Karlsruher Institut für Technologie

Parallele Simulation des Sandhaufen-Zellularautomaten für Systeme mit gemeinsamem Speicher

Bachelorarbeit
von

Johannes Lorenz

an der Fakultät für Informatik,
Institut für Kryptographie und Sicherheit (IKS)

Betreuer:	Prof. Dr.-Ing. Roland Vollmar
Betreuender Mitarbeiter:	Dr. rer. nat. Thomas Worsch

Inhaltsverzeichnis

Kapitel 1

Einführung

Diese Bachelorarbeit befasst sich mit Sandhaufen-Zellularautomaten gemäß Per Bak, Chao Tang und Kurt Wiesenfeld [BTW87]. Dabei soll die effiziente Parallelisierung auf Mehrkernrechnern im Vordergrund stehen.

1.1 Motivation

Das Modell des Sandhaufen-Zellularautomaten wurde in den 1980er Jahren von Per Bak, Chao Tang und Kurt Wiesenfeld ins Leben gerufen. Seitdem konnten ihm viele interessante theoretische Eigenschaften entlockt werden. So bilden die rekurrenten Konfigurationen mit der Operation, die komponentenweise addiert und danach stabilisiert, eine abelsche Gruppe [Wor11]. Das Modell hat die Eigenschaft der selbstorganisierten Kritikalität. Sandhaufen-Zellularautomaten sind in gewisser Weise reversibel. Außerdem findet man NP-vollständige Probleme in ihnen [Ada08].

Nichtsdestotrotz hat dieses Modell interessante Eigenschaften, die auch für die Praxis relevant sein können. Tatsächlich stimmt die Häufigkeitsverteilung der Stärke von Erdbeben mit der der Lawinengrößen im Sandhaufenmodell überein[Wor11]. Sandhaufen-Zellularautomaten folgen einem zweidimensionalen Wachstum nach dem Eden Growth Model, einem Modell, in dem Wachstum durch zufällige Ansammlung bedingt ist. Dieses Modell beschreibt auch das Wachstum von Bakterienkolonien. Außerdem folgt die Lawinengröße einer $1/f$-Verteilung.

Bei manchen dieser Anwendungen, wie zum Beispiel Erdbebensimulation, ist eine schnelle Berechnung besonders wichtig. In einem Zeitalter, in dem CPU-Hersteller auf Mehrkernprozessoren setzen, bietet sich oft eine parallele Berechnung an, um einen Algorithmus schneller auszuführen. Optimal kann man mit n Kernen die Laufzeit mit Faktor n senken, also auf $\frac{1}{n}$. Ist die Laufzeit noch besser, so spricht man von superlinearem Speedup. Dies kann zum Beispiel daraus resultieren, dass mehrere Kerne nicht nur mehr Rechenleistung, sondern insgesamt auch mehr Cache zur Verfügung haben.

Das Ziel dieser Bachelorarbeit ist, die bisher bekannten schnellsten Algorithmen für das Sandhaufen-Modell noch schneller zu gestalten und neue, noch schnellere Algorithmen zu entwickeln.

1.2 Genaue Problemstellung

Die Simulation von Sandhaufen-Zellularautomaten kann durch unterschiedliche Problemstellungen motiviert sein. Dabei können Anforderungen an Eingabe und Ausgabe gestellt sein. Die Eingabe ist meist, wie auch in unserem Fall, eine vorberechnete Reihenfolge von Sandkörnern. Diese seien als zufällig zu betrachten. Die Anforderungen an die Ausgabe können sehr unterschiedlich sein. Typische Forderungen sind:

1. Größe einer jeden Lawine
2. Nummern aller Zellen, die im Zuge einer jeden Lawine feuern
3. Konfiguration des Gitters nach der Simulation

Verzichtet man auf Anforderungen (1) und (2), so kann man die Laufzeit sehr stark senken. Dies ist in der Diplomarbeit von Felix Bodarenko [Bod09] nachzulesen. Wir fordern für unsere Zwecke mindestens (1) und (3). Für die Laufzeitanalysen am Ende dieser Bachelorarbeit wird auch auf (1) verzichtet, damit der Algorithmus frei von Störeinflüssen von Logging-Methoden laufen kann. Formal müssen aber alle Algorithmen Eigenschaft (1) erfüllen können, damit der Algorithmus der Anforderung gerecht wird, jede Lawine einzeln zu berechnen.

Eigenschaft (2) verlangsamt viele Algorithmen deutlich und ist sinnvoll, falls man an genaueren Details über die Lawinen interessiert ist.

1.3 Gliederung

Diese Arbeit beginnt mit einem theoretischen Kapitel, in welchem ein paar Grundlagen zu Sandhaufen-Zellularautomaten erörtert werden. Von Interesse sind die Eigenschaften der Kommutativität und die der exakten Größe einer Lawine unter bestimmten oder beliebigen Voraussetzungen an eine Startkonfiguration. Diese Eigenschaften sind nicht nur von theoretischer Bedeutung; viele spielen für die hier beschriebenen Algorithmen eine fundamentale Rolle. Den Abschluss des Kapitels bildet ein Algorithmus, der mit den bisherigen Resultaten eine „perfekte Parallelisierbarkeit“ anstrebt, aber aus Zeitgründen nicht mehr implementiert werden konnte.

Im folgenden Kapitel wird von der Portierung der effizientesten Algorithmen aus der von Sebastian Frehmel geschriebenen Diplomarbeit [Fre10] „Parallelisierung des Sandhaufen-Zellularautomaten für Systeme mit gemeinsamem Speicher in Java“ Bericht erstattet. Die Algorithmen an sich wurden dabei nicht verändert. Besonderes Augenmerk wird auf die Technik gerichtet. Dies beinhaltet Fragen wie „Welche Arten von Mutexen sind besonders geeignet?“, „Was sind Auswahlkriterien für eine Thread-Bibliothek?“ oder „Welche Fehler wurden behoben?“.

Danach folgt der Fokus dieser Bachelorarbeit. Es werden neue Algorithmen aufgezeigt, wobei Parallelisierbarkeit eine entscheidende Rolle spielt. Zunächst wird der Repository-Algorithmus vorgestellt, der anstatt wie bisher üblich, nicht nur eine Lawine, sondern mehrere Lawinen gleichzeitig entwickelt. Dennoch kann er der Anforderung an die Ausgabe, die Lawinen einzeln und chronologisch sortiert auszugeben, gerecht werden. Dabei stellt sich natürlich die Frage, wie viele Lawinen man gleichzeitig effizient simulieren kann. Den zweiten Teil dieses Kapitels bildet eine Klasse von Algorithmen, die alle jeweils ein Quadrat von 16 Zellen in einem nur 64 Bit großen Integer speichert. Insbesondere können diese Algorithmen leicht von einer Hash-Tabelle Gebrauch machen, die hilft, einen Wurf eines Korns auf einige solche Quadrate in $O(1)$ zu berechnen. Eine entscheidende Frage wird hier sein, welche Kombinationen von Zellen vorberechnet werden sollen, da die Cache-Speicher einer CPU meist nur einige MegaByte umfassen.

Den Abschluss bildet ein Kapitel, welches die Algorithmen analysiert und untereinander vergleicht. Dabei wird diskutiert, was die Gründe für besonders schlechte oder besonders gute Laufzeiten sind. Anschließend werden alle Algorithmen untereinander verglichen. Auch werden bei den portierten Algorithmen die Unterschiede bezüglich der Programmiersprache gezeigt.

Kapitel 2

Theorie

2.1 Festlegungen

Die folgenden Festlegungen gelten in allen folgenden Kapiteln.

Es sei $0 \notin \mathbb{N}$ und $\mathbb{N}_0 := \mathbb{N} \cup 0$.

Im ganzen Kapitel sei, falls nicht anders angegeben, $R := \{0, ..., \mathfrak{b}\} \times \{0, ..., \mathfrak{h}\} \subseteq \mathbb{Z}^2$ mit $\mathfrak{b}, \mathfrak{h} \in \mathbb{N}_0$ der *Koordinatenraum*. Sei $N := \{(0,0), (0,-1), (0,1), (-1,0), (1,0)\}$ die Nachbarschaft zu jeder Zelle; sie wird auch *1-dimensionale Von-Neumann-Nachbarschaft* genannt.

Es stehe $Q \subseteq \mathbb{Z}$ für die Menge der Zustände. Falls nicht explizit angegeben, so sei $Q := \{0, 1, 2, 3, 4, 5, 6, 7\}$.

Als *Größe einer Lawine* bezeichnen wir die Anzahl des Ereignisses, dass sich der Zustand einer Zelle um 4 verringert. Als Startzelle bezeichnen wir diejenige Zelle, die vor der Stabilisation in einem Zustand $a > 4, a \in Q$ ist, falls diese eindeutig ist.

Falls $M \subseteq R$, so bezeichnet $c|_M$ die *Einschränkung* auf M, so dass

$$\forall i \in M : c|_M(i) := \begin{cases} c(i), & \text{falls } i \in M \\ 0, & \text{sonst} \end{cases}.$$

Es handelt sich hier also streng genommen nicht um eine Einschränkung des Definitionsbereichs, sondern um eine Modifikation der Funktion.

Wir werden Q^R als $\mathbb{Z}$-Modul auffassen. Seien $c_1, c_2 \in Q^R, z \in \mathbb{Z}$ und $i \in R$, dann legen wir fest:

$$(c_1 + c_2)(i) := c_1(i) + c_2(i) \text{ und } (z \cdot c_1)(i) := z \cdot (c_1(i)).$$

Wir definieren $\{e_i \in Q^R, i \in R\}$ so, dass

$$\forall j \in R : e_i(j) : \begin{cases} 1, & \text{falls } j = i \\ 0, & \text{sonst} \end{cases}.$$

Eine Basis für Q^R ist $\{e_i, i \in R\}$, denn diese e_i sind linear unabhängig und erzeugend.

Die Nullkonfiguration oder kurz $0 \in Q^R$ sei hier die Konfiguration, in der gilt: $\forall i \in R : 0(i) = 0$. Analog seien die Konfigurationen 1, 2 und 3 definiert. Weiter definieren wir für ein Quadrat $K \subseteq R$ die Konfiguration

$$\delta_K := \begin{cases} 1, & \text{falls } i \text{ auf einer Diagonalen von } K \\ 0, & \text{sonst} \end{cases}.$$

Definition 2.1.1 (Die Stabilisierungsfunktion)**.** Es sei

$$s : \eta(\mathbb{Z}^R) \to \{0, 1, 2, 3\}^R$$

der übliche Algorithmus des Zellularautomaten, der das Gitter solange modifiziert, bis eine stabile Konfiguration entsteht. Dabei bezeichne $\eta(\mathbb{Z}^R) \subseteq \mathbb{Z}^R$ diejenigen Konfigurationen, auf denen s nach endlich vielen Schritten anhält. Falls das Gitter Zellen in negativen Zuständen enthält, so berechnet s nämlich Rückwärtsschritte, so dass das Ergebnis auch negativ sein kann. Bei bestimmten Eingaben kann dann s für bestimmte R nicht terminieren, zum Beispiel für $-e_i, i \in R$ und $R = \mathbb{Z}^2$. Falls s aber terminiert, so ist das Ergebnis wohldefiniert, wie man sich leicht überlegt.

Wir nennen s *Stabilisierungsfunktion*. Gelegentlich nennen wir für $c \in Q^R$ die Konfiguration $s(c)$ auch *Endkonfiguration von* c.

Definition 2.1.2 (Die Lawinenfunktion)**.** Die Funktion

$$l : \eta(\mathbb{Z}^R) \times R \to \mathbb{Z}, (c, i) \mapsto l(c, i)$$

gibt an, wie oft die Zelle i genau feuert, wenn c stabilisiert wird. η sei definiert wie für s.

Analog definieren wir

$$l_c : \eta(Q^R) \to (R \to \mathbb{Z}), c \mapsto l_c \text{ ,wobei } l_c(i) := l(c, i).$$

Wenn keine Verwechslungsgefahr besteht, so schreiben wir in beiden Fällen l. In beiden Fällen nennen wir l *Lawinenfunktion*.

Die Kenntnis des folgenden Satzes ist notwendig.

Satz 2.1.1. *Die Stabilisierung des Sandhaufen-Zellularautomaten ist kommutativ, d.h. für $c \subseteq Q^R$, $c_1, c_2 \subseteq \mathbb{Z}^R$ gilt:*

$$s(s(c + c_1) + c_2) = s(s(c + c_2) + c_1) \text{ und}$$
$$l(s(c + c_1)) + l(s(c + c_1) + c_2) = l(s(c + c_2)) + l(s(c + c_2) + c_1).$$

Beweis. Ein Beweis findet sich in [Spy02]. □

2.2 Kommutativität

Es ist bekannt, dass Kommutativität beim Werfen von Sandkörnern gilt. Was passiert aber, wenn man temporär Sandkörner subtrahiert oder addiert?

Satz 2.2.1. *Seien $c \in \{0,1,2,3\}^R$, $c_1, c_2 \in \mathbb{Z}^R$ globale Konfigurationen. Falls $\forall i \in R : (c - c_1)(i) \geq 0$, so gilt:*

$$l(c - c_1) + l(c - c_1 + c_2) + l(s(c - c_1 + c_2) + c_1) = l(c + c_2).$$

Analog gilt: falls $\forall i \in R : (s(c + c_1 + c_2) - c_1)(i) \geq 0$, so ist

$$l(c + c_1) + l(s(c + c_1) + c_2) + l(s(c + c_1 + c_2) - c_1) = l(c + c_2).$$

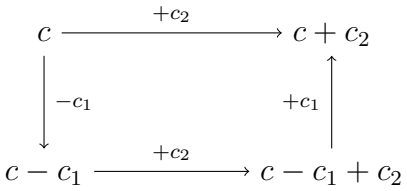

Abbildung 2.1: Diagramm für den 1. Fall

Beweis. Es gilt:

$$\begin{aligned} l(c - c_1) + l(c - c_1 + c_2) + l(s(c - c_1 + c_2) + c_1) &= \\ l(c - c_1 + c_2) + l(s(c - c_1 + c_2) + c_1) &= \\ l(c - c_1 + c_1) + l(s(c - c_1 + c_1) + c_2) &= \\ l(c) + l(s(c) + c_2) &= \\ l(c + c_2). & \end{aligned}$$

Die zweite Gleichheit gilt dabei wegen Satz 2.1.1.

Der zweite Teil des Satzes gilt wegen:

$$\begin{aligned} l(c + c_1) + l(s(c + c_1) + c_2) + l(s(c + c_1 + c_2) - c_1) &= \\ l(c + c_1) + l(s(c + c_1) + c_2) &= \\ l(c + c_2) + l(s(c + c_2) + c_1) &= \\ l(c + c_2), & \end{aligned}$$

wobei $l(s(c + c_2) + c_1) = 0$ aus $l(s(c - c_1 + c_2) + c_1) = 0$ folgt. Dies sei dem Leser als kleine Übung überlassen. □

Der erste Teil dieses Satzes ermöglicht es also, zum Stabilisieren einer Konfiguration zunächst eine andere Konfiguration zu subtrahieren, dann zu stabilisieren, und schließlich die fehlenden Sandkörner wieder aufzuaddieren. Das wird für den Cache-Algorithmus von Nutzen sein. Der zweite Teil ist eher von theoretischem Nutzen, da vor Berechnung von $s(c + c_2)$ schon bekannt sein muss, dass $\forall i \in R : (s(c + c_1 + c_2) - c_1)(i) \geq 0$ gilt.

2.3 Kreisvermutung

In diesem Kapitel sei $R := \mathbb{Z}^2$, $m \in R$ fest.

Sei M_n für $n \in \mathbb{N}_0$ die Menge aller Zellen in $s(n \cdot e_m)$ bis auf die äußere zusammenhängende Menge von Zellen im Zustand 0. Es fällt auf, dass die M_n Muster bilden, die Ähnlichkeiten zu Kreisen aufweisen.

Um die Vermutung zu überprüfen, wurde ein Programm geschrieben, das die Endkonfigurationen darauf prüft, ob es sich um Kreise handelt. Tatsächlich handelt es sich meist nicht um exakte Kreise, die Abweichung vom Radius ist aber in den Messungen konstant bei ca. 2-3 Zellen.

Vermutung 1. *Sei K_r, $r \in \mathbb{N}_0$ definiert als die Menge $\{i \in R : \|i - m\| \leq r\}$, wobei hier die euklidische Norm gemeint sei. Es existiert ein c, so das gilt: Zu allen $n \in \mathbb{N}$ findet sich ein $r \in \mathbb{N}$, so dass gilt:*

$$K_r \subseteq M_n \subseteq K_{r+c}.$$

Nun ist man hauptsächlich an der Ausbreitung, also am Radius, der $s(n \cdot e_i), n \in \mathbb{N}$ interessiert. Falls die Vermutung gilt, so lässt sich dieser bestimmen. Sei dazu $c \approx 2,43$ der durchschnittliche Zustandswert in der kritischen Phase, der hier für den Kreis geschätzt wird. Dann gilt:

$$t \approx \pi c r^2 \Leftrightarrow r \approx \sqrt{\frac{t}{\pi c}} \approx 0,36\sqrt{t}.$$

Es konnten keine Beweise für die Vermutung hergestellt werden.

2.4 Vergrößerung eines Rechtecks

Die Schlüsselfrage dieses Abschnitts ist, wie weit sich ein Rechteck bei einem Wurf eines Korns ausdehnen kann. Dabei sei ein Rechteck eine Menge $K \subseteq R$ von Koordinaten, so dass gilt:

$$\begin{aligned} \exists i \in R, a, b \in \mathbb{N} : K \quad := \quad & \{i + (0,0), i + (0,1), ..., i + (0,a)\} \\ \cup \quad & \{i + (1,0), i + (1,1), ..., i + (1,a)\} \\ \cup \quad & ... \\ \cup \quad & \{i + (b,0), i + (b,1), ..., i + (b,a)\}. \end{aligned}$$

Für ein Quadrat ist dabei offensichtlich $a = b$. In den meisten Fällen nimmt man $c|_{R \setminus K} = 0$ an.

Wir definieren eine Abstandsfunktion $d : R \times \mathcal{P}(R) \to \mathbb{N}, (i, S) \mapsto d(i, S)$, die zu jeder Zelle i die Länge des kürzesten Pfades $\{i_0 := i, i_1, ..., i_n\}, n \in \mathbb{N}_0$ angibt, für den $\forall k \in 1, ..., n : (i_k - i_{k-1}) \in N$ und $i_n \in S$ gilt.

Sei K das Rechteck. Man kann K wie folgt in disjunkte Mengen aufteilen. Die Ringe $V_j \subseteq K, j \in \mathbb{N}_0$ seien definiert als $V_j := \{i \in K; d(i, R \setminus K) = j\}$. V_1 ist insbesondere der äußerste Ring und per Definition ist $V_0 = R \setminus K$.

Nun zu den Aussagen. Es reicht ohne Einschränkung, den Fall $3|_K$ zu betrachten:

Lemma 1. *Seien $c \in \{0,1,2,3\}^R|_K$, $c_2 \in {\mathbb{N}_0}^R$, dann gilt:*

$$l(c + c_2) \leq l(3|_K + c_2).$$

Beweis. Es gilt: $\exists \zeta \in \{0,1,2,3\}^R|_K : l(c + c_2) = l(3|_K - \zeta + c_2)$. Aus der 1. Gleichung über Kommutativität aus 2.1.1 folgt

$$l(c + c_2) = l(3|_K - \zeta + c_2) = l(3|_K + c_2) - l(s(3|_K - \zeta + c_2) + \zeta) \leq l(3|_K + c_2).$$

□

Proposition 1. *Sei K ein Rechteck mit Konfiguration c, so dass $c|_{R \setminus K} = 0$. Man werfe ein Korn auf eine beliebige Zelle δ. Dann gilt:*

$$\forall z \in R : l(c + e_\delta, z) \leq d(z, R \setminus K).$$

Insbesondere feuern Zellen mit Nachbarn unmittelbar am Rand maximal einmal, und das Rechteck muss an jeder Seite maximal um eine Zelle „vergrößert“ werden.

Bevor wir mit den Beweisen beginnen, soll der folgende Satz ein Beispiel liefern, welches später auch hilfreich sein wird.

Satz 2.4.1 (Invarianz von „3“ mit Wurf vom Rand)**.** *Sei $K \subseteq R$ ein Quadrat mit $c = 3|_K$. Dann gilt:*

$$s(c + 1|_{V_1})|_K = c \text{ und } l(c + 1|_{V_1}) = 1|_K.$$

Beweis. Wir nehmen an, dass die Seitenlänge gerade sei. Für ungerade Seitenlänge ist die mittlere Zelle ein einfacher Spezialfall, der aber nichts am Resultat ändert.

Wir stabilisieren schrittweise die Ringe V_i und führen Induktion durch, mit der Induktionsannahme, dass, nachdem alle Zellen aus V_i genau einmal feuern, gilt:

1. $c|_{\bigcup_{j<i} V_j} = 3$,
2. $c|_{V_i} = 2 + \delta_K$.

Wenn also alle Zellen von V_{i+1} nach Induktionsannahme genau 1 mal feuern werden, so ist $c|_{V_i} = 3$ und die Behauptung des Satzes gilt.

Was macht s auf V_i? Die instabile Konfiguration auf V_i ist $3|_{V_i} + (1|_{V_i} + \delta_K|_{V_i}) = 4|_{V_i} + \delta_K|_{V_i}$. Klar ist $l(5|_{V_i}) \geq 1$. Würde eine Zelle die erste sein, die 2 mal feuert, so

müsste sich ihr Wert um 3 erhöhen. Da sie aber in V_i nur 2 Nachbarn hat, und diese bisher nur einmal gefeuert haben, ist $l(5|_{V_i}) = 1$. Wir erhalten

$$1 \leq l(4|_{V_i}) \leq l(4|_{V_i} + \delta_K|_{V_i}) \leq l(5|_{V_i}) = 1,$$

also gilt überall Gleichheit und insbesondere $l(4|_{V_i} + \delta_K|_{V_i}) = 1$. Es folgt

$$s(3|_{V_i} + 1|_{V_i} + \delta_K|_{V_i})|_{V_i} = (4 + \delta_K - 4 + 2)|_{V_i} = (2 + \delta_K)|_{V_i}.$$

□

Korollar 2.4.1. *Sei $K \subseteq R$ ein Quadrat mit $c = (3 - 2\delta_K)|_K$. Dann gilt:*

$$s(c + 1|_{V_1})|_K = c \text{ und } l(c + 1|_{V_1}) = 1|_K.$$

Beweis. Das ist ein direktes Resultat aus Satz 2.2.1 (Fall 2). □

Wir betrachten nun zunächst den Fall eines Quadrats mit ungerader Seitenlänge.

Satz 2.4.2. *Sei $K \subseteq R$ ein Quadrat mit ungerader Seitenlänge und $c = 3|_K$. Man werfe ein Korn auf die zentrale Zelle $m \in R$. Dann gilt:*

$$\forall z \in R : l(3 + e_m, z) = d(z, R \setminus K).$$

Beweis. Wir benennen verschiedene Arten von Konfigurationen auf Ringen. Für $c \in Q^R$ heißt $c|_{V_j}$

- A-Ring, falls $c|_{V_j} = 3|_{V_j}$
- B-Ring, falls $c|_{V_j} = (2 - \delta_K)|_{V_j}$
- B^I-Ring, falls $c|_{V_j} = (3 - 2\delta_K)|_{V_j}$
- B^{IO}-Ring, falls $c|_{V_j} = (4 - \delta_K)|_{V_j}$
- C-Ring, falls $c|_{V_j} = (1 - \delta_K)|_{V_j}$

Lemma 2. *Sei j fest, $a \in Q^{V_j}$ ein A-Ring, $b \in Q^{V_j}$ ein B-Ring usw. Seien außerdem*

$$i, o : \mathbb{Z}^R \to \mathbb{Z}^R$$

$$i : c \mapsto c + (1 - \delta_K)|_{V_j}, o : c \mapsto c + (1 + \delta_K)|_{V_j},$$

also ist i die Funktion, die eine Ring-Konfiguration so erhöht, als hätte jede Zelle des nächstinneren Rings V_{j+1} genau einmal gefeuert und o analog für V_{j-1}.

Für die Ring-Konfigurationen gilt:

1. $i(a) = b^{io}$

2. $i(b) = b^i$
3. $(o \circ i)(b) = b^{io}$
4. $s(b^{io}) = b$
5. *Beim Stabilisieren von* b^{io} *feuert jede Zelle exakt ein mal.*
6. $i(0) = c$
7. b^{io} *ist instabil;* a, b, b^i, c, 0 *sind stabil.*

Beweis des Lemmas. (1) Gilt offensichtlich, da $3|_{V_j} + (1 - \delta_K)|_{V_j} = (4 - \delta_K)|_{V_j}$. (2) und (3) sind ebenso offensichtlich. (4) gilt, denn aus 2.4.1 ist bekannt, dass $s(b^{io}) = s(o(b^i)) = s(o(3|_{V_j} - 2\delta_K|_{V_j})) = (3|_{V_j} - 2\delta_K|_{V_j}) - (1|_{V_j} - \delta_K|_{V_j}) = (2|_{V_j} - \delta_K|_{V_j})) = b$. (5) folgt ebenso aus 2.4.1. (6) und (7) sind trivial. □

Abbildung 2.2 skizziert nun eine Entwicklung der Ring-Konfigurationen, wobei links die zentrale Zelle des Quadrats steht. Bildlich breitet sich eine Welle von innen nach

0	4	A	…				A	A	0
1	0	B^{IO}	A	…			A	A	0
2	4	B	B^{IO}	A	…		A	A	0
3	0	B^{IO}	B	B^{IO}	A	…	A	A	0
…									
r-1	0	B^{IO}	B	B^{IO}	B	…	B	B^{IO}	0
r	4	B	B^{IO}	B	B^{IO}	…	B^{IO}	B	C
r+1	0	B^{IO}	B	B^{IO}	B	…	B	B^{I}	C
r+2	4	B	B^{IO}	B	B^{IO}	…	B^{I}	B^{I}	C
…									
2r-4	4	B	B^{IO}	B	B^{I}	…	B^{I}	B^{I}	C
2r-3	0	B^{IO}	B	B^{I}	B^{I}	…	B^{I}	B^{I}	C
2r-2	4	B	B^{I}	B^{I}	B^{I}	…	B^{I}	B^{I}	C
2r-1	0	B^{I}	B^{I}	B^{I}	B^{I}	…	B^{I}	B^{I}	C

Abbildung 2.2: Entwicklung der Ring-Konfigurationen. Grau hinterlegte Zellen sind instabil.

außen aus. Wenn die Welle zum Zeitpunkt $r - 1$ außen angekommen ist, zieht sie sich mit gleicher „Geschwindigkeit“ zurück.

Anmerkung. Das Prozedere funktioniert analog, wenn die innerste Zelle beim ersten Vorkommen des C-Rings 0 ist. Außerdem sei angemerkt, dass die Stabilisierung des Rechtecks nicht in der Reihenfolge passieren muss. Wegen der Kommutativität ist aber die Reihenfolge irrelevant für die Endkonfiguration.

Wenn man die Entwicklung wie oben skizziert in Zeitpunkte $\{t_k, k \in \mathbb{N}_0\}$ aufteilt, so ergibt sich die Formel

$$c_{t_k}|_{V_j} = \begin{cases} a_j, & \text{falls } k < j < r \\ {b^i}_j, & \text{falls } 2r-k < j < r \\ b_j, & \text{falls } k \leq j \leq 2r-1-k \text{ und } j-k \in 2\mathbb{Z} \\ {b^{io}}_j, & \text{falls } k \leq j \leq 2r-1-k \text{ und } j-k \notin 2\mathbb{Z} \\ c_j, & \text{falls } j = r \text{ und } k \geq r \\ 0, & \text{sonst} \end{cases},$$

wobei a_j die eindeutigen A-Ring-Konfigurationen des Rings V_j sind usw. Für die mittlere Zelle definieren wir außerdem $b_0 := 0$ und ${b^{io}}_0 := 4$, damit die Formel stimmt. Dies verträgt sich mit den Aussagen des Lemmas.

Wegen Aussage (7) des Lemmas gilt nun: Eine Zelle feuert genau dann, wenn gilt: $k \leq j \leq 2r-1-k$ und $j-k \notin 2\mathbb{Z}$. Daraus folgt sofort die Behauptung des Satzes, denn für $z \in V_j \in K$ für ein j gilt:

$$\begin{aligned} l(3+m,z) &= \sum_{k=0}^{2r-1} \begin{cases} 1, & \text{falls } k \leq j \leq 2r-1-k \text{ und } j-k \notin 2\mathbb{Z} \\ 0, & \text{sonst} \end{cases} \\ &= \sum_{k=j}^{2r-1-j} \begin{cases} 1, & \text{falls } j-k \notin 2\mathbb{Z} \\ 0, & \text{sonst} \end{cases} \\ &= \frac{1}{2}((2r-1-j)-(j-1)) \cdot 1 + \frac{1}{2}((2r-1-j)-(j-1)) \cdot 0 \\ &= r-j \\ &= d(V_j, R \setminus K) \\ &= d(z, R \setminus K). \end{aligned}$$

□

Der letzte Satz gilt auch für Quadrate gerader Seitenlänge:

Satz 2.4.3. *Sei $K \subseteq R$ ein Quadrat mit Seitenlänge a und $c = 3|_K$. Man werfe ein Korn auf eine Zelle $m \in R, d(m, R \setminus K) = \lceil \frac{a}{2} \rceil$, also auf die „Mitte“. Dann gilt:*

$$\forall z \in R : l(3+e_m, z) = d(z, R \setminus K).$$

Beweis. Das einzige, was dabei gezeigt werden muss, ist, dass die mittleren 4 Zellen sich nach außen genau so verhalten wie eine einzelne Zelle. Ein Wurf auf 4 Zellen im Zustand 3 führt zu einer Zelle im Zustand 2, die 2 Nachbarzellen im Zustand 1 hat, und die übrige Zelle ist im Zustand 0. Nun stellt man noch fest, dass $s \circ o$ die Identität ist, und für den nächstäußeren Ring i „ausführt“. □

Für den Fall, dass das Korn nicht auf eine Zelle in der „Mitte“ geworfen wird, so gilt $d(z, R \setminus K)$ immer noch als Oberschranke für $l(3 + e_m, z)$. Der folgende Satz zeigt das zunächst für den Fall, dass das Sandkorn auf eine Diagonale des Quadrats geworfen wird, der darauf folgende für den allgemeinen Fall.

Satz 2.4.4. *Sei K ein Quadrat mit Konfiguration c, so dass $c = 3|_K$. Man werfe ein Korn auf eine Zelle $x \in R$, die auf einer Diagonalen liegt. Dann gilt:*

$$\forall z \in R : l(c + e_x, z) \leq d(z, R \setminus K).$$

Beweis. Sei $i \in R$. Dann ist $s(3+m+i) = s(3+m)$, denn in der Konfiguration $s(3+m)$ sind nach letzterem Satz alle Ringkonfigurationen vom Typ 0, B oder C, also sind dort alle Zellen im Zustand 0 oder 1, und das Werfen auf solche Zellen hat keinen Effekt. Es gilt:

$$l(3 + m) = l(3 + m) + l(s(3 + m) + i) = l(3 + i) + l(s(3 + i) + m) \geq l(3 + i),$$

wobei die zweite Gleichheit aus der allgemein bekannten Kommutativität folgt. □

Anmerkung. Falls das Sandkorn nicht mehr auf der Mitte, sondern nur noch auf einer Diagonale geworfen wurde, so gilt die echte Gleichheit im letzten Satz schon allgemein nicht mehr. Das sieht man leicht, wenn man sich ein 3x3-Eck aufmalt.

Satz 2.4.5. *Sei K ein Quadrat mit Konfiguration c, so dass $c = 3|_K$. Man werfe ein Korn auf eine beliebige Zelle $z \in R$. Dann gilt:*

$$\forall j \in R : l(c + e_z, j) \leq d(j, R \setminus K).$$

Beweis. Die Zelle, auf die das Sandkorn geworfen wurde, impliziert zwei Diagonalen d_1, d_2, auf deren Schnittpunkt sie liegt. Man konstruiere nun für jede der 4 Richtungen $s \in \{0, 1, 2, 3\}$ zwei Quadrate $Q_{s_1}, Q_{s_2} \subseteq R$ mit folgenden Eigenschaften:

1. $Q_{s_1} \cap Q_{s_2} \neq \emptyset$
2. $K \subseteq (Q_{s_1} \cup Q_{s_2})$
3. K und $(Q_{s_1} \cup Q_{s_2})$ haben in Richtung s genau den selben Rand

Wenn man das alte Quadrat K entlang seiner Diagonalen in 4 Teile zerteilt, so erhält man zu jeder Richtung s ein Teilstück $T_s \subseteq K$, nämlich das, welches den Strahl vom Mittelpunkt von K in Richtung s enthält. Dabei seien die Diagonalen in angrenzenden Teilstücken enthalten. Können wir nun Q_{s_1}, Q_{s_2} mit geforderten Eigenschaften konstruieren, so würde es folgende Ähnlichkeit zu K haben:

$$\forall i \in T_s : l(c, i) \leq max(l(c_1, i), l(c_2, i)),$$

falls $c_1 = 3|_{Q_{s_1}}$ und $c_2 = 3|_{Q_{s_2}}$. Alle Q_{s_1}, Q_{s_2} für alle s vereinigt würden mit letztem Satz daher die Behauptung dieses Satzes ergeben.

Es sei nun die fehlende Konstruktion der Q_{s_1}, Q_{s_2} gegeben. Man betrachte die Gerade a_s der äußersten Zellen in Richtung s. Sie schneidet d_1 in dem ersten Eckpunkt von Q_{s_1}. Wir sprechen von „dem Eckpunkt“. Die anderen Eckpunkte wähle man so, dass $K \cap Q_{s_1}$ maximal wird und ein weiterer Eckpunkt auf d_1 liegt. Analog konstruiere man Q_{s_2} mit d_2. Man betrachte Abbildung 2.3.

Eigenschaft (3) folgt sofort aus der Konstruktion. Für Eigenschaft 1 und 2 betrachtet man den Eckpunkt von Q_{s_1} (alles gilt ohne Einschränkung auch für Q_{s_2}). Ist er außerhalb K, so enthält Q_{s_1} nach Konstruktion schon K, d.h. $K \subseteq Q_{s_1}$. Das zeigt schon (2), und da $K \cap Q_{s_2} \neq \emptyset$ folgt auch (1). Im anderen Fall seien nun beide Eckpunkte für Q_{s_1} und Q_{s_2} in K. Dann enthält auch Q_{s_1} den Eckpunkt von Q_{s_2}, denn Q_{s_1} hat 2 Eckpunkte auf d_1, die auf den Rand projiziert offenbar ein Intervall liefern, in dem der Eckpunkt von Q_{s_2} liegt. Das zeigt (1) und (2).

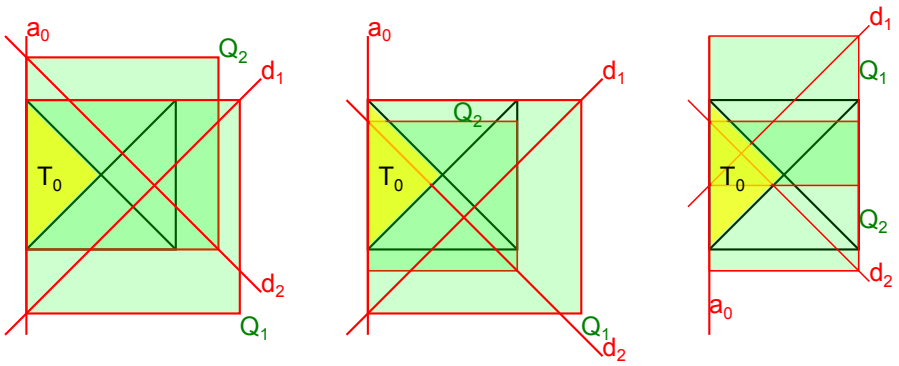

Abbildung 2.3: Fall 1 (links und Mitte) und Fall 2 (rechts).

□

Der letzte Satz über Quadrate gilt allgemein für Rechtecke, wie folgender Satz zeigt.

Satz 2.4.6. *Sei K ein Rechteck mit Konfiguration c, so dass $c = 3|_K$. Man werfe ein Korn auf eine beliebige Zelle z. Dann gilt:*

$$\forall i \in R : l(c + e_z, i) \leq d(i, R \setminus K).$$

Beweis. Wir können offensichtlich 4 Quadrate $Q_1, Q_2, Q_3, Q_4 \in R$ konstruieren, so dass für alle $s \in 0, 1, 2, 3$ gilt:

1. $K \subseteq Q_s$
2. K und (Q_s) haben in Richtung s genau den selben Rand

Die vier Quadrate geben ähnlich wie im Beweis vom letzten Satz zusammen genau die angegebenen Maximalitätsbeschränkungen. □

Korollar 2.4.2. *Für ein Quadrat $K \subseteq R$ mit Seitenlänge a gilt für die maximale Anzahl feuernder Zellen beim Wurf eines Korns $f_{max}(a)$ die Gleichung*

$$f_{max}(a) = \frac{1}{6}a(a+1)(a+2).$$

Diese Zahl kann sogar erreicht werden, wenn man das Sandkorn auf eine zentrale Zelle wirft. Insbesondere gilt für ein Quadrat mit Seitenlänge 1000 *(was die vorgegebene Einstellung für alle Algorithmen ist):*

$$f_{max}(1000) = 167.167.000.$$

Beweis. Es sei $r := \lceil \frac{a}{2} \rceil$ der *Radius* von Q. Sei $z \in R$ die zentrale Zelle.

Da $\forall i \in R : l(c + e_z, i) \leq d(i, R \setminus K)$ nimmt die Anzahl, wie oft eine Zelle feuert, von außen nach innen pro Zelle um 1 zu. Also ist die gesuchte Summe eine Summe von übereinander liegenden Quadraten, die je um 2 kürzere Seitenlänge haben.

Fall 2.4.6.1. Das Quadrat hat gerade Seitenlänge. Dann ist

$$\sum_{i=1}^{r}(2i)^2 = 4\sum_{i=1}^{r} i^2 = 4\frac{(r+1)(2r+1)r}{6} = \frac{1}{3}(2r+2)(2r+1)r.$$

Fall 2.4.6.2. Im Fall ungerader Seitenlänge gilt:

$$\begin{aligned}\sum_{i=1}^{r}(2i-1)^2 &= \sum_{i=1}^{2r} i^2 - \sum_{i=1}^{r}(2i)^2 \\ &= \frac{1}{6}(2r+1)(4r+1)2r - \frac{1}{3}(2r+2)(2r+1)r \\ &= \frac{r(2r+1)}{3}(4r+1-(2r+2)) \\ &= \frac{1}{3}r(2r+1)(2r-1).\end{aligned}$$

Insgesamt gilt also: $\frac{1}{3}r(2r+1)(2r+y)$, wobei $y = 2$ falls $a \in 2\mathbb{Z}$ und $y = -1$ sonst.
Im ungeraden Fall gilt nun $r = \frac{a+1}{2}$ bzw. $a = 2r - 1$. Damit ergibt sich

$$f_{max}(a) = \frac{1}{3}(\frac{a+1}{2})(a+2)a.$$

Analog gilt im geraden Fall $r = \frac{a}{2}$ bzw. $a = 2r$, also

$$f_{max}(a) = \frac{1}{3}(\frac{a}{2})(a+1)(a+2).$$

Also stimmt die Gleichung für beide Fälle. □

Korollar 2.4.3. *Der Stabilisierungsalgorithmus ist in $O(n^3)$ realisierbar, wobei n eine Seitenlänge des Rechtecks sei. Der Sandhaufen-Algorithmus, also das nacheinander ausgeführte Stabilisieren von zufällig geworfenen Sandkörnern, ist nicht schneller als in $O(n^2)$ realisierbar.*

Beweis. Klar ist das Verhältnis der Seitenlängen eine Konstante, wir beschränken uns also auf Quadrate mit Seitenlänge a. Der vorherige Beweis zeigt, dass es $O(n^3)$ mal passiert, dass eine Zelle feuert. Ein Algorithmus kann nach dem Vorbild des sequentiellen Stack-Algorithmus realisiert werden. Es ist dann klar, dass $f_{max}(a)$ Zellen stabilisiert werden müssen. Führt nun ein Algorithmus zu jeder feuernden Zelle eine gewisse separate Simulation durch, so führt das zwangsläufig zu mindestens n^3 Berechnungen.

Die zweite Aussage des Satzes beruht auf folgendem Szenario: Betrachte die Konfiguration 3. Werfe ein Korn auf die Mitte, so entsteht eine Lawine der Größe $\theta(n^3)$. Andererseits fallen maximal $4n$ Sandkörner aus dem Gitter. Man erhöhe nun jede Zelle so lange, bis sie im Zustand 3 ist. Also vergehen maximal $4n$ Würfe mit Lawinen der Größe 0. Insgesamt feuern $\theta(n^3)$ Zellen bei $4n + 1$ Sandkörnern, also ist die Laufzeit amortisiert in $O(n^2)$. □

2.5 Vergrößerung beliebiger Mengen

Die Formeln für die exakten Werte von l aus 2.5.3 gelten bei beliebigen Mengen schon nicht mehr. Es gelten nicht einmal die Abschätzungen für Maxima von l aus 2.4.6, wie Abbildung 2.4 demonstriert.

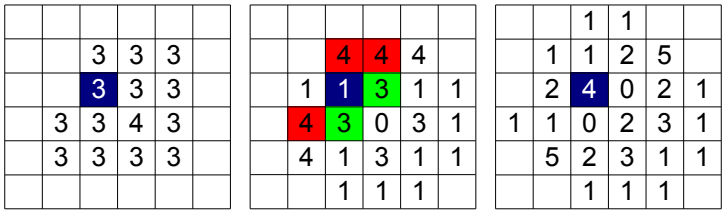

Abbildung 2.4: Ein Fall, in dem eine Zelle auf dem Rand zwei mal feuert. Dabei stabilisieren wir in der ersten Konfiguration nur das Quadrat der Seitenlänge 3, dessen Zentrum im Zustand 4 ist. Dabei kann 2.5.3 helfen. In der zweiten Konfiguration stabilisieren wir nur alle roten und grünen Zellen. Am Ende erreicht die blau hinterlegte Zelle zwei mal Zustand 4.

Als Motivation betrachte man eine beliebige Menge $M \subseteq R$, die von einem eindeutigen minimalen Rechteck K mit $M \subseteq K \subseteq R$ umgeben ist. Offenbar gelten die maximalen Feuerzahlen von K dann auch für M. Dies kann aber unzufriedenstellend sein. Wenn zum Beispiel die Startzelle der Lawine in der Nähe des Randes von K liegt, so findet man viele Beispiele wie in Abbildung 2.5, in denen die mittlere Zelle nur sehr selten

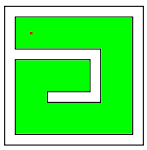

Abbildung 2.5: Eine Klasse von Mengen, bei denen das umgebende Rechteck für mittlere Zellen eine besonders schlechte Abschätzung liefert. Dabei sind alle weißen Zellen im Zustand 0, alle grünen in Zuständen aus $\{1, 2, 3\}$ und die rote Zelle im Zustand 4. Wegen der hohen Entfernung auf der „Schnecke“ ist es eher unwahrscheinlich, dass die mittlere Zelle auch nur genau so häufig wie die Startzelle feuert.

feuert. Hier liefern die bisherigen Formeln unrealistisch hohe Abschätzungen. Folgende Sätze liefern für ein Quadrat nicht nur eine maximale Obergrenze für l, sondern den genauen Wert. Dabei wird zuerst die Einschränkung getroffen, dass das Sandkorn auf eine der Diagonalen geworfen wird, und anschließend folgt der allgemeine Fall.

Satz 2.5.1. *Sei $K \subseteq R$ ein Quadrat und $c = 3|_K$. Man werfe ein Korn auf eine Startzelle $z \in R$ auf eine der Diagonalen. Dann gilt:*

$$\forall i \in R : l(c + e_z, i) = min(d(i, R \setminus K), d(z, R \setminus K)).$$

Beweis. Sei k fest, so dass $z \in V_k$. Alle Zellen auf V_k feuern genau k mal, denn würde eine Zelle weniger als ihre Nachbarzelle feuern, so könnte man aus Symmetriegründen eine Kette von Zellen bilden, an deren Ende eine Zelle auf V_k gar nicht feuert, was aber nicht der Fall sein kann. Nach Satz 2.4.6 ist damit schon das Maximum für alle Ringe $V_j, j \leq k$ erreicht.

Sei $M := \bigcup_{j>k} V_j$. Es bleibt nur noch zu zeigen:

$$\forall y \in M : l(c + e_s, y) = k.$$

Der Beweis geht induktiv: Sei c_{f_r} die Konfiguration, nachdem jede Zelle in V_k exakt r mal gefeuert hat. Als Induktionsannahme sei für ein $n \in \mathbb{N}_0$ $c_{f_k}|_M = 3$ und $\forall y \in M$: y hat genau k mal gefeuert. Die Induktionsvoraussetzung ist klar. Wir schließen auf n von $n + 1$ direkt mit Satz 2.4.1. Damit gilt die Induktionsannahme für alle $n \in \mathbb{N}_0$. □

Satz 2.5.2. *Sei $K \subseteq R$ ein Quadrat und $c = 3|_K$. Man werfe ein Korn auf eine Startzelle $z \in R$. Dann gilt:*

$$\forall i \in R : l(c + e_z, i) = min(d(i, R \setminus K), d(z, R \setminus K)).$$

Beweis. Sei zunächst z auf einer Diagonalen. Dann ist die Behauptung gezeigt. Sei also nun z nicht auf einer der Diagonalen. Sei $x \in R$ eine Zelle auf einer der Diagonalen mit $d(x, R \setminus K) = d(z, R \setminus K)$. Anhand der Häufigkeit, wie oft die Nachbarzellen von

e_x feuern, liest man ab, dass $(s(c+e_x))(z) = 2 < 3$, und daher $l(s(c+e_x)+e_z) = 0$. Andererseits ist aus Symmetriegründen $(s(c+e_z+e_x))(x) = (s(c+e_x))(x) = 2 > 0$, also erhalten wir nach 2.2.1 $l(s(c+e_z)+e_x) = 0$.

Auf K gilt:

$$l(c+e_x) = l(c+e_x) + l(s(c+e_x)+e_z) = l(c+e_z) + l(s(c+e_s)+e_x) = l(c+e_z).$$

Die zweite Gleichung gilt wieder wegen der bekannten Kommutativität aus Satz 2.1.1. □

Schließlich gilt der letzte Satz auch für Rechtecke:

Satz 2.5.3. *Sei $K \subseteq R$ ein Rechteck und $c = 3|_K$. Man werfe ein Korn auf eine Startzelle $z \in R$. Dann gilt:*

$$\forall i \in R : l(c+e_z, i) = min(d(i, R \setminus K), d(z, R \setminus K)).$$

Beweis. Ohne Einschränkung sei die Höhe des Rechtecks kleiner als die Breite. Sei $z \in V_k$ für ein $k \in \mathbb{N}$. Alle Quadrate, die oberen und unteren Rand mit K gemeinsam haben, bilden die Menge N_0. Man betrachte die Vereinigung

$$M_0 := \bigcup_{L \in N_0, z \in V_k(L)} L,$$

wobei $V_i(L) \subset R$ der Ring V_i von L sei. Aus Satz 2.5.2 folgt die Bedingung des Satzes auf ganz M_0. Bilde nun die Menge N_i aus Rechtecken der Breite von M_{i-1} für alle $i \in \mathbb{N}$. Man erhält so jedes mal echt breitere Rechtecke $M_j, j \in \mathbb{N}_0$, bis auch ein Rechteck gefunden wird, das die Bedingung für ganz K erfüllt. □

Korollar 2.5.1. *Für ein Quadrat $K \subseteq R$ mit Seitenlänge a sei $f(a,z)$ die exakte Anzahl des Ereignisses, dass eine Zelle feuert beim Wurf eines Korns auf eine Zelle $z \in R$. Dann gilt:*

$$f(a,z) = f_{max}(a) - 2f_{max}(d(z, R \setminus K)),$$

wobei

$$f_{max}(x) := \frac{1}{6}x(x+1)(x+2).$$

2.6 Synchrone Stabilisierung von Lawinen

Im Folgenden geben wir an, dass allein von der Nachbarschaft der Startzelle einer Lawine abhängt, ob alle Zellen höchstens einmal feuern. Dabei betrachten wir Zustände, die niemals feuern werden.

Satz 2.6.1. *Sei $c \in \{0,1,2,3\}^R$ beliebig, $i \in R$ fest.*

$$\exists \nu \in N : l(c+e_i, i+\nu) = 0 \Rightarrow \forall j \in R : l(c+e_i, j) \leq 1.$$

Beweis. Angenommen, eine Zelle feuert als erstes zum zweiten Mal. Dann muss sich ihr Wert insgesamt um mindestens 5 erhöht haben. Falls die Zelle diejenige ist, auf die das Korn geworfen hat, so wurde dessen Wert um 1 durch den Wurf und wegen der Annahme höchstens 3 mal um eins durch die Nachbarn erhöht. Jede andere Zelle wird maximal um 4 durch ihre Nachbarn erhöht. Also ein Widerspruch! □

Korollar 2.6.1. *Sei $c \in \{0,1,2,3\}^R$ beliebig, $z \in R$ fest. Führe s aus mit der Einschränkung, dass z nur einmal stabilisiert werden darf. Wir führen dafür s_z und l_z ein. Dann gilt:*

$$\forall j \in R : l_z(c + e_z, j) \leq 1.$$

Beweis. Wähle zunächst ein $\nu \in N$ und führe s auf allen Zellen bis auf $z + \nu$ genau 1 mal aus. Nun sind bis auf Zelle $z + \nu$ alle Zellen sicher stabil und Zelle $z + \nu$ maximal im Zustand 7. Stabilisiere nun ausschließlich $z + \nu$. Offenbar ist j in einem der Zustände $\{0,1,2\}$ für $j \in (z + \nu) + (N \setminus (\{0,0\} \cup -\nu))$, also feuern diese Zellen nicht. Nur z kann in $\{0,1,2,3\}$ sein. Daraus folgt die Behauptung. □

Korollar 2.6.2. *Sei $M \subseteq R$ und K ein Rechteck, so dass $M \subseteq K \subseteq R$. Sei $c = 3|_M$. Man werfe ein Sandkorn auf $z \in R$. Dann gilt: Der Wurf kann in $d(z, R \setminus K)$ Schritten stabilisiert werden, wobei in jedem Schritt jede Zelle höchstens ein mal feuert.*

Beweis. Sei $c_0 := c$ und $c_k := s_z(c_{k-1}), k \in \mathbb{N}$. Nach 2.6.1 gilt induktiv: $\forall i \in R \setminus \{z\} : c_k(i) \leq 3$ und $c_k(z) \leq 4$ für alle $k \in \mathbb{N}_0$. Andererseits gilt nach 2.4.6 $\forall i \in R : l(c+e_z, z) \leq d(z, R \setminus K)$. □

Korollar 2.6.3 (Maximalität der Startzelle)**.** *Sei $c \in \{0,1,2,3\}^R$ beliebig, $z \in R$ fest.*

$$\forall j \in R : l(c + e_z, j) \leq l(c + e_z, z).$$

2.6.1 Synchrone Parallelisierung einer Lawine

Mit 2.6.2 kann nun der Sandhaufen-Algorithmus auf eine völlig neue Weise simuliert werden. Wenn nämlich jede Zelle maximal einmal feuert, reicht es, die Menge der feuernden und dadurch veränderten Zellen zu ermitteln. Ist dies geschehen, so können diese Zellen völlig synchron neu berechnet werden, also mit perfekter Parallelisierung! Algorithmus 1 funktioniert genau so, wobei hier die for-Schleifen beliebig parallelisiert werden können. Das einzig kritische an diesem Algorithmus ist, die Menge der feuernden Zellen zu bestimmen.

2.6.2 Bestimmung der veränderten Zellen

Der gesuchte Algorithmus kann durch Markieren von Zellen durchgeführt werden. Sei $c \in \{0,1,2,3\}^R$ beliebig und man werfe ein Korn auf Zelle $z \in R$. Im Folgenden sei immer angenommen, dass z dann feuert. Setze

$$L_0 := M_0 := \{z\}.$$

Algorithmus 1 Stabilisiere Zelle i auf A[]

```
A[i] ← A[i] + 1
while A[i] = 4 do
  M ← feuernde(i)
  L ← veraenderte(i)
  for all z ∈ M do
    A[z] ← A[z] − |{y ∈ z + N, y ∉ M}|
  end for
  for all z ∈ L do
    A[z] ← A[z] + |{y ∈ z + N, y ∈ M}|
  end for
end while
```

Berechne nun zunächst L_j, dann $M_j, j \in \mathbb{N}_0$, mit

$$L_j := \{i \in R | (i + N) \cap M_{j-1} \neq \emptyset\},$$

$$M_j := \{i \in L_j | |\{n \in N : i + n \in M_{j-1}\}| + c(i) > 3\}.$$

Offenbar feuern alle Zellen in $M_j, j \in \mathbb{N}_0$ und alle Zellen in L_j werden verändert, und falls $\exists s \in \mathbb{N}_0 : M_s = M_{s+1}, L_s = L_{s+1}$, so ist M_s genau die Menge aller feuernden Zellen und N_s die Menge aller Zellen, die feuern oder sich verändern.

Die Berechnung kann sehr effizient mit dem Algorithmus „Paralleler Stack“ aus der Diplomarbeit von Sebastian Frehmel [Fre10] ausgeführt werden. Dabei werden keinerlei Mutexe benötigt. Eine vollständige Implementierung wurde unter */src/parallel/synchronous/stack* eingereicht. Sie funktioniert noch nicht, verdeutlicht aber die Funktionsweise.

2.6.3 Alternativen

Andere Möglichkeiten, die Menge der feuernden Zellen zu bestimmen, wären möglich. Man kann versuchen, den Umriss der Lawine durch „Right-First“- oder „Left-First“-Suche auf einem planaren Graphen zu bestimmen. Beachten muss man auch, dass dabei „Löcher“ entstehen können.

1. Man könnte alle zusammenhängenden „3-er“-Mengen während der gesamten Simulation aktuell halten, zum Beispiel, indem man deren Rand mit einer verketteten Liste einrahmt. Das Hinzufügen von Zellen, wenn diese von 2 zu 3 übergehen, geht dann schnell. Auch der Zerfall und die Konkatenation dieser Mengen geht sehr schnell. Damit könnte man die oben beschriebenen $M_j, j \in \mathbb{N}_0$ vermutlich recht schnell gewinnen.

2. Bestimmte Zell-Kombinationen bilden Barrieren für das Wachstum der $M_j, j \in \mathbb{N}_0$. Ein Beispiel dafür ist ein Rahmen von Zellen im Zustand 2. Mit Hilfe solcher Barrieren kann man vielleicht die Lawine auf geschickte Weise von außen einkreisen.

Korollar 2.6.2 liefert damit schon genug Umfang für eine eigenständige Bachelorarbeit.

Kapitel 3

Portierung auf C++

Dieses Kapitel umfasst zuerst eine Begründung für die Wahl von C++ und deren einzelne Bibliotheken. Anschließend werden die portierten Algorithmen vorgestellt und auf einige Änderungen und Erweiterungen am Code hingewiesen. Für algorithmische Details zu den portierten Algorithmen sei man auf [Fre10] verwiesen.

3.1 Wahl der Programmiersprache

Alle Algorithmen wurden in einer einheitlichen Programmiersprache verfasst. Es war interessant zu sehen, ob die bestehenden Algorithmen, welche in Java verfasst waren, bei einer Portierung auf eine andere Programmiersprache noch spürbare Laufzeitverbesserungen erfahren konnten. Daher wurde eine Sprache gesucht, die im Vergleich zu Java als „maschinennahe“ gewertet werden kann.

Die Programmiersprachen C und C++ erfüllen diese Anforderung. Der Code wird beim Kompilieren sofort in ausführbaren Code übersetzt. Auch bieten sie zum Beispiel eine granularere Typisierung und eine Vielzahl von direkten Systemaufrufen des Betriebssystems. C und C++ sind trotz ihrer Laufzeitvorteile als Hochsprachen anzusehen. C hat nur eine geringe Anzahl an Schlüsselwörtern, und C++ bietet eine darauf aufbauende Objektorientierung.

Es gibt viele Gründe, die die Vermutung stützen, dass ein Java-Programm langsamer sein könnte als ein C++-Programm. Zum Beispiel behandelt Java alle Funktionen als *virtual*, während dies in C++ ausdrücklich so deklariert werden muss. Ein weiterer Punkt ist, dass man in Java Objekte nicht auf dem Stack ablegen kann, sondern immer mit *new* erzeugen muss. Aber das wahrscheinlich wichtigste Argument ist, dass Java permanent mit der Übersetzung von Code beschäftigt ist. Die JVM kann aber den übersetzten Code auch zwischenspeichern und sogar während der Laufzeit analysieren, wie der Code am besten übersetzt werden könnte. Es gibt also hier auch Gegenargumente.

3.2 Unterschiede zwischen C++ und Java

Java gilt für manche Programmierer als der Nachfolger von C++ und beide Sprachen haben syntaktisch sehr viel gemeinsam. Die Unterschiede zwischen C++ und Java sind aber nicht immer offensichtlich. Bei einer Portierung von Java-Code in C++-Code gibt es durchaus Gefahren, deren Kenntnis hilfreich sein könnte, um resultierende Fehler zu finden.

Es gibt viele Unterschiede zwischen C++ und Java, die man schnell sieht. Dazu gehören unterschiedliche Bezeichner, unterschiedlicher Funktionsumfang und natürlich ein unterschiedlich assoziierter Coding-Style. Ein einfaches ersetzen bestimmter Bezeichner reicht bei einer Portierung aber oft nicht aus. Folgende Punkte sollte man unbedingt beachten:

- In C++ ist das Verhalten undefiniert, wenn virtuelle Funktionen aus dem Konstruktor der Basisklasse aufgerufen werden; selbst, wenn der Aufruf über mehrere Funktionen geschieht. Der g++-Compiler gibt diesbezüglich in Version 4.5 nur eine Warnung aus, die man übersehen könnte. Andere Compiler warnen vielleicht gar nicht oder bemerken verschachtelte Aufrufe nicht. In Java hingegen ist dieses Vorgehen üblich, da es bequem ist.

```
1 public abstract class AbstractOptimizedCacheStrategy extends Header
      implements ICache {
2   //...
3   public AbstractOptimizedCacheStrategy() {
4     //...
5     setNoCacheStrategy(); // virtuell, definiert in ICache
6     //...
7   }
8 }
```

  In C++ geht dies nicht. Wird in C++ eine Klasse mit einer Oberklasse instantiiert, so wird zuallererst der Konstruktor der Oberklasse aufgerufen. Bis zu einschließlich diesem Ereignis behandelt C++ das Objekt jedoch wie eines der Basisklasse. Dieses Verhalten wird dadurch gerechtfertigt, dass die Attribute der Unterklasse noch nicht initialisiert sind und es daher unsicher sein kann, mit diesen zu arbeiten [Mey05]. Daher verhalten sich die virtuellen Funktionen hier noch nicht virtuell. In C++ kann man dieses Problem recht einfach beheben, auch, wenn es softwaretechnisch nicht unbedingt optimal ist:

```
1 class AbstractOptimizedCacheStrategy : public ICache {
2   public:
3   AbstractOptimizedCacheStrategy() {}
4   // ...
5   void init() {
6     // ...
7     if(/*...*/) {
8       performPostOperation(); // virtuell, definiert in ICache
```

```
9       } else {
10        setNoCacheStrategy(); // virtuell, definiert in ICache
11      }
12    }
13  }
```

Der Aufrufer muss in diesem Fall sicher stellen, dass er nach dem Konstruktoraufruf die Funktion *init()* aufruft. Falls man dies dem Aufrufer nicht zutraut, so kann man Absicherungen mit Hilfsvariablen einfügen.

- C++ erlaubt keine zyklischen Header-Inklusionen. Ein Header darf also keinen Header einbinden, der ihn selbst einbindet. Manche Compiler, wie der g++ in Version 4.5, lösen ein Glied des Zyklus und geben dann einen Fehler über eine undefinierte Referenz zurück. Dies kann dazu führen, dass der Compiler manche Typen nicht kennt, obwohl deren Header eingebunden ist. Hat man das Problem identifiziert, so ist es nicht immer trivial zu beheben, da sich manchmal Header gegenseitig benötigen. Oft ist das Problem zu beheben, indem man die Implementierung in eine „.cpp-Datei“ verlagert, da diese üblicherweise nicht eingebunden werden.

- C++ bietet keine automatische Speicherbereinigung. Man muss also darauf achten, dass Speicherplatz bereinigt wird, bevor man die letzte Referenz darauf verliert, aber erst nach dem letzten Zugriff. Insbesondere ist ein Aufruf von *exit()* mitten im Programm nicht akzeptabel, da so der Speicher nicht ordnungsgemäß freigegeben werden kann.

- Java initialisiert Attribute automatisch mit Standardwerten; insbesondere Integer-Variablen mit 0. Es kann zu einem Laufzeitfehler in C++ führen, wenn man die Initialisierung dort vergisst. Es folgt ein Code-Beispiel dazu.

```
1 class AnyClass {
2   int index;
3   char data[4];
4   void gelElem() { return data[index]; }
5 };
```

Diese Klasse kann man sowohl in C++ als auch in Java übersetzen. In C++ kann dies aber zu einem Fehler führen, da *index* beliebig initialisiert werden kann. Das Verhalten ist also undefiniert und kann sogar Speicherzugriffsfehler auslösen.

- Besondere Vorsicht gilt auch bezüglich des Zeitpunktes der Initialisierung statischer Variablen. Diese geschieht bei C++ vor Aufruf der main()-Methode, während Java dies erst bei Initialisierung der Klasse tut.

- Beim Arbeiten mit Threads gibt es Unterschiede. Die JVM wartet am Ende, bis alle Threads ihre Arbeit beendet haben. In C++ führt ein Terminieren des

Main-Threads sofort zum Ende des Programms. Warten die Threads nicht korrekt aufeinander, so können Speicher-Lecks oder sogar Speicherzugriffsfehler entstehen.

3.3 C oder C++

Die Entscheidung zwischen C und C++ fiel zugunsten von C++ aus. Wie groß ist aber der Unterschied zwischen beiden Sprachen? Wie steht es bezüglich der Laufzeit und der Programmierparadigmen?

C++ wird oft als „C mit Klassen“ bezeichnet. Die Arbeit an C++ wurde 1979 von Bjarne Stroustroup begonnen, und erst im Jahre 1998 erstmals vom ISO-Komitee standardisiert. Als Stroustroup begann, C++ zu entwickeln, behalf er sich anfangs mit Makros, so dass C++ eine reine Erweiterung zu C war [Str07]. Mittlerweile sind Klassen, Templates und anderes längst Konstrukte, die dem Compiler bekannt sind. Auch ist C++ nicht nur eine syntaktische Obermenge, denn C++-Compiler schränken die C-Syntax zum Teil ein. C++ ist eine eigenständige Programmiersprache.

Ein typisches Beispiel für das Argument, dass C++ keine syntaktische Obermenge sein kann, ist die strengere Handhabung von Typumwandlungen. Etwas weniger bekannt ist, dass Funktionszeiger ohne Argumente in C, anders als in C++, ohne Argument nicht das gleiche sind wie Funktionszeiger mit dem Argument void. Erstere dürfen nämlich auf Funktionen mit beliebigen Argumenten zeigen. In C++ ist das nicht mehr möglich. Hier ein Beispiel:

```
1  void (*funcptr)();
2
3  void f2(int i) {
4    printf("i=%d\n", i);
5  }
6
7  int main() {
8    funcptr = &f2;
9    funcptr();
10   return 0;
11 }
```

Bezüglich der Softwarequalität ist C++ als objektorientierte Programmiersprache erste Wahl. In diesem Fall ist aber Laufzeit sehr kritisch. Diesbezüglich gilt C++ im Allgemeinen als geringfügig langsamer. In der Tat besteht beim Programmieren mit C++ die Gefahr, die Laufzeit geringfügig zu erhöhen. Zum Beispiel kann durch überflüssige virtuelle Funktionen unnötig Speicher und Rechenzeit in Anspruch genommen werden. Außerdem können Informationen über exceptions den Speicher geringfügig belasten.

Es gibt aber auch Benchmarks, die das Gegenteil behaupten. Es gibt kein Gegenstück zu Templates in C. Templates helfen dem Programmierer, Routinen für mehrere Typen von Variablen oder sogar verschieden Konstanten zu schreiben, ohne den Code zu duplizieren. Dies bietet dem Compiler Vorteile, wie zum Beispiel die Vorberechnung von

Potenzen. Auch gibt es nach ISO-Standards keine inline-Funktionen in C++. Letzten Endes kann bessere Softwarequalität auch zu übersichtlicherem Code führen, in dem es einfacher ist, Routinen zu finden, die ein Hindernis für die Laufzeit darstellen.

Im Allgemeinen kann man C und C++ als in etwa gleich schnell ansehen. Die Wahl fällt wegen Objektorientierung und Verfügbarkeit der STL auf C++.

3.4 Thread-Bibliothek

Da das Programm auf verschiedenen Linux- und Unix-Derivaten funktionieren sollte, war es sinnvoll, eine POSIX-standardisierte API zu nutzen. Für Threads ist dies die Bibliothek „pthreads“.

Pthreads ist eine C-API ohne C++-Konzepte. Sie läuft nicht nur auf allen „POSIX-konformen“ Betriebssystemen, sondern wurde auch auf Windows portiert. Sie gilt als sehr portabel. Außerdem ist pthreads nicht besonders „high-level“.

Eine portable C++-API sollte also auf pthreads basieren. Eine Möglichkeit dafür ist die Bibliothek „Threads“ aus boost, einer der „angesehensten und mit am meisten Expertenwissen entworfenen C++-Bibliothek-Projekten der Welt“ (Übersetzung aus [SA05]). Die boost-Threads-Bibliothek verwendet nicht auf allen Betriebssystemen die Pthreads-Bibliothek und kann dadurch theoretisch eine höhere Portabilität erreichen. Außerdem ist sie ebenfalls sehr „low-level“.

Aufgrund der Vorteile der boost-Bibliothek wurden keine Alternativen in Betracht gezogen.

3.5 Wahl der GUI

Die GUI sollte ebenfalls auf C++ portiert werden. Da es jedoch unter C++ keine standardisierte GUI gibt, wurde eine Auswahl getroffen. Bekannte GUIs für C++ sind Qt, GTKmm und WxWidgets. Alle GUIs sind portabel. Wichtig ist vor allem eine gute Laufzeit, um eine hohe Bildrate zu erreichen. Alle GUIs sind dazu in der Lage. Persönliche Präferenzen zu der Qt-GUI beruhen vor allem auf der sehr ausführlichen Dokumentation. Außerdem wird Qt ständig weiterentwickelt. Im Rahmen dieser Bachelorarbeit fiel die Entscheidung zugunsten von Qt.

3.6 Portierte Algorithmen

Im Laufe dieser Arbeit wurden folgende Algorithmen aus der Diplomarbeit von Sebastian Frehmel [Fre10] portiert:

- Der Algorithmus von Walter und Worsch [WW04]
- Der sequentielle Stack-Algorithmus

- Der Vierzustands-Algorithmus in der parallelen Variante
- Der parallele Stack-Algorithmus

Außerdem wurde das entsprechende Rahmenwerk inklusive der GUI portiert. An den Algorithmen selbst war es nicht notwendig, viel zu ändern. Für das Rahmenwerk mussten jedoch einige Anpassungen vorgenommen werden.

3.7 Besondere Anpassungen im Quellcode

3.7.1 Klassen CPUAffinitySetter und ThreadTracker

Die Affinität eines Threads zu setzen ist mit C unkomplizierter, da man hier nicht auf ein Kommandozeilenprogramm zurückgreifen muss, sondern direkt entsprechende C-Funktionen aufrufen kann. Letztere werden von der boost-Bibliothek für verschiedene Betriebssysteme bereitgestellt. Außerdem ist es nicht notwendig, die Prozess-IDs zu verfolgen, da man diese keinem externen Programm mitteilen muss. Durch letzteres wurde die Klasse ThreadTracker obsolet.

Die Klasse CPUAffinitySetter konnte außerdem deutlich vereinfacht werden. Durch das Weglassen von Kommandozeilenprogramm-Aufrufen ist das Programm auch portabler. Zwar ist der entsprechende Code zum Zeitpunkt der Fertigstellung noch nicht Bestandteil der boost-Bibliothek und benötigt die pthreads-Bibliothek, kann aber portabel angepasst werden.

3.7.2 Mutex-Klasse

Boost stellt Mutex-Klassen zur Verfügung. Unter Linux nutzen diese die Pthreads-Bibliothek. Diese Art von Mutexen ist aber deutlich langsamer als das Nutzen der „compare and swap“-Instruktion, bei der der Zähler der Mutex-Variablen über einen einzigen atomaren Befehl gelesen und geschrieben wird. Da Mutexe bei den parallelen Algorithmen eine ganz entscheidende Rolle spielen, wurden „compare and swap“-Mutexe entworfen. Auch die Spinlocks wurden so geschrieben. In der Tat konnten so in etwa 50 Prozent Laufzeit gespart werden.

3.7.3 Namensraum os

Der Namensraum os ist neu im Rahmenwerk und definiert Funktionen, die von den Betriebssystemen unterschiedlich implementiert werden. Diese werden von Java meist schon mitgeliefert, in C++ sind sie hingegen kein verlässlicher Standard. Es befinden sich hier Funktionen zum Messen der Laufzeit, zur Rückgabe von Zufallswerten und zum Erstellen von Verzeichnissen. Insbesondere befindet sich hier auch die Initialisierungsroutine für Zufallswerte. Man kann hier den Code einfach ändern, standard ist die *random()*-Routine.

3.7.4 AbstractLogOperator

Für den Repository-Algorithmus wurde diese Klasse abgeändert, da sie den Anforderungen an Parallelisierbarkeit nicht entsprach. Das Hauptproblem dabei war, dass Threads unter Umständen bereits Informationen an den Log-Operator übermittelt haben, die im Nachhinein wieder gelöscht werden müssen.

Ansonsten gab es nur kleine Änderungen bei der Pufferung der Ausgabe.

3.7.5 GUI

Im Wesentlichen gibt es in Qt zwei verschiedene Klassen, um mit Bild-Daten umzugehen, und zwar QImage und QPixmap. Für eine Grafik-Bibliothek wie X11, die als Server angesprochen wird, gilt dabei folgendes: Während QImage die Pixel im Client hält und dort manipuliert, geschieht dies bei QPixmap im Server. Da in der Implementierung des Rahmenwerks ständig ein Pixel nach dem anderen manipuliert wird, ist QImage hier die passende Wahl.

3.7.6 Fehlerbehebung

In den alten Algorithmen wurden einige Fehler gefunden, die in der C++-Version ausgebessert wurden. Anschließend wurden alle Speicher-Lecks gesucht und behoben.

Als Sonderfall kann man auch eine unverständliche Ausgabe oder unerwartete Reaktion des Programms zählen. Für diese Fälle wurde ein Makro „CC_STRICT“ definiert, welches das Programm abbricht oder nicht kompiliert. Dabei wird eine aussagekräftige Begründung geliefert.

In folgenden Situationen wird das Kompilieren verweigert:

- Es gibt Algorithmen, in denen die Zustände der Zellen immer nur im Bereich $\{0, 1, 2, 3\}$ liegen. Das ist zum Beispiel bei der „fourstates“-Simulation der Fall. Wenn der Benutzer in diesem Fall die GUI benutzt und eine Farbpalette verwendet, in der die Werte $\{0, 1, 2, 3\}$ eine gleiche Farbe haben, bleibt die Anzeige der GUI dauerhaft unverändert. Das macht keinen Sinn und kann zur Verwirrung führen.

- Programme, die aufgrund von Makros nicht ausreichend Informationen loggen, terminieren nie, auch wenn man eine Obergrenze für die Anzahl der Sandkörner spezifiziert. Das kann zu einem endlos laufendem Programm und sogar Speicherzugriffsfehlern führen.

In folgenden Situationen bricht das Programm vor der Simulation ab:

- Falls der Anwender nur sehr wenig Sandkörner wirft, kann es sein, dass das Programm nach dem Erreichen der geforderten Anzahl Sandkörner so bald keine Lawine verursacht. Dies kann zu einem Überschreiten des Arrays mit den vordefinierten Koordinaten für die Würfe führen, was zu einem dem Anwender unverständlichen Speicherzugriffsfehler führt.

- Falls der Anwender sehr kleine Gitter wählt, bei denen die Weite in einer Dimension kleiner als die angegebene Größe der Cache-Zeilen ist, so wird die Cache-Optimierung mit einem Speicherzugriffsfehler fehlschlagen.

3.7.7 Lesen aus Textdateien

Im Zuge dieser Arbeit wurden noch Erweiterungen am Rahmenwerk und zusätzliche Tools geschrieben. Eine besondere Erweiterung ist, dass vordefinierte Start-Konfigurationen in Form von Textdateien übergeben werden können, was für theoretische Analysen vorteilhaft sein kann.

Kapitel 4

Neue Algorithmen

Im ganzen Kapitel sei $Q = \{0, 1, 2, 3\}$.

4.1 Parelleler SmallCell-Stack-Algorithmus

4.1.1 Motivation

Der SmallCell-Stack-Algorithmus entstand als Grundlage für den Hashing-Algorithmus (siehe 4.2). Es war das Ziel, möglichst viele Zellen in einem Wort speichern zu können. Dies war wichtig, um das Array für die Hash-Funktion möglichst klein zu halten.

Als erfreulicher Nebeneffekt ergibt sich natürlich eine geringere Speicherauslastung, die mit einer besseren Cache-Effizienz verbunden ist. Eine Herausforderung ist das Rechnen mit diesen komprimierten Daten. Der folgende Algorithmus hat dieses Thema zur Grundlage.

4.1.2 Übersicht zum Algorithmus

Der SmallCell-Stack-Algorithmus lässt sich grob in vier Schichten unterteilen:

1. Berechnung eines einzelnen Wurfs in einer 16-er Zelle
2. Speicherung mehrerer Würfe pro 16-er Zelle
3. Austausch der austretenden Bits zwischen den 16-er Zellen
4. Algorithmus, der eine 16-er Zelle wie eine Einer-Zelle bearbeitet

Dies entspricht dem Klassendiagramm in Abbildung 4.1

4.1.3 Aufbau einer Zelle

Jede kleinste Zelle enthält 16 untergeordnete Zellen. Jede dieser 16 Zellen benötigt 2 Bits für die 4 möglichen Zustände. Die restlichen 32 Bits werden genutzt, um für jede der

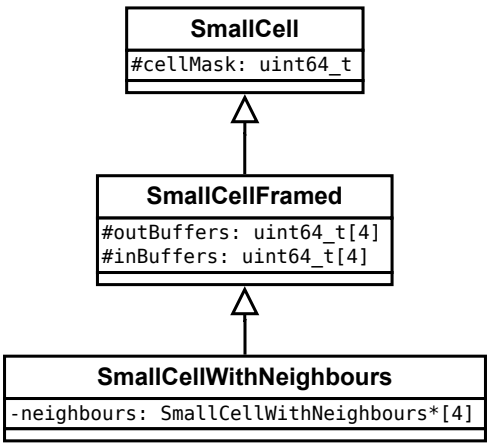

Abbildung 4.1: Stark vereinfachtes Klassendiagramm.

Seiten einen Puffer zu bilden, der aus der 16er-Zelle austretende Bits temporär speichert. Diesen Puffer bezeichnen wir im Folgenden als „Rand-Puffer“.

Eine Zelle wird als 64-Bit-Integer repräsentiert. $Q \cong (\mathbb{Z}/4\mathbb{Z}, +)$ sei wieder der natürliche $\mathbb{Z}$-Modul. Sei λ der natürliche Isomorphismus

$$\lambda : (\mathbb{Z}/4\mathbb{Z}, +)^{32} \to (\mathbb{Z}/2\mathbb{Z}, +)^{64},$$

bei dem $a \in (\mathbb{Z}/4\mathbb{Z}, +)^{32}$ so auf $\lambda(a)$ abgebildet wird, dass

$$\sum_{i=1}^{32} {a_i}^i = \sum_{j=1}^{64} \lambda(a)_j{}^j.$$

Dann nennen wir die Repräsentation der Folge in Bits, in der Zelle $\lambda(a)_i$ das Bit 2^i angibt, *Zell-Maske*.

Abbildung 4.2 skizziert die Positionierung der Bits, wobei jede Zelle 2 Bits enthält. Diese Bit-Positionen bezeichnen wir als *Positionen der Zellen*. Es handelt sich um die

	0	1	2	3	
4	5	6	7	8	9
10	11	12	13	14	15
16	17	18	19	20	21
22	23	24	25	26	27
	28	29	30	31	

Abbildung 4.2: Aufbau einer kleinsten Zelle. Der Rand-Puffer ist grau hinterlegt.

Zahlen $\{0, ..., 31\} \subseteq \mathbb{N}_0$.

Nach Satz 2.4.6 feuert jede Randzelle höchstens einmal bei einem geworfenem Sandkorn. Demnach reicht der temporäre Randpuffer also für bis zu drei Würfe und ist damit ausreichend.

Der Algorithmus muss aber auch die Bit-Position der Nachbarn einer jeden Zelle effizient berechnen können. Der linke und rechte Nachbar sind einfach durch Subtraktion bzw. Addition von 1 zur Position der Zelle zu finden. Hingegen ist dies beim unteren und oberen Nachbar arithmetisch nicht ohne Fallunterscheidung zu bewerkstelligen. Zum Beispiel wäre Feld 6 der obere Nachbar zu Feld 12, aber der obere Nachbar zu Feld 6 ist Feld 1. Um hier an Laufzeit zu sparen, wurden diese beiden Funktionen in einem Array vorberechnet.

Außerdem wurde eine weitere Eigenschaft des Layouts ausgenutzt. Sei Bit 0 das „least significant bit“. Wenn eine Zelle, die die Bits $2n, 2n+1$ für ein $n \in \mathbb{N}_0$ enthält, beim Feuern einen Übertrag nach „rechts“ liefert, bedeutet dies, dass sich ihr Zustand von 3 auf 0 ändert und gleichzeitig der Zustand der rechten Zelle mit Bits $2n+2, 2n+3$ um 1 erhöht wird. Dieser Vorgang kann sich über mehrere Zellen fortsetzen. Dies ist dann äquivalent zu der Formulierung, dass man $2^{2n} = 4^n$ zur Zell-Maske hinzuaddiert.

Der interne Algorithmus zum Wurf eines Sandkorns auf eine 16er-Zelle ist nun in folgendem C-Code dargestellt. Dabei wurden zur Vereinfachung Logging-Methoden entfernt.

```
1  bool throwOnFieldWithoutAdd(unsigned int xlatedNum) {
2    if((cellMask & ((cellType) 3 << (xlatedNum << 1))) == 0) {
3
4      // the add of this cell was already done by integer arithmetic
5      // => do not add, only check
6      throwOnFieldWithoutAdd(xlatedNum + 1);
7
8      throwOnField(xlatedNum - 1);
9      throwOnField(_upperCell[xlatedNum]);
10     throwOnField(_lowerCell[xlatedNum]);
11
12     return true;
13   }
14   return false;
15 }
16
17 bool throwOnField(unsigned int xlatedNum) {
18   cellMask += (cellType) 1 << (xlatedNum << 1);
19   return throwOnFieldWithoutAdd(xlatedNum);
20 }
```

4.1.4 Speicherung mehrerer Würfe

Nach spätestens 3 Würfen sichert 2.4.6 nicht mehr ab, dass die 2-Bit großen Puffer der Rand-Zellen ausreichen. Daher werden die Bits zurückgesetzt und entsprechende Zähler erhöht.

Zur Speicherung eines jeden Zählers dienen jeweils 16 Bits, so dass jede Seite des Quadrats mit einem 64-Bit-Integer auskommt. Nach Satz 2.4.6 ist das ausreichend für Quadrate von einer Seitenlänge d mit $d \leq 2^{15}$. Wir nennen diese 16 Bits *Ausgabe-Puffer*. Die jeweils niedrigsten 2 Bits der genannten 16 Bits sind der Teil der 16 Bits, auf die der Zustand der entsprechenden Zelle des Rand-Puffers aufaddiert werden muss. Wir nennen sie die *Übertrags-Bits* zu dem jeweiligen Ausgabe-Puffer.

Tatsächlich werden die 16 Bits direkt hintereinander in einem 64-Bit-Integer zu jeder Seite gespeichert, so dass man auch die Ausgabe-Puffer pro Seite ebenfalls als eine Zell-Maske auffassen kann.

Man betrachte eine feste Seite. Die Zell-Positionen der Übertrags-Bits der Zell-Maske der Ausgabe-Puffer zu jeder Seite sind immer $\{0, 8, 16, 24\}$. Da die Differenzen der Zell-Positionen der Randpuffer-Zellen in der ursprünglichen Zell-Maske nicht jeweils 8, sondern 5 für die westlichen und östlichen Puffer bzw. sonst 1 betragen, muss vor der Addition eine Transformierung durchgeführt werden. Glücklicherweise geht dies schon mit je 4 Bitshift-, einer dreistelligen Oder- und zwei Und-Operationen. Der Code dafür ist:

```
1  cellType northBitsToBufferFormat() const {
2    const cellType northBits = cellMask & BORDER_MASK_NORTH;
3    return (northBits | northBits << 14 | northBits << 28 | northBits << 42)
4      & BITS_ADD_MASK;
5  }
6
7  cellType southBitsToBufferFormat() const {
8    const cellType southBits = cellMask & BORDER_MASK_SOUTH;
9    return (southBits >> 56 | southBits >> (56-14)
10     | southBits >> (56-28) | southBits >> (56-42))
11     & BITS_ADD_MASK;
12 }
13
14 cellType eastBitsToBufferFormat() const {
15   const cellType eastBits = cellMask & BORDER_MASK_EAST;
16   return (eastBits >> 18 | eastBits >> 14 | eastBits >> 10 | eastBits >>
         6)
17     & BITS_ADD_MASK;
18 }
19
20 cellType westBitsToBufferFormat() const {
21   const cellType westBits = cellMask & BORDER_MASK_WEST;
22   return (westBits >> 8 | westBits >> 4 | westBits | westBits << 4)
23     & BITS_ADD_MASK;
24 }
25
26 void storeBordersInOutBuffers() {
27   outBuffers[HeaderCache::NORTH] += northBitsToBufferFormat();
28   outBuffers[HeaderCache::SOUTH] += southBitsToBufferFormat();
29   outBuffers[HeaderCache::EAST] += eastBitsToBufferFormat();
30   outBuffers[HeaderCache::WEST] += westBitsToBufferFormat();
```

```
31   resetBorders();
32 }
```

Abbildung 4.3 zeigt, welche Zellen von den Bitshift-Operationen betroffen sein können. Diese Prozedur ist effizienter als alle Bits einzeln mit je einem „Und“ zu filtern und

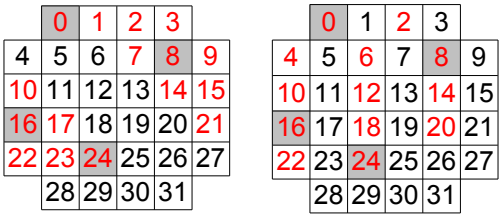

Abbildung 4.3: Zellen, die nach den Bitshift-Operationen nicht 0 sein müssen, sind rot gefärbt. Links der Fall für Puffer im Norden oder Süden, rechts der für Puffer im Osten oder Westen.

dann zu verschieben, denn es spart 8 Und-Operationen pro Aufruf *von storeBordersInOutBuffers()*. Es muss gezeigt werden, dass die Bits, die bei den Bitshift-Operationen unnötig geschoben werden, nicht die Übertrags-Bits bei den Oder-Operationen beeinflussen können. Zur Korrektheit sei folgender Satz gegeben:

Satz 4.1.1. *Die Übertrags-Bits können nur durch je ein Resultat einer Bitshift-Operation von 0 verschieden werden.*

Beweis. Für die Rand-Puffer im Norden und Süden ist es klar. Betrachte den anderen Fall. Angenommen, es gibt zwei Zellen, die Übertrags-Bits enthalten und deren Bitmasken sich überlagern. Deren Zell-Positionen seien i und j mit $i, j \in \{0, ..., 31\}$. Damit sich die Bitmasken überlagern muss $j - i \in \{0, 6, 12, 18\}$ gelten, da die Puffer vor dem Bitshift jeweils jede sechste Zelle belegt haben und es insgesamt nur 4 Zellen pro Puffer waren. Andererseits sollen i und j nach dem Bitshift je die Übertragsbits in der Zell-Maske belegen, d.h. $8|j - i$. Zusammen ist $j - i = 0$, also $j = i$. □

Man beachte, dass die Zähler für eine parallele Version unter Umständen mit Mutexen ausgestattet werden müssen. Für diese Version wurde dies so gehandhabt.

4.1.5 Austausch zwischen zwei 16er-Zellen

Der Austausch zwischen zwei 16er-Zellen erfolgt über die Ausgabe-Puffer. Deren Werte werden zunächst in den sogenannten *Eingabe-Puffer* einer anderen Zelle kopiert, bevor sie dort weiter verarbeitet werden.

Der Arbeitsalgorithmus einer Zelle lässt sich in folgendem Code-Ausschnitt nachvollziehen.

```
void catchNeighboursGrains() {
  for(unsigned int i = 0; i < 4; i++) {
    // i+2 % 4 is the opposite direction
    inBuffers.cellTypeVal[i] = neighbours[i]->fetchNeighboursOutBuffer((i
        +2)%4);
  }
}
void throwAllGrains() {
  boost::uint16_t *sixteenBits = inBuffers.shortVals;
  for(int i = 0; i < 16; i++, sixteenBits++) {
    while(*sixteenBits) {
      throwGrainXlatedWithoutStore(_throwIndexToXlated[i]);
      (*sixteenBits)--;
      storeBordersInOutBuffers();
    }
  }
}
void doCompleteWorkUnit() {
  catchNeighboursGrains();
  throwAllGrains();
}
```

4.1.6 Algorithmus auf den 16er-Zellen

Der Algorithmus entspricht dem der parallelisierten Stack-Version, mit einem Unterschied. Anstatt 1x1-Zellen werden nun 4x4-Zellen als kleinste Einheit betrachtet. Bis auf kleine Anpassungen verhält er sich aber sonst gleich.

4.1.7 sequentielle Variante

Der sequentielle SmallCell-Stack-Algorithmus ist eine Modifikation. Offensichtlich werden keine Mutexe gebraucht. Außerdem wurde auf „Eingabe-Puffer“ verzichtet, da es nicht möglich ist, dass eine Zelle in ihren „Ausgabe-Puffer“ schreibt, während eine andere von genau diesem liest.

Eine besondere Änderung kam den Rand-Puffern zugute. Man sieht, dass wenn man die dort liegenden Sandkörner nach jedem Wurf sofort in den 16-Bit-Speicher verschiebt, pro Rand-Zelle 1 Bit ungenutzt gelassen werden. Folgender Satz gibt genauere Auskunft.

Satz 4.1.2 (Rand-Puffer voll laufen lassen)**.** *Sei K ein Quadrat der Seitenlänge 4 und $c_1 \in \{0,1,2,3\}^R$ mit $c_1 = 3|K$. Dann gilt:*

$$\forall c_2 \in \{0,1,2,3\}^K : \sum_{i \in R} c_2(i) \leq 6 \Rightarrow l(c_1 + c_2)|_{R \setminus K} = 0 \text{ und}$$

$$\exists c_2 \in \{0,1,2,3\}^K : \sum_{i \in R} c_2(i) = 7, l(c_1 + c_2)|_{R \setminus K} \neq 0.$$

Beweis. Der Beweis des ersten Teils wurde maschinell durchgeführt, indem alle möglichen Kombinationen ausprobiert wurden. Es genügt, die maximale Konfiguration zu untersuchen, also diejenige, in der alle Zellen im Zustand 3 sind. Der Code befindet sich im Test-Ordner zu dieser Version.

Für den zweiten Teil genügt ein Beispiel, wie in Abbildung 4.4 zu sehen ist. Wir werfen sechs mal auf Zelle 1 und danach ein mal auf Zelle 0. Danach hat sich eine Rand-Zelle des oberen Randes um insgesamt 4 erhöht. □

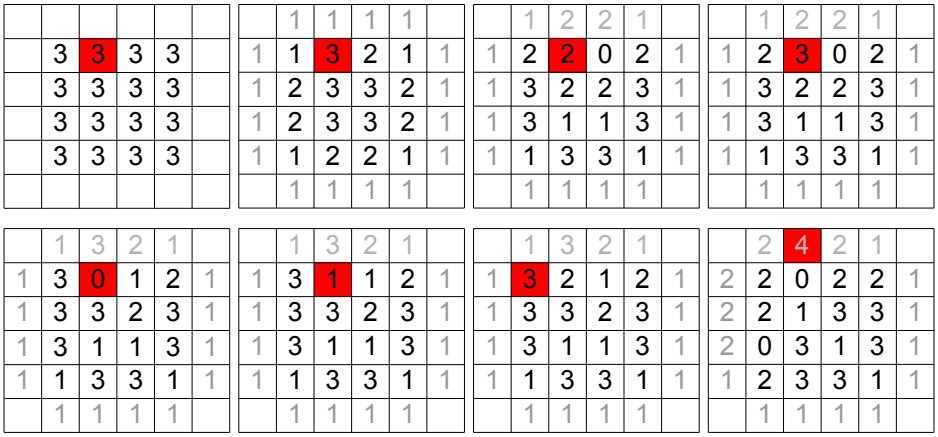

Abbildung 4.4: Ein Beispiel für die zweite Aussage des Satzes; zeilenweise zu lesen.

Aus diesem Satz ergibt sich, dass man erst nach maximal 6 Würfen die Sandkörner der Randzellen in den größeren Puffer verlagern muss. So kann man sich viele Schreibe-Vorgänge sparen. Im Prinzip ist diese Technik auch für die parallele Variante interessant. Allerdings wurde sie hier nicht angewandt, denn obwohl man so sogar viele Zugriffe auf Mutexe sparen kann, wird die Parallelität durch das Warten auf die Ausgabe-Puffer stark eingeschränkt, was zu einer schlechteren Laufzeit führt.

4.2 Hashing-Algorithmus

4.2.1 Motivation und Problemstellung

Mit Hilfe von Hash-Funktionen kann man in vielen Situationen Laufzeit sparen. Hier war die Idee, die Ergebnisse zu speichern, die durch das Werfen eines Sandkorns entstehen. Somit kann man mehrere feuernde Zellen mit einem einzigen Nachschlagen berechnen. Die Wahrscheinlichkeit, dass es oft mehrere feuernde Zellen gibt, ist in der kritischen Phase bei Zellen mit durchschnittlich 2,43 Sandkörnern [Wor11] hoch.

Es bleibt dabei die Frage offen, wie groß man diese Hash-Funktion wählen kann, ohne an Laufzeit zu verlieren. Eine zu große Hash-Funktion kann zu starken Laufzeit-

Einbußen führen, da das das Auslagern von Daten und Code in immer langsamere Speicher nach sich zieht.

Es soll nun eine Funktion

$$\{\text{16er-Zell-Kombinationen}\} \times \{\text{Eingabe-Bits}\} \to \{\text{Ausgabe-Konfigurationen}\}$$

berechnet werden. Für das Hashing soll also ein Integer-Array genutzt werden, das 2^{20} Einträge hat.

Bei einer Größe von 16 Zellen klingt es zunächst unrealistisch, alle Kombinationen zu speichern; bei 4 Zuständen pro Zelle sind es zunächst $4^{16} = 2^{32}$ Stück. Pro 16-er-Zelle gibt es nun 12 Möglichkeiten, ein Sandkorn von einer anderen aus einzuwerfen. Jedes Ergebnis fasst außerdem 8 Byte. Insgesamt käme man auf einen Speicherverbrauch von mehr als $2^{32+3+3} = 2^{38} = 2^8 GiB$.

Diese Zahl könnte man noch um Faktor 8 senken, indem man die Spiegelungen und Drehungen vernachlässigt. Der Faktor wäre 8, weil man so zu jeder gespeicherten Kombination 7 andere hätte, die man berechnen könnte. Dies folgt daraus, dass das Viereck genau durch Gruppenoperation mit der achtelementigen Diedergruppe D_4 gespiegelt und gedreht wird.

Einsparungen an den Kombinationen selbst werden schwierig. Man könnte sich leicht überlegen, dass man ein paar Kombinationen, die zu wenigen feuernden Zellen führen, gar nicht speichern will. Wie kann man aber systematisch Leerräume auslassen, ohne die Hashfunktion stark zu verlangsamen? Wäre die Hash-Map als binärer Baum organisiert, so würde eine Suche im Baum logarithmisch sein. Das wäre in unserem Fall viel zu aufwendig.

Alles in allem wird also eine Hashfunktion gesucht, die

- sehr schnellen Zugriff gewährleistet und
- dabei nicht übermäßig viel Speicherplatz belegt.

4.2.2 Hashing der Zustände im Wert 3

Diese Lösung beruht auf der Kommutativität aus 2.2.1, Fall 1. Diesen Satz kann man ausnutzen, indem man nur bestimmte Zell-Masken im Hash-Array speichert und alle anderen verändert, bis man eine in der Hash-Funktion vorhandene erhält. Da die Hash-Funktion besonders viele Ereignisse vorberechnen sollte, in denen eine Zelle feuert, ist es sinnvoll, vor allem die Zellen im Zustand 3 nicht vor dem Hashing zu reduzieren.

Im Folgenden ist die Funktion abgebildet, die zu einer 16er-Zelle von einer bestimmten Position aus ein Sandkorn wirft.

```
1 inline void SmallCellFramed::throwGrainWithoutStore(unsigned int side)
2 {
3   // contains a 1 for each cell containing a 3
4   const cellType bits_only3 = cellMask & (cellMask >> 1) & BITS_0_MOD_2;
```

```
5
6   const cellType oldBits = cellMask & ~BORDER_MASK;
7   const int hashvalue_20bit = ((side << 8) | (bits_only3 >> 10) | (
        bits_only3 >> 33)) & 0xFFFFF;
8   const cellType hashedValue = HeaderCache::hashFunction(hashvalue_20bit);
9
10  if(hashedValue == HashFunction::NOT_HASHED) {
11    cellMask = bits_only3 | (bits_only3<<1);
12    throwOnFieldXlated(_throwIndexToXlated[side]);
13    HeaderCache::hashFunction.insert(hashvalue_20bit, cellMask);
14  } else {
15    cellMask = hashedValue;
16  }
17
18  operator+=(oldBits ^ (bits_only3 | (bits_only3<<1))); // add 1s and 2s
19
20  storeBordersInOutBuffers();
21 }
```

Man sieht, dass das Finden der Zellen im Zustand 3 recht einfach ist, und zwar mit 2 Und-Operationen und einer Bitshift-Operation. Die Berechnung des Keys für die Hashfunktion ist etwas schwieriger.

Das Ziel beim Entwurf des Formats für den Key war, diesen so kurz wie möglich zu wählen, um das Hash-Array klein zu halten und so häufig genutzte Einträge im CPU-Cache zu behalten. Der Aufbau eines Keys ist im folgenden Diagramm gegeben: Dieses

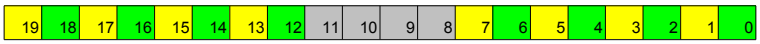

Abbildung 4.5: Die Zusammensetzung des Hash-Keys. Gelbe und grüne Zellen repräsentieren die eigentlichen Zell-Werte. Der Zwischenraum (grau) wird für die Seite genutzt.

Format ist praktisch, da in der Variablen, die die 3er-Bits der Zelle markiert, diese markierenden Bits jeweils ein Bit Zwischenraum auslassen. Es ist auch bezüglich der Bit-Anzahl minimal. Man muss hier aber anmerken, dass die Nummern der feuernden Zellen sowie die Gesamtzahl hier nicht gespeichert wurde. Dazu später mehr.

In Zeile 8 wird der Wert der Hash-Funktion abgefragt. Falls dieser noch nicht berechnet wurde, wird dies im Folgenden getan und in das Hash-Array eingetragen. Diese Fallunterscheidung kann man auch durch eine Vorberechnung des Arrays unterlassen, um so Laufzeit in der if-Bedingung zu sparen.

4.2.3 Detaillierte Ausgabe der Hash-Funktion

Auf eine Implementierung der detaillierten Ausgabe der Hash-Funktion wurde aus Zeitgründen verzichtet. Tatsächlich kann die Hash-Funktion ohne Mehraufwand an Bits eine

detaillierte Ausgabe der Anzahl und sogar der Positionen speichern. Der folgende Satz beweist das.

Satz 4.2.1. *Für die Anzahl benötigter Bits $b \in \mathbb{N}_0$ zum Speichern der Informationen, welche Zelle wie oft feuert, sowie der Gesamtzahl (die nicht aus den einzelnen Werten durch Addition gewonnen werden soll) gilt:*

$$b = 8.$$

Beweis. Sei $K \subseteq R$ das Quadrat und $c \in Q^R$ mit $c = 0|_{R \setminus K}$. Wir wollen zunächst die Positionen der feuernden Zellen speichern. Da die Startzellen der Lawinen in der Hash-Funktion nur am Rand liegen, gilt nach Satz 2.5.3 $\forall z \in K : l(c, z) \leq 1$. Wir machen eine Fallunterscheidung bezüglich der Position der Zelle $z \in K$.

Fall 4.2.1.1. $d(z, R \setminus K) = 1$. Es gilt, dass eine Nachbarzelle zu z im Rand-Puffer genau dann im Zustand 0 ist, wenn z nicht gefeuert hat. Also gibt es eine Äquivalenz zwischen dem Feuern von z und dem Wert der Nachbarzelle im Rand-Puffer. Daher werden keine zusätzlichen Bits benötigt, um diesen Fall abzudecken.

Fall 4.2.1.2. $d(z, R \setminus K) = 2$. Es gibt genau 4 solcher Zellen. Damit sind zur Speicherung von 2^4 Möglichkeiten 4 Bits ausreichend.

Nun zur Gesamtzahl. Nach Korollar 2.5.1 ist diese $f := \frac{1}{6}(4(4+1)(4+2) - 2(2+1)(2+2)) = \frac{1}{6}(4 \cdot 5 \cdot 6 - 2 \cdot 3 \cdot 4) = 20 - 4 = 16 = 2^4$. Also reichen insgesamt $b = 4 + 4 = 8$ Zellen aus. □

Korollar 4.2.1. *Die 64 Bit große Zell-Maske ist ausreichend, um diese Informationen zusätzlich zu speichern.*

Beweis. Zur Speicherung des Resultats nach dem Wurf werden auf jeden Fall 16 Zellen belegt. Die restlichen 16 Zellen, die die Rand-Puffer bilden, benötigen aber für herausgefallene Bits nur 1 von 2 Bits; das ist gerade ein Resultat von Satz 2.4.6. Damit sind noch $64 - (16 \cdot 2 + 16 \cdot 1) = 16 > 8$ Bit frei. □

Es ist klar, dass die Anzahl ohne größeren Aufwand extrahiert werden kann. Die Weiterverarbeitung der genauen Positionen gestaltet sich etwas schwieriger, da man die oben erwähnt Fallunterscheidung hat und anschließend noch die Offsets der eigentlichen 16er-Zelle mit einberechnen muss.

4.2.4 Varianten

4.2.4.1 Hash-Funktion nur bedingt anwenden

Wenn der Wert der Zelle, auf die man wirft, nicht 3 ist, ist es eher nicht sinnvoll, den Wert mit Hilfe der Hash-Funktion zu berechnen. Die Anwendung dieser Erkenntnis bewirkte Laufzeit-Verbesserungen von Faktor 2.0 in der kritischen Phase.

4.2.4.2 Zuerst Sandkörner werfen, die keine Lawinen erzeugen

Die Hash-Funktion ist dann besonders effizient, wenn sie große Lawinen berechnet. Es liegt wegen Variante 4.2.4.1 nahe, zuerst die Sandkörner zu werfen, die keine Lawinen erzeugen, damit dann bei den restlichen Lawinen eine möglichst große Anzahl von Zellen im Zustand 3 ist.

Zunächst ist diese Variante jedoch langsamer, was durch den höheren Aufwand an Überprüfungen, ob eine Zelle eine Lawine auslöst, bedingt ist. Tatsächlich bringt es jedoch einen erfreulichen Laufzeit-Unterschied, wenn man in der Implementierung des Algorithmus „Paralleler Stack“ den Stack durch eine Warteschlange ersetzt. Dazu wurde die Klasse „FastRingListStack“ verwendet. Diese Modifikation hat den Vorteil, dass die meisten 16-er Zellen, wenn sie an der Reihe sind, längere Zeit nicht beachtet wurden und so eher mehr Sandkörnern in ihren Eingabe-Puffern gesammelt haben. Daher können so für die meisten Lawinen mehr Zellen in den Zustand 3 gebracht werden.

Dieser Laufzeitvorteil steht aufgrund der Warteschlange im Widerspruch zu einer guten Cache-Auslastung. Damit zeigt der Algorithmus, dass es sich unter Umständen lohnen kann, Cache-Ineffizienz in Kauf zu nehmen, um andere Ziele zu erreichen.

Die Lösung wurde als „cache2“ eingereicht.

4.3 Repository-Algorithmus

4.3.1 Motivation

Die Idee dieses Algorithmus ist, die Lokalität von Lawinen auszunutzen, um Parallelität zwischen verschiedenen Lawinen zu erreichen. Die Größe einer Lawine kann stark variieren. Man hofft sozusagen, dass verschiedene Lawinen nicht aufeinander stoßen. Falls dies doch der Fall ist, müssen einige Lawinen abgebrochen und überschrieben werden. Abbildung 4.6 skizziert das Prinzip.

Der Algorithmus funktioniert am besten, wenn die Lawinen selten aufeinander treffen. Folgende Tabelle zeigt die durchschnittliche Lawinengröße $\lambda(a)$ in der kritischen Phase bei einem Gitter der Seitenlänge a. Die Ergebnisse wurden experimentell ermittelt.

a	10	100	1000
$\lambda(a)$	5,01	365,48	ca. 34150

Es scheint eine Lawine ca. $3 - 5\%$ des Gitters zu belegen. Das spricht für eine Parallelisierung mit zumindest 2 Threads.

Als Voraussetzung für den Algorithmus nehmen wir an

1. Eine Zelle wird zu jeder Zeit nur von maximal einem Thread bearbeitet.
2. Es arbeiten nie 2 Threads an der selben Lawine.

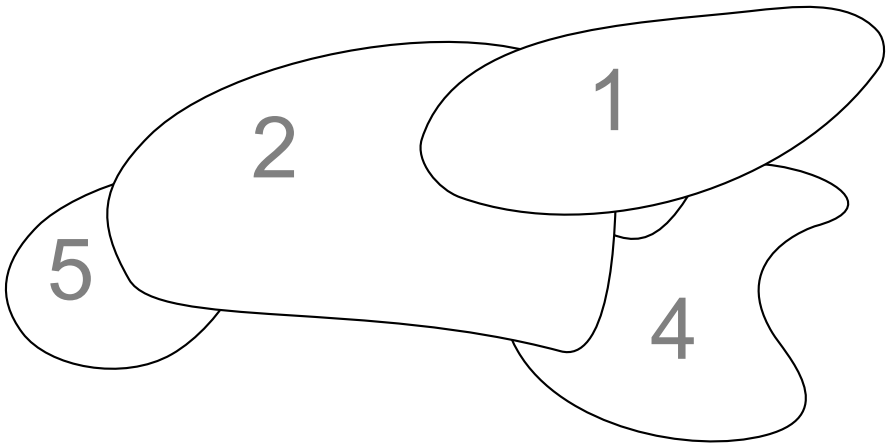

Abbildung 4.6: Im Falle von Überschneidungen muss der Thread, dessen Sandkorn-Nummer zeitlich zuerst geworfen wird, die Lawine des anderen Threads überschreiben.

4.3.2 Struktur einer Zelle

Wenn wir von einem *geworfenen Sandkorn* zu einer Lawine reden, so sei damit immer das eindeutige Sandkorn gemeint, welches die Lawine ausgelöst hat. Die *Nummer* dieses Sandkorns bezieht sich auf die vorher festgelegte Menge von Sandkörnern, die in eindeutiger Reihenfolge auf das Gitter geworfen werden.

Für einen sequentiellen Algorithmus braucht man mindestens für jede Zelle deren aktuellen Zustand. Welche Attribute muss eine Zelle haben, um dem hier beschriebenen Algorithmus zu genügen?

Um die Reihenfolge der Lawinen einhalten zu können, müssen Lawinen zwei Operationen anbieten:

1. Information über die Nummer des geworfenen Sandkorns und
2. Zurücksetzen der Lawine.

Für die erste Eigenschaft muss jede Zelle die Nummer des Sandkorns speichern, dessen Wurf seinen aktuellen Zustand bewirkt hat. Die zweite Eigenschaft erfordert, dass man Zustand und geworfenes Sandkorn der vorletzten Lawine speichert. Insgesamt muss eine solche Zelle also von allen vorhergehenden Lawinen, in denen sie enthalten war, je folgende Attribute speichern:

- Zustand (nach Bearbeitung) und
- Nummer des geworfenen Sandkorns.

Das würde aber bedeuten, dass unendlich viel Speicherplatz notwendig wäre. Die Eigenschaft der „sicheren Konfiguration“ erlaubt uns, die Daten nur für endlich viele Lawinen zu speichern.

Definition 4.3.1. Eine Lawine heißt dann sicher, wenn die Nummer ihres geworfenen Sandkorns echt kleiner als die aller Lawinen ist, welche gerade von einem Thread bearbeitet werden. Eine Konfiguration $c \in Q^R$ heißt *sicher*, wenn sie das Resultat von nur sicheren, fertig entwickelten Lawinen ist.

Die Konfiguration, die zu Beginn einer Simulation vorliegt, kann man als sicher ansehen. Danach gibt es immer mindestens einen Thread, dessen Lawine durch das niedrigste geworfene Sandkorn ausgelöst wurde. Da ein solcher Thread nie von anderen Threads unterbrochen wird, folgt:

1. Es arbeitet immer ein Thread an der nächsten sicheren Konfiguration.
2. Der Algorithmus terminiert.

Obwohl man nun eine Obergrenze an unsicheren Konfigurationen, die eine Zelle speichern soll, festlegen kann, ist dies noch nicht zufriedenstellend. Offensichtlich benötigt man deutlich mehr Arbeitsspeicher, was eine schlechte Cache-Auslastung zur Folge hätte. Abgesehen davon stelle man sich folgendes Szenario vor:

- Thread 0 hat Zelle $i \in R$ nacheinander mit Lawinen der Nummern $[0, ..., 499], [501, ..., 1000]$ beschrieben.
- Thread 1 versucht anschließend, das Sandkorn mit Nummer 500 zu werfen.

In diesem Fall müsste Thread 1 erst eine Suche durchführen, welche Lawinen zu löschen wären und welche nicht. Um so etwas zu vermeiden, fordern wir:

Definition 4.3.2. Eine Zelle kann maximal Daten für eine unsichere Konfiguration speichern.

Es muss noch geklärt werden, wie eine Lawine zurück gesetzt werden kann. Eine erste Idee ist es, ein kleinstes Rechteck um alle Zellen zu entwickeln. Um dann die Zellen zurück zu setzen, muss man alle Zellen in diesem Rechteck überprüfen. Dieser Ansatz hat aber Nachteile. Zunächst sind Lawinen im Allgemeinen keine Rechtecke, und so müssen oft viel mehr Zellen als nötig überprüft werden. Außerdem muss das Zurücksetzen vollständig unter wechselseitigem Ausschluss stattfinden, was zu starken Laufzeitdifferenzen führen kann und eine Absicherung gegen Deadlocks erfordert.

Eine andere Möglichkeit wäre es, die Nummer jedes geworfenen Sandkorns, das eine Lawine ausgelöst hat, zu vermerken. Threads, die dann auf Zellen einer solchen Lawine stoßen, könnten diese dann Zelle für Zelle zurück setzen. Dabei besteht aber das Problem, dass ein Thread, der das gleiche Sandkorn zweimal versucht, nicht seine aktuelle von der fehlgeschlagenen Lawine unterscheiden kann.

Um den Ansatz, fehlgeschlagene Lawinen zu speichern, realisieren zu können, bedarf es also noch einer weiteren Kennzeichnung. Diese wurden *commit-IDs* genannt; und so wird auch der Name des Algorithmus klar: Die Threads stellen selbst Commits in Form

von Lawinen zusammen, versuchen diese „einzuchecken“, müssen aber gelegentlich auch Commits zurücksetzen.

Für die Klasse SharedCell ergeben sich unter anderem folgende Attribute:

- Start-Sandkorn-Nummer und Zustand (nach Bearbeitung) von
  - der letzten sicheren Lawine
  - einer unsicheren Lawine
- Aktuelle unsichere Commit-ID.

4.3.3 Struktur des Zellgitters

Das Zellgitter enthält folgende Informationen, auf die jeweils nur mit entsprechendem Mutex zugegriffen wird:

- Einen Zähler, der die Nummer des letzten geworfenen Sandkorns enthält, dessen Lawine sicher ist.
- Eine Relation „ Sandkorn-Nummer $\rightarrow \mathbb{N}$ “, die jeder fehlgeschlagenen Lawine zuordnet, wie viele Felder von dieser sich noch auf dem Brett befinden.
- Eine Prioritätswarteschlange, in die fehlgeschlagene Commit-IDs eingetragen werden.

4.3.4 Konfliktfälle

Trifft ein Thread auf eine unsichere Lawine (d.h. ein Thread bearbeitet eine Zelle, die noch unsicher ist), so gibt es 3 Fälle:

1. Der Thread hat eine kleinere Sandkorn-Nummer als die Lawine. Dann war die Lawine fehlerhaft und wird überschrieben.
2. Der Thread hat eine größere Sandkorn-Nummer als die Lawine. Dann hat der Thread selbst eine fehlerhafte Lawine berechnet. Da nur eine unsichere Lawine pro Zelle erlaubt ist, kann die Bearbeitung nicht fortschreiten und wird abgebrochen.
3. Der Thread stößt auf seine eigene Lawine. Das ist nicht kritisch.

4.3.5 Wechselseitiger Ausschluss

Eine Möglichkeit zum wechselseitigen Ausschluss in jeder Zelle besteht darin, auf Zellen nur via Mutex zuzugreifen.

Eine weitere Idee war, um jede Lawine Rechtecke zu zeichnen, welche die Lawine vollständig enthalten. Zu jeder Lawine wird immer ein Rechteck mitgeführt, welches

diese und einen Platzhalter enthält, der mindestens eine Zelle breit ist. Wir nennen diese Rechtecke einfach nur *Rechtecke*.

Einen *kritischen Bereich* nennen wir einen Bereich, der in der Schnittmenge zweier Rechtecke liegt.

Es bleibt sicherzustellen, dass eine Zelle zu jeder Zeit nur von maximal einem Thread bearbeitet wird. Entweder liegt die Zelle nicht in einem *kritischen Bereich*. Dann ist es klar. Falls doch, so erfolgt sein Zugriff von allen Threads nur per Mutex, sobald alle Threads über die Bereichsvergrößerung informiert sind. Dies ist dann der Fall, wenn alle anderen Threads ihre Arbeit an ihrer jeweils aktuellen Zelle beendet haben, denn bei der nächsten Zelle prüfen sie selbst, ob sich ihre Zelle in einem kritischen Bereich befindet.

4.3.6 Algorithmus

Der Algorithmus ähnelt dem des Algorithmus „Parallelen Stack“. Die Bearbeitung der Zellen erfolgt nach der bereits erwähnten Fallunterscheidung.

Kapitel 5

Laufzeitvergleiche

Sei ein Algorithmus fest gewählt. Es bezeichne $t_i, i \in \mathbb{N}$ die durchschnittliche gemessene Laufzeit für 1000 Würfe zu diesem Algorithmus für t Threads in der kritischen Phase. Dabei sei t in Sekunden angegeben.

5.1 Der Testrechner

Bei dem Testrechner handelt es sich um einen PC. Die CPU ist vom Typ „Intel Core i7 2600“. Sie hat 4 Kerne und bietet Hyper-Threading; man spricht dann von 8 virtuellen Kernen. Jeder der 4 Kerne hat eine Leistung von 3,4 GHz. Der L3-Cache hat 8MB.

Alle Tests wurden mit dem Betriebssystem „Gentoo Linux “ in einer aktuellen Version durchgeführt. Gentoo ist dafür bekannt, dass ein Großteil der Software auf dem eigentlichen Rechner kompiliert wird.

5.2 Die parallelen Algorithmen im Detail

5.2.1 Repository-Algorithmus

Der Repository-Algorithmus liefert die schlechtesten Resultate. Mit einem Thread ist er noch in etwa so schnell wie der sequentielle Stack-Algorithmus, dem er in der Implementierung ähnlich ist. Schon mit 2 Threads ist er erheblich langsamer, und mit 4 Threads ist die Berechnung so langsam, dass die Laufzeit nicht mehr sinnvoll in ein Diagramm mit den anderen Threads eingetragen werden kann.

5.2.2 SmallCell-Stack-Algorithmen

Es gab 3 Varianten dieser Algorithmen. Zum einen den Algorithmus ohne Hash-Funktionen. Dieser ist auch erheblich besser als die anderen beiden Varianten, die eine Hash-Funktion nutzen. Offensichtlich ist der Speicherzugriff auf die Hash-Tabelle und die Vorberechnung für den Hash-Key der Grund dafür.

Die zweite Cache-Version unterscheidet sich von der ersten lediglich dadurch, dass sie versucht, wenige möglichst große Lawinen zu entwickeln, wobei die Reihenfolge der

zu werfenden Sandkörner eine Rolle spielt. Tatsächlich hat diese Version eine deutlich bessere Laufzeit. Das zeigt, dass es sinnvoll sein kann, einen Algorithmus an eine Hash-Funktion anzupassen, um sie besser nutzen zu können.

5.3 Sequentielle Algorithmen im Vergleich

Die Laufzeiten der sequentiellen Algorithmen sind in folgender Tabelle aufgelistet.

	t_1
Algorithmus von Walter und Worsch	6,45
Sequentieller Stack-Algorithmus	1,17
Sequentieller SmallCell-Stack-Algorithmus	1,65

Der neue SmallCell-Stack-Algorithmus ist dabei etwas langsamer als der vorher bekannte Stack-Algorithmus. Einer der Hauptgründe für dieses Resultat ist, dass der SmallCell-Stack-Algorithmus durch das Überführen der Bits in die Ausgabe-Puffer etwas Laufzeit verliert.

5.4 Parallele Algorithmen im Vergleich

	t_1	t_8
Paralleler Stack-Algorithmus	4,17	0,65
Vier-Zustands-Algorithmus	4,45	0,80
Paralleler SmallCell-Stack-Algorithmus	4,41	0,68
Cache-Algorithmus (Version 1)	6,19	1,41
Cache-Algorithmus (Version 2)	5,85	0,92
Repository-Algorithmus	5,95	128,61 [1]

5.5 Java gegen C++

5.5.1 Die Algorithmen

Folgende Tabellen stellen für die vier portierten Algorithmen die durchschnittlichen Laufzeiten der jeweiligen Java- und C++-Versionen gegenüber.

	t_1 (Java)	t_1 (C++)
Algorithmus von Walter und Worsch	5,09	6,45
Sequentieller Stack-Algorithmus	0,98	1,17

[1] Dies ist der Wert von t_4.

	t_8 (Java)	t_8 (C++)
Paralleler Stack-Algorithmus	0,86	0,65
Vier-Zustands-Algorithmus	1,19	0,85

Die sequentiellen Lösungen sind mit Java etwas schneller. Das könnte daran liegen, dass der Fokus der Portierung auf die parallelen Algorithmen gelegt wurde. Diese sind hingegen mit C++ um ca. 30 Prozent schneller. Damit gewinnt C++ diesen Vergleich. Es zeigt sich hier, dass die Wahl der Programmiersprache tatsächlich sehr entscheidend für die Laufzeit eines Programms sein kann.

5.5.2 Die GUI

Bei der GUI kann man die Bilder pro Sekunde messen. Für die schnellsten Algorithmen wurden diese Werte verglichen. Auch hier zeigt C++ seine Vorzüge. Die Bildrate war im Schnitt 2-3 mal so hoch wie bei Java. Allerdings erhält man hier Schwankungen, die von der Anzahl der freien Threads abhängt. Außerdem können die Werte bei anderen Grafikkarten ganz anders sein.

Kapitel 6

Fazit

In dieser Bachelor-Arbeit konnte in verschiedenen Bereichen neue Kenntnis erworben werden.

Im Theorieteil wurden neue Formeln geliefert. Als Grundlage diente eine Formel über Kommutativität, die eine Berechnung über eine gewisse „Umleitung“ erlaubt. Dies konnte als Grundlage für weitere Formeln genutzt werden. Es wurde gezeigt, wie sich die Lawinenzahlen bei rechteckigen Mengen exakt verhalten. Eines der wichtigsten Resultate war die Tatsache, dass eine Lawine in einer leicht berechenbaren Anzahl von Schritten entwickelt werden kann, wobei in jedem Schritt jede Lawine nur ein mal feuert. Dies konnte genutzt werden, um einen Algorithmus zu entwerfen, der jeden dieser Schritte perfekt parallelisieren kann.

Die Portierung des Java-Codes zu C++ konnte erfolgreich durchgeführt werden. Es wurden die beiden schnellsten Algorithmen portiert und Laufzeitverbesserungen festgestellt.

Es wurden außerdem 2 weitere Algorithmen neu entworfen und implementiert. Weder der Repository-Algorithmus, bei dem mehrere Lawinen gleichzeitig entwickelt werden, noch der Hashing-Algorithmus, der viele Ergebnisse schon vorher im Cache speichert, konnten sich dabei profilieren. Die Laufzeitergebnisse fielen eher schlechter aus. Es konnten keine schnelleren Algorithmen entworfen werden.

Insgesamt konnte mit typischen algorithmischen Methoden also keine Laufzeit mehr gewonnen werden. Der Algorithmus aus dem Theorieteil zeigt aber, dass es sich durchaus lohnen kann, die Eigenschaften des Sandhaufen-Zellularautomaten genau zu studieren, um neue parallele Algorithmen zu entwickeln.

Literaturverzeichnis

[Ada08] Andrew Adamatzky. *Automata-2008: Theory and Applications of Cellular Automata*. Luniver Press, 2008.

[Bod09] Felix Bodarenko. Eine parallele Implementierung des Sandhaufenmodells für Systeme mit gemeinsamem Speicher, September 2009.

[BTW87] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality - An explanation of 1/f noise. *Physical Review Letters*, 59:381–384, July 1987.

[Fre10] Sebastian Frehmel. Parallelisierung des Sandhaufen-Zellularautomaten für Systeme mit gemeinsamem Speicher in Java, February 2010.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition*. Addison-Wesley Professional, May 2005.

[SA05] Herb Sutter and Andrei Alexandrescu. *C++ coding standards 101 rules, guidelines, and best practices*. Boston: Addison-Wesley, 2005.

[Spy02] Frank Spychalski. Versuch einer effizienten parallelen Simulation des Sandhaufen-Modells, 2002.

[Str07] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, sixteenth edition, March 2007.

[Wor11] Thomas Worsch. Algorithmen in Zellularautomaten. Skript zur Vorlesung, 2011.

[WW04] Richard Walter and Thomas Worsch. Efficient Simulation of CA with Few Activities. In *ACRI '04: Proceedings of the 6th international conference on Cellular Automata for Reseach and Industry*, 2004.

Erklärung der Urheberschaft

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Desweiteren versichere ich, dass ich die Satzung der Universität Karlsruhe (TH) zur Sicherung guter wissenschaftlicher Praxis in der zur Zeit gültigen Fassung beachtet habe.