Praktikumsgruppe 5 SS95: Atilla Ünlü, Frank Pilhofer, Kay Römer und Thomas Holubar

Der Lattice Gas-Automat

Beschreibung

Der Automat (GA) beschreibt auf diskrete Weise das Strömungsverhalten von nichtkomprimierbaren Gasen oder Flüssigkeiten im 2-dimensionalen Raum.
Zunächst möchte ich auf die Datenrepräsentation eingehen, damit dem Leser das Verhalten des GA deutlicher wird. Die Oberfläche des simulierten Stoffes (also die Matrix mit der die Simulation läuft) wird in Sechsecke eingeteilt (Abb.1). Jede dieser Zellen (Sechsecke) hat nun genau 6 Nachbarzellen (ausgenommen die Ränder, doch dazu später mehr). Es existieren also von jedem "Zellkern" aus 6 Pfade zu Nachbarzellen, auf denen sich entweder genau ein Teilchen befinden kann - oder eben nicht. Somit läßt sich jede Zelle mit 6 Bits kodieren. Ein gesetztes Bit gibt die Existenz eines Teilchens an, ein ungesetztes die Abwesenheit.
Wie aber läuft nun die Simulation innerhalb einer Zelle ab? Hierzu muß man wissen, daß jedem Pfad in einer Zelle eine Richtung zugeordnet ist. Somit kann eine Zelle alle Teilchen ermitteln, die sich auf sie zubewegen (Abb.2). Indem man dieses Verfahren auf alle Zellen anwendet, hat man bereits ein rudimentäres Bewegungsverhalten implementiert. Was nun noch fehlt ist das Erkennen von Kollisionen, d.h. bei der Bewegung kann es sein, daß 2 oder mehr Teilchen zusammenstossen. Wie dies im einzelnen behandelt wird kann man im Anhang (oder Sourcecode) nachlesen.
So weit, so gut. Bis jetzt habe ich einen (recht passablen) Automaten zur Simulation von Teilchenbewegungen beschrieben. Daraus ein Strömungsverhalten zu erzeugen ist allerdings sehr simpel: Angenommen, es soll eine Strömung von rechts nach links simuliert werden: Alles was zu tun ist, ist das Bewegungsverhalten für alle Zellen zu bestimmen und nach jedem Schritt am rechten Rand Teilchen mit einer Linksbewegung hinzuzufügen. Außerdem ist noch zu berücksichtigen, daß am oberen und unteren Rand keine Teilchen verlorengehen (alle "ausbrechenden" Teilchen kann man z.B. am Rand einfach reflektieren).
Das war's im Prinzip schon. Um die ganze Sache aber noch interessanter zu gestalten, kann man in die Strömung noch Barrieren einbauen. Trifft ein Teilchen nun auf eine Barriere, so wird es reflektiert. Auf diese Weise können Wirbel und andere interessante Strömungen entstehen.

Optimierungen: Um die Simulation möglichst zügig ausführen zu können, habe ich einige Optimierungen vorgenommen.
- Jede Zelle wird in einem Byte kodiert. Nachteil: 33% mehr Platzbedarf für die Matrix, allerdings vernachlässigbar, da der Speicher auf den Transputern locker ausreicht um 4000x3000 Matrizen zu berechnen, was zeitlich gar nicht mehr sinnvoll ist (einige Minuten pro Berechnungsschritt).
- Barrieren werden nicht schon bei der Simulation einer Zelle berücksichtigt, sondern die fehlerhaft (da Barrieren n. ber.) berechnete Matrix nach jedem Rechenschritt korrigiert.

Parallelisierung: Im Prinzip läßt sich das recht einfach bewerkstelligen (der Teufel steckt im Detail): jeder Slave erhält einen Teil der Matrix (Zeilenanzahl/Slaveanzahl) Zeilen (aufgerundet). Außerdem sucht er sich noch die Anteile der Barrieren heraus, die für seinen Abschnitt relevant sind - fertig, d.h. eine kleine Einschränkung gibt es noch: da ein Transputer nicht mehr das ganze Feld kennt, muß nach jedem Berechnungsschritt ein Informationsaustausch zwischen zwei benachbarten Slaves stattfinden (s. Anhang).

Für die Kommunikation habe ich eine ähnliche Struktur gewählt, wie sie sich bei Aufgabe 2 (Life) bewährt hat: Eine Kombination aus Ring und Farm (Abb.3).
Hierbei gibt es folgende Arbeitsteilung zwischen Ring und Farm: der Ring wird zur Initialisierung der Slaves genutzt, d.h. auf ihm werden die Kommandos verteilt, und die Updates (s.u.) zwischen den einzelnen Rechenschritten laufen über ihn; die Farm dient dazu, die Ausgabedaten von den Slaves an den Master zu schicken.

Eine typische Kommunikation sieht also wie folgt aus:
aus der Perspektive des Masters:
- zählen der Slaves (Ring)
- versenden der Matrixinformationen (Ring)
- versenden der Barrieren (Ring)
- starten der Berechnung (Ring)
- LOOP: warten auf Ausgabedaten (Farm) ->LOOP
- beenden der Slaves (Ring)

aus der Perspektive der Slaves
:- LOOP1: warten auf Kommandos (Ring) ->LOOP1
(= zählen, lesen d. Informationen+Initialisierung, Barrieren empfangen, Start d. Berechnung)
- LOOP2: [berechnen], versenden und empfangen der Updates (TeilRing)
- versenden der Ausgabedaten (Farm) ->LOOP2

Hierbei bedeutet TeilRing daß nur die Ringkanäle genutzt werden, die zwischen Slaves liegen, d.h. der Master bleibt in puncto Berechnung arbeitslos und kann sich ganz der Ausgabe widmen. Die Idee dabei ist, daß die X11-Ausgabe relativ lange dauert und der Master ohnehin nichts anderes zu tun hätte, als das Update zu kontrollieren. Also lag es nahe, die Slaves bei der Berechnung (und dazu gehört auch das Update) von ihm abzukoppeln. Mit anderen Worten: Master und Slaves arbeiten während der Berechnung völlig unabhängig voneinander und damit parallel: die Slaves berechnen die neue Matrix, der Master bereitet die Ausgabe auf und schickt sie an einen X-Server. Lediglich beim Versenden/Empfangen der Ausgabedaten ist eine (kurzfristige und idealerweise minimale) Synchronisation notwendig. Der Vorteil dieser Architektur liegt in der enormen Effizienz die damit erzielt werden kann (Tab.1). Ich möchte allerdings noch ein paar Worte zur Effizienz verlieren: vergleicht man in Tabelle 1 Spalte 2 mit Spalte 3 so fällt ein frappierender Unterschied auf. Dies rührt daher, daß bei den Tabelle 2 zugrundeliegenden Messungen jeder Berechnungsschritt ausgegeben wurde, im anderen Fall nur jeder zehnte. Damit gibt Spalte3 nur an, wieviel Zeit für die Ausgabe benötigt wurde, da aufgrund der Parallellisierung von Berechnung und Ausgabe nur der jeweils größere Zeitbetrag gemessen wird, während Spalte 2 die tatsächliche Rechenzeit angibt.
Zu den Tabellen noch einige Worte vorweg: Es war mir leider nicht möglich, mehr als 9 Transputer zu allokieren. Versuche mit längeren Pipes oder einer 4x4-Map schlugen fehl (es gelang zwar einmal, 13 Transputer zu booten (+ 3 Totgeburten), aber die Resultate waren doch recht merkwürdig. Deshalb beziehen sich alle Messungen auf eine 3x3-Map.
Allerdings war es auch hier nötig, die Transputer mehrfach neu zu booten. Alle Meßwerte geben deshalb Mittelwerte an. Die Abweichungen betrugen in allen Fällen weniger als 10 Prozent zwischen Minimum und Maximum.

Tabelle 1: Meßwerte für LATTICE.DAT (Zeit in Sekunden)

Slaveanzahl	Ausgabe alle 10 Berechnungsschritte	Ausgabe pro Berechnungsschritt
1	281,06	327,49
2	142,71	298,35
3	96,38	
5	58,86	
7	42,59	
9	33,68	312,44

Tabelle 2: Speedup

Slaveanzahl	Ausgabe alle 10 Berechnungsschritte	Ausgabe pro Berechnungsschritt
1	1,000	1,000
2	1,969	1,098
3	2,916	
5	4,775	
7	6,599	
9	8,345	1,048

Tabelle 3: Effizienz (%)

Slaveanzahl	Ausgabe alle 10 Berechnungsschritte	Ausgabe pro Berechnungsschritt
1	100	100
2	98	55
3	97	
5	96	
7	94	
9	93	12

Fazit: Ein Speedup kann nur erreicht werden, wenn die Berechnung länger dauert als die X11-Ausgabe. Darüberhinaus läßt sich nicht mehr viel tun. Die Effizienz liegt durchweg deutlich über 90%. Kleine Verbesserungen kann man allerdings dadurch erreichen, daß man die Ausgabe weniger aktiviert oder größere Matrizen berechnet. Der Kommunikationsaufwand für eine Ausgabe liegt etwa in der Größenordnung von 50% dessen, was ein Update benötigt.
Ein Beispiel: Wäre die zu berechnende Matrix in obigem Fall doppelt so groß, würde der Effizienz verlust um 50% reduziert (also z.B. statt 96% wie oben 98%). Würde man dagegen nur halb so oft ausgeben, wäre der Gewinn (deutlich) geringer:
5% bei einer Ausgabe alle 20 Schritte statt alle 10 Schritte; 50% bei einer Ausgabe alle 2 Schritte statt jeden Schritt. Allerdings jeweils unter der Voraussetzung, daß die Berechnung für die gemessene Zeit verantwortlich ist (s.o.: wenn es die Ausgabe ist, läßt sich keine Vorhersage über die Effizienzsteigerung machen - man müsste nachmessen).

ANHANG:

Kollisionen:
Es gibt zwei Grundkonfigurationen bei Kollisionen: zwei Teilchen stoßen frontal (also in einem 180° Winkel) zusammen oder drei Teilchen stoßen in "Mercedesformation" (also in je einem 120° Winkel) zusammen. In ersterem Fall werden die Flugbahnen der Teilchen je um 60° im oder gegen (aber bei beiden gleich) den Uhrzeigersinn gedreht (Abb. 4), in zweiterem wird das Bitmuster einfach invertiert (entspricht ebenfalls einer 60° Drehung).
Alle anderen Kollisionsformen ergeben sich prinzipiell aus diesen Regeln. Zu beachten ist hierbei, das das energetische Potential innerhalb der Matrix gleichbleibend ist.

Barrieren:
Da die Reflektionen an den Barrieren im Falle von Einfalls- = Ausgangswinkel die Tendenz hatten Muster zu bilden, habe ich die Reflektionen um eine (optionale) Zufallsstreuung erweitert. Das Ergebnis läßt sich sehen - und, was viel wichtiger ist, statistisch bleibt auch das energetische Potential gleich.

Update/Informationsaustausch:
Damit ein Slave alle Zellen seiner Matrix berechnen kann, benötigt er Informationen von seinen Nachbarn - jedenfalls dann, wenn die Matrix auf mindestens zwei Slaves verteilt ist. Der Grund dafür liegt darin, daß für die Berechnung einer Zeile (ich berechne Zeilenweise) sowohl die vorherige als auch die nachfolgende Zeile bekannt sein müssen (Ausnahme: Rand). Da ein Slave diese aber nicht berechnen kann, muß er sie sich von seinen Nachbarn holen, d.h. jeder Slave hat zu jedem seiner Nachbarn zwei Zeilen Überlappung, von denen er eine berechnet und seinem Partner übermittelt und die andere übermittelt bekommt. Das Übermitteln dieser Zeilen nenne ich Update.

Abbildungen:

Abb.1:

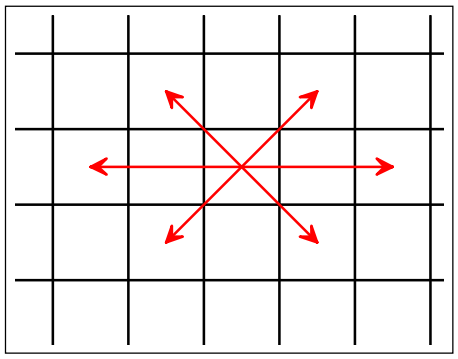
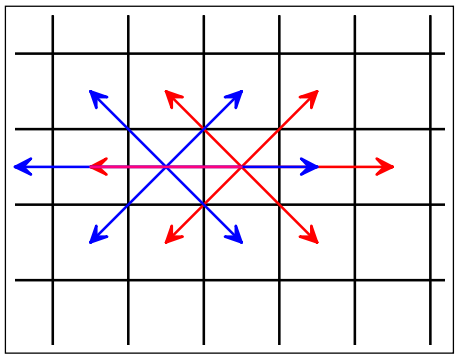
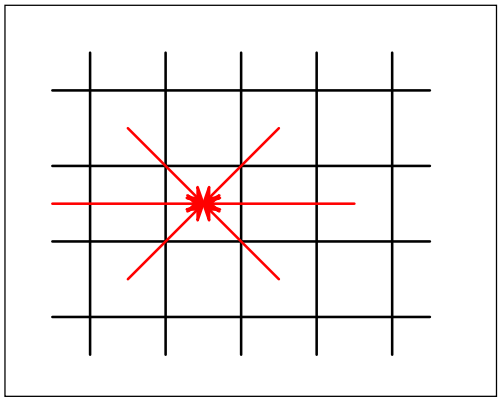


Abb.1b:



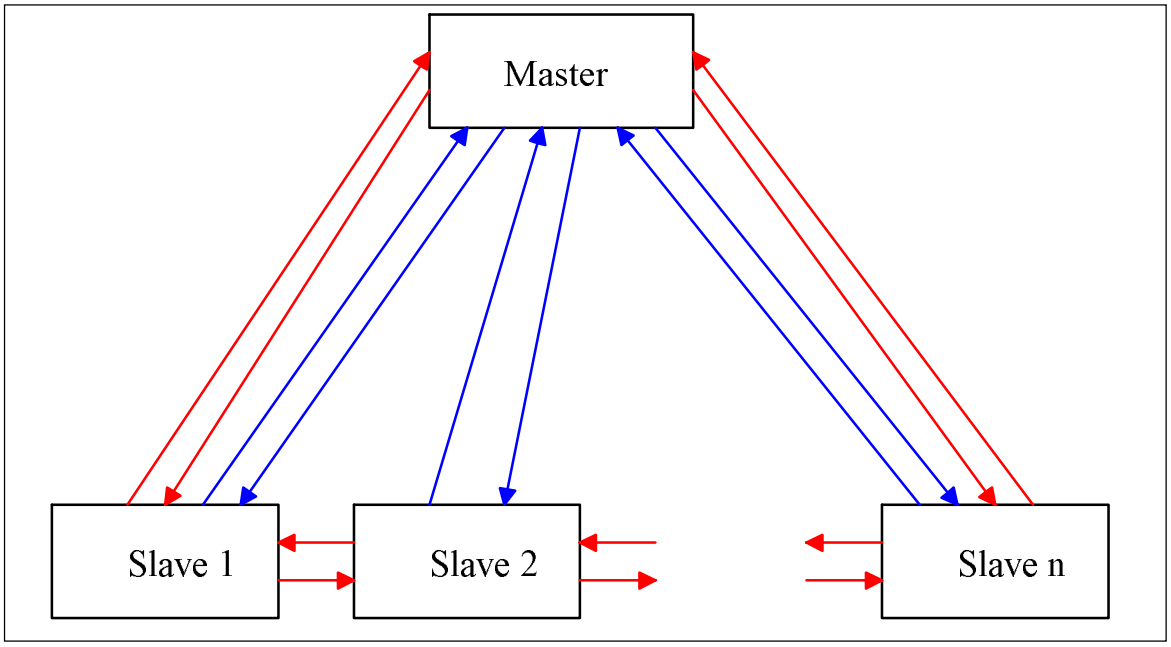
Die Einteilung der Matrix. Abb.1 stellt dar, wie die Pfade zwischen Zellen verlaufen. Um geometrisch korrekt zu sein und trotzdem eine relativ simple Struktur verwenden zu können, wird nur jedes zweite Byte im Speicher genutzt. Diesen Nachteil kann man aber sehr einfach dadurch kompensieren, daß man zwei Matrizen um ein Byte versetzt im Speicher anlegt: schon funktioniert die Sache wunderbar.

Abb.2:



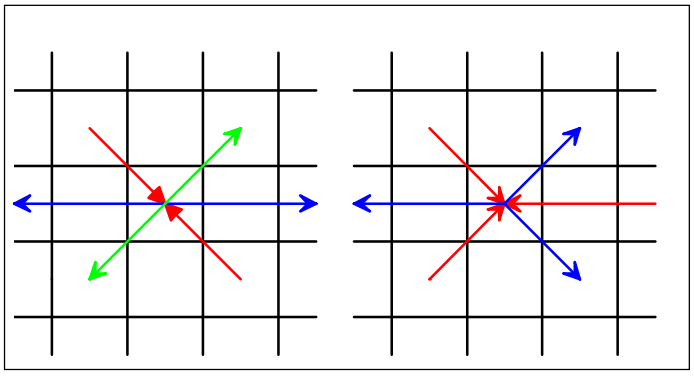
Sechs Teilchen bewegen sich auf die Zelle zu.

Abb.3: Kommunikationsstruktur



Die roten Pfeile zeigen den Ring, die blauen die Farm. Zu beachten ist, daß für die Updatephasen lediglich die Ringabschnitte zwischen Slaves (also die horizontalen roten Pfeile) relevant sind.

Abb.4: Kollisionen



Die roten Pfeile symbolisieren die eingehenden Teilchen. Im linken Fall treten mit 50%iger Wahrscheinlichkeit entweder die grüne oder die blaue Variante als ausgehende Teilchen auf, im rechten Fall repräsentieren die blauen Pfeile die ausgehenden Teilchen.

Noch ein paar Worte zur X11-Ausgabe:
Ich benutze hierzu die von Kay stammende RX11-Bibliothek, da sie sehr leistungsfähig und zudem (relativ) einfach zu benutzen ist. Insgesamt haben wir vier verschiedene Ausgabefunktionen ausprobiert, sind aber zuletzt auf die ursprüngliche Version (die ich bereits unter DOS implementiert hatte) zurückgekehrt. Diese Routinen stammen in der X11-Version von Kay. Jede Zelle wird hierbei in einem von sieben Farbtönen dargestellt, je heller, desto dichter ist die Teilchenkonzentration, d.h. kein Teilchen = schwarz, sechs Teilchen = weiß.
Die anderen drei Routinen befinden sich z.T. auch noch im Sourcecode (auskommentiert). Bei entsprechendem Interesse ist es kein Problem sie auszuprobieren.

Syntax der Eingabedatei:
Die Datei enthält die zur Initialisierung nötigen Werte. Am einfachsten ist es sicherlich, eine bestehende Datei zu verändern, trotzdem hier nochmal eine kurze Auflistung der erforderlichen Parameter:

```
## /*Anzahl Zeilen*/
## /*Anzahl Spalten*/
## /*Anzahl Barrieren*/
## /*Seed*/
## /*Display - x*/
## /*Display - y*/
## /*Anzahl Rechenschritte zwischen zwei Ausgaben*/
## /*total Steps*/
## /*Belegungsdichte in %*/
## /*Fluss durch Matrix*/
     /*hier kaemen jetzt die Barrieren in folgendem Format: */
## <x-LinksOben>
## <y-LinksOben>
## <x-RechtsUnten>
## <y-RechtsUnten>
/* ACHTUNG!: Es müssen mindestens soviele Barrieren angegeben
werden, wie in der Anzahl oben spezifiziert. */
 /* und fertig...*/
P.S.: Die C-Kommentarsyntay ist optional, "/*" bzw. "*/" kann
weggelassen werden.
```

"##" steht hierbei für einen Integerwert. **Wichtig!:** ASCII-Werte zwischen 48 und 57 dürfen nur anstelle der "##"-Ints vorkommen - auch nicht in Kommentaren.

Der **Programmaufruf** geschieht mit `lattice.z`. Das Programm erwartet dabei eine Datei mit Namen `lattice.dat` als Parameterdatei im aktuellen Verzeichnis. Alternativ kann man dem Programm auch eine Datei als Parameter mitgeben, etwa durch `lattice.z MeineLatticeParameter`. Es gibt noch eine Besonderheit: will man die Parameter interaktiv eingeben, ruft man das Programm mittels `lattice.z Console` auf.