Effiziente Implementierung von Zellularautomaten

Praktisches Beispiel: „Game of Life“

Beitrag zum Proseminar
„Zellularautomaten“
Von Johannes Singler

Gliederung

1. Einführung

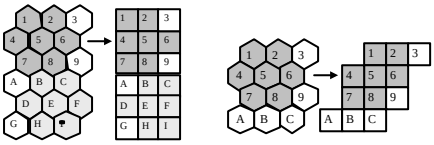
- Zellularautomaten massiv parallel aufgebaut
- Alle Zustandswechsel simultan
- Daher auch auf „normalen“ Rechnern mit von-Neumann-Architektur u. U. nicht einfach zu simulieren
- Sehr große Optimierungsmöglichkeiten

2.1 Raster

- 1-dimensional: trivial, lineares Array
- 3-dimensional: sehr viele mögliche Raster, bei Kuben 3-dimensionales Array, sonst wenig Aussagemöglichkeit
- Generell andere Möglichkeit für beliebig-dimensionale Raster:
  - Verzeigerung der Nachbarschaft zu Simulationsbeginn
  - Nachteil: hoher Speicherverbrauch, da Referenzen im Vergleich zu Zustandsspeicherplatz groß

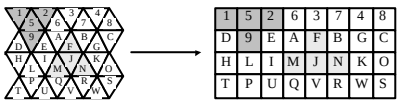
2.1 Raster (Sechsecke)

- Quadrate: Array direkt
- Dreiecke, Sechsecke: Abbilden auf quadratisches Raster
- Zwei Möglichkeiten bei hexagonaler Anordnung: einmal mit konstanten Randbedingungen, einmal mit konstanter Nachbarschaft



2.1 Raster (Dreiecke)

- Möglichkeit der Abbildung



- Nachbarschaft ändert sich jeder Zeile: Fallunterscheidung nötig

2.2 Nachbarschaften

- Von-Neumann- und Moore-Nachbarschaft auf Grund ihrer Regelmäßigkeit leicht mit Arrays zu implementieren
- Wichtig: „Sichtweite“ klein gegenüber Universum, daraus Optimierungsmöglichkeiten
- Berechnung innerhalb von Teilen des Raums möglich

2.3 Begrenzungen

- Unendlich große Räume i. A. nicht implementierbar
- Periodische Randbedingungen (Index modulo n)
- Reflexive Randbedingungen
- Feste Ränder: Zellen am Rand machen Zustandsübergang nicht mit
- Optimierung durch Mehrfachspeicherung möglich
  - Vorteil: Vermeidung von Fallunterscheidung
  - Nachteil: nachträgliche Synchronisation nötig
- Im Extremfall spezielle Übergangsfunktion für Ränder und Ecken

2. 4 Überführungsfunktion

- Berechnung mittels
  - Tabelle (vgl. endliche Automaten)
    - Auch mit Wildcards
    - Nach Priorität sortiert (vgl. Routing-Tabellen)
  - Boolescher Funktion
- Optimierungsmöglichkeiten, falls
  - Funktion nur von Anzahl Zellen bestimmten Zustands in der Nachbarschaft abhängig („Outer Totalistic CA“)
  - Zellen der Nachbarschaft und evtl. auch betrachtete Zelle nicht unterscheidbar
- Mehrere Zellen zu einem Automaten zusammenfassen
- Mehrere Schritte auf einmal berechnen (Achtung: Sichtweite vergrößert sich damit effektiv)

3.1 Praktische Anwendung am „Game of Life“

- „Game of Life“ von Conway 1970, sehr populärer Zellularautomat, turing-vollständig
- Raster: quadratisch
- Moore-Nachbarschaft: alle 8 direkt umliegenden Zellen
- Überführungsfunktion:
  Zelle lebt, wenn
  - 3 Zellen in Umgebung leben
  - 4 Zellen in Umgebung leben und Zelle selbst auch schon lebt
- „Game of Life“ ist ein „Outer Totalistic CA“, d. h. die Zellen in der Nachbarschaft müssen nicht unterscheidbar sein

3.2 Konkrete Implementierung

- Geschrieben in C++ unter Win-32
- 5 verschiedene Algorithmen
- Periodische Randbedingungen
- Aktueller Zustand darf bei Berechnung nicht sofort wieder überschrieben werden, daher zwei Speicher, zwischen denen gewechselt wird

3.2.1 Primitive Implementierung (Simple)

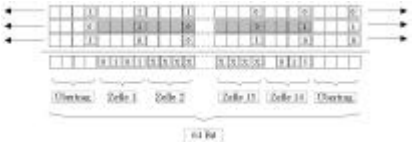
- Schön verständlich und lehrhaft ausprogrammiert
- Verschachtelte Schleife zählt lebende Zellen in der Nachbarschaft
- Mit Fallunterscheidung Folgezustand herausfinden
- Dient als Referenz, als nicht optimierte Variante

3.2.2 Anzahl der lebenden Zellen mitspeichern (Good)

- Anzahl der lebenden Zellen in der Nachbarschaft wird in jeder Zelle mitgespeichert
- Vorteil: schnelle Updates, falls sich nichts ändert, da nur 2 Lesezugriffe
- Nachteil: Anzahlen müssen auch aktualisiert werden, falls sich der Zustand einer Zelle ändert, das kostet 9 Schreibzugriffe

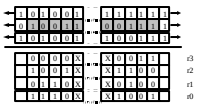
3.2.3 Mehrere Zellen gleichzeitig berechnen

- Mehrere Zellen werden in einem Langwort gespeichert und parallel neu berechnet
- 4 Bit pro Zelle, 0 für tot, 1 für lebendig
- Direktes Addieren innerhalb der Langwörter, da kein Überlauf
- Findung des Nachfolgezustands über booleschen Ausdruck
- Zellen am Rand können nicht verwertet werden -> Verschnitt



3.2.4 Speichern in Bitvektoren

- Raum wird durch zweidimensionales Array von Bitvektoren dargestellt, 1 Bit pro Zelle
- Addition durch boolesche Funktion entsprechende Dualzähler in 4 Bitvektoren als Zwischenspeicher
- Berechnung der neuen Zustände mittels booleschem Ausdruck stark parallel



3.2.5 Nur Bereiche berechnen, in denen „Leben herrscht“

- Speicherung des aktiven Gebiets in Menge von Kacheln
- Ausschließliche Berechnung nur dieser Kacheln, andere Zellen werden nicht „angefasst“
- Kacheln können werden vergrößert, falls sich ihr Rand ändert
- Kacheln fallen auseinander, falls ihr Füllungsgrad unter 10% fällt
- Kacheln werden zusammengefasst, falls das Ersparnis bringt
- Geschwindigkeit hängt stark von Konfiguration ab (extrem schnell bis extrem langsam)

3.3 Benchmark-Ergebnisse (1)

- Plattform: P3 733 MHz, 128 MB RAM, Windows 98 SE
- Feldgrößen 100×100, 868×868 (kein Verschnitt)
- Bei Algorithmen 2 und 5 unterschiedliche Konfigurationen getestet (Jeweils erste 100 Schritte):
  - Leeres Feld
  - Bistabile Konfiguration (wechselt zwischen zwei Zuständen)
  - Nur eine „Gosper‘s Gun“ (wenig los)
  - Zufällige Konfiguration (50% lebende Zellen)

3.3 Benchmark-Ergebnisse (2)

- Geschwindigkeit in „Millionen Zustandsübergänge · Anzahl Zellen pro Sekunde“ (MUpds) angegeben, Speicherverbrauch in KByte

	Kleines Feld (100×100)		Gro˚es Feld (868×868)	
Algorithmus	Geschwindigkeit	Speicher	Geschwindigkeit	Speicher
1 (Simple)	1,5	78	1,4	5886
2 (Good)	34,6/29,4/ 32,8/14,3	19	28,3/25,0/ 28,0/12,6	1471
3 (BitPacks)	55,3	12	51,0	840
4 (BitVector)	165,7	3	243,0	189
5 (Smart)	6500/11,0/3 0,2/1,1	546	502000/ /755/	41000