

# Modelle der Parallelverarbeitung

## 4. Parallele Registermaschinen

Thomas Worsch

Institut für Theoretische Informatik  
Karlsruher Institut für Technologie

Sommersemester 2018

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

Eine nichtdeterministische Variante von PRAM

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

Eine nichtdeterministische Variante von PRAM

## Bestandteile einer RAM

- ▶ *Registermaschine*, engl. *random access machine (RAM)*
- ▶ Idee: idealisierter Prozessor
  - ▶ *Akkumulator*
  - ▶ *lokaler Speicher*:  
unbeschränkt viele Zellen  $L[0], L[1], L[2], \dots$   
für je eine unbeschränkt große nichtnegative ganze Zahl
  - ▶ gelegentlich Vorstellung: *Rechenalphabet*  $B$   
 $|B|$ -adische Darstellung von Zahlen
  - ▶ *Programmspeicher* mit endlicher Folge von  
Maschinenbefehlen  
(Adressierung über *Befehlszähler* PC)

## $b$ -adische $b$ -adische Darstellung von Zahlen

- ▶ Symbole für „Ziffern“:  $1, 2, \dots, b$   
mit den naheliegenden Wertigkeiten
- ▶ Stellenwertigkeiten wie üblich von hinten nach vorne  
 $b^0, b^1, b^2, \dots$
- ▶ Beispiel: 2-adische (dyadische) Darstellung von Zahlen
  - ▶ Symbole **1** und **2**
  - ▶ z. B. Darstellung der Zahlen 0 bis 9:

0	1	2	3	4	5	6	7	8	9
$\epsilon$	<b>1</b>	<b>2</b>	<b>11</b>	<b>12</b>	<b>21</b>	<b>22</b>	<b>111</b>	<b>112</b>	<b>121</b>

- ▶ eindeutige Zahlendarstellungen
  - ▶ Hin- und Herwechseln zwischen Zahlen und Wörtern

## Programme für RAMs

Befehl		Wirkung
LOAD	<i>operand</i>	Wert in den Akku
STORE	<i>operand</i>	Akku-Inhalt wird abgespeichert.
ADD	<i>operand</i>	Akku-Inhalt wird erhöht.
SUB	<i>operand</i>	Akku-Inhalt wird erniedrigt falls Differenz negativ: Akku $\leftarrow 0$
JUMP	<i>zeile</i>	unbedingter Sprung
JZERO	<i>zeile</i>	bedingter Sprung, falls ACCU = 0
HALT		RAM hält an.

## Adressierungsarten

- ▶ *Konstanten* wie 123.  
Ausnahme: bei STORE-Befehlen keine Konstanten
- ▶ *direkt adressierte* Speicherzellen des lokalen Speichers,  
z. B. L[7], usw.
- ▶ *indirekt adressierte* Speicherzellen,  
z. B. L[L[42]]

# Ausführung von Programmen

- ▶ initial:  
PC = 1
- ▶ RAM führt Befehl in Zeile, deren Nummer der Inhalt von PC ist, aus
- ▶ anschließend in der Regel Erhöhung um 1, Ausnahmen:
  - ▶ Sprungbefehle
  - ▶ HALT



## Kostenmaße

- ▶ *uniformes Kostenmaß:*  
Ausführung jedes Befehles dauert genau einen Schritt  
*das nehmen wir*
- ▶ *logarithmisches Kostenmaß:*  
Ausführung eines Arithmetik-Befehles benötigt  
Anzahl Schritte proportional Größe der Argumente  
(Anzahl Bits für ihre Repräsentation in Binärdarstellung)
- ▶ beachte:

## Kostenmaße

- ▶ *uniformes Kostenmaß:*  
Ausführung jedes Befehles dauert genau einen Schritt  
*das nehmen wir*
- ▶ *logarithmisches Kostenmaß:*  
Ausführung eines Arithmetik-Befehles benötigt  
Anzahl Schritte proportional Größe der Argumente  
(Anzahl Bits für ihre Repräsentation in Binärdarstellung)
- ▶ beachte:  
keine Befehle für Multiplikation, Shifts, ...

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

Grundlagen

Erkennung formaler Sprachen

Werkzeugkiste

Beispiele für Erkennung formaler Sprachen

Zusammenhang zwischen PRAM und TM

Eine nichtdeterministische Variante von PRAM

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

Grundlagen

Erkennung formaler Sprachen

Werkzeugkiste

Beispiele für Erkennung formaler Sprachen

Zusammenhang zwischen PRAM und TM

Eine nichtdeterministische Variante von PRAM

## Bestandteile

- ▶ *globaler Speicher* mit unbeschränkt vielen Registern  $G[0]$ ,  $G[1]$ , ...
- ▶ abzählbar unendliche viele *Prozessoren*  $P_0, P_1, \dots$ 
  - ▶ jeder Prozessor analog zu RAM aufgebaut
  - ▶ aktiv oder inaktiv
  - ▶ alle aktiven Prozessoren arbeiten nach dem gleichen Programm,
  - ▶ können sich aber in verschiedenen Zeilen befinden

# Programme für PRAM

- ▶ analog zu RAM-Programmen
- ▶ zusätzliche Adressierungsarten
  - ▶ Bezugnahme auf globalen Speicher, z. B.  $L[G[42]]$
- ▶ zusätzlicher Maschinenbefehl *FORK zeile*
  - ▶ aktiviert einen bislang inaktiven Prozessor  $Q$
  - ▶ Akku-Inhalt wird nach  $Q$  kopiert
  - ▶  $Q$  startet mit PC-Inhalt *zeile*
- ▶ Bei Ausführung von *HALT* wird Prozessor inaktiv

# Konflikte im globalen Speicher

# Konflikte im globalen Speicher

man unterscheidet

- ▶ EREW – *exclusive read exclusive write*
- ▶ CREW – *concurrent read exclusive write*  
*das nehmen wir*
- ▶ CRCW – *concurrent read concurrent write*  
(verschiedene Varianten)



# Überblick

Sequenzielle Registermaschinen

**Parallele Registermaschinen (Standard-Modell)**

Grundlagen

**Erkennung formaler Sprachen**

Werkzeugkiste

Beispiele für Erkennung formaler Sprachen

Zusammenhang zwischen PRAM und TM

Eine nichtdeterministische Variante von PRAM

## Weitere Definitionen

- ▶ *Eingabealphabet*  $A \subset B$
- ▶ Anfangskonfiguration für  $w = x_1 \cdots x_n \in A^n$ :
  - ▶  $G[0]$  enthält  $n$
  - ▶  $G[1], \dots, G[n]$  enthalten die Symbole  $x_1, \dots, x_n$
  - ▶ Alle anderen Zellen des globalen und aller lokalen Speicher sind mit 0 initialisiert.
  - ▶ Zu Beginn nur  $P_0$  aktiv, startet in Zeile 1 des Programmes
- ▶ *Endkonfiguration* erreicht, wenn  $P_0$  HALT ausführt.
- ▶ dann Eingabe
  - ▶ abgelehnt, wenn 0 im Akku von  $P_0$  steht
  - ▶ akzeptiert, wenn Wert  $\neq 0$  im Akku von  $P_0$  steht

# Überblick

Sequenzielle Registermaschinen

**Parallele Registermaschinen (Standard-Modell)**

Grundlagen

Erkennung formaler Sprachen

**Werkzeugkiste**

Beispiele für Erkennung formaler Sprachen

Zusammenhang zwischen PRAM und TM

Eine nichtdeterministische Variante von PRAM

## Einige nützliche Hilfsmittel

- ▶ Baum von Prozessoren
- ▶ Parameterübergabe zwischen ihnen
- ▶ „parallele Schleifen“

## Baum von Prozessoren

- ▶ endlicher Baum von Prozessoren
  - ▶ Verzweigungsgrad  $m$
  - ▶ Wurzel:  $P_0$
  - ▶  $P_i$  hat Nachfolger  $P_{mi+1}, \dots, P_{mi+m}$
- ▶ beschrifte die Kante von  $P_i$  zu  $P_{i+j}$  mit  $j$
- ▶ Folge der Kantenbeschriftungen von der Wurzel zu  $P_i$  ist die  $m$ -adische Darstellung von  $i$

# Parameterübergabe zwischen Prozessoren

## Parameterübergabe zwischen Prozessoren

- ▶ endlicher Baum von Prozessoren
- ▶ Kommunikation von  $P_i$  mit seinem Vorgänger:  
konstant  $k$  viele Speicherstellen  $G[a_i + 0], \dots, G[a_i + k - 1]$  mögen genügen.
- ▶ Prozessor  $P_i$  erhält beim Start im ACCU Adresse  $a_i$   
übergeben, ab der im globalen Speicher Eingabewerte zu lesen und Ausgabewerte zu schreiben sind.
- ▶ Wähle  $a_i = o + ki$  (für geeigneten Offset  $o$ , z. B.  $n + 1$ ).
- ▶  $P_i$  kommuniziert mit den  $P_{mi+x}$  für  $x = 1, \dots, m$ , jeweils ab Adresse  $a_{mi+x} = o + k(mi + x) = o + m(a_i - o) + kx$ .
- ▶ Beachte: Ausgehend von  $a_i$  sind die  $a_{mi+x}$  sukzessive in jeweils konstanter Zeit berechenbar.

# Parameterübergabe zwischen Prozessoren

- ▶ Beispiel für  $m = 3$ ,  $k = 2$  und  $o = 8$ :

$$P_0 \rightarrow P_1 : G[8 + 2 \cdot 1] \quad G[8 + 2 \cdot 1 + 1]$$

$$: G[10] \quad G[11]$$

$$P_0 \rightarrow P_2 : G[8 + 2 \cdot 2] \quad G[8 + 2 \cdot 2 + 1]$$

$$: G[12] \quad G[13]$$

$$P_0 \rightarrow P_3 : G[8 + 2 \cdot 3] \quad G[8 + 2 \cdot 3 + 1]$$

$$: G[14] \quad G[15]$$

$$P_1 \rightarrow P_4 : G[8 + 2 \cdot 4] \quad G[8 + 2 \cdot 4 + 1]$$

$$: G[16] \quad G[17]$$



## Parallele Bearbeitung aller Zahlen in einem Bereich

$adr \leftarrow ACCU$

$i \leftarrow G[adr] ; ub \leftarrow G[adr + 1]$

**if**  $i \leq ub$  **then**

**for**  $x \leftarrow 1$  **to**  $m$  **do**

$adrx \leftarrow o + m(adr - o) + kx$

$G[adrx] \leftarrow mi + x ; G[adrx + 1] \leftarrow ub$

$ACCU \leftarrow adrx$

FORK ...

**od**

*⟨und nun die lokal zu erledigende Arbeit für Zahl  $i$ ⟩*

*⟨evtl. warten auf Ergebnisse von Nachfolgern⟩*

*⟨zusammengefasstes Ergebnis „nach oben“ liefern⟩*

**fi**

## Beispiel: Abspalten des letzten Symbols

- ▶ Gegeben: Zahl mit Darstellung  $w = vx$  mit  $v \in B^*$ ,  $x \in B$
- ▶ Gesucht: die Zahlen mit den Darstellungen  $v$  und  $x$

## Beispiel: Abspalten des letzten Symbols

- ▶ Gegeben: Zahl mit Darstellung  $w = vx$  mit  $v \in B^*$ ,  $x \in B$
- ▶ Gesucht: die Zahlen mit den Darstellungen  $v$  und  $x$
- ▶ Idee:
  - ▶ Starte Baum von Prozessoren
  - ▶ Eingaben:  $w$  und ein  $u$  (das potenzielle  $v$ )
    - ▶ Wurzel startet mit  $u = \varepsilon$
  - ▶ prüfe alle  $x$  darauf hin, wie groß  $ux$  ist ...

## Beispiel: Abspalten des letzten Symbols (2)

Genauer:

```
function  $(v, x, flag) \leftarrow \text{lastsymbol}(u, w)$   
if  $w = \varepsilon$  then return  $(\_, \_, 0)$  fi  
for  $x \leftarrow 1$  to  $m$  do  
    if  $mu + x = w$  then  
         $res[x] \leftarrow (u, x, 1)$   
    elseif  $mu + x < w$  then  
         $res[x] \leftarrow \text{lastsymbol}(mu + x, w)$      $\langle \text{FORK ...} \rangle$   
    else  
         $res[x] \leftarrow (\_, \_, 0)$   
    fi  
od
```

## Beispiel: Abspalten des letzten Symbols (3)

```
success ← 0
for  $x \leftarrow 1$  to  $m$  do
    if  $res[x] = (\dots, \dots, 1)$  then
        success ← 1
        successx ←  $x$ 
    fi
od
if success then
    return  $res[successx]$ 
else
    return  $(\_, \_, 0)$ 
fi
```

## Ähnliche Verfahren

- ▶ Anhängen  $(w, x) \mapsto wx$  ist einfach:  $m \cdot w + x$
- ▶ Abspalten des letzten Symbols:  $wx \mapsto (w, x)$
- ▶ Abspalten des ersten Symbols:  $xw \mapsto (x, w)$
- ▶ Voranstellen eines einzelnen Symbols:  $(x, w) \mapsto xw$
- ▶ allgemeiner: Konkatination von Wörtern:  $(w_1, w_2) \mapsto w_1w_2$
- ▶ Zerlegen eines Wortes  $w_1xw_2 \mapsto (w_1, x, w_2)$  mit  $w_1w_2 \in A_1^*$  und  $x \in A_2$  für disjunkte Alphabete  $A_1, A_2$
- ▶ in allen Fällen Zeitbedarf proportional zur Länge der Wörter

# Überblick

Sequenzielle Registermaschinen

**Parallele Registermaschinen (Standard-Modell)**

Grundlagen

Erkennung formaler Sprachen

Werkzeugkiste

**Beispiele für Erkennung formaler Sprachen**

Zusammenhang zwischen PRAM und TM

Eine nichtdeterministische Variante von PRAM

$L_{pal}$

- ▶ Algorithmus: an der Tafel



$L_{pal}$

- ▶ Algorithmus: an der Tafel
- ▶ Ergebnis:

$$L_{pal} \in \text{PRAM-TIME}(\Theta(\log n))$$

$L_{VV}$

► Analog:

$$L_{VV} \in \text{PRAM-TIME}(\Theta(\log n))$$

$L_{VV}$

- ▶ Analog:

$$L_{VV} \in \text{PRAM-TIME}(\Theta(\log n))$$

- ▶ Wo kam im Zusammenhang mit  $L_{VV}$  schon mal  $\log n$  vor?

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

Grundlagen

Erkennung formaler Sprachen

Werkzeugkiste

Beispiele für Erkennung formaler Sprachen

**Zusammenhang zwischen PRAM und TM**

Eine nichtdeterministische Variante von PRAM

# Simulation von TM durch PRAM

Idee:

# Simulation von TM durch PRAM

Idee:

1. Repräsentiere TM-Konfiguration  $c$  durch Zahl  $\text{cod}(c)$ ,
  - ▶ und zwar so, dass PRAM aus  $\text{cod}(c)$  leicht  $\text{cod}(\Delta_T(c))$  ausrechnen kann.
2. Erzeuge schnell
  - ▶ Kodierungen *aller* „relevanten“ TM-Konfigurationen
  - ▶ und der zugehörigen Nachfolgekonfigurationen
3. Bestimme zu (Kodierung der) Anfangskonfiguration schnell (Kodierung der) zugehörigen Endkonfiguration

## Kodierung von TM-Konfigurationen

- ▶ Es sei  $C_k$  die Menge aller Konfigurationen, bei denen die Felder mit Nummern  $|i| > k$  nicht benutzt sind.
- ▶ wähle  $B_P = \{1, 2, \dots, m\}$  mit  $m = |S_T| + |B_T|$   
o. B. d. A. entspreche  $\{1, 2, \dots, |S_T|\}$  der Zustandsmenge  $S_T$  und ...
- ▶ codiere  $c = (s, b, p) \in C_k$  als Wort der Länge  $2k + 2$  über  $B_P$  in der Form  $\text{cod}_k(c) =$

$$b(-k) \cdot b(-k + 1) \cdots b(p - 1) \cdot s \cdot b(p) \cdot b(p + 1) \cdots b(k)$$

# Kodierung von TM-Konfigurationen

## Lemma

1. Jede TM-Konfiguration hat beliebig lange Codierungen.
2. Je endlich viele TM-Konfigurationen besitzen (u.a.) gleich lange Codierungen.
3. Zu jeder TM  $T$  gibt es eine PRAM  $P$ , so dass für alle  $k$  gilt:  
Sind  $c \in C_k$  und  $\Delta_T(c) \in C_k$ , dann kann  $P$  in einer Zeit proportional zu  $k$  aus  $\text{cod}_k(c)$  die Codierung  $\text{cod}_k(\Delta_T(c))$  berechnen.



## Beweis des Lemmas

- ▶ Es sei  $m = |B_P|$ .
- ▶ Algorithmus:
  - ▶ Zerlege  $\text{cod}_k(c) = w_1 x w_2$  in  $w_1$ ,  $x \in S_T$  und  $w_2$ , sowie  $w_1 = w'_1 b_1$  in  $w'_1$  und  $b_1$  und  $w_2 = b_2 w'_2$  in  $b_2$  und  $w'_2$ .
  - ▶ bestimme anhand von  $x$  und  $b_2$ , was die TM tun würde:  $x'$  und  $b'_2$
  - ▶ „simuliere“ Kopfbewegung durch ggf. „Verschieben“ von  $x'$  um ein Bandsymbol
  - ▶ berechne  $\text{cod}_k(\Delta_T(c)) = w'_1 \cdots w'_2$

# Simulation von TM durch PRAM

## Satz

Für alle  $s(n) \geq n$  gilt:

$$\mathbb{W}_1\text{-TM-SPC}(s(n)) \subseteq \text{PRAM-TIME}(\Theta(s(n)))$$

# Simulation von TM durch PRAM

## Satz

Für alle  $s(n) \geq n$  gilt:

$$\mathbb{W}_1\text{-TM-SPC}(s(n)) \subseteq \text{PRAM-TIME}(\Theta(s(n)))$$

Anmerkung:

- ▶ Ein analoges Ergebnis kann man für  $\mathbb{E}\text{-}\mathbb{W}_*\text{-TM}$  und  $s(n) \geq \log n$  beweisen.
- ▶  $L_{\forall\forall}$  kann man mit TM auf logarithmischem Platz erkennen, also auch mit PRAM in logarithmischer Zeit.

# Simulation von TM durch PRAM: Algorithmus (1)

Beweis in 3 Teilen

Teil 1: (unabhängig von PRAM)

- ▶ Es sei  $T$  eine TM,  $w \in A^+$  und  $k = \text{Space}(|w|)$
- ▶ Sei  $G_k$  der Graph mit Knotenmenge  $C_k$  und Kanten für die TM-Übergänge.
- ▶ An Knoten für Endkonfigurationen sind Schleifen.
- ▶  $w \in L(T)$  gdw. in  $G_k$  ein Weg mit Länge exakt  $d^k$  von Anfangskonfig.  $\text{init}(w)$  zu einer Endkonfig. existiert (für geeignetes  $d$ )

## Simulation von TM durch PRAM: Algorithmus (2)

Teil 2:

unter der Annahme, dass die PRAM zu  $w$  auch  $k$  kennt:

- ▶ Aktiviere Baum von Prozessoren, deren Blätter für Zahlen  $\text{cod}_k(c)$  stehen.
- ▶ Jeder solche Prozessor führt durch:
  - ▶ Phase 1: berechne in Zeit  $\Theta(k)$  Codierung  $\text{cod}_k(\Delta_T(c))$  und speichert diesen Wert unter Adresse  $\text{cod}_k(c)$ :  
 $G[\text{cod}_k(c)] \leftarrow \text{cod}_k(\Delta_T(c))$
  - ▶ Phase 2:

## Simulation von TM durch PRAM: Algorithmus (2)

Teil 2:

unter der Annahme, dass die PRAM zu  $w$  auch  $k$  kennt:

- ▶ Aktiviere Baum von Prozessoren, deren Blätter für Zahlen  $\text{cod}_k(c)$  stehen.
- ▶ Jeder solche Prozessor führt durch:
  - ▶ Phase 1: berechne in Zeit  $\Theta(k)$  Codierung  $\text{cod}_k(\Delta_T(c))$  und speichert diesen Wert unter Adresse  $\text{cod}_k(c)$ :  
 $G[\text{cod}_k(c)] \leftarrow \text{cod}_k(\Delta_T(c))$
  - ▶ Phase 2: iteriere  $\lceil \log_2 |C_k| \rceil$  mal „Zeigerspringen“:  
 $G[\text{cod}_k(c)] \leftarrow G[G[\text{cod}_k(c)]]$

# Simulation von TM durch PRAM: Algorithmus (2)

Teil 2:

unter der Annahme, dass die PRAM zu  $w$  auch  $k$  kennt:

- ▶ Aktiviere Baum von Prozessoren, deren Blätter für Zahlen  $\text{cod}_k(c)$  stehen.
- ▶ Jeder solche Prozessor führt durch:
  - ▶ Phase 1: berechne in Zeit  $\Theta(k)$  Codierung  $\text{cod}_k(\Delta_T(c))$  und speichert diesen Wert unter Adresse  $\text{cod}_k(c)$ :  

$$G[\text{cod}_k(c)] \leftarrow \text{cod}_k(\Delta_T(c))$$
  - ▶ Phase 2: iteriere  $\lceil \log_2 |C_k| \rceil$  mal „Zeigerspringen“:  

$$G[\text{cod}_k(c)] \leftarrow G[G[\text{cod}_k(c)]]$$
- ▶ Nach  $i$  Iterationen von Phase 2:  $G[j]$  enthält  $\Delta_T^{2^i}(j)$ , also
- ▶  $\lceil \log_2 |C_k| \rceil$  Iterationen: in  $G[j]$  erreichte Endkonfiguration

## Simulation von TM durch PRAM: Algorithmus (3)

Teil 3:

Da im allgemeinen  $k$  unbekannt:

- ▶ Die PRAM startet nacheinander den in Teil 2 beschriebenen Algorithmus für  $k = 2, 3, \dots$ ,
- ▶ bis irgendwann für ein  $k$ 
  - ▶ eine Antwort „ $w \in L(T)$ “ oder „ $w \notin L(T)$ “
  - ▶ aber *nicht* die Antwort „Platz reicht nicht aus“geliefert wird.



# Polynomielle Ressourcenschranken

## Korollar

$$\mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \text{PRAM-TIME}(\text{Pol}(n))$$

- ▶ Man kann **PSPACE**-vollständige Probleme in Polynialzeit auf PRAM lösen,

# Polynomielle Ressourcenschranken

## Korollar

$$\mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \text{PRAM-TIME}(\text{Pol}(n))$$

- ▶ Man kann **PSPACE**-vollständige Probleme in Polynialzeit auf PRAM lösen,
- ▶ muss dafür aber (bei allen bekannten Konstruktionen) mehr als polynomiell viele Prozessoren aktivieren ...

## Polynomielle Ressourcenschranken (2)

- ▶ anderer Beweis für  $\mathbf{PSPACE} \subseteq \text{PRAM-TIME}(\text{Pol}(n))$ :
  - ▶ löse ein  $\mathbf{PSPACE}$ -vollständiges Problem in Polynomialzeit auf einer PRAM

## Polynomielle Ressourcenschranken (2)

- ▶ anderer Beweis für  $\mathbf{PSPACE} \subseteq \text{PRAM-TIME}(\text{Pol}(n))$ :
  - ▶ löse ein  $\mathbf{PSPACE}$ -vollständiges Problem in Polynomialzeit auf einer PRAM
- ▶ Beispiel: „Quantified Boolean Formula“ QBF (QSAT)  
Probleminstanzen: Formeln der Form  $Q_1x_1 \cdots Q_kx_k : F(x_1, \dots, x_k)$  wobei
  - ▶ jedes  $Q_i$  ein  $\exists$ - oder ein  $\forall$ -Quantor ist und
  - ▶  $F(x_1, \dots, x_k)$  eine boolesche Formel mit freien Variablen  $x_1, \dots, x_k$ .

# Simulation von PRAM durch TM

# Simulation von PRAM durch TM

## Satz

Für alle  $t(n) \geq n$  gilt:

$$\text{PRAM-TIME}(t(n)) \subseteq \mathbb{W}_* \text{-TM-SPC}(\text{Pol}(t(n)))$$

# Simulation von PRAM durch TM: Algorithmus

nutze folgende 5 Prozeduren:

- ▶ **bool** *check* (**proc**  $p$ , **time**  $start$ , **time**  $t$ , **line**  $l$ , **nat**  $a$ )
- ▶ **bool** *findForker* (**proc**  $p$ , **time**  $start$ , **line**  $l$ , **nat**  $a$ )
- ▶ (**bool,line,nat**) *findLineAccu* (**proc**  $p$ , **time**  $t$ , **time**  $start$ )
- ▶ **bool** *checkLine* (**proc**  $p$ , **time**  $t$ , **line**  $l'$ , **nat**  $a'$ , **line**  $l$ )
- ▶ **bool** *checkAccu* (**proc**  $p$ , **time**  $t$ , **nat**  $a'$ , **line**  $l$ , **nat**  $a$ )

## Simulation von PRAM durch TM: Algorithmus (2)

**function**

**bool** *check* (**proc** *p*, **time** *start*, **time** *t*, **line** *l*, **nat** *a*)

**begin**

**if** (*t* < *start*) **then return** (*false*); **fi**

**if** (*t* = *start*) **then**

**if** (*p* = 0 **and** *start* = 0) **then**

*⟨der einfachste Fall: es muss  $l = 0$  sein, usw.⟩*

    ...

**elsif** (*p* > 0 **and** *start* > 0) **then**

**return** (*findForker*(*p*, *start*, *l*, *a*));

**else** *⟨andere Fälle sind unmöglich⟩*

**return** (*false*);

**fi**

**fi**



## Simulation von PRAM durch TM: Algorithmus (3)

**function**

**bool** *check* (**proc** *p*, **time** *start*, **time** *t*, **line** *l*, **nat** *a*)

**begin**

...

**if** (*t* > *start*) **then**

*⟨Informationen zum vorangegangenen Schritt:⟩*

  (*found*, *l'*, *a'*) ← *findLineAccu*(*p*, *t* - 1, *start*)

**if not** *found* **then return** (*false*); **fi**

*⟨Konsistenz vorangegangenen Schrittes mit diesem:⟩*

*c*<sub>1</sub> ← *checkLine*(*p*, *t*, *l'*, *a'*, *l*);

*c*<sub>2</sub> ← *checkAccu*(*p*, *t*, *a'*, *l*, *a*);

**return** (*c*<sub>1</sub> **and** *c*<sub>2</sub>);

**fi**

**end**

## Simulation von PRAM durch TM: Algorithmus (4)

**bool** *findForker* (**proc** *p*, **time** *start*, **line** *l*, **nat** *a*)

## Simulation von PRAM durch TM: Algorithmus (4)

```

bool findForker (proc  $p$ , time  $start$ , line  $l$ , nat  $a$ )
for  $p' \leftarrow p - 1$  to 0 step -1 do
    for  $s' \leftarrow start - 1$  to 0 step -1 do
        for  $a' \leftarrow 0$  to  $n \cdot alphSize^t$  do
            if ( $checkAccu(p, start, a', l, a)$ ) then
                for  $l' \leftarrow$  each  $forkLineWithLabel(l)$  do
                    if ( $check(p', start - 1, s', l', a')$ ) then
                        return ( $true$ );
                    fi
                od
            fi
        od
    od
od
return ( $false$ );

```

## Simulation von PRAM durch TM: Algorithmus (5)

- ▶ andere Funktionen analog
- ▶ Beobachtung: Parameter **time**  $t$  wird spätestens bei jedem zweiten (indirekt rekursiven) Aufruf einer Funktion um 1 erniedrigt. Also
- ▶ maximale Rekursionstiefe:  
proportional zum Zeitbedarf der PRAM
- ▶ Platzbedarf auf jeder Rekursionsstufe:  
polynomial zum Zeitbedarf  
(betrachte größtmögliche Schleifenzählerwerte)

# Zusammenhang von PRAM und TM

## Korollar

Für alle  $f(n) \geq n$  gilt:

$$\text{PRAM-TIME}(\text{Pol}(f(n))) = \mathbb{W}_*\text{-TM-SPC}(\text{Pol}(f(n)))$$

- ▶ salopp gesprochen: *polynomielle Verknüpfung von sequenziellem Platzbedarf und parallelem Zeitbedarf*

# Zusammenhang von PRAM und TM

## Korollar

Für alle  $f(n) \geq n$  gilt:

$$\text{PRAM-TIME}(\text{Pol}(f(n))) = \mathbb{W}_* \text{-TM-SPC}(\text{Pol}(f(n)))$$

- ▶ salopp gesprochen: *polynomielle Verknüpfung von sequenziellem Platzbedarf und parallelem Zeitbedarf*
  - ▶ der Preis: viele Prozessoren
- ▶ besseres Verständnis im nächsten Kapitel

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

**Eine nichtdeterministische Variante von PRAM**

Grundlagen

Beziehung zu sequenziellen Turingmaschinen

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

**Eine nichtdeterministische Variante von PRAM**

**Grundlagen**

Beziehung zu sequenziellen Turingmaschinen



# NLPRAM



W. J. Savitch

Parallel Random Access Machines with  
Powerful Instruction Sets.

*Mathematical Systems Theory*, 15:191–210, 1979.

„Nondeterministic List-processing PRAM“: NLPRAM

# Bestandteile

- ▶ binärer Baum von Prozessoren, jeweils sequenzielle RAM
- ▶ Kommunikation jeweils über  $k$  Register:
  - ▶  $CL[1], \dots, CL[k]$ : zum linken Nachfolger
  - ▶  $CR[1], \dots, CR[k]$ : zum rechten Nachfolger
  - ▶  $C[1], \dots, C[k]$ : zum Vorgänger
- ▶ Beachte:
  - ▶ Was für einen Prozessor die  $CL[i]$  sind, sind für seinen linken Nachfolger die  $C[i]$ .
  - ▶ Was für einen Prozessor die  $CR[i]$  sind, sind für seinen rechten Nachfolger die  $C[i]$ .
- ▶ Maschinenbefehle: wie bei RAM  
zusätzlich: siehe nächste Folie

## Neue Maschinenbefehle

- ▶ **FORKL**  $\langle op_1, \dots, op_k \rangle$  und **FORKR**  $\langle op_1, \dots, op_k \rangle$  zum Starten des linken resp. rechten Nachfolgers nach Laden der angegebenen Operanden in die CL- resp. CR-Register in Zeile 1 (!) des Programmes
- ▶ **RETURN**  $\langle op_1, \dots, op_k \rangle$  stoppt Prozessor nach Laden der Operanden in die C-Register
- ▶ **JRETL** und **JRETR** zum Testen, ob der linke resp. rechte Nachfolger fertig ist
- ▶ *Nichtdeterminismus*: ein Sprungbefehl kann mehrere Sprungziele „anbieten“
- ▶ **CONC** zum Konkatenieren zweier Operanden (als Wörter)
- ▶ **HEAD** und **TAIL** zur Extraktion der vorderen resp. hinteren „Hälfte“ des Wortes im ACCU

# Erkennung formaler Sprachen

- ▶ Anfangskonfiguration:
  - ▶ nur Wurzelprozessor aktiv
  - ▶ startet in Zeile 1 des Programms
  - ▶  $L[0]$  enthält  $w$
  - ▶ alle anderen Register enthalten 0
- ▶ Endkonfiguration:
  - ▶ Wurzelprozessor führt RETURN aus
  - ▶  $w$  akzeptiert, wenn es
    - ▶ eine Berechnung gibt, bei der
    - ▶ am Ende Inhalt von ACCU ungleich 0 ist

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Überblick

Sequenzielle Registermaschinen

Parallele Registermaschinen (Standard-Modell)

**Eine nichtdeterministische Variante von PRAM**

Grundlagen

Beziehung zu sequenziellen Turingmaschinen

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Beispiel: schnelles Raten großer Zahlen

```
JUMP    deeper, choose_1, choose_2, choose_e
deeper: FORKL  0
        FORKR  0
        wL:   JRETL wR
            JUMP  wL
        wR:   JRETR c
            JUMP  wR
        c:   LOAD  CL[1]
            CONC  CR[1]
            RETURN ACCU
choose_1: RETURN 1
choose_2: RETURN 2
choose_e: RETURN 0
```

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Beispiel: schnelles Raten großer Zahlen

- ▶ Beobachtung: Jede Zahl, die eine Darstellung der *Länge*  $\ell$  hat, kann in *Zeit*  $\log \ell$  bei dem Rateprozess als Ergebnis geliefert werden.

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Codierung von TM-Konfigurationen

### Lemma

1. Jede TM-Konfiguration hat beliebig lange Codierungen.
2. Je endlich viele TM-Konfigurationen besitzen (u.a.) gleich lange Codierungen.
3. Zu jeder TM  $T$  gibt es eine NLPRAM  $P$ , so dass für alle  $k$  gilt:  
Sind  $c \in C_k$  und  $\Delta_T(c) \in C_k$ , dann kann  $P$  in einer Zeit proportional zu  $\log k$  aus  $\text{cod}_k(c)$  die Codierung  $\text{cod}_k(\Delta_T(c))$  berechnen.



- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Codierung von TM-Konfigurationen

- ▶ Aussagen des Lemmas
  - ▶ ähnlich wie bei PRAM, aber nun
  - ▶ alles in Zeit  $\log k$  (statt in  $k$ )
- ▶ Beweise: nicht besonders schwere Übungen

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Simulation von NTM durch NLPRAM

### Satz

Für alle  $t(n) \geq n$  gilt:

$$\mathbb{W}_1\text{-NTM-TIME}(t(n)) \subseteq \text{NLPRAM-TIME}(O(\log t(n)))$$

Beweis gleich

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Simulation von NTM durch NLPRAM

## Korollar

Für alle  $t(n) \geq n$  gilt:

$$*W_*\text{-NTM-TIME}(t(n)) \subseteq \text{NLPRAM-TIME}(O(\log t(n)))$$

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Simulation von NTM durch NLPRAM

### Korollar

Für alle  $t(n) \geq n$  gilt:

$$*W_*\text{-NTM-TIME}(t(n)) \subseteq \text{NLPRAM-TIME}(O(\log t(n)))$$

### Korollar

$$\mathbf{NP} \subseteq \text{NLPRAM-TIME}(O(\log n))$$

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Simulation von NTM $N$ durch NLPRAM: Algorithmenskizze

►  $h = \lceil \log_2 t_N(n) \rceil$

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Simulation von NTM $N$ durch NLPRAM: Algorithmenskizze

- ▶  $h = \lceil \log_2 t_N(n) \rceil$
- ▶ Baum der Höhe  $h$  von Prozessoren aktivieren
  - ▶ Blätter raten, dass sie die richtige Höhe haben

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Simulation von NTM $N$ durch NLPRAM: Algorithmenskizze

- ▶  $h = \lceil \log_2 t_N(n) \rceil$
- ▶ Baum der Höhe  $h$  von Prozessoren aktivieren
  - ▶ Blätter raten, dass sie die richtige Höhe haben
- ▶ jedes Blatt rät zwei  $N$ -Konfigurationen  $c, c'$ 
  - ▶ falls  $c \vdash_N c'$ :  $(c, c')$  nach oben geben
  - ▶ falls nicht: Fehlermeldung

- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

# Simulation von NTM $N$ durch NLPRAM: Algorithmenskizze

- ▶  $h = \lceil \log_2 t_N(n) \rceil$
- ▶ Baum der Höhe  $h$  von Prozessoren aktivieren
  - ▶ Blätter raten, dass sie die richtige Höhe haben
- ▶ jedes Blatt rät zwei  $N$ -Konfigurationen  $c, c'$ 
  - ▶ falls  $c \vdash_N c'$ :  $(c, c')$  nach oben geben
  - ▶ falls nicht: Fehlermeldung
- ▶ jedes Nichtblatt, das  $(c, c')$  und  $(d, d')$  bekommt:
  - ▶ falls  $c' = d$  ist:  $(c, d')$  nach oben geben
  - ▶ sonst: Fehlermeldung



- └ Eine nichtdeterministische Variante von PRAM
- └ Beziehung zu sequenziellen Turingmaschinen

## Simulation von NLPRAM durch NTM

### Satz

Für alle  $t(n) \geq n$  gilt:

$$\text{NLPRAM-TIME}(t(n)) \subseteq *W_*\text{-NTM-TIME}(16^{t(n)})$$

Der Beweis ist langweilig.

# Zusammenfassung

- ▶ Zeitbedarf von PRAM ist polynomiell verknüpft mit Platzbedarf von TM
  - ▶ **PSPACE**-Probleme in Polynomialzeit lösbar
  - ▶ Preis: viele Prozessoren

# Zusammenfassung

- ▶ Zeitbedarf von PRAM ist polynomiell verknüpft mit Platzbedarf von TM
  - ▶ **PSPACE**-Probleme in Polynomialzeit lösbar
  - ▶ Preis: viele Prozessoren
- ▶ NLPRAM schaffen exponentielle Beschleunigung gegenüber NTM
  - ▶ alle **NP**-Probleme in logarithmischer Zeit lösbar
  - ▶ Preis: viele Prozessoren, Nichtdeterminismus, mächtige Maschinenbefehle

# Zusammenfassung

- ▶ Zeitbedarf von PRAM ist polynomiell verknüpft mit Platzbedarf von TM
  - ▶ **PSPACE**-Probleme in Polynomialzeit lösbar
  - ▶ Preis: viele Prozessoren
- ▶ NLPRAM schaffen exponentielle Beschleunigung gegenüber NTM
  - ▶ alle **NP**-Probleme in logarithmischer Zeit lösbar
  - ▶ Preis: viele Prozessoren, Nichtdeterminismus, mächtige Maschinenbefehle
- ▶ Man überlege, für wie sinnvoll man die Modelle hält!
  - ▶ wir kommen in späterem Kapitel darauf zurück