8 Vorträge

1

Kapitel 7

Komplexitätsklassen

In diesem Kapitel wollen wir untersuchen, welche Beziehungen zwischen den PRAM Klassen EREW(p,t), CREW(p,t), CRCW(p,t) und den Turing-Maschinen Klassen DTIME(f), NTIME(f), DSPACE(f), NSPACE(f) existieren. In Abschnitt 7.1 werden zunächst einige grundlegende Definitionen bzgl. Turing-Maschinen wiederholt. Abschnitt 7.2 zeigt an Beispielen den Zusammenhang zwischen Entscheidungsproblemen und den von Turing-Maschinen akzeptierten Sprachen. Anschließend wird die Technik der Reduktion einer Sprache auf eine andere vorgestellt. In Abschnitt 7.3 werden die PRAM Klassen näher untersucht. Insbesondere wird dort auf die nur schwer parallelisierbaren Probleme eingegangen, welche die Klasse der $\mathcal{P}$-vollständigen Probleme bilden.

7.1 Grundlegende Definitionen bzgl. Turing-Maschinen

Definition 7.1 (Turing-Maschine)

Eine k-Band Turing-Maschine M ist beschrieben durch $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ mit:

Q : endliche Menge von Zuständen
Σ : endliche Menge von Eingabesymbolen
Γ : endliche Menge von Bandsymbolen
δ : Übergangsfunktion mit
$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{-1, 0, +1\}^k$ (det. TM)
$\delta : Q \times \Gamma^k \mapsto \text{Pot}(Q \times \Gamma^k \times \{-1, 0, +1\}^k)$ (nicht-det. TM)
q_0 : Startzustand, $q_0 \in Q$
F : endliche Menge von Endzuständen, $F \subseteq Q$

Definition 7.2 (Konfiguration)

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine k-Band Turing-Maschine. Ein Tupel $(q, w_1, \ldots, w_k, h_1, \ldots, h_k)$ aus $Q \times (\Gamma^*)^k \times \mathbb{N}^k$ heißt Konfiguration von M. Dabei bezeichnet q den Zustand, in dem sich die TM befindet. Die Worte $w_1, \ldots, w_k$ geben den Inhalt der k Arbeitsbänder an. Die Zahlen $h_1, \ldots, h_k$ beschreiben die Positionen der k Leseköpfe auf den k Arbeitsbändern.

Definition 7.3 (akzeptierte Sprache)

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine k-Band Turing-Maschine. Die von M akzeptierte Sprache $L(M) \subseteq \Sigma^*$ ist definiert als:

$$L(M) = \{w \in \Sigma^*;\ (q_0, w, \underbrace{\epsilon, \ldots, \epsilon}_{k-1}, \underbrace{1, \ldots, 1}_{k}) \overset{*}{\longmapsto} (t, \ldots) \text{ mit } t \in F\}$$

Definition 7.4 (Zeit-Komplexitätsklassen)

Sei M eine Turing-Maschine. Für $w \in L(M)$ bezeichne $t_M(w)$ die Anzahl der Schritte, die M bei Eingabe w benötigt, um einen Endzustand zu erreichen. Sei ferner $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann sind die Klassen DTIME(f), NTIME(f) und $\mathcal{P}$, $\mathcal{NP}$ wie folgt definiert:

$$\text{DTIME}(f) := \{L;\ \exists \text{ det. } k\text{-Band TM } M \text{ mit } L(M) = L \text{ und } t_M(w) \leq c \cdot f(|w|)\ \forall w \in L\}$$

$$\text{NTIME}(f) := \{L;\ \exists \text{ nicht-det. } k\text{-Band TM } M \text{ mit } L(M) = L \text{ und } t_M(w) \leq c \cdot f(|w|)\ \forall w \in L\}$$

$$\mathcal{P} := \bigcup_d \text{DTIME}(n^d)$$

$$\mathcal{NP} := \bigcup_d \text{NTIME}(n^d)$$

Definition 7.5 (off-line Turing-Maschine)

Eine k-Band off-line Turing-Maschine ist eine spezielle $(k+1)$-Band Turing-Maschine, die neben den k Arbeitsbändern ein zusätzliches Eingabeband besitzt, auf das nur lesend zugegriffen werden darf. Eine k-Band off-line Turing-Maschine mit Ausgabeband besitzt darüberhinaus ein Band, auf das nur geschrieben werden darf. Der Kopf wandert dabei von links nach rechts.

Definition 7.6 (Platz-Komplexitätsklassen)

Sei M eine off-line Turing-Maschine. Für $w \in L(M)$ bezeichne $s_M(w)$ die Anzahl der Speicherzellen auf den Arbeitsbändern, die M bei Eingabe w benötigt, um einen Endzustand zu erreichen. Sei ferner $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann sind die Klassen DSPACE(f), NSPACE(f) und $\mathcal{L}$, $\mathcal{NL}$ wie folgt definiert:

$$\text{DSPACE}(f) := \{L;\ \exists \text{ det. } k\text{-Band off-line TM } M \text{ mit } L(M) = L \text{ und } s_M(w) \leq c \cdot f(|w|)\ \forall w \in L\}$$

$$\text{NSPACE}(f) := \{L;\ \exists \text{ nicht-det. } k\text{-Band off-line TM } M \text{ mit } L(M) = L \text{ und } s_M(w) \leq c \cdot f(|w|)\ \forall w \in L\}$$

$$\mathcal{L} := \text{DSPACE}(\log n)$$

$$\mathcal{NL} := \text{NSPACE}(\log n)$$

Definition 7.7 (Berechnung einer Funktion)

Sei Σ ein endliches Alphabet und $g : \Sigma^* \to \Sigma^*$ eine Funktion. Eine k-Band off-line Turing-Maschine M berechnet g, falls M auf Eingabe $w \in \Sigma^*$ $g(w)$ auf das Ausgabeband schreibt.

Schreibweise: Sei $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Gilt $s_M(w) \leq c \cdot f(|w|)\ \forall w \in \Sigma^*$, so schreibt man $g \in \text{DSPACE}(f)$, falls M deterministisch und $g \in \text{NSPACE}(f)$, falls M nicht deterministisch.

7.2 Entscheidungsprobleme und Sprachen

Dem Studium der in Abschnitt 7.1 definierten Komplexitätsklassen kommt in der Praxis eine große Bedeutung zu (vgl. [23]). Dies liegt daran, daß jedes kombinatorische Optimierungsproblem als Entscheidungsproblem formuliert werden kann. Solche Probleme besitzen als zulässige Lösungen nur die Antworten Ja oder Nein. Mit Hilfe eines geeigneten Kodierungsverfahrens kann das Entscheidungsproblem wiederum in eine Sprache über einem endlichen Alphabet transformiert werden. Auf diese Weise ist es möglich, Optimierungsprobleme in „Schwierigkeitsklassen“ einzuteilen. Dies soll am Beispiel des *Traveling Salesman Problem (TSP)* veranschaulicht werden. Eingabe des Problems ist eine endliche Menge von Städten $C = \{c_1, \ldots, c_m\}$ und eine Distanzmatrix $D \in M(m, \mathbb{N})$, die für jedes Paar c_i, c_j angibt, wie weit die Städte auseinander liegen. Lösung des Problems ist eine Rundreise $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}, c_{\pi(1)} >$, auf der jede Stadt genau einmal besucht wird und deren Länge

$$\left\{ \sum_{i=1}^{m-1} D(c_{\pi(i)}, c_{\pi(i+1)}) \right\} + D(c_{\pi(m)}, c_{\pi(1)})$$

minimal ist. Das Traveling Salesman Problem kann wie folgt als Entscheidungsproblem formuliert werden:

Gegeben: Eine endliche Menge von Städten $C = \{c_1, \ldots, c_m\}$, eine Distanzmatrix $D \in M(m, \mathbb{N})$ und eine obere Schranke $S \in \mathbb{N}$.

Frage: Gibt es eine Rundreise $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}, c_{\pi(1)} >$, auf der jede Stadt genau einmal besucht wird mit

$$\left\{ \sum_{i=1}^{m-1} D(c_{\pi(i)}, c_{\pi(i+1)}) \right\} + D(c_{\pi(m)}, c_{\pi(1)}) \leq S$$

Jeder Algorithmus, der das Minimierungsproblem in Polynomzeit löst, kann auch das Entscheidungsproblem in Polynomzeit lösen. Hierfür müßte der Algorithmus nach Bestimmung der minimalen Rundreise nur noch zusätzlich ihre Länge berechnen und mit S vergleichen. Aus dieser Beobachtung folgt unmittelbar: das Minimierungsproblem ist mindestens genauso schwer zu lösen wie das Entscheidungsproblem. Kann man also zeigen, daß die Lösung des Entscheidungsproblems nur mit hohem Zeit- oder Platzaufwand berechnet werden kann, so gilt dies erst recht für die Lösung des Optimierungsproblems.

Um das Entscheidungsproblem als Sprache über einem endlichen Alphabet Σ formulieren zu können, werden die Städte in C, die Eintragungen der Distanzmatrix D und die obere Schranke S geeignet kodiert. Dabei wird jede natürliche Zahl als Binärzahl dargestellt. Im folgenden bezeichne $\text{code}(C, D, S)$ diese Kodierung. Dann gilt:

$$\begin{aligned} L_{\text{TSP}} = \{ & \text{code}(C, D, S) \in \Sigma^*; \ \exists \text{ Rundreise, die jede Stadt genau einmal besucht} \\ & \text{mit } \left\{ \textstyle\sum_{i=1}^{m-1} D(c_{\pi(i)}, c_{\pi(i+1)}) \right\} + D(c_{\pi(m)}, c_{\pi(1)}) \leq S \} \end{aligned}$$

Man kann leicht zeigen, daß L_{TSP} eine Sprache aus $\mathcal{NP}$ ist, d.h. $L_{\text{TSP}} \in \text{NTIME}(n^d)$ für ein $d \in \mathbb{N}$. Dabei entspricht n der Länge der Kodierung $\text{code}(C, D, S)$. Ist $z \in \mathbb{N}$ die betragsmäßig größte Zahl in der Distanzmatrix D, so gilt: $n = O(m^2 \cdot \log z)$.

Normalerweise unterscheidet man nicht zwischen dem Entscheidungsproblem und der daraus abgeleiteten Sprache. Statt $L_{\mathrm{TSP}} \in \mathcal{NP}$ schreibt man einfach TSP$\in \mathcal{NP}$. Wir wollen das oben Gesagte an einem weiteren Beispiel verdeutlichen. Das *Graph Accessibility Problem (GAP)* ist wie folgt definiert:

Gegeben: gerichteter Graph $G = (V, E)$ und zwei Knoten $u, v \in V$

Frage: existiert ein Weg von u nach v in G ?

Lemma 7.1

$$\mathrm{GAP} \in \mathcal{NL}$$

Beweis:

Sei $V = \{0\ldots., n-1\}$. Der folgende nicht deterministische Algorithmus rät einen Weg $u = w_1 \to w_2 \to \ldots \to w_r = v$. Dazu werden zwei Hilfsvariablen x, y benötigt. Eine nicht deterministische off-line Turing-Maschine speichert diese Variablen auf zwei Hilfsbändern ab.

nicht deterministischer Algorithmus
var x, y : integer;
begin
  $x := u$;
  while $x \neq v$ do begin
    wähle einen beliebigen Knoten $y \in V$ mit $(x, y) \in E$;
    $x := y$;
  end;
  accept;
end;

Die Eingabelänge beträgt bei einer binären Kodierung der Knoten $N = O(|E| \cdot \log n)$, $|E| \geq n$. Zur Abspeicherung der Hilfsvariablen wird $O(\log n) = O(\log N)$ Platz benötigt. Daher kann der Algorithmus von einer logarithmisch platzbeschränkten, nicht deterministischen off-line Turing-Maschine ausgeführt werden.

■

Bei der Einordnung von Problemen in Komplexitätsklassen ist die Technik der Reduktion eines Problems auf ein anderes von entscheidender Bedeutung. Stephen Cook [24] begründete 1971 mit Hilfe dieser Technik die Theorie der $\mathcal{NP}$-Vollständigkeit. Formal kann die Reduktion eines Problems auf ein anderes wie folgt definiert werden:

Definition 7.8 (Reduktion)

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen. L_1 heißt „log-SPACE“ reduzierbar auf L_2 genau dann, wenn eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$ existiert mit:

1.) $f \in \mathrm{DSPACE}(\log n)$
2.) $w \in L_1 \Leftrightarrow f(w) \in L_2 \quad \forall w \in \Sigma_1^*$

Schreibweise: $L_1 \leq_{\log} L_2$

Durch $\leq_{\log}$ wird eine Relation auf die Probleme einer Komplexitätsklasse induziert. Dies ermöglicht eine Definition der „schwersten“ Probleme einer Klasse.

Definition 7.9 (Vollständigkeit)

Sei $\mathcal{L}_0$ eine Klasse von Sprachen. Eine Sprache L heißt $\mathcal{L}_0$-vollständig (bzgl. $\leq_{\log}$) genau dann, wenn

1.) $L \in \mathcal{L}_0$
2.) $L' \leq_{\log} L \quad \forall L' \in \mathcal{L}_0$

Satz 7.1

GAP ist $\mathcal{NL}$-vollständig

Beweis:

Wir müssen zeigen, daß für jede Sprache $L \in \mathcal{NL}$ gilt: $L \leq_{\log}$ GAP. Sei also L eine beliebige Sprache aus $\mathcal{NL}$. Dann existiert eine nicht deterministische off-line Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ mit $L(M) = L$ und $s_M(w) \leq c \cdot \log|w| \ \forall w \in L$. Sei K_w die Menge aller Konfigurationen, die M bei Eingabe w mit Platzbeschränkung $c \cdot \log|w|$ annehmen kann. O.B.d.A sei angenommen, daß M nur ein Arbeitsband besitzt. Dann haben die Elemente aus K_w die Form (q, h_1, u, h_2) mit

$q \in Q$	Zustand der TM
$1 \leq h_1 \leq \lvert w\rvert$	Kopfposition auf dem Eingabeband
$u \in \Gamma^{\leq c \cdot \log\lvert w\rvert}$	Inschrift auf dem Arbeitsband
$1 \leq h_2 \leq c \cdot \log\lvert w\rvert$	Kopfposition auf dem Arbeitsband

Es folgt: $|K_w| = |Q| \cdot |w| \cdot |\Gamma|^{c \cdot \log|w|} \cdot c \cdot \log|w| \leq |w|^r$ für ein $r \in \mathbb{N}$. Wir konstruieren jetzt einen Graphen $G_w = (K_w, E_w)$ mit

$$E_w = \{(k_1, k_2);\ k_1, k_2 \in K_w \text{ und } k_1 \mapsto k_2\}$$

Ausgezeichnete Knoten des Graphen sind $u = (q_0, 1, \epsilon, 1)$ und $v = (f, 1, \epsilon, 1)$. Knoten u entspricht der Startkonfiguration von M. Desweiteren sei angenommen, daß v die einzige Endkonfiguration ist, d.h. $F = \{f\}$. Das Tupel (G_w, u, v) ist nun Eingabe des Graph Accessibility Problems. Dann gilt:

$$w \in L \Leftrightarrow f(w) = (G_w, u, v) \in \text{GAP}$$

Es verbleibt zu zeigen, daß f mit $O(\log|w|)$ Platz berechnet werden kann. Sei dazu $\widetilde{M}$ eine deterministische off-line Turing-Maschine mit jeweils einem Eingabe-, Arbeits- und Ausgabeband. $\widetilde{M}$ arbeitet auf Eingabe w wie folgt:

1. schreibe auf das Arbeitsband die Startkonfiguration $k = (q_0, 1, \epsilon, 1)$ von M
2. simuliere einen Schritt von M
3. schreibe für alle k' mit $k \mapsto k'$ das Tupel (k, k') auf das Ausgabeband
4. setze $k :=$ lexikographisch nächste Konfiguration und gehe nach 2.

Nach Termination steht auf dem Ausgabeband von $\widetilde{M}$ die Kantenmenge E_w des Graphen G_w. Wegen $|K_w| \leq |w|^r$ gilt: $|E_w| \leq |w|^{2r}$. Die Ausgabe von $\widetilde{M}$ besitzt also polynomielle Länge. Auf dem Arbeitsband von $\widetilde{M}$ ist in jeder Iteration eine Konfiguration von M abgespeichert. Die Konfigurationen enthalten als größten Wert die Inschrift u des Arbeitsbandes von M. Da dieses $c \cdot \log |w|$ platzbeschränkt ist, wird zur Abspeicherung der gesamten Konfiguration ebenfalls nur $O(\log |w|)$ Platz benötigt. Damit ist auch $\widetilde{M}$ $O(\log |w|)$ platzbeschränkt.

■

Die $\leq_{\log}$-Reduktion besitzt eine wichtige Eigenschaft: sie ist transitiv. Diese Eigenschaft vereinfacht den Nachweis, daß eine Sprache L_1 aus einer Klasse $\mathcal{L}_0$ zu den schwersten Sprachen der Klasse gehört. Korollar 7.1 zeigt, daß hierzu lediglich eine Sprache L_2 benötigt wird, von der bereits gezeigt wurde, daß sie $\mathcal{L}_0$-vollständig ist.

Satz 7.2 (Transitivität der Reduktion)

Seien $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ und $L_3 \subseteq \Sigma_3^*$ drei Sprachen mit $L_1 \leq_{\log} L_2$ und $L_2 \leq_{\log} L_3$, dann gilt: $L_1 \leq_{\log} L_3$.

Der Beweis von Satz 7.2 ist dem Leser als Übungsaufgabe überlassen. An dieser Stelle sei jedoch darauf hingewiesen, daß nicht einfach zwei logarithmisch platzbeschränkte, deterministische off-line Turing-Maschinen hintereinandergeschaltet werden können, da die Ausgabe der ersten Turing-Maschine – wie im Beweis zu Satz 7.1 gesehen – polynomielle Länge besitzen kann.

Korollar 7.1

Sei $\mathcal{L}_0$ eine Sprachklasse und seien L_1, L_2 zwei Sprachen aus $\mathcal{L}_0$. Sei weiter L_2 eine $\mathcal{L}_0$-vollständige Sprache und $L_2 \leq_{\log} L_1$. Dann ist auch L_1 eine $\mathcal{L}_0$-vollständige Sprache.

Beweis:

Da L_2 $\mathcal{L}_0$-vollständig ist, gilt für alle Sprachen $L' \in \mathcal{L}_0$: $L' \leq_{\log} L_2$. Wegen $L_2 \leq_{\log} L_1$ und aufgrund der Transitivität von $\leq_{\log}$ folgt sofort: $L' \leq_{\log} L_1$ $\forall L' \in \mathcal{L}_0$.

■

Als nächstes wird definiert, was unter der Abgeschlossenheit einer Sprachklasse gegenüber der $\leq_{\log}$-Reduktion zu verstehen ist. Satz 7.3 zeigt eine wichtige Eigenschaft abgeschlossener Sprachklassen.

Definition 7.10 (Abgeschlossenheit einer Sprachklasse)

Sei $\mathcal{L}_0$ eine Sprachklasse. $\mathcal{L}_0$ heißt abgeschlossen gegenüber der $\leq_{\log}$-Reduktion, wenn aus $L_1 \in \mathcal{L}_0$ und $L_2 \leq_{\log} L_1$ folgt: $L_2 \in \mathcal{L}_0$.

Satz 7.3 (Eigenschaft abgeschlossener Sprachklassen)

Seien $\mathcal{L}_0, \mathcal{L}_0'$ zwei Sprachklassen. Sei $\mathcal{L}_0$ abgeschlossen gegenüber der $\leq_{\log}$-Reduktion. Sei weiter L eine $\mathcal{L}_0'$-vollständige Sprache. Dann folgt aus $L \in \mathcal{L}_0$ sofort: $\mathcal{L}_0' \subseteq \mathcal{L}_0$.

Satz 7.4

Die Sprachklassen $\mathcal{L}, \mathcal{NL}, \mathcal{P}, \mathcal{NP}$ sind abgeschlossen gegenüber der $\leq_{\log}$-Reduktion.

Beweis:

Wir zeigen an dieser Stelle lediglich, daß $\mathcal{P}$ gegenüber der $\leq_{\log}$-Reduktion abgeschlossen ist. Seien also $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen mit $L_1 \in \mathcal{P}$ und $L_2 \leq_{\log} L_1$. Dann existiert eine deterministische Turing-Maschine M_1, die in Zeit $c \cdot n^d$ die Sprache L_1 akzeptiert. Wegen $L_2 \leq_{\log} L_1$ existiert eine Funktion $f : \Sigma_2^* \to \Sigma_1^*$ mit $f \in \text{DSPACE}(\log n)$ und $w \in L_2 \Leftrightarrow f(w) \in L_1$. Da $\text{DSPACE}(\log n) \subseteq \mathcal{P}$, kann f von einer Turing-Maschine M' in Polynomzeit berechnet werden. Aus M_1 und M' kann eine Turing-Maschine M_2 konstruiert werden, die L_2 akzeptiert. M_2 arbeitet wie folgt:

> Auf Eingabe $w \in \Sigma_2^*$, berechnet M_2 zunächst $f(w)$. Dies ist in Zeit $|w|^r$ möglich. Anschließend simuliert M_2 die Turing-Maschine M_1 auf Eingabe $f(w)$. Wegen $|f(w)| \leq |w|^r$ werden hierfür $c \cdot (|w|^r)^d = c \cdot |w|^{r \cdot d}$ Zeiteinheiten gebraucht.

Insgesamt folgt: $L(M_2) = L_2$ und $L_2 \in \mathcal{P}$.

■

7.3 Eigenschaften der PRAM Klassen

In diesem Abschnitt werden einige Beziehungen zwischen den in Abschnitt 7.1 definierten Sprachklassen und den in Abschnitt 2.2 definierten PRAM Klassen vorgestellt. Wir definieren zunächst eine neue Sprachklasse, welche alle „gut parallelisierbaren“ Probleme umfaßt. Diese wurde zum ersten Mal von Nick Pippenger erwähnt und trägt deshalb den Namen *Nick's Class* oder kurz $\mathcal{NC}$.

Definition 7.11 (Nick's Class)

$$\mathcal{NC} = \bigcup_{p,q} \text{CRCW}(n^p, (\log n)^q)$$

Mit Hilfe des nachfolgenden Satzes kann man leicht zeigen, daß auch die Klasse $\mathcal{NC}$ gegenüber der $\leq_{\log}$-Reduktion abgeschlossen ist.

Satz 7.5

Sei $f : \mathbb{N} \to \mathbb{R}$, $f(n) \geq \log n$, eine Funktion. Es gilt:

(1) $\text{DSPACE}(f) \subseteq \bigcup_d \text{CREW}(d^{f(n)}, f(n))$

(2) $\text{NSPACE}(f) \subseteq \bigcup_d \text{CRCW}(d^{f(n)}, f(n))$

Beweis:

Sei zunächst $L \in \text{DSPACE}(f)$ beliebig und sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine deterministische off-line Turing-Maschine mit $L(M) = L$ und $s_M(w) \leq c \cdot f(|w|)\ \forall w \in L$. Wie im Beweis zu Satz 7.1 gesehen, besteht K_w aus Tupeln der Form (q, h_1, u, h_2). Für $|w| = n$ gilt: $|K_w| = |Q| \cdot n \cdot |\Gamma|^{c \cdot f(n)} \cdot c \cdot f(n) \leq d^{f(n)}$, für ein $d \in I\!N$. Der Graph $G_w = (K_w, E_w)$ mit $E_w = \{(k_1, k_2);\ k_1, k_2 \in K_w \text{ und } k_1 \mapsto k_2\}$ besitzt also höchstens $d^{f(n)}$ Knoten. Mit $d^{2 \cdot f(n)}$ Prozessoren kann E_w berechnet werden. Dabei wird E_w als Adjazenzmatrix A dargestellt. Jedem Prozessor wird ein Konfigurationspaar (k_i, k_j) zugeordnet. Der Prozessor bestimmt die Nachfolgekonfiguration von k_i und vergleicht diese mit k_j. Stimmen beide überein, so setzt er $A[k_i, k_j] = 1$ und $A[k_i, k_j] = 0$ sonst. Für den Vergleich benötigt der Prozessor $O(f(n))$ Zeiteinheiten (Vergleich der Inschrift auf dem Arbeitsband). Wir haben gesehen, daß $w \in L$ genau dann gilt, wenn es in G_w von dem Startknoten $u = (q_0, 1, \epsilon, 1)$ einen Weg zu dem Endknoten $v = (f, 1, \epsilon, 1)$ gibt. Da M deterministisch, hat jeder Knoten in G_w den Ausgangsgrad ≤ 1. In diesem Fall kann auf einer CREW mit $|K_w|^3$ Prozessoren in Zeit $O(\log |K_w|)$ bestimmt werden, ob ein Weg zwischen u und v in G_w existiert (Bildung des transitiven Abschluß). Wegen $|K_w| \leq d^{f(n)}$, folgt also insgesamt: $L \in \text{CREW}(d^{3 \cdot f(n)}, f(n))$. Gilt $L \in \text{NSPACE}(f)$, so hat eine Konfiguration in der Regel mehrere Nachfolgekonfigurationen. Daher können bei der Bildung des transitiven Abschluß Schreibkonflikten auftreten. In diesem Fall gilt daher: $L \in \text{CRCW}(d^{3 \cdot f(n)}, f(n))$.

■

Nach Satz 7.5 gilt $\text{DSPACE}(\log n) \subseteq \bigcup_d \text{CREW}(n^d, \log n)$ und $\mathcal{NL} \subseteq \bigcup_d \text{CRCW}(n^d, \log n)$. Daher folgt:

Korollar 7.2

$\mathcal{NC}$ ist abgeschlossen gegenüber $\leq_{\log}$

Korollar 7.3

$\mathcal{NL} \subseteq \mathcal{NC}$

Bezeichne im folgenden $\text{CRCW}(p(n), t(n), z(n))$ die Klasse aller Funktionen, die auf einer CRCW-PRAM mit $O(p(n))$ Prozessoren in Zeit $O(t(n))$ berechnet werden können, wobei alle während der Berechnung auftretenden Zahlen $\leq z(n)$ sind. D.h. $z(n)$ ist die größte Zahl, die bei einer Eingabe der Länge n erzeugt wird. Dann gilt:

Satz 7.6

$$\text{CRCW}(p(n), t(n), z(n)) \subseteq \text{DSPACE}(t(n) \cdot (\log p(n) + \log t(n) + \log z(n)))$$

Beweis: (nur für EREW-PRAM)

Sei M eine EREW-PRAM. Wir betrachten ein beliebiges PRAM-Programm, welches von einem Compiler in die auf Seite 9 beschriebenen, elementaren Instruktionen von M übersetzt worden ist. Dann gilt:

- den Variablen und Datenstrukturen sind einzelne Speicherzellen (Register) zugeordnet
- Schleifen sind zu GOTO-Anweisungen übersetzt worden

Betrachte jetzt die Funktionen

$\pi(p,t)$: Stellung des Programmzählers von Prozessor p zum Zeitpunkt t
$x(i,t)$: Inhalt des Registers $x[i]$ zum Zeitpunkt t

mit den Anfangsbedingungen

$\pi(p,1) = 1, \quad 1 \leq p \leq p(n)$
$x(i,0) \cong$ „Eingabe“, $\quad 1 \leq i \leq n, \quad$ /* $x[1] \ldots x[n]$ enthalten die Eingabe */

und folgender rekursiver Definition

1.) falls $\pi(p,t-1) \cong$ „$x[i] := x[j] \otimes x[k]$“, $\quad \otimes \in \{+,-,*,/\}$

$x(i,t) = x(j,t-1) \otimes x(k,t-1)$
$\pi(p,t) = \pi(p,t-1) + 1$

2.) falls $\pi(p,t-1) \cong$ „$x[i] := x[x[j]]$“

$x(i,t) = x(x(j,t-1),t-1)$
$\pi(p,t) = \pi(p,t-1) + 1$

3.) falls $\pi(p,t-1) \cong$ „$x[x[i]] := x[j]$“

$x(x(i,t-1),t) = x(i,t-1)$
$\pi(p,t) = \pi(p,t-1) + 1$

4.) falls $\pi(p,t-1) \cong$ „if $x[i] = 0$ then goto m“

$$\pi(p,t) = \begin{cases} \pi(p,t-1)+1, & \text{falls } x(i,t-1) \neq 0 \\ m, & \text{falls } x(i,t-1) = 0 \end{cases}$$

Eine deterministische off-line Turing-Maschine, deren endliches Gedächnis das PRAM Programm von M enthält, kann die Funktionen $\pi(p,t)$ und $x(i,t)$ für alle $1 \leq p \leq p(n)$, $1 \leq t \leq t(n)$, $1 \leq i \leq z(n)$ berechnen. Beachte: es gilt stets $i \leq z(n)$, da der von M benutzte Speicher beschränkt ist durch den größten während einer Berechnung auftretenden Registerinhalt (vgl. die indirekten Lade- und Speicheranweisungen „$x[i] := x[x[j]]$“ und „$x[x[i]] := x[j]$“).

Wir zeigen jetzt, daß die Funktionswerte $\pi(p,t), x(i,t)$ mit Platz $d \cdot t \cdot (\log p(n) + \log t(n) + \log z(n))$ berechnet werden können. Betrachte dazu die Fallunterscheidungen 1. bis 4. Zur Berechnung eines Funktionswertes werden höchstens c Funktionswerte zum Zeitpunkt $t-1$ benötigt. Diese c Funktionswerte werden nacheinander wie folgt berechnet:

(a) Lege die aktuellen Parameter p, t, i in einer Rekursionsschachtel ab
(b) Berechne den Funktionswert zum Zeitpunkt $t-1$

Für jede Rekursionsschachtel wird Platz $d \cdot (\log p(n) + \log t(n) + \log z(n))$ benötigt. Bei einer Rekursionstiefe von t ergibt sich so ein Platzbedarf von $d \cdot t \cdot (\log p(n) + \log t(n) + \log z(n))$.

■

Korollar 7.4

$$\bigcup_{p,q,r} \mathrm{CRCW}(n^p, (\log n)^q, n^r) \subseteq \bigcup_{d} \mathrm{DSPACE}((\log n)^d) \quad \text{und insbesondere}$$
$$\mathrm{CRCW}(n^p, \log n, n^r) \subseteq \mathrm{DSPACE}((\log n)^2)$$

Alle sequentiellen Berechnungsmodelle, die in der Literatur vorgeschlagen worden sind (z.B. Turing-Maschinen, Registermaschinen oder μ-rekursive Funktionen), haben sich als gleich mächtig erwiesen. Man kann also prinzipiell alle Aufgaben, die ein realer Computer bewältigt, auch mit einer Turing-Maschine lösen. Diese Erkenntnis hat die Bezeichnung *Church'sche These* erhalten. Ein Analogon bezogen auf parallele Berechnungsmodelle lautet:

Parallel Computation Thesis:

Sei $\mathcal{M}$ ein „vernünftiges" Berechnungsmodell. Bezeichne $\mathcal{M}(p(n), t(n))$ die Klasse aller Probleme, die auf dem Berechnungsmodell mit $O(p(n))$ Prozessoren in Zeit $O(t(n))$ berechnet werden können. Dann gilt:

$$\bigcup_{p,q} \mathcal{M}(n^p, (\log n)^q) \subseteq \bigcup_{d} \mathrm{DSPACE}((\log n)^d)$$

Unter der Annahme, daß die in einem PRAM Programm auftretenden Zahlen höchstens polynomiell zur Länge der Eingabe anwachsen, kann man nach Korollar 7.4 schreiben: $\mathcal{NC} \subseteq \bigcup_d \mathrm{DSPACE}((\log n)^d)$. Es wird vermutet, daß $\mathcal{P}$ keine Teilmenge von $\bigcup_d \mathrm{DSPACE}((\log n)^d)$ ist. Unter dieser Annahme folgt für eine $\mathcal{P}$-vollständige Sprache L:

$$L \notin \bigcup_{d} \mathrm{DSPACE}((\log n)^d) \text{ und damit } L \notin \mathcal{NC}$$

Gelingt es jedoch für eine $\mathcal{P}$-vollständige Sprache L ein nach Definition 2.2 effizientes PRAM Programm anzugeben, so folgt aufgrund der Abgeschlossenheit von $\mathcal{NC}$ nach Satz 7.3 sofort:

$$\mathcal{P} \subseteq \mathcal{NC} \subseteq \bigcup_{d} \mathrm{DSPACE}((\log n)^d)$$

Die Frage $\mathcal{P} \stackrel{?}{\subseteq} \mathcal{NC}$ ist also von ähnlicher Bedeutung wie die Frage $\mathcal{NP} \stackrel{?}{\subseteq} \mathcal{P}$. Abbildung 35 veranschaulicht nocheinmal die vermutete Lage der Komplexitätsklassen.

Beispiele für $\mathcal{P}$-vollständige Probleme sind:

- Wert eines Schaltkreises (Ladner; 1975)
- Berechnung des maximalen Flusses bei exponentieller Kantenbewertung
- Lexikographische Tiefensuche (Reif; 1985)
- Berechnung einer lexikographisch maximalen unabhängigen Menge (Cook; 1985)
- Lineares Programmieren (Doblin,Lipton,Reiss; 1979)

Zum Abschluß des Kapitels zeigen wir, daß die (konkurrierenden) Schreibzugriffe einer CRCW PRAM sehr effizient von einer CREW PRAM mit derselben Prozessorenanzahl simuliert werden können. Unter Verwendung des optimalen Sortieralgorithmus von Cole verursacht die Simulation lediglich einen logarithmischen Zeitverlust.

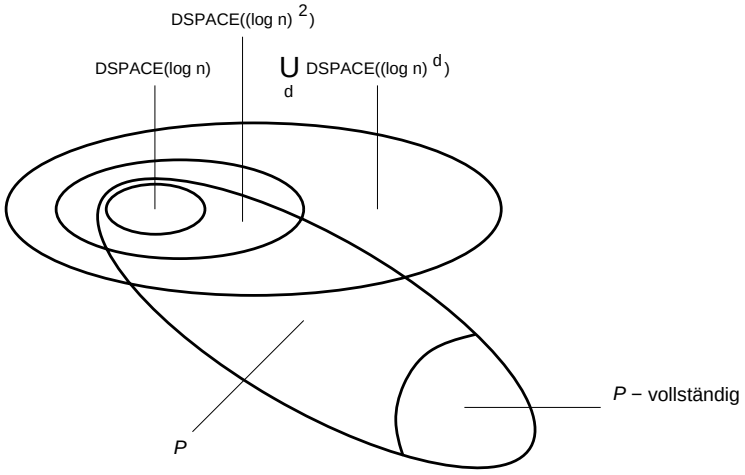

Abbildung 35: *Lage der Klassen unter der Annahme* $\mathcal{P} \not\subseteq \bigcup_d DSPACE((\log n)^d)$

Satz 7.7

$$\text{CRCW}(p(n), t(n)) \subseteq \text{CREW}(p(n), t(n) \cdot \log p(n))$$

Beweis:

Sei M eine CRCW-PRAM, die mit $p(n)$ Prozessoren in Zeit $t(n)$ arbeitet. Wir konstruieren daraus eine CREW-PRAM $\widetilde{M}$, die einen Schritt von M in $\log p(n)$ Schritten simulieren kann. Sei $N = p(n)$. Definiere drei Hilfsarrays B, H, W : array$[0 \ldots N-1]$ of integer. Möchte innerhalb einer parallelen for-Schleife Prozessor p_i, $0 \leq p_i < N$, schreibend auf die Speicherzelle s_i des globalen Speichers zugreifen, so wird die Anweisung

0. for $0 \leq i < N$ parallel do
1. $\quad A[s_i] := x_i$;

von der CREW-PRAM $\widetilde{M}$ wie folgt abgearbeitet.

0. for $0 \leq i < N$ parallel do
1. $\quad H[i] := x_i$;
2. $\quad W[i] := (s_i, i)$;
3. $\quad$ SORT(W);
4. $\quad$ CHECK(W, H, B);
5. $\quad$ ODER(B);
6. $\quad$ if $B[0] = 0$ then SWRITE(W, H);
   $\quad$ else Abbruch;

Wir werden im folgenden auf die einzelnen Schritte näher eingehen.

ad 1.:

Zunächst wird in $H[i]$ abgespeichert, was Prozessor i schreiben möchte.

ad 2.:

In $W[i]$ wird durch das Tupel (s_i, i) abgespeichert, wohin Prozessor i schreiben möchte. Das Tupel (s_i, i) wird kodiert durch die Zahl $s_i \cdot N + i$.

ad 3.:

Das Array W wird mit Hilfe des Algorithmus von Cole sortiert. Nach der Sortierung stehen die Tupel aller Prozessoren, die auf die gleiche Speicherzelle zugreifen wollen, hintereinander in W.

ad 4.:

Das Unterprogramm CHECK testet, ob Schreibkonflikte existieren. Dazu untersucht Prozessor i die Einträge $W[i]$ und $W[i+1]$. Wird ein Schreibkonflikt entdeckt, der von der COMMON-CRCW M nicht hätte aufgelöst werden können. so wird $B[i]$ auf 1 gesetzt, sonst auf 0. Formal kann das Unterprogramm CHECK wie folgt beschrieben werden:

```
s_i := W[i] div N;                 /* dekodiere Tupel in W[i] */
p_i := W[i] mod N;
s_{i+1} := W[i+1] div N;           /* dekodiere Tupel in W[i+1] */
p_{i+1} := W[i+1] mod N;
if (s_i = s_{i+1}) then            /* p_i, p_{i+1} möchten auf dieselbe Speicherzelle zugreifen */
  if (H[p_i] = H[p_{i+1}]) then    /* beide Prozessoren schreiben dasselbe */
    B[i] := 0;
  else
    B[i] := 1;                     /* Schreibkonflikt könnte von M nicht aufgelöst werden */
```

ad 5.:

Über die Einträge $B[i]$ wird das logische Oder gebildet.

ad 6.:

Nur wenn alle Schreibkonflikte von der COMMON-CRCW M hätten aufgelöst werden können, überträgt $\widetilde{M}$ die Einträge von H nach A. Das Unterprogramm SWRITE kann wie folgt beschrieben werden:

```
s_i := W[i] div N;
p_i := W[i] mod N;
s_{i+1} := W[i+1] div N;
if (s_i ≠ s_{i+1}) then
  A[p_i] := H[p_i];
```

Beachte: von allen Prozessoren, die schreibend auf s_i zugreifen wollen, schreibt derjenige Prozessor, dessen Tupel in der Sortierung ganz hinten steht.

Die CREW-PRAM $\widetilde{M}$ benötigt zur Ausführung der Schritte 1,2,4,6 jeweils konstante Zeit. Die Schritte 3 und 5 können jeweils in Zeit $O(\log N)$ ausgeführt werden.

■