr()
      else
        ! Is a global label. Three cases possible:
        ! 1. labels in border_recv(i) and globneigh(idx,-local(nrlab)/2) are
        !    the same -> don't do anything;
        ! 2. globneigh(idx,-local(nrlab)/2) = MAXLOC -> connect the global
        !    label to border_recv(i);
        ! 3. labels in border_recv(i) and globneigh(idx,-local(nrlab)/2) are
        !    different -> tell our neighbour to join these (pairing).
        first = border_recv(i)
        second = globneigh(idx,-local(nrlab)/2)
        if(second.eq.MAXLOC) then
          if(first.eq.0) print *, 'Error: label 0 in neighbour_exchange'
          globneigh(idx,-local(nrlab)/2) = first
        else if(first .ne. second) then
          smaller = min(first, second)
          larger  = max(first, second)
          if(smaller.eq.0) print *, 'Error: label 0 in neighbour_exchange'
          globneigh(idx,-local(nrlab)/2) = smaller
          pairdata(pairptr(idx),idx)   = smaller
          pairdata(pairptr(idx)+1,idx) = larger
          call advance_pairptr(idx)
        end if
      end if
    end if
  end do
end subroutine

! Receive pairing information from idx neighbour. Process it. During the
! processing, pairing information for the 1-idx neighbour can arise.
! pred-succ-direction:        receive       send
!   idx = idx_pred      pred --------> me --------> succ
!   pe_recv = pe_pred
!   pe_send = pe_succ
! succ-pred-direction:         send        receive
!   idx = idx_succ      pred <-------- me <-------- succ
!   pe_recv = pe_succ
!   pe_send = pe_pred
subroutine pairing_exchange(idx, pe_recv, pe_send)
```

```
  integer idx
  integer*4, intent(in) :: pe_recv, pe_send
  integer recvptr

  tag = 1
  ! Add sentinel
  pairdata(pairptr(1-idx), 1-idx) = 0
  call mpi_sendrecv(&
         pairdata(0,1-idx), MAXPAIR+1, MPI_INTEGER8, pe_send, 4,&
         pairrecv(0), MAXPAIR+1, MPI_INTEGER8, pe_recv, 4,&
         MPI_COMM_WORLD, status, err)
  pairptr(1-idx) = 0
  recvptr = 0
  do
    if(pairrecv(recvptr) .eq. 0) exit           ! reached sentinel
    call pair_a_pair(pairrecv(recvptr), pairrecv(recvptr+1), idx)
    recvptr = recvptr+2
    if(sanity_checks.and.(recvptr .ge. MAXPAIR)) then              ! CANTHAPPEN
      print *, 'Error: recvptr .ge. MAXPAIR: CANTHAPPEN'
      call mpi_abort(MPI_COMM_WORLD, 0, 0)
    end if
  end do
end subroutine

subroutine pair_a_pair(ione, itwo, idx)
  integer*8, intent(in) :: ione, itwo
  integer, intent(in)   :: idx
  integer*8 :: lone, ltwo, gone, gtwo, pone, ptwo, smaller, larger

  lone = root(ione)
  ltwo = root(itwo)
  if(lone .ne. ltwo) then
    smaller = min(lone, ltwo)
    larger  = max(lone, ltwo)
    lone = smaller
    ltwo = larger
    gone = -local(lone)/2             ! the label which will be kept
    gtwo = -local(ltwo)/2             ! the label which will be discarded
    ! As pairing info came from idx neighbour, the labels are already
    ! joined there. Pick the smaller label and keep it, ignore the larger.
    if(sanity_checks.and.(min(globneigh(idx,gone),globneigh(idx,gtwo)).eq.0)) &
    print *, 'Error: label 0 in pair_a_pair, pe = ',pe_me,',line=',&
    glob_line,'site=',glob_site
    globneigh(idx,gone) = min(globneigh(idx,gone),globneigh(idx,gtwo))
    pone = globneigh(1-idx,gone)
    ptwo = globneigh(1-idx,gtwo)
    smaller = min(pone, ptwo)
    larger  = max(pone, ptwo)
    pone    = smaller
    ptwo    = larger
    if((pone.ne.MAXLOC).and.(ptwo.ne.MAXLOC).and.(pone.ne.ptwo)) then
      pairdata(pairptr(1-idx)  ,1-idx) = pone
      pairdata(pairptr(1-idx)+1,1-idx) = ptwo
      call advance_pairptr(1-idx)
    end if
    if(sanity_checks.and.(pone.eq.0)) print *, &
    'Error: label 0 in pair_a_pair, pe = ',pe_me,',line=',glob_line,&
    'site=',glob_site
    globneigh(1-idx,gone) = pone
    globsites(gone) = 2*(globsites(gone)/2+globsites(gtwo)/2)-1
    globsites(gtwo) = 0
    globneigh(0:1,gtwo) = MAXLOC
    local(ltwo) = lone
  end if
end subroutine

! Exchange pairing information until no new pairing information arises.
```

```
subroutine full_relaxation()
  logical once_more, once_more_ored

  do
    call pairing_exchange(idx_pred, pe_pred, pe_succ)
    call pairing_exchange(idx_succ, pe_succ, pe_pred)
    once_more = .false.
    if((pairptr(idx_pred).ne.0).or.(pairptr(idx_succ).ne.0)) once_more = .true.
    call mpi_allreduce(once_more, once_more_ored, 1, MPI_LOGICAL, MPI_LOR, &
           MPI_COMM_WORLD, err)
    if(.not. once_more_ored) exit
  end do
end subroutine

subroutine concentrate()
  integer num, max_num

  do
    num = concentrate_once()
    call mpi_allreduce(num,max_num,1,MPI_INTEGER,MPI_MAX,MPI_COMM_WORLD,err)
    if(max_num .eq. 0) exit
  end do
end subroutine

integer function concentrate_once()
  integer   i, ptr, num
  integer*8 lab
  logical   once_more, any_wants_more

  once_more = .true.
  num = 0
  i = 1
  do
    ptr = 0
    do
      if((local(i).lt.0).and.(iand(local(i),one).ne.0)) then
        lab = -local(i)/2
        if((globneigh(idx_succ,lab).eq.MAXLOC).and.(globsites(lab).lt.0)) then
          ! no succ. neighb., global label is valid (important when used
          ! during global recycling).
          sendbuf(ptr) = globneigh(idx_pred,lab)
          sendbuf(ptr+1) = globsites(lab)
          globsites(lab) = 0
          globneigh(0:1,lab) = MAXLOC
          local(i) = 0
          ptr = ptr + 2
          num = num + 1
        end if
      end if
      i = i + 1
      if((ptr.ge.MAXBUF-1).or.(i.eq.MAXLOC)) exit
    end do
    if(i.eq.MAXLOC) once_more = .false.
    sendbuf(ptr) = 0     ! add sentinel
    call conc_helper()
    call mpi_allreduce(once_more, any_wants_more, 1, MPI_LOGICAL, MPI_LOR, &
           MPI_COMM_WORLD, err)
    if(.not. once_more) exit
  end do
  sendbuf(0) = 0      ! No more data to transmit
  do
    if(.not. any_wants_more) exit
    call conc_helper()
    call mpi_allreduce(once_more, any_wants_more, 1, MPI_LOGICAL, MPI_LOR, &
           MPI_COMM_WORLD, err)
  end do
  concentrate_once = num
```

```
end function

subroutine conc_helper()
  integer   ptr
  integer*8 lab, ici, gl, signum

  call mpi_sendrecv(sendbuf(0), MAXBUF+1, MPI_INTEGER8, pe_pred, 6, &
                    recvbuf(0), MAXBUF+1, MPI_INTEGER8, pe_succ, 6, &
                    MPI_COMM_WORLD, status, err)
  ptr = 0
  do
    if(recvbuf(ptr).eq.0) exit     ! reached sentinel
    lab = root(recvbuf(ptr))
    if(sanity_checks.and.(iand(local(lab),one).eq.0)) then
      print *, 'Error: local label received in conc_helper, pe = ', pe_me, &
      ', lab = ', recvbuf(ptr)
      call mpi_abort(MPI_COMM_WORLD, 0, 0)
    end if
    gl = -local(lab)/2
    if(sanity_checks.and.(recvbuf(ptr+1).ge.0)) print *, &
    'Error: illegal recvbuf in conc_helper'
    ! When doing concentration during recycling, we must honor and preserve the
    ! sign of globsites (positive if label is alive in this strip, negative if
    ! label is dead in this strip).
    if(globsites(gl).lt.0) then
      signum = -1
    else
      signum = 1
    end if
    ici = 2 * (signum*globsites(gl)/2 - recvbuf(ptr+1)/2) + 1 ! ici is positive!
    globsites(gl) = signum*ici          ! can be positive or negative during recycling
    globneigh(idx_succ, gl) = MAXLOC
    if(globneigh(idx_pred,gl).eq.MAXLOC) then    ! convert to local
      ici = globsites(gl)/2
      if(ici.gt.0) ici=-ici     ! For local labels, always negative
      globsites(gl) = 0
      globneigh(0:1,gl) = MAXLOC
      local(recvbuf(ptr)) = 2 * ici
    end if
    ptr = ptr + 2
  end do
end subroutine

subroutine local_accounting()
  integer i
  integer*8 :: gl

  call prepare_recycling(idx_pred,pe_pred,pe_succ)
  call prepare_recycling(idx_succ,pe_succ,pe_pred)
  do i = 1, MAXLOC
    if((local(i).lt.0).and.(iand(local(i),one).eq.0)) call account_label(i)
    if((local(i).lt.0).and.(iand(local(i),one).eq.1)) then
      gl = -local(i)/2
      if(globsites(gl).ne.0) &
        print *, '# Leftover global label g=',gl,&
        ', local=', i, &
        ', succ=', globneigh(idx_succ,gl), &
        ', pred=', globneigh(idx_pred,gl), &
        ', pe=',pe_me,', size=', globsites(gl)
        num_leftover = num_leftover - globsites(gl)/2
    end if
  end do
end subroutine

subroutine reclassify_borders()
  integer i

```

```
    do i = 0, Lline-1
      plane(i) = root(plane(i))
    end do
    do i = Lplane-Lline, Lplane-1
      plane(i) = root(plane(i))
    end do
  end subroutine

  subroutine garbage_collection()
    integer i
    integer*8 :: lab, gl, numglolabs

    nrecyc = nrecyc + 1
    call full_relaxation()   ! Exchange pairing info until there no longer is any
    call prepare_recycling(idx_pred, pe_pred, pe_succ) ! Reclassify all of our
    call prepare_recycling(idx_succ, pe_succ, pe_pred) ! neighbours' pointers
    call reclassify_planes() ! Reclassify upper() and plane()
    ! According to theory, no one anywhere references any longer our non-root
    ! labels (neither local nor global ones), so we are allowed to delete them
    ! all. While doing this, mark all global labels with inverted sign, so that
    ! they aren't discarded in the later process.
    do i = 1, MAXLOC
      if(local(i) .ge. 0) local(i) = 0
    end do
    ! Walk through upper() and plane() and mark all living labels with an
    ! inverted sign. globsites() of global labels are marked, too.
    do i = 0, Lplane-1
      lab = plane(i)
      if(local(lab).lt.0) then
        if(iand(local(lab),one).ne.0) then  ! is global
          gl = -local(lab)/2
          if(sanity_checks.and.(globsites(gl).ge.0)) then
            print *, 'Error: local label pointing to illegal global label'
            stop 7
          end if
          globsites(gl) = -globsites(gl) ! Mark living global label with pos. sign
        end if
        local(lab) = -local(lab) ! Mark living label with pos. sign
      end if
      if(fully_periodic) then
        lab = upper(i)
        if(local(lab).lt.0) then
          if(iand(local(lab),one).ne.0) then  ! is global
            gl = -local(lab)/2
            if(globsites(gl).ge.0) then
              print *, 'Error: local label pointing to illegal global label'
              stop 7
            end if
            globsites(gl) = -globsites(gl) ! Mark lvng glbl label with pos. sign
          end if
          local(lab) = -local(lab) ! Mark living label with pos. sign
        end if
      end if
    end do
    ! Throw away labels with negative sign, unless they are global labels.
    ! (otherwise they are dead local labels).
    ! Count them. Flip back labels with positive sign. Don't yet flip the
    ! corresponding globsites() for global labels.
    if(sanity_checks) then
      do i = 1, MAXLOC
        if(local(i).lt.0) then
          if(iand(local(i),one).ne.0) then
            if(globsites(-local(i)/2).gt.0) print *, 'Error: local<0, global>0'
            if(globsites(-local(i)/2).eq.0) print *, 'Error: local<0, global=0'
          end if
        end if
        if(local(i).gt.0) then
          if(iand(local(i),one).ne.0) then
            if(globsites(local(i)/2).lt.0) print *, 'Error: local>0, global<0'
            if(globsites(local(i)/2).eq.0) print *, 'Error: local>0, global=0'
          end if
        end if
      end do
    end if
    do i = 1, MAXLOC
      if((local(i).lt.0).and.(iand(local(i),one).eq.0)) then
        call account_label(i)
        local(i) = 0
      end if
      if(local(i).gt.0) local(i) = -local(i)
    end do
    ! Now we have the following situation: there are no indirect labels.
    ! All dead local clusters are cleaned. Present are living local clusters
    ! and all local labels that point to global labels. The globsites for
    ! global labels are negative if they are dead in this strip, and positive,
    ! if they are alive in this strip.
    ! Now we can do concentration.
    if(sanity_checks) then
      ! All locals must be negative or zero
      do i = 1, MAXLOC
        if(local(i).gt.0) print *, 'Error: positive local during gc'
        if((local(i).lt.0).and.(iand(local(i),one).ne.0).and.(globsites(-local(i)/2).eq.0)) &
        print *, 'Error: dangling local to global'
      end do
    end if
    call concentrate()
    ! Flip back all pos. globsites to negative (these were the globals that are
    ! alive in this strip).
    numglolabs = 0
    do i = 1, MAXGLO
      if(globsites(i).gt.0) globsites(i)=-globsites(i)
      if(globsites(i).ne.0) numglolabs = numglolabs + 1
    end do
    num_glo_labels = max(num_glo_labels, numglolabs) ! Number of alive globals after recyc
    ! Let locptr point to the first free label.
    ! Let gloptr point to the first free entry.
    locptr = 0
    call advance_locptr()
    gloptr = 0
    call advance_gloptr()
  end subroutine

  subroutine preprec_forthnback(pe_recv, pe_send)
    integer*4, intent(in) :: pe_recv, pe_send
    integer ptr

    call mpi_sendrecv(preprec_send(0), MAXPREP+1, MPI_INTEGER8, pe_send, 7, &
                      preprec_recv(0), MAXPREP+1, MPI_INTEGER8, pe_recv, 7, &
                      MPI_COMM_WORLD, status, err)
    ! reclassify labels
    ptr = 0
    do
      if(preprec_recv(ptr).eq.0) exit      ! reached sentinel
      if(sanity_checks.and.(preprec_recv(ptr).lt.0)) print *, 'Error: negative preprec_recv!'
      preprec_recv(ptr) = root(preprec_recv(ptr))
      ptr = ptr + 1
    end do
    call mpi_sendrecv(preprec_recv(0), MAXPREP+1, MPI_INTEGER8, pe_recv, 8, &
                      preprec_send(0), MAXPREP+1, MPI_INTEGER8, pe_send, 8, &
                      MPI_COMM_WORLD, status, err)
  end subroutine

  ! pred-succ-direction:        receive       send
  !   idx = idx_pred      pred --------> me --------> succ
  !   pe_recv = pe_pred
  !   pe_send = pe_succ
  ! succ-pred-direction:         send        receive
  !   idx = idx_succ      pred <-------- me <-------- succ
  !   pe_recv = pe_succ
  !   pe_send = pe_pred
  subroutine prepare_recycling(idx, pe_recv, pe_send)
    integer idx
    integer*4, intent(in) :: pe_recv, pe_send
    integer ptr, i, begin
    logical once_more, any_wants_more

    once_more = .true.
    i = 1
    begin = i
    do
      ptr = 0
      do
        if((globsites(i).ne.0).and.(globneigh(1-idx,i).ne.MAXLOC)) then
          preprec_send(ptr) = globneigh(1-idx,i)
          ptr = ptr + 1
        end if
        i = i + 1
        if((ptr.ge.MAXPREP-1).or.(i.eq.MAXGLO)) exit
      end do
      if(i.eq.MAXGLO) once_more = .false.
      preprec_send(ptr) = 0      ! add sentinel
      call preprec_forthnback(pe_recv, pe_send)
      call mpi_allreduce(once_more, any_wants_more, 1, MPI_LOGICAL, MPI_LOR, &
            MPI_COMM_WORLD, err)
      ptr = 0
      i = begin
      do
        if(preprec_send(ptr).eq.0) exit    ! reached sentinel
        if((globsites(i).ne.0).and.(globneigh(1-idx,i).ne.MAXLOC)) then
          if(sanity_checks.and.(preprec_send(ptr).eq.0)) print *, 'Error: label 0 in pre-
  pare_recycling'
          globneigh(1-idx,i) = preprec_send(ptr)
          ptr = ptr + 1
        end if
        i = i + 1
      end do
      begin = i
      if(.not. once_more) exit
    end do
    preprec_send(0) = 0      ! We no longer have data to transmit
    do
      if(.not. any_wants_more) exit
      call preprec_forthnback(pe_recv, pe_send)
      call mpi_allreduce(once_more, any_wants_more, 1, MPI_LOGICAL, MPI_LOR, &
            MPI_COMM_WORLD, err)
    end do
  end subroutine

  subroutine reclassify_planes()
    integer i

    do i = 0, Lplane-1
      plane(i) = root(plane(i))
      if(fully_periodic) upper(i) = root(upper(i))
    end do
  end subroutine

  ! Let locptr point to the next free label
  subroutine advance_locptr()

    do
```

```
      locptr = locptr + 1
      if( locptr .ge. MAXLOC) then
        print *, 'Error: locptr .ge. MAXLOC'
        call mpi_abort(MPI_COMM_WORLD, 0, 0)
      end if
      if( local(locptr) .eq. 0 ) exit
    end do
  end subroutine

  ! Let gloptr point to the next free label
  subroutine advance_gloptr()

    do
      gloptr = gloptr + 1
      if( gloptr .ge. MAXGLO) then
        print *, 'Error: gloptr .ge. MAXGLO'
        call mpi_abort(MPI_COMM_WORLD, 0, 0)
      end if
      if( globsites(gloptr) .eq. 0 ) exit
    end do
  end subroutine

  subroutine advance_pairptr(idx)
    integer idx

    pairptr(idx) = pairptr(idx)+2
    if(pairptr(idx) .ge. MAXPAIR) then
      print *, 'Error: pairptr(idx) .ge. MAXPAIR'
      call mpi_abort(MPI_COMM_WORLD, 0, 0)
    end if
  end subroutine

  subroutine connect_top_and_bottom()
    integer   :: i
    integer*8 :: ui, pi, smaller, larger
    logical   :: ui_loc, pi_loc

    do i = 0, Lplane-1
      glob_site = i
      ui = root(upper(i))
      pi = root(plane(i))
      if((ui.eq.MAXLOC).or.(pi.eq.MAXLOC)) cycle
      if(ui.eq.pi) cycle
      ui_loc = ((iand(local(ui),one)).eq.0)
      pi_loc = ((iand(local(pi),one)).eq.0)
      smaller = min(ui,pi)
      larger  = max(ui,pi)
      if(ui_loc .and. pi_loc) then        ! both local
        local(smaller) = local(smaller) + local(larger)
        local(larger) = smaller
      else if((.not. ui_loc).and.(.not. pi_loc)) then  ! both global
        globsites(-local(smaller)/2) = &
          2* (globsites(-local(smaller)/2)/2 &
        + globsites(-local(larger)/2)/2) -1
        call gen_pair(smaller, larger)
        globsites(-local(larger)/2) = 0
        globneigh(0:1,-local(larger)/2) = MAXLOC
        local(larger) = smaller
      else                                ! one local, one global
        if(ui_loc) then
          smaller = pi     ! smaller is the global
          larger  = ui     ! larger is the local
        else
          smaller = ui     ! smaller is the global
          larger  = pi     ! larger is the local
        end if
        globsites(-local(smaller)/2) = 2 * (globsites(-local(smaller)/2)/2 + &
                                          local(larger)/2) - 1
        local(larger) = smaller
      end if
    end do
  end subroutine

  subroutine account_label(lab)
    integer lab
    integer ibin

    num_acc_sites = num_acc_sites - local(lab)/2
    call account_bin(-local(lab)/2)
    call account_chi(-local(lab)/2)
  end subroutine

  subroutine account_bin(sites)
    integer*8, intent(in) :: sites
    integer ibin

    ibin = dlog(1.0d0*sites)*rcplog+0.00001d0
    if(ibin .le. MAXBIN) ns(ibin) = ns(ibin) + 1
  end subroutine

  subroutine account_chi(sites)
    integer*8, intent(in) :: sites

    chi = chi + sites*sites
  end subroutine

  ! 32 bit LFG: R(471,1586,6988,9689)
  subroutine rnd_randomize(iseed, seedshift)
    integer*4 :: iseed, seedshift
    integer   :: i, ii
    integer*4 :: ibm, ici

    ibm = 2*iseed-1
    do i = 0, seedshift
      ibm = 65539 * ibm
    end do
    do i = 0, RNDMAX
      ici = 0
      do ii = 1, 32               ! 32-bit version
        ici = ishft(ici, 1)
        ibm = ibm * 16807
        if( ibm .lt. 0 ) ici = ior(ici, 1)
      end do
      rnd_data(i) = ici
    end do
    rnd_idx = 0
    do i = 1, 8*(RNDMAX+1)        ! "heat up" generator
      ibm = rnd_gen()
    end do
  end subroutine

  integer*4 function rnd_gen()
    rnd_idx = iand(rnd_idx + 1, RNDMAX)
    rnd_data(rnd_idx) = ieor(&
    ieor(rnd_data(iand(rnd_idx-471,RNDMAX)),rnd_data(iand(rnd_idx-1586,RNDMAX))),&
    ieor(rnd_data(iand(rnd_idx-6988,RNDMAX)),rnd_data(iand(rnd_idx-9689,RNDMAX))))
    rnd_gen = rnd_data(rnd_idx)
  end function

  ! Only for debugging purposes:
  subroutine sanity_check()
    integer   i
    integer*8 g
    do i = 1, MAXLOC
      if(local(i).eq.0) cycle
      if(local(i).gt.0) then
        call check_indirect(i)
      else
        if(iand(local(i),one).ne.0) then
          g = -local(i)/2
          if((g.lt.1).or.(g.ge.MAXGLO)) then
            print *, 'Error: Out of range for global label'
            cycle
          end if
          if((globsites(g).eq.0).or.(globneigh(0,g).eq.0).or.&
              (globneigh(1,g).eq.0)) then
            print *, 'Error: Zeroed global label ', g
            cycle
          end if
          if((globneigh(0,g).eq.MAXLOC).and.(globneigh(1,g).eq.MAXLOC)) then
            print *, 'Error: Unsane global label ', g
            cycle
          end if
        end if
      end if
    end do
    do i = 1, MAXGLO
      if(globsites(i).eq.0) cycle
      if((globneigh(0,i).eq.MAXLOC).and.(globneigh(1,i).eq.MAXLOC)) &
        print *, 'Error: filled global label with no neighbours! pe=',pe_me,&
                 ', line=',glob_line,', globlabel=',i
    end do
  end subroutine

  ! Only for debugging purposes:
  subroutine check_indirect(lab)
    integer, intent(in) :: lab
    integer*8 :: nxt
    integer   :: i

    i = 0
    nxt = local(lab)
    do
      i = i + 1
      if((nxt.lt.1).or.(nxt.ge.MAXLOC)) then
        print *, 'Error: Local pointer out of range for label ', lab
        exit
      end if
      if(local(nxt).eq.0) then
        print *, 'Error: Dead end for label ', lab
        exit
      end if
      if(local(nxt).lt.0) exit
      if(i.ge.100) then
        print *, 'Error: Too much indirections for label ', lab
        exit
      end if
      nxt = local(nxt)
    end do
  end subroutine

  end
```

F.3 Percolation on growing lattices

This program is serial and uses only standard Fortran 90, no extensions. Parallelization is achieved via replication, i. e. running many instances independently, using different seed values for the random number generator.

```
program Varyl3d

implicit none

integer, parameter :: Lmax = 5000
integer, parameter :: Stepmax = 64
integer, parameter :: MAX = 5e7
integer, parameter :: LIMIT = 0.7*MAX
integer, parameter :: MAXBIN = 63
integer, parameter :: RNDMAX = 16383
integer, parameter :: MAXINF = 63
integer, parameter :: VERSION_NUMBER = 1

integer*8 :: A(0:Lmax,0:Lmax), B(0:Lmax,0:Lmax), C(0:Lmax,0:Lmax)
integer*8 :: AB(0:Lmax), BC(0:Lmax), CA(0:Lmax)
integer*8 :: ABC
integer*8 :: label(0:MAX)      ! label(0) is never used!
integer*8 :: label_peri(0:MAX) ! label_peri(0) is never used!
integer*8 :: X(0:Lmax,0:Lmax), Y(0:Lmax,0:Lmax), Z(0:Lmax,0:Lmax)

integer*4 :: IP, rnd_data(0:RNDMAX), rnd_idx        ! 32-bit version
integer*4, parameter :: one32 = 1

integer*8 :: ns_inrun(0:MAXBIN)
integer*8 :: ns_open(0:MAXBIN, 0:Stepmax-2) ! L=1 is not counted
integer*8 :: ns_peri(0:MAXBIN, 0:Stepmax-2) ! L=1 is not counted
integer*8 :: largest_inrun, largest_peri
integer*8 :: largest_sum(0:Stepmax-2), largest_max(0:Stepmax-2)
integer*8 :: inf_label(0:MAXINF), inf_sum(0:Stepmax-2)
integer   :: inf_ptr
real*8    :: chi_inrun, chi_peri, chi_sum(0:Stepmax-2)
integer   :: L, step_to_L(0:Stepmax-2)
integer   :: nrlab
integer   :: stepwidth, step_nr, run_nr
integer   :: initial_seed
real*8    :: p
real*8    :: rcplog
character :: filename *100
integer   :: Runmax
integer   :: nrecyc
real*4    :: etime, dummy, tstart(2), tstop(2), tcurr(2)
real*4    :: t_firstini, t_ini, t_perc, t_anal

read *, p
read *, initial_seed
read *, Runmax

print *, '# Lmax = ', Lmax
print *, '# Stepmax = ', Stepmax
print *, '# Runmax = ', Runmax
print *, '# MAX = ', MAX
print *, '# LIMIT = ', LIMIT
print *, '# MAXBIN = ', MAXBIN
print *, '# p = ', p
print *, '# initial_seed = ', initial_seed

t_ini = 0.0
t_perc = 0.0
t_anal = 0.0

dummy = etime(tstart)
call first_initialize()
dummy = etime(tstop)
t_firstini = tstop(1)-tstart(1)
do run_nr = 0, Runmax-1
  call rnd_randomize(initial_seed+run_nr)
  dummy = etime(tcurr)
```

```fortran
    call initialize()
    dummy = etime(tstop)
    t_ini = t_ini + (tstop(1)-tcurr(1))
    dummy = etime(tcurr)
    call percolate()
    dummy = etime(tstop)
    t_perc = t_perc + (tstop(1)-tcurr(1))
  end do
  call do_output()
  dummy = etime(tstop)
  print *, '# Required runtime: ', (tstop(1)-tstart(1)), ' seconds'
  print *, '# Speed: ', 1.0d-6*Runmax*(1.0d0*Lmax)**3/(tstop(1)-tstart(1)),&
           ' MSites/s'

  contains

  subroutine do_output()
    integer :: i, j
    real*8 :: ns_out(0:MAXBIN)

    print *, '# File format: L chi_avg largest_avg largest_max inf_avg n(s)_binned'
    print *, '#            : 1    2          3          4         5      6 - ...'
    do i = 0, Stepmax-2
      do j = MAXBIN-1, 0, -1
        ns_peri(j,i) = ns_peri(j,i) + ns_peri(j+1,i)
        ns_out(j) = 1.0d0*ns_peri(j,i)/Runmax
      end do
      if(step_to_L(i) .ne. 0) then
        print *, step_to_L(i), dsqrt(chi_sum(i)/Runmax),&
                 -1.0d0*largest_sum(i)/Runmax, -largest_max(i),&
                 1.0d0*inf_sum(i)/Runmax, (ns_out(j),j=0,MAXBIN)
      end if
    end do

    print *, '# t_firstini = ', t_firstini, ' seconds'
    print *, '# t_ini = ', t_ini, ' seconds'
    print *, '# t_perc = ', t_perc, ' seconds'
    print *, '# t_anal = ', t_anal, ' seconds'
    print *, '# nrecyc = ', nrecyc
  end subroutine

  subroutine first_initialize()
    integer :: i, j

    IP = 2147483648.0d0*(2.0d0*p-1.0d0)
    rcplog = 1.0d0/dlog(2.0d0)
    nrecyc = 0

    ns_peri = 0
    largest_sum = 0
    largest_max = 0
    chi_sum = 0.0d0
    inf_sum = 0
  end subroutine

  subroutine initialize()
    integer :: i

    ns_inrun = 0
    nrlab = 1
    stepwidth = 1
    step_nr = 0
    largest_inrun = 0
    chi_inrun = 0
    label = 0
  end subroutine

```

```
subroutine percolate()
  integer :: i, j
  real*4  :: tsubstart(2), tsubstop(2)

  ! L = 1
  AB(0)  = MAX
  BC(0)  = MAX
  CA(0)  = MAX
  call site_cubic(ABC, AB(0), BC(0), CA(0))
  X(0,0) = ABC
  Y(0,0) = ABC
  Z(0,0) = ABC

  ! L > 1
  do L = 2, Lmax
    A(L-1,0:L-2) = MAX
    A(0:L-2,L-1) = MAX
    B(L-1,0:L-2) = MAX
    B(0:L-2,L-1) = MAX
    C(L-1,0:L-2) = MAX
    C(0:L-2,L-1) = MAX
    AB(L-1) = MAX
    BC(L-1) = MAX
    CA(L-1) = MAX
    ! Inner part of A
    do j = L-2, 1, -1
      do i = L-2, 1, -1
        call site_cubic(A(i,j), A(i+1,j), A(i,j+1), A(i-1,j-1))
      end do
    end do
    ! Inner part of B
    do j = L-2, 1, -1
      do i = L-2, 1, -1
        call site_cubic(B(i,j), B(i+1,j), B(i,j+1), B(i-1,j-1))
      end do
    end do
    ! Inner part of C
    do j = L-2, 1, -1
      do i = L-2, 1, -1
        call site_cubic(C(i,j), C(i+1,j), C(i,j+1), C(i-1,j-1))
      end do
    end do
    ! Outer part of A
    do i = L-2, 1, -1                               ! edges
      call site_cubic(A(i,0), A(i+1,0), A(i,1), CA(i-1))
      call site_cubic(A(0,i), A(0,i+1), A(1,i), AB(i-1))
    end do
    call site_cubic(A(0,0), A(1,0), A(0,1), ABC)    ! corner
    ! Outer part of B
    do i = L-2, 1, -1                               ! edges
      call site_cubic(B(i,0), B(i+1,0), B(i,1), AB(i-1))
      call site_cubic(B(0,i), B(0,i+1), B(1,i), BC(i-1))
    end do
    call site_cubic(B(0,0), B(1,0), B(0,1), ABC)    ! corner
    ! Outer part of C
    do i = L-2, 1, -1                               ! edges
      call site_cubic(C(i,0), C(i+1,0), C(i,1), BC(i-1))
      call site_cubic(C(0,i), C(0,i+1), C(1,i), CA(i-1))
    end do
    call site_cubic(C(0,0), C(1,0), C(0,1), ABC)    ! corner
    ! AB
    do i = L-2, 0, -1
      call site_cubic(AB(i), AB(i+1), A(0,i), B(i,0))
    end do
    ! BC
    do i = L-2, 0, -1
      call site_cubic(BC(i), BC(i+1), B(0,i), C(i,0))
    end do
    ! CA
    do i = L-2, 0, -1
      call site_cubic(CA(i), CA(i+1), C(0,i), A(i,0))
    end do
    ! ABC
    call site_cubic(ABC, AB(0), BC(0), CA(0))
    ! Now fill the buffer planes for the periodic b.c.
    do i = 0, L-2
      X(L-1,i) = B(L-2,L-2-i)
      X(i,L-1) = A(L-2-i,L-2)
      Y(L-1,i) = A(L-2,L-2-i)
      Y(i,L-1) = C(L-2-i,L-2)
      Z(L-1,i) = C(L-2,L-2-i)
      Z(i,L-1) = B(L-2-i,L-2)
    end do
    X(L-1,L-1) = AB(L-2)
    Y(L-1,L-1) = CA(L-2)
    Z(L-1,L-1) = BC(L-2)
    if ( nrlab .ge. LIMIT) then
      call garbage_collector()
    end if
    stepwidth = stepwidth - 1
    if(stepwidth .eq. 0) then
      dummy = etime(tsubstart)
      call analyze()
      largest_sum(step_nr) = largest_sum(step_nr) + largest_peri
      largest_max(step_nr) = min(largest_max(step_nr), largest_peri)
      chi_sum(step_nr) = chi_sum(step_nr) + chi_peri
      inf_sum(step_nr) = inf_sum(step_nr) + inf_ptr + 1
      step_to_L(step_nr) = L
      call new_accounting_step()
      dummy = etime(tsubstop)
      t_anal = t_anal + (tsubstop(1)-tsubstart(1))
    end if
  end do
end subroutine

subroutine site_cubic(here, pred, top, back)
  integer*8, intent(out) :: here
  integer*8, intent(in)  :: pred, top, back
  integer*8 :: npred, ntop, nback, new, ici
  logical :: pred_occ, top_occ, back_occ

  call rnd_next()
  if( rnd_get() .ge. IP ) then
    ! This site is not occupied
    here = MAX
  else
    ! This site is occupied
    ici = 1
    top_occ  = (top  .lt. MAX)
    pred_occ = (pred .lt. MAX)
    back_occ = (back .lt. MAX)
    if( .not. (top_occ .or. pred_occ .or. back_occ) ) then
      ! Neighbours are not occupied, start a new cluster
      new = nrlab
      call advance_nrlab()
      here = new
      label(new) = -ici
    else
      ! At least one neighbour is occupied
      npred = MAX
      ntop  = MAX
      nback = MAX
      if( pred_occ ) npred = root(pred)
      if( top_occ  ) ntop  = root(top)
      if( back_occ ) nback = root(back)
      new = min(npred, ntop, nback)
      if( pred_occ ) then
        ici = ici - label(npred)
        if( npred .ne. new ) label(npred) = new
      end if
      if( top_occ ) then
        if( ntop .ne. npred ) ici = ici - label(ntop)
        if( ntop .ne. new ) label(ntop) = new
      end if
      if( back_occ ) then
        if( (nback.ne.npred).and.(nback.ne.ntop) ) ici = ici - label(nback)
        if( nback .ne. new ) label(nback) = new
      end if
      here = new
      label(new) = -ici
    end if
  end if
end subroutine

! Analyze with fully periodic b. c.
subroutine analyze()
  integer :: i, j

  call reclassify_planes()
  label_peri = label
  ns_peri(0:MAXBIN,step_nr) = ns_peri(0:MAXBIN,step_nr) + ns_inrun(0:MAXBIN)
  largest_peri = largest_inrun
  chi_peri = chi_inrun
  inf_ptr = -1
  !
  ! Search for infinite clusters
  ! Z is opposite to A:
  do j = 0, L-2
    do i = 0, L-2
      call identify_infinite(A(i,j), Z(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call identify_infinite(AB(i), Z(L-2-i,L-1))
    call identify_infinite(CA(i), Z(L-1,L-2-i))
  end do
  call identify_infinite(ABC, Z(L-1,L-1))
  ! Y is opposite to B:
  do j = 0, L-2
    do i = 0, L-2
      call identify_infinite(B(i,j), Y(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call identify_infinite(BC(i), Y(L-2-i,L-1))
    call identify_infinite(AB(i), Y(L-1,L-2-i))
  end do
  call identify_infinite(ABC, Y(L-1,L-1))
  ! X is opposite to C:
  do j = 0, L-2
    do i = 0, L-2
      call identify_infinite(C(i,j), X(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call identify_infinite(CA(i), X(L-2-i,L-1))
    call identify_infinite(BC(i), X(L-1,L-2-i))
  end do
  call identify_infinite(ABC, X(L-1,L-1))
  !
  ! Make boundaries fully periodic
  ! Z is opposite to A:
  do j = 0, L-2
    do i = 0, L-2
      call connect_two_sites(A(i,j), Z(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call connect_two_sites(AB(i), Z(L-2-i,L-1))
    call connect_two_sites(CA(i), Z(L-1,L-2-i))
  end do
  call connect_two_sites(ABC, Z(L-1,L-1))
  ! Y is opposite to B:
  do j = 0, L-2
    do i = 0, L-2
      call connect_two_sites(B(i,j), Y(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call connect_two_sites(BC(i), Y(L-2-i,L-1))
    call connect_two_sites(AB(i), Y(L-1,L-2-i))
  end do
  call connect_two_sites(ABC, Y(L-1,L-1))
  ! X is opposite to C:
  do j = 0, L-2
    do i = 0, L-2
      call connect_two_sites(C(i,j), X(L-2-j,L-2-i))
    end do
  end do
  do i = 0, L-2
    call connect_two_sites(CA(i), X(L-2-i,L-1))
    call connect_two_sites(BC(i), X(L-1,L-2-i))
  end do
  call connect_two_sites(ABC, X(L-1,L-1))
  !
  ! Reclassify infinite clusters
  if( inf_ptr .gt. MAXINF ) print *, 'Warning: too many infinite clusters'
  ! Reclassify infinite labels
  do i = 0, min(inf_ptr, MAXINF)
    inf_label(i) = root_peri(inf_label(i))
  end do
  ! Account labels
  do i = 1, MAX
    if( label_peri(i) .lt. 0 ) call account_label_peri(i)
  end do
end subroutine

subroutine account_label_inrun(lab)
  integer, intent(in) :: lab
  integer :: ibin

  if(label(lab) .lt. 0) then
    ibin = dlog(-1.0d0*label(lab))*rcplog+0.00001d0
    if (ibin .le. MAXBIN) ns_inrun(ibin) = ns_inrun(ibin)+1
    largest_inrun = min(largest_inrun, label(lab))
    chi_inrun = chi_inrun+(1.0d0*label(lab))*label(lab)
  end if
end subroutine

subroutine account_label_peri(lab)
  integer, intent(in) :: lab
  integer :: ibin, i
  logical :: is_not_infinite

  if(label_peri(lab) .lt. 0) then
    ibin = dlog(-1.0d0*label_peri(lab))*rcplog+0.00001d0
    if (ibin .le. MAXBIN) ns_peri(ibin,step_nr) = ns_peri(ibin,step_nr)+1
    largest_peri = min(largest_peri, label_peri(lab))
    is_not_infinite = .true.
    do i = 0, min(inf_ptr, MAXINF)
      if( lab .eq. inf_label(i) ) then
        chi_peri = chi_peri+(1.0d0*label_peri(lab))*label_peri(lab)
      end if
    end do
  end if
end subroutine

!root(MAX) gives MAX, to make life easier
integer*8 recursive function root(lab) result(rootlabel)
  integer*8, intent(in) :: lab

  if(lab .eq. MAX) then
    rootlabel = MAX
  else if(label(lab) .lt. 0) then
    rootlabel = lab
  else if (label(lab) .eq. 0) then
    print *, 'ERROR: Dead end! L = ', L
    stop 5
  else
    label(lab) = root(label(lab))
    rootlabel = label(lab)
  end if
end function

!root_peri(MAX) gives MAX, to make life easier
!root_peri works on label_peri()
integer*8 recursive function root_peri(lab) result(rootlabel)
  integer*8, intent(in) :: lab

  if(lab .eq. MAX) then
    rootlabel = MAX
  else if(label_peri(lab) .lt. 0) then
    rootlabel = lab
  else if (label_peri(lab) .eq. 0) then
    print *, 'ERROR: Dead end! L = ', L
    stop 6
  else
    label_peri(lab) = root_peri(label_peri(lab))
    rootlabel = label_peri(lab)
  end if
end function

subroutine connect_two_sites(a, b)
  integer*8, intent(in) :: a, b
  integer*8 :: ra, rb

  ra = root_peri(a)
  rb = root_peri(b)
  if( (ra .ne. MAX) .and. (rb .ne. MAX) .and. (ra .ne. rb) ) then
    label_peri(ra) = label_peri(ra) + label_peri(rb)
    label_peri(rb) = ra
  end if
end subroutine

subroutine identify_infinite(a, b)
  integer*8, intent(in) :: a, b
  integer*8 :: ra, rb
  logical :: is_a_new_one
  integer :: i

  ra = root_peri(a)
  rb = root_peri(b)
  if( (ra .ne. MAX) .and. (ra .eq. rb) ) then
    is_a_new_one = .true.
    do i = 0, min(inf_ptr, MAXINF)
```

```
      if( ra .eq. inf_label(i) ) is_a_new_one = .false.
    end do
    if( is_a_new_one ) then
      inf_ptr = inf_ptr + 1
      if( inf_ptr .le. MAXINF ) inf_label(inf_ptr) = ra
    end if
  end if
end subroutine

subroutine garbage_collector()
  integer :: i, j

  nrecyc = nrecyc + 1
  ! First reclassify all labels in working arrays and buffers
  call reclassify_planes()
  ! Delete all indirect labels
  do i = 1, MAX
    if(label(i) .gt. 0) label(i) = 0
    ! Alternatively: label(i) = min(label(i),0)  ! May be faster on some CPUs
  end do
  ! Mark all living root labels with a positive sign, MAX is treated like all
  ! other labels. For this to work, label(MAX) has to be 0.
  do j = 0, L-2
    do i = 0, L-2
      call mark_living_label(A(i,j))
      call mark_living_label(B(i,j))
      call mark_living_label(C(i,j))
    end do
  end do
  do j = 0, L-1
    do i = 0, L-1
      call mark_living_label(X(i,j))
      call mark_living_label(Y(i,j))
      call mark_living_label(Z(i,j))
    end do
  end do
  do i = 0, L-2
    call mark_living_label(AB(i))
    call mark_living_label(BC(i))
    call mark_living_label(CA(i))
  end do
  call mark_living_label(ABC)
  ! Throw away labels with negative sign. Do not forget to count them.
  ! Invert labels with positive sign.
  do i = 1, MAX
    if(label(i) .lt. 0) then
      call account_label_inrun(i)
      label(i) = 0
    else
      label(i) = -label(i)
    end if
  end do
  ! Find the first free label and let nrlab point to it
  nrlab = 0
  call advance_nrlab()
end subroutine

subroutine reclassify_planes()
  integer :: i, j

  do j = 0, L-2
    do i = 0, L-2
      A(i,j) = root(A(i,j))
      B(i,j) = root(B(i,j))
      C(i,j) = root(C(i,j))
    end do
  end do
  do j = 0, L-1
    do i = 0, L-1
      X(i,j) = root(X(i,j))
      Y(i,j) = root(Y(i,j))
      Z(i,j) = root(Z(i,j))
    end do
  end do
  do i = 0, L-2
    AB(i) = root(AB(i))
    BC(i) = root(BC(i))
    CA(i) = root(CA(i))
  end do
  ABC = root(ABC)
end subroutine

! Mark living label (negative sign) with positive sign.
subroutine mark_living_label(lab)
  integer*8, intent(in) :: lab

  if(label(lab) .lt. 0) label(lab) = -label(lab)
end subroutine

! Let nrlab point to the next free label
subroutine advance_nrlab()

  do
    nrlab = nrlab + 1
    if( nrlab .ge. MAX) then    ! never utilize label(MAX) !
      print *, 'ERROR: not enough free labels!'
      stop 3
    end if
    if( label(nrlab) .eq. 0 ) exit
  end do
end subroutine

! Enter a new accounting step: set stepwidth
subroutine new_accounting_step()

  !stepwidth = L / 100 + 1
  if( L .lt. 10 ) then
    stepwidth = 1
  else if( L .lt. 100 ) then
    stepwidth = 5
  else if( L .lt. 1e3 ) then
    stepwidth = 50
  else if( L .lt. 1e4 ) then
    stepwidth = 5e2
  else if( L .lt. 1e5 ) then
    stepwidth = 5e3
  else
    stepwidth = 5e4
  end if
  step_nr = step_nr + 1
end subroutine

! 32 bit LFG: R(471,1586,6988,9689)
subroutine rnd_randomize(iseed)
  integer   :: iseed
  integer   :: i, ii
  integer*4 :: ibm, ici

  ibm = 2*iseed-1
  do i = 0, RNDMAX
    ici = 0
    do ii = 1, 32               ! 32-bit version
      ici = ishft(ici, 1)
      ibm = ibm * 16807
      if( ibm .lt. 0 ) ici = ior(ici, one32)
    end do
    rnd_data(i) = ici
  end do
  rnd_idx = 0
  do i = 1, 8*(RNDMAX+1)        ! "heat up" generator
    call rnd_next()
  end do
end subroutine

subroutine rnd_next()
  rnd_idx = iand(rnd_idx + 1, RNDMAX)
  rnd_data(rnd_idx) = ieor(&
  ieor(rnd_data(iand(rnd_idx-471,RNDMAX)),rnd_data(iand(rnd_idx-1586,RNDMAX))),&
  ieor(rnd_data(iand(rnd_idx-6988,RNDMAX)),rnd_data(iand(rnd_idx-9689,RNDMAX))))
end subroutine

integer*4 function rnd_get()
  rnd_get = rnd_data(rnd_idx)
end function

end
```

F.4 Ising model

This program uses MPI (version 1.1). It should be portable with one possible exception: some compilers may need the hex-constants for `MASK_0`, `MASK_1`, etc. in a different format, consult the handbook of your compiler; but most modern compilers handle these without problems.

```
! Two-dimensional ising model with glauber kinetics.
! Uses multi-spin coding and mpi routines for parallelization
! Uses one bit per spin and compression in multi-spin-coded direction.
! Due to "historical reasons", the communication pattern works in the
! opposite direction as one would expect (i. e. pe_succ is the left neighbour).

PROGRAM Ising

implicit none

include 'mpif.h'
integer, parameter   :: LL = 128    ! LL must be a multiple of 4
integer, parameter   :: L = LL*16   ! L/NPROCS must be a natural number
integer*8, parameter :: NPROCS = 4
integer, parameter   :: MAXSTEP = 20
integer*8, parameter :: IDIM = 2
integer*8, parameter :: ISEED = 113
integer*8, parameter :: one = 1
integer*8, parameter :: fifteen = 15
integer*8, parameter :: MASK_3 = '8888888888888888'X
integer*8, parameter :: MASK_2 = '4444444444444444'X
integer*8, parameter :: MASK_1 = '2222222222222222'X
integer*8, parameter :: MASK_0 = '1111111111111111'X
integer, parameter   :: tag = 1

real*8    :: JkbT    ! J/(k_B*T)

integer status(MPI_STATUS_SIZE)
integer err
integer mpi_nprocs

integer*8 spin(0:LL/4-1, 0:L/NPROCS+1)
integer*8 decom(0:LL+1, 0:2)
integer*8 prob(0:2*IDIM)
integer   pe_me, pe_pred, pe_succ
integer   j

integer*8 ibm
integer*8 IMULT

!real*4    etime, dummy, t_start(2), t_end(2) ! Sun
real*8    rtc, t_start, t_end                ! IBM Regatta
!integer*8 irtc, t_start, t_end              ! Cray T3E

JkbT = 0.5d0*(dlog(1.0d0+dsqrt(2.0d0)))
!dummy = etime(t_start) ! Sun
t_start = rtc()        ! IBM Regatta
!t_start = irtc()       ! Crat T3E

if( (L/NPROCS)*NPROCS .ne. L ) then
  print *, 'L/NPROCS must be a natural number!'
  stop 1
end if
if( (LL/4)*4 .ne. LL ) then
  print *, 'LL/4 must be a natural number!'
  stop 1
end if

call mpi_init(err)
call mpi_comm_size(MPI_COMM_WORLD, mpi_nprocs, err)

if( NPROCS .ne. mpi_nprocs ) then
  print *, 'Wrong number of processors!'
  stop 1
end if

IMULT = 13**7              ! Unfortunately necessary, as the IBM compiler
IMULT = IMULT * 13**6      ! has its problems with 64-bit constants.

call mpi_comm_rank(MPI_COMM_WORLD, pe_me, err)
pe_pred = pe_me - 1
if( pe_pred .eq. -1) pe_pred = NPROCS - 1
pe_succ = pe_me + 1
if( pe_succ .eq. NPROCS ) pe_succ = 0
if( pe_me .eq. 0 ) then
  print *, '# J/kbT = ', JkbT, ', L = ', L, ', NPROCS = ', NPROCS,&
           ', ISEED = ', ISEED
  print *, '# PRNG: ibm*', IMULT, ', new version'
end if

! Initialize the PRNG
ibm = 2*ISEED-1
do j = 0, pe_me
  ibm = ibm * 65539
end do

call init_spin()
call init_prob()

do j = 1, MAXSTEP
  call mcs_step(j)
end do

call mpi_finalize(err)

!dummy = etime(t_end) ! Sun
t_end = rtc()        ! IBM Regatta
!t_end = irtc()       ! Cray T3E

!print *, '# Elapsed time: ', t_end(1) - t_start(1)           ! Sun
print *, '# Elapsed time: ', t_end - t_start                 ! IBM Regatta
!print *, '# Elapsed time: ', (t_end - t_start)*1.33333333d-5 ! Cray T3E
```

```
contains

subroutine init_spin()
  integer :: ii, i

  do i = 0, L/NPROCS + 1
    do ii = 0, LL/4-1
      spin(ii,i) = 0
    end do
  end do
end subroutine

subroutine init_prob()
  integer :: i
  real*8  :: ex

  do i = 0, 2*IDIM
    ex = exp( (i-IDIM)*4 * JkbT )
    prob(i) = 2.0d0*2147483648.0d0*2147483648.0d0*(2.0d0*ex/(1.0d0+ex)-1.0d0)
  end do
end subroutine

subroutine mcs_step(timestep)
  integer, intent(in) :: timestep
  integer   ::  ii, i, ibit
  integer*8 :: word, ici, mask
  integer, save :: old = 0, current = 1, new = 2
  integer   :: tmp
  integer*8 :: mag, mag_acc

  mag = 0
  ! Inititalize decom( ,old) and decom( ,current)
  do ii = 0, LL/4-1
    ici = spin(ii,0)
    decom(4*ii+1,old) = ishft(iand(ici,MASK_3),-3)
    decom(4*ii+2,old) = ishft(iand(ici,MASK_2),-2)
    decom(4*ii+3,old) = ishft(iand(ici,MASK_1),-1)
    decom(4*ii+4,old) =       iand(ici,MASK_0)
    ici = spin(ii,1)
    decom(4*ii+1,current) = ishft(iand(ici,MASK_3),-3)
    decom(4*ii+2,current) = ishft(iand(ici,MASK_2),-2)
    decom(4*ii+3,current) = ishft(iand(ici,MASK_1),-1)
    decom(4*ii+4,current) =       iand(ici,MASK_0)
  end do
  decom(0,current) = ishftc(decom(LL,current),-4)
  do i = 1, L/NPROCS
    ! Decompress a line
    do ii = 0, LL/4-1
      ici = spin(ii,i+1)
      decom(4*ii+1,new) = ishft(iand(ici,MASK_3),-3)
      decom(4*ii+2,new) = ishft(iand(ici,MASK_2),-2)
      decom(4*ii+3,new) = ishft(iand(ici,MASK_1),-1)
      decom(4*ii+4,new) =       iand(ici,MASK_0)
    end do
    decom(0,new) = ishftc(decom(LL,new),-4)
    do ii = 1, LL
      ici = decom(ii,current)
      word = ieor(decom(ii-1,current),ici)+ieor(decom(ii+1,current),ici)+&
             ieor(decom(ii,old),ici)+ieor(decom(ii,new),ici)
      mask = one
      do ibit = 0, 15
        ibm = ibm * IMULT
        if( ibm .lt. prob(iand(word,fifteen))) then
          ici = ieor(ici, mask)
        end if
        mag = mag + ishft(iand(ici, mask),-4*ibit)
        word = ishft(word, -4)
        mask = ishft(mask, 4)
      end do
      decom(ii,current) = ici
      ! take care of the boundaries
      if( ii .eq. 1) then
        decom(LL+1,current) = ishftc(decom(1,current),4)
      end if
    end do
    ! Compress a line
    do ii = 0, LL/4-1
      spin(ii,i-1) = ishft(decom(4*ii+1,old),3) + ishft(decom(4*ii+2,old),2) &
                   + ishft(decom(4*ii+3,old),1) +       decom(4*ii+4,old)
    end do
    tmp = old
    old = current
    current = new
    new = tmp
    ! Exchange boundaries with neighbours
    if( i .eq. 1 ) then
      call mpi_sendrecv( spin(0,1), LL/4, MPI_INTEGER8, pe_succ, tag, &
                         spin(0,L/NPROCS+1), LL/4, MPI_INTEGER8, &
                         pe_pred, tag, MPI_COMM_WORLD, status, err )
    end if
  end do
  ! Sum up magnetization on PE 0 and print it out:
  call mpi_reduce( mag, mag_acc, 1, MPI_INTEGER8, MPI_SUM, 0, MPI_COMM_WORLD, err )
  if(pe_me .eq. 0) then
    print *, timestep, -2.0d0*mag_acc/(1.0d0*L*L)+1.0d0
  end if
  do ii = 0, LL/4-1
    spin(ii,L/NPROCS) = ishft(decom(4*ii+1,old),3) + ishft(decom(4*ii+2,old),2) &
                      + ishft(decom(4*ii+3,old),1) +       decom(4*ii+4,old)
  end do
  call mpi_sendrecv( spin(0,L/NPROCS), LL/4, MPI_INTEGER8, pe_pred, &
                     tag, spin(0,0), LL/4, MPI_INTEGER8, pe_succ, &
                     tag, MPI_COMM_WORLD, status, err )
end subroutine

end
```

Appendix G

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie abgesehen von unten angegebenen Teilpublikationen noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Stauffer betreut worden.

Köln, den

Teilveröffentlichungen

D. Tiggemann, *Percolation on growing lattices*, Int. J. Mod. Phys. C **17**, 1141 (2006).

D. Tiggemann, *New results for the dynamical critical behaviour of the two-dimensional Ising model*, Int. J. Mod. Phys. C **15**, 1069 (2004).

D. Tiggemann, *Fluctuations of cluster numbers in percolation*,
Int. J. Mod. Phys. C **13**, 777 (2002).

D. Tiggemann, *Simulation of percolation on massively-parallel computers*,
Int. J. Mod. Phys. C **12**, 871 (2001).

D. Tiggemann, *Simulation of percolation on parallel computers*, Diploma thesis at the Institute for Theoretical Physics at the University of Cologne (2001).

This page intentionally left blank