Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

Prof. Dr. Stefan Bosse

Universität Koblenz - FB Informatik - FG Praktiksche Informatik

Parallele und Verteilte Zelluläre Automaten

Wie kann das ZA Modell für die Parallelisierung und Verteilung genutzt werden?

Parallele und Verteilte Zelluläre Automaten

Wie kann das ZA Modell für die Parallelisierung und Verteilung genutzt werden?

Wie können geteilte und verteilte Speichermodelle sinnvoll eingesetzt werden?

Parallele und Verteilte Zelluläre Automaten

Wie kann das ZA Modell für die Parallelisierung und Verteilung genutzt werden?

Wie können geteilte und verteilte Speichermodelle sinnvoll eingesetzt werden?

Wie muss die Synchronisation aussehen? Synchron versa Asynchrone Ausführung

Parallele und Verteilte Zelluläre Automaten

Wie kann das ZA Modell für die Parallelisierung und Verteilung genutzt werden?

Wie können geteilte und verteilte Speichermodelle sinnvoll eingesetzt werden?

Wie muss die Synchronisation aussehen? Synchron versa Asynchrone Ausführung

Wie kann CALUA parallelisiert werden?

Arten von ZA

- Klassische ZA
  - Statische Regeln
  - Statische Nachbarschaft
- Neuronale ZA
  - Kombiniert ZA Strukturen mit Neuronalen Netzwerken
  - Maschen- oder Gitterperzeption
- Differenzierbare Logik ZA

Neuronale Zelluläre Automaten

[Horibe et al., arXiv:2206.06674]

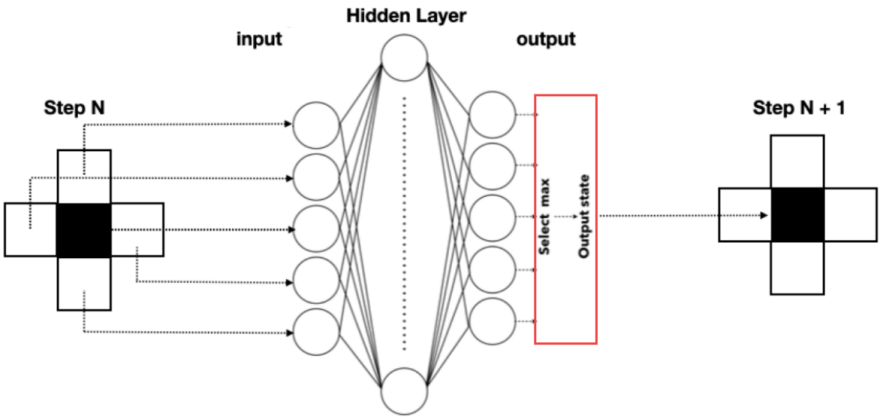


Abb. 1. Architekture eines NZA: Die nachbarschaftszellen liefern die Eingabe für ein Perzeptron dessen Ausgabe den nächsten Zustand der zelle bestimmt.

Ausführungsmodell

1. Synchron mit Zentraltakt
2. Synchron in Phasen
3. Asynchron in Phasen
4. Asynchron

Anforderungen

Die **Taxonomie** paralleler und verteilter Anwendungen bezieht sich auf Daten- und Anwendungsklassen, z.B.,

- Datenklassen: Vektor-, Matrix-, Tensor-, Funktionsdaten (zelluläre Automaten);
- Algorithmenklassen: Matrixoperationen im Allgemeinen, datengetriebene und iterative Optimierungsprobleme, Zelluläre Automatenverarbeitung, Simulation, Gleichungslösung, Regression, Statistische Analyse;
- Datenabhängigkeitsklassen: Lokaler, globaler, gruppierter, statischer und dynamischer Inhalt, statische und dynamische Größen sowie horizontale (zeitliche) und vertikale Abhängigkeiten;
- Verarbeitungsflussklassen: Datenfluss ⇔ Funktionsfluss, Kontrollfluss ⇒ Synchronisation;
- Partitionierungsklassen: Einzeldaten- und Einzelmodell- versa Mehrdaten- und Mehrmodellberechnung (z. B. Ensemblemodell ML mit Modellfusion, Datenströme);

Anforderungen

- Modellklassen: Datengetriebene Modellierung, z. B. unter Verwendung von ML-Methoden, aufgeteilt in Trainings-, Test- und Validierungsphasen, Hypothesentest und Modellauswahl (parallel), Exploration (Suche) in einem Modellraum, Exploration (Suche) in einem Hyperparameterraums;

- Größenklassen: Statische Größe versa Probleme mit dynamischer (wachsender) Größe.

Das 1:1 Modell

- Die einzelnen Zellen eines ZA sind zunächst völlig unabhängig voneinander → **Kontrollpfadparallelität**
- Die **Datenabängigkeit** einer Zelle ist auf die Zellen seiner unmittelbaren Umgebung beschränkt → **Kurzreichweitige Datenabhängigkeit**
- **Synchronisation** erfolgt primär (implizit) durch einen **zentralen Takt**, der aber nur die einzelnen Phasen des ZA (*before*, *activity*, *after*) einleitet.
- Weitere implizite Synchronisation beim Zugriff auf Zustand (Variablen) von Nachbarzellen

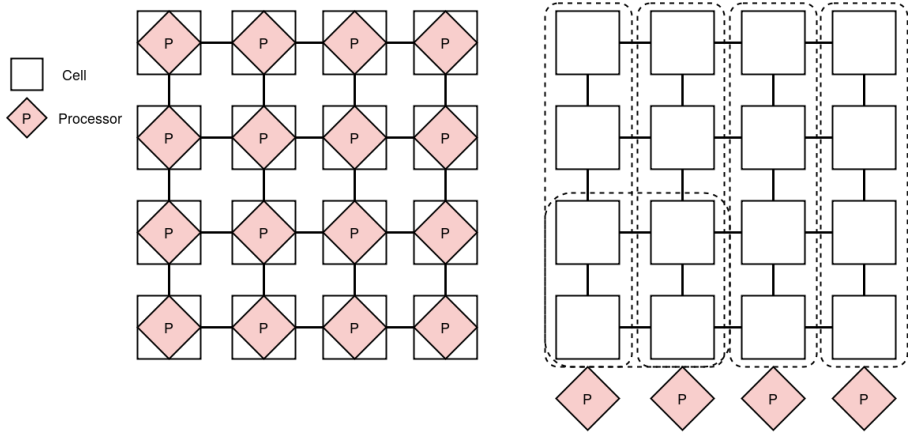


Abb. 2. (a) Das parallele (oder verteilte) 1:1 ZA Modell wo jede Zelle von einem physischen Prozessor ausgeführt wird → Verteiltes Speichermodell (b) Das partitionierte 1:N ZA Modell wo ein Bereich des ZA von einem Prozessor ausgeführt wird → Geteiltes und verteiltes Speichermodell

Das 1:1 Modell

Das 1:1 Modell ist nur in digitalen Hardwaresystemen sinnvoll. Es ist der Overhead der Kommunikation und Synchronisation zu beachten!

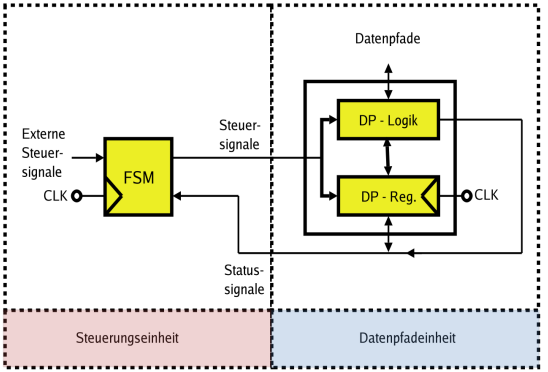


Abb. 3. Ein Zustandsautomatmit Register-Transfer Logik: Kontroll- und Datenpfad sind hier getrennt und statisch (nicht progranmmierbar), d.h. programmspezifisch.

Das Partitionsmodell

- Partitionierung der Zellen des ZA in parallele Felder:
  - Jedes Feld besteht aus einer Gruppe aus Zellen $F_i=\{z \in Z\} \subset Z$ für sich: Ein *P* mit *SM*, Zellen eng gekoppelt
  - Alle Felder: Verteilter Zellenrechner, Multi-*P*, Felder lose gekoppelt über Verteilten und geteilten Speicher (DSM)
- Man unterscheidet:
  - Kernbereiche eines Feldes (Untergruppe von Zellen aus *F*) mit reinem SM Modell, und
  - Überlappende Randbereiche mit DM Modell
- Es gibt weiterhin einen gemeinsamen Takt (Synch.)

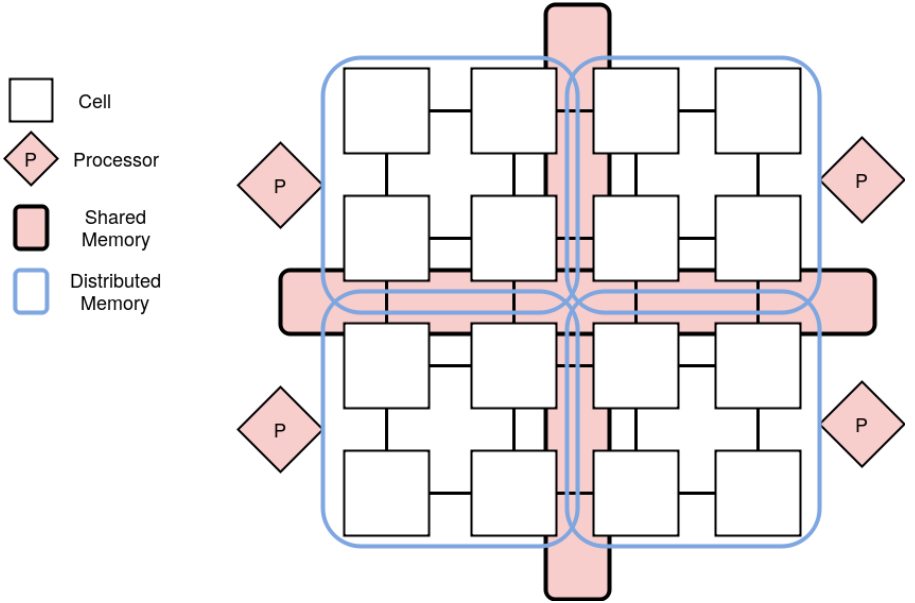


Abb. 4. Partitionierter ZA mit gemischten SM und DSM Modellen (Kern- und Randbereiche)

Parallele Programmierung des ZA

> Da das Speichermodell der Kern- und Randzellen unterschiedlich ist müsste explizit bei der Programmierung zwischen Zellen im eigenen Feld und im Nachbarfeldern unterschieden werden!

- Die Kommunikation bei Zellen innerhalb eines Feldes ist ein direkter Speicherzugriff,
  - bei verteilten Feldern (physisch getrennte Rechner) ein Nachrichtenversand (DSM),
  - und bei SM auf einem Parallelrechner (Multi-core Rechner) wieder ein direkter Speicherzugriff, aber bei Verwendung einer VM auf ein spezielles geteiltes Speicherobjekt!

Strukturierter Geteilter Speicher

- Auf geteilten Speicher (Shared Memory) kann durch direkten Speicherzugriff das Lesen und Schreiben von Speicherzellen erfolgen - zunächst aber keine Datenstrukturierung, reiner Byte Speicher
- Zusammengesetzte Daten (Arrays, Recors, Hashtabellen usw.) können prinzipiell über die Startadresse im Speicher von verschiedenene Prozessen direkt genutzt werden.
- Virtuelle Maschinen könnten Datenstrukturen über SM teilen.

Aber: Wenn es automatisches Speichermanagement gibt können Datenstrukturen im Speicher verschoben werden (Dynamische Speicheradressen). Statische Adressen sind erforderlich.

Strukturierter Geteilter Speicher

- Strukturierter Geteilter Speicher bedarf daher einer Serialisierung und Deserialisierung von Daten.

Serialisierung

Interne (binäre) Datenstrukturen werden in ein austauschbares Format übersetzte und von einer VM *A* zu einer anderen VM *B* übertragen (oder direkt im geteilten Speicher abgelegt)

Deserialisierung

Das austauschbare Datenformat (z.B. JSON) wird wieder in interne Datenstrukturen übersetzt.

Aber: Hier haben wir dann wieder ("teure") nachrichtenbasierte Kommunikation

- Besser: Die Struktur von Daten wird direkt im geteilten Speicher abgebildet und über Funktionen oder Monitore wird die Vm Datenstruktur auf die geteilte und vereinheitlichte Datenstruktur abgebildet ⇒ Strukturierter Geteilter Speicher

Strukturierter Geteilter Speicher

[Bosse, Appl.Sci. SI PC, MDPI, 2022]

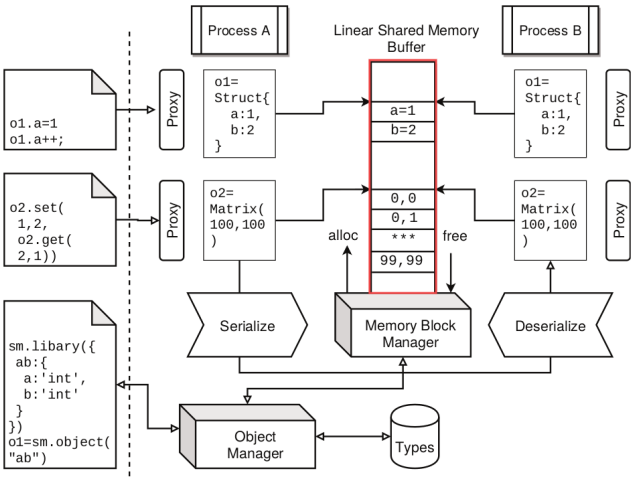


Abb. 5. Architektur für gemeinsam genutzte strukturierte Speicherobjekte und programmatischer Zugriff über Proxy Funktionen (Monitore)

Strukturierter Geteilter Speicher

Viele VM unterstützen dynamisch typisierte Programmierung (Lua, Python, JavaScript). Für einen strukturierten geteilten Speicher brauchen wir zwingend statische Typisierung.

- Die geteilten Datenstrukturen müssen also explizit erzeugt werden, eine direkte Konvertierung von internen durch dynamsiche Typsignaturen geprägte Daten ist nicht möglich.

```
typesDefs = {
  xy = {
    x='int',
    y='int'
  }
}
sm = BufferSegment:new(1000)
( erzeugt Array mit Datenstrukturen {x,y} )
o  = BufferSegment:object(sm,typesDefs.xy,10)
a  = BufferSegment:AddrOf(o)
o[1].x = 100
```

Bsp. 1. Prototypische Beispiel einer SSM API. Die Adresse *a* kann an einen anderen VM Prozess übergeben werden.

Monitore

- Um heterogene und nicht einheitliche Programmierung (Siehe Entwurfsprinzipien von Verteilten Systemen!) können **Objektmonitore** eingesetzt werden
  - Sogenannte getter/setter Funktionen überwachen den Zugriff auf Feldern von Objekten (Proxy)
  - Bei Nachbarzellen innerhalb eines Feldes wird direkt auf die Objektvariablen zugegriffen
  - Bei Nachbarzellen außerhalb des Feldes wird entweder Nachrichtenkommunikation verwendet (Achtung: ausführungsblockierend!) oder bei SM auf ein spezielles geteiltes Speicherobjekt zugegriffen

```
function C:initialize (..)
    for k,v in pairs(parameter.state) do
      if shared then
        self.__state[k]=v
      else
        self[k]=v;
      end
    end
end
function C:__newindex( index, value )
   if index ~= '__state' and self.__state ~= nil and self.__state[index] ~= nil then
     if shared then model.shared[index]:write(value,self.y,self.x)
     else self.__state[index] = value end
   else
     rawset( self, index, value )
   end
end
function C:__index( index )
    if index ~= '__state' and self.__state ~= nil and self.__state[index] ~= nil then
      if shared and then return model.shared[index]:read(self.y,self.x)
      else return self.__state[index] end
    else
      local value = rawget( self, index )
      if value == nil then return rawget (getmetatable(self), index)
      else return value end
    end
end
```

Code 1. Lua Objektmonitor mit Objektgetter und Settergateways. Bei geteilten Zellen werden die Zellzustandsvariablen (*state*) in einem Hintergrundobjekt verwaltet.

- Bei einem Zugriff auf die Zustandsvariablen einer Zelle, also z.B. `self.open`, werden die Setter und Getter Funktionen aufgerufen

- Handelt es sich um eine geteilte Randzelle und bei dem Objektzugriff um eine Zustandsvariable (aus der *state* Definition der Zelle), wird ein von allen Randzellen geteilter Speicher verwendet, ansonsten die lokale private Speicher einer Zelle

    - Anstelle des geteilten Speicherzugriffs könnte auch nachrichtenbasierte Kommunikation verwendet werden (*read*/*write* Remote Procedure Call)

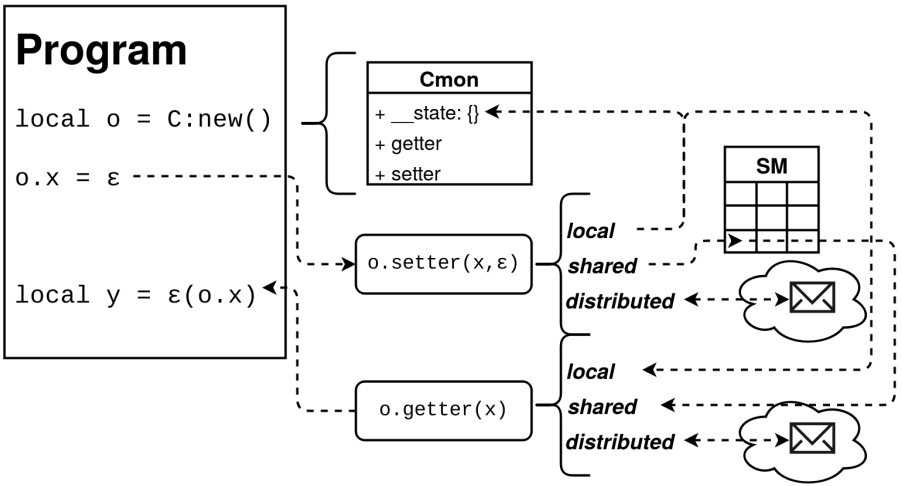


Abb. 6. Objektmonitore implementieren lokalen, geteilten, und verteilten Speicherzugriff: Virtualisierung des Speicherzugriffs durch getter und setter Methoden

- Aber: Sowohl der Zugriff auf geteilten (und geschützen) als auch vor allem auf verteilten Speicher verursacht erhöhte Laufzeitkosten!

- Da die Zelloperationen i.A. geringe Rechenkomplexität aufweisen kann das parallele ZA Modell schnell ineffizient werden (Skalierunsgproblem)

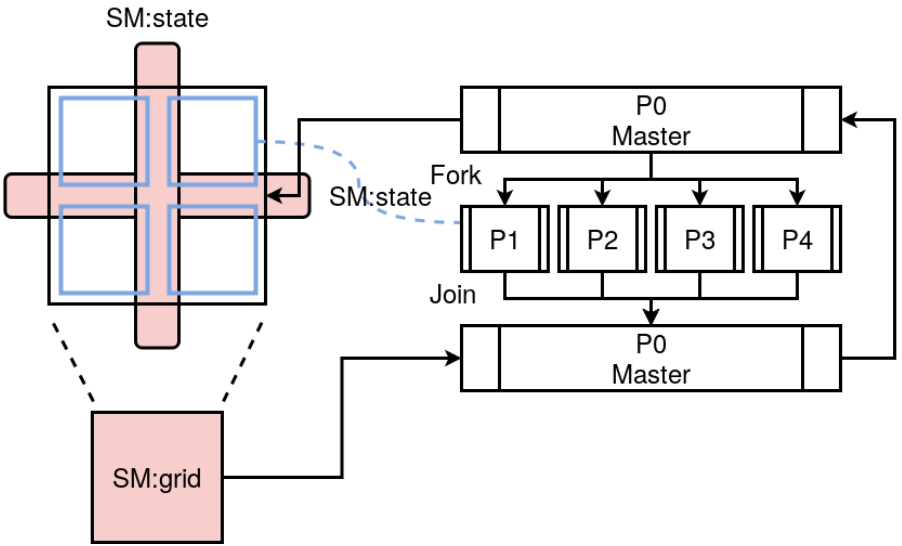
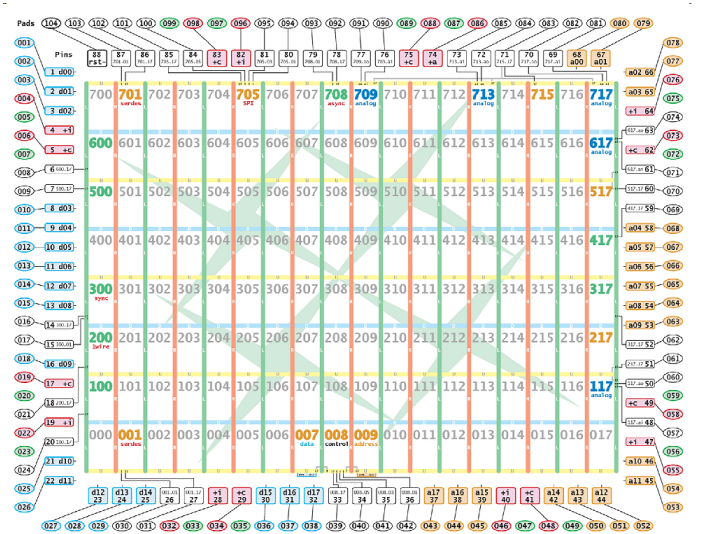


Abb. 7. (Links) Partitionierter ZA mit lokalen und geteilten Speicher in den Randbereichen (Rechts) Prozessablaufmodell: Fork & Join Phasen (jeweils einzeln für *before*, *activity*, *after*)

Multi-Computer Parallelrechner: GreenArrays GA144

[www.greenarraychips.com]



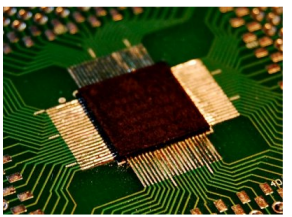


Abb. 8. GA144 Array Rechner (ein Mikrochip!)

Der GA144 Parallelrechner bietet:

- 144 F18A FORTH Prozessoren (FORTH: Interpretersprache, stackbasiert)
  - Anordnung der einzelnen Prozessoren in einem 2D Gitter mit Nachbarknotenkommunikation
  - Pro F18A Knoten: 9k Word RAM
- 96GIPS!
- 14µW - 650mW Leistungsaufnahme
- Kein zentraler Takt! Asynchrone Digitallogik

Zusammenfassung

Parallelisierung und Verteilung von Zellulären Automaten erfordert ein hybrides Speichermodell (lokal privat, geteilt, und verteilt)

Zusammenfassung

Parallelisierung und Verteilung von Zellulären Automaten erfordert ein hybrides Speichermodell (lokal privat, geteilt, und verteilt)

Durch die kurzreichweitige Datenabhängigkeit der Zellen kann durch Feldpartitionierung der Kommunikationsaufwand deutlich reduziert werden (nur Randzellen benötigen geteiltes Speichermodell mit direkten oder nachrichtenbasierten Zugriff)

Zusammenfassung

Parallelisierung und Verteilung von Zellulären Automaten erfordert ein hybrides Speichermodell (lokal privat, geteilt, und verteilt)

Durch die kurzreichweitige Datenabhängigkeit der Zellen kann durch Feldpartitionierung der Kommunikationsaufwand deutlich reduziert werden (nur Randzellen benötigen geteiltes Speichermodell mit direkten oder nachrichtenbasierten Zugriff)

Objektmonitore können ein einheitliches Programmierkonzept erzielen und Virtualisierung von dem darunterliegenden Speichermodell erreichen