Das GCA-Modell im Vergleich zum PRAM-Modell[1]

Andre Osterloh
Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Rechnerarchitektur
Hochschulstraße 10
64289 Darmstadt

Jörg Keller
FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Parallelität und VLSI
Universitätsstraße 1
58084 Hagen

[1]Unterstützt durch Deutsche Forschungsgemeinschaft, Projekt HO 893/7-1 „Massiv-parallele Systeme zur Globalen Zellularen Verarbeitung“.

Kapitel 1

Einleitung

Das PRAM-Modell [9] ist eines der bekanntesten Modelle der theoretischen Informatik zur Beschreibung und Analyse von parallelen Algorithmen. Es gilt allerdings als realitätsfern, weil Simulationen auf realistischen parallelen Rechnerarchitekturen sehr aufwändig sind.

In den letzten Jahren wurde das Modell des Globalen Zellularautomaten [14] entwickelt, bei dem die effiziente Implementierbarkeit in Hardware von Beginn an mit im Fokus stand. Die Einordnung des GCA-Modells in die Berechnungshierarchien der bekannten Modelle stand bisher allerdings aus.

In diesem Bericht wird untersucht, wie mächtig das GCA-Modell im Vergleich zur PRAM ist, und wie sich PRAM-Algorithmen effizient auf das GCA-Modell abbilden lassen. Mit der Beantwortung dieser Frage wird nämlich die Fülle der PRAM-Algorithmen für den GCA zugänglich.

Die Berechnungsmächtigkeit des GCAs im Sinne der Berechenbarkeitstheorie betrachtet Kapitel 2.2. Eine untere Schranke für die zur Lösung von Problemen nötige Anzahl der Generationen eines a-armigen GCAs gibt Lemma 2.7 in Kapitel 2.3.1. Der Vergleich des GCAs mit verschiedenen PRAM-Modellen findet ausführlich in Kapitel 2.3.1 statt. Das Hauptergebnis ist, dass ein Algorithmus, der auf einer Priority CRCW PRAM mit p Prozessoren t Schritte und m Speicherstellen benötigt, auf einem 1-armigen GCA mit $O(\max\{p, m\})$ Zellen in $O(t \log p)$ Generationen abgearbeitet werden kann. Außerdem zeigt Lemma 2.7, dass der erreichte Slowdown von $\log p$ selbst bei einer Simulation von EREW PRAMs durch 1-armige GCAs im Allgemeinen nicht mehr verbessert werden kann. Die in dem Kapitel geführten Beweise geben auch Antwort darauf mit welchen Techniken PRAM-Algorithmen effizient auf dem GCA simuliert werden können. In Kapitel 3 werden die bisher implementierten GCA-Algorithmen mit PRAM-Algorithmen verglichen.

Bezüglich der Simulation von mächtigeren GCA-Varianten auf einfacheren Varianten gibt das Lemma 2.7 erste Hinweise.

Die hier beschriebenen Untersuchungen fanden im Rahmen des von der Deutschen Forschungsgemeinschaft geförderten Projekts *Massiv-parallele Systeme zur Globalen Zellularen Verarbeitung* (HO 893/7-1) statt.

Kapitel 2

Globaler Zellularer Automat (GCA)

2.1 Einführung

2.1.1 Das Modell des GCAs

Zur Beschreibung des Modells wird an dieser Stelle die in [14] gegebene Modellbeschreibung wiedergegeben:

> Das GCA-Modell kann durch folgende Eigenschaften charakterisiert werden:
>
> 1. Der GCA besteht aus einem m-dimensionalen Zellfeld. Jede Zelle kann eindeutig durch ihre Raumkoordinaten identifiziert werden.
> 2. Jede Zelle hat n individuelle globale Nachbarn. Die Nachbarn sind durch relative Koordinaten zur Zentrumszelle definiert.
> 3. Der Zustand einer Zelle besteht aus einem Datenfeld und n Adressfeldern, $State = (Data, p_1, \ldots, p_n)$.
> 4. Jede Zelle hat über die Pointer Lesezugriff auf beliebig entfernte Nachbarn.
> 5. Jede Zelle enthält eine lokale Regel, die den nächsten Zustand in Abhängigkeit von dem eigenen Zustand und den Zuständen der Nachbarn definiert. Durch eine Zustandsänderung können nicht nur die Daten sondern auch die Adressen neu berechnet werden, so dass in der nächsten Generation auf andere Nachbarn zugegriffen werden kann.
> 6. Die Zellzustände werden synchron neu berechnet, die neue Zellengeneration hängt nur von der alten Zellgeneration ab.
> 7. Das Modell ist massivparallel, weil alle neuen Zellzustände parallel berechnet werden können.
> 8. Raum- und zeitabhängige Regeln können durch geeignete zusätzliche Zustandsbits implementiert werden.

Die oben gemachten Festlegungen bieten einen breiten Interpretationsspielraum. Interpretiert man den GCA als eine PRAM, dann kann deren Speicher mitsamt der Prozessoren als 1-dimensionales Zellfeld aufgefasst werden. Der Befehlssatz der Prozessoren einer PRAM (siehe Kapitel 4) ist in den lokalen Regeln kodiert. Diese Regeln sind zeit- und raumabhängig. Die Anzahl der globalen Nachbarn ist durch eins nach oben beschränkt, da ein Prozessor einer PRAM in einem Schritt maximal von einer Speicherstelle etwas lesen bzw. in eine Speicherstelle etwas schreiben darf. Die Registerinhalte der Prozessoren bzw. die Inhalte der Speicherstellen werden in den Zuständen der Zellen kodiert. Simuliert man eine PRAM mit endlicher Registergröße, endlicher Speicherstellengröße, endlicher Speicherstellen- und endlicher Prozessoranzahl, dann ist der Zustandsraum einer Zelle endlich. Dieses war aber in den Festlegungen nicht gefordert. Da Zellen nur den eigenen Zustand verändern dürfen, ist man auf CROW PRAMs (siehe Kapitel 2.1.2) beschränkt. In [10] wird die Realisierung verschiedener Varianten des Modells diskutiert.

2.1.2 Das Modell der PRAM

Die PRAM (parallel random-access machine) ist ein synchrones Modell zur Analyse von parallelen Algorithmen. Eine PRAM besteht aus einer Menge von Prozessoren und einem globalen Speicher. Zusätzlich besitzt jeder Prozessor noch einen lokalen Speichen (Register) und kennt seine eigene Adresse. Ein Beispiel für einen Befehlssatz eines PRAM-Prozessors ist in Kapitel 2.1.2 zu finden. Bei der Behandlung von gleichzeitigen Zugriffen mehrerer Prozessoren auf eine Speicherstelle des gemeinsamen Speichers unterscheidet man verschiedene Modelle. Bei gemeinsamen Lese- und Schreibzugriffen existieren die folgenden Modi:

CR **C**oncurrent **R**ead: Beliebig viele Prozessoren dürfen in einem Schritt von einer Speicherstelle lesen.

ER **E**xlusive **R**ead: In einem Schritt darf maximal ein Prozessor von einer Speicherstelle lesen.

OR **O**wner **R**ead: Nur der Prozessor mit Leserecht auf die Speicherstelle darf von der Speicherstelle lesen. Für jede Speicherstelle besitzt maximal ein Prozessor das Leserecht.

CW **C**oncurrent **W**rite: Beliebig viele Prozessoren dürfen in einem Schritt auf eine Speicherstelle schreiben. Hier muss noch festgelegt werden was passieren soll, wenn verschiedene Werte in die Speicherstelle geschrieben werden sollen. Man unterscheidet die folgenden Untertypen:

- **common**: Ein gleichzeitiges Schreiben auf einer Speicherstelle ist nur erlaubt, wenn alle Prozessoren den gleichen Wert schreiben.
- **arbitrary**: Ein beliebiger von den Schreibwünschen wird in die Speicherstelle geschrieben.
- **priority**: Der Prozessor mit der höchsten Priority darf seinen Wert in die Speicherstelle schreiben.

EW **E**xclusive **W**rite: In einem Schritt darf maximal ein Prozessor auf eine Speicherstelle schreiben.

OW **O**wner **W**rite: Nur der Prozessor mit Schreibrecht auf die Speicherstelle darf auf die Speicherstelle schreiben. Für jede Speicherstelle besitzt maximal ein Prozessor das Schreibrecht.

Eine ausführlichere Diskussion des PRAM Modells kann man beispielsweise in [9, 17, 24, 26] finden.

2.2 Mächtigkeit des Modells

Aus der Sicht der Komplexitäts- und Berechenbarkeitstheorie ist das Modell des GCAs, so wie es in [13, 14, 16] eingeführt wurde, sehr mächtig. In [10] wurde gezeigt, dass jede turingberechenbare Funktion auf einem GCA berechnet werden kann. Da Turingmaschinen und PRAMs gleichmächtige Modelle im Sinne der Berechenbarkeit sind, kann somit jedes Problem, das auf einer PRAM gelöst werden kann, auch auf einem GCA gelöst werden. Benutzt man alle Freiheiten, die die Modellbeschreibung des GCAs bietet, so kann jede turingberechenbare k-stellige Funktion $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ in $O(1)$ Generationen auf einem GCA berechnet werden. Dabei gehen wir davon aus, dass die k Eingaben der Funktion f anfangs in den Zellen 1 bis k des GCAs gespeichert sind und dass die Ausgabe in Zelle 1 erfolgen soll. Der entsprechende Algorithmus dazu ist sehr simpel, beispielsweise könnte die Zelle 1 eines GCAs mit k Zellen, folgendes Programm ausführen:

(1) Lese die Inhalte $i_1, \ldots, i_k$ der Zellen 1 bis k.

(2) Berechne $f(i_1, \ldots, i_k)$ und speichere das Ergebnis in Zelle 1.

Schritt (1) kann auf einem GCA mit k Zellen in $O(1)$ Generationen ausgeführt werden, da auf einem solchen GCA in einer Generation das Lesen der Zustände von allen $k - 1$ Nachbarzellen, sowie das Lesen des eigenen Zustands möglich ist. Da in der Beschreibung des GCAs in [13, 14, 16] keine Einschränkungen angegeben wurden, was durch

die dort eingeführte Zustandsüberführungsfunktion $Rule$ berechnet werden darf, gehen wir an dieser Stelle davon aus, dass $Rule$ jede berechenbare Funktion sein darf. Somit kann auch Schritt (2) in $O(1)$ Generationen auf einem GCA berechnet werden.

Dieser Ansatz läßt sich leicht auf mehrwertige Funktionen anwenden. Im Fall, dass eine turingberechenbare Funktion $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ gegeben ist, gehen wir wieder davon aus, dass die Eingabe in den Zellen 1 bis k steht. Die Ausgabe der i-te Komponente der Funktion f, $1 \leq i \leq m$, soll in Zelle i erfolgen. Die Zelle i eines GCAs mit $\max\{k, m\}$ Zellen, führt dann das folgende Programm aus:

(1) Lese die Inhalte $i_1, \ldots, i_k$ der Zellen 1 bis k.

(2) Berechne die i-te Komponente von $f(i_1, \ldots, i_k)$ und speichere das Ergebnis in Zelle i.

Zusammenfassend erhält man das folgende Ergebnis:

Theorem 2.1 *Eine turingberechenbare Funktion $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ kann auf einem $k-1$-armigen GCA mit $\max\{k, m\}$ Zellen in $O(1)$ Generationen berechnet werden.*

Ein GCA wie in Theorem 2.1 ist nicht besonders praxisnah und auch nicht dazu geeignet eine tiefere Einsicht in die Mächtigkeit des GCA Modells unter realistischeren Voraussetzungen zu erlangen. Um also das GCA Modell mit anderen Berechenbarkeitsmodellen vergleichen zu können, ist die Untersuchung der Fähigkeiten des GCAs unter bestimmten Beschränkungen nötig. Die obige Diskussion legt nahe, dass die Anzahl der Zellen auf die eine Zelle in einer Generation zugegriffen werden darf und die Möglichkeiten, was von einer Zelle in einer Generation berechnet werden darf, beschränkt werden müssen. Aber auch andere Faktoren, wie beispielsweise die Anzahl der Zellen oder die Menge der in einer Zelle gespeicherten Informationen, können Einfluß haben.

2.3 GCA im Vergleich zu anderen parallelen Modellen

2.3.1 GCA und PRAM

Im Gegensatz zur PRAM besitzt ein GCA keinen gemeinsamen Speicher. Somit muss eine Lösung gefunden werden, wie Zugriffe auf den gemeinsamen Speicher der PRAM simuliert werden können.

Bei genauerer Betrachtung der Beschreibung eines GCAs [13, 14, 16], stellt man fest, dass eine Zelle eines GCAs in einem Schritt gleichzeitig auf alle anderen Zellen *lesend* zugreifen darf. Somit ist die Simulation eines lesenden Speicherzugriffs einer PRAM (sowohl im Fall CR[1] als auch in den Fällen ER[2] und OR[3]) kein Problem. Da ein Prozessor einer PRAM in einem Schritt nur auf eine Speicherzelle lesend zugreifen darf, benötigt man für die Simulation der Lesezugriffe nur einen 1-armigen GCA. Im Folgenden gehen wir davon aus, dass sowohl jeder der p Prozessoren der PRAM als auch jede der m benutzten Speicherstellen durch eine Zelle des GCAs simuliert wird. Die Zellen, die die Prozessoren simulieren nennen wir Prozessorzellen, die anderen Speicherzellen. Somit würde ein PRAM Algorithmus der m Speicherplätze und p Prozessoren benutzt auf einem GCA mit $p + m$ Zellen simuliert werden. Aus den obigen Betrachtungen zum GCA folgt sofort, dass alle Befehle der PRAM, die keine Schreibbefehle sind, von den Prozessorzellen ohne Zeitverlust simuliert werden können.

Das Schreiben der PRAM auf den gemeinsamen Speicher stellt bei der Simulation ein größeres Problem dar. Ein Prozessor der PRAM darf auf jede der m Speicherzellen schreibend zugreifen, eine Zelle eines GCAs darf nur auf den eigenen Speicher schreibend zugreifen.

Ein wichtiger Teilaspekt beim Simulieren von Schreibzugriffen ist die Frage, wie bei Schreibkonflikten verfahren wird. Ist zur Lösung eines Schreibkonfliktes eine sehr komplexe Funktion zu berechnen, dann beeinträchtigt dies die

[1] concurrent read
[2] exclusive read
[3] owner read

Effizienz der Simulation. Wir gehen im Folgenden davon aus, dass die zur Lösung von Schreibkonflikten nötigen Prioritäten allen Zellen bekannt sind. Sie sind als Feld $Prio[\cdot]$ gegeben mit $\{Prio[1], \ldots, Prio[p]\} = \{1, \ldots, p\}$. Zur Vereinfachung legen wir fest, dass immer der Prozessor mit der höchsten Nummer gewinnt, also $Prio[p] > Prio[p-1] > \cdots > Prio[1]$. Durch Umnummerierung der Prozessoren kann dieses immer erreicht werden.

Eine Möglichkeit der Simulation eines Schreibbefehls kann dadurch erreicht werden, dass alle m Speicherzellen des GCAs parallel die p Prozessorzellen nacheinander abfragen, ob ein Schreibwunsch vorliegt. Schreibkonflikte können dann durch die gewählte Abfragereihenfolge der p Prozessorzellen gelöst werden.

Der Preis dieser einfachen Methode ist die erhöhte Laufzeit. Ein Schritt einer PRAM vom Typ Priority CRCW mit p Prozessoren muss mit $O(p)$ Generationen eines 1-armigen GCAs mit $m + p$ Zellen simuliert werden. Die Anzahl der nötigen Zellen kann auf $\max\{m, p\}$ reduziert werden. Bezüglich der O-Notation ist dieses aber keine Verbesserung [4].

Theorem 2.2 *(siehe auch in [25] für eine ähnliche Aussage) Jedes Problem, das in t Schritten auf einer Priority CRCW PRAM mit p Prozessoren unter Benutzung von m Speicherstellen gelöst wird, kann auf einem 1-armigen GCA mit $\max\{m, p\}$ Zellen in $O(p \cdot t)$ Generationen gelöst werden. Dabei besitzen die Zellen des GCAs, bis auf Schreibbefehle, den gleichen Befehlsatz[5] wie die PRAM Prozessoren.*

Beweis: Die Verbesserung von $p + m$ auf $\max\{p, m\}$ kann durch eine Fallunterscheidung erreicht werden. Im Fall $p \geq m$ übernehmen m der p Prozessorzellen zusätzlich die Arbeit der Speicherstellen und analog im Fall $m \geq p$ übernehmen die Speicherzellen die Arbeit der Prozessorzellen. In beiden Fällen bleibt die Anzahl der Generationen, die zur Simulation eines Schrittes nötig sind bei $O(p)$. $\diamond$

Die Zeit, die für die Simulation eines Schrittes einer PRAM durch einen GCA benötigt wird[6], kann durch die Benutzung von mehr Speicherzellen oder durch die Benutzung eines mehrarmigen GCA reduziert werden.

Spendiert man dem GCA mehr einfache Speicherzellen, dann lässt sich die Anzahl der Generationen zur Simulation eines Schrittes auf $O(\log p)$ reduzieren.

Theorem 2.3 *Jedes Problem, das in t Schritten auf einer Priority CRCW PRAM mit p Prozessoren unter der Benutzung von m Speicherstellen gelöst wird, kann auf einem 1-armigen GCA mit $p \cdot m$ Zellen in $O(t \cdot \log p)$ Generationen gelöst werden. Der Befehlssatz der Zellen des GCAs ist dabei, bis auf Schreibbefehle, mit dem Befehlssatz der PRAM Prozessoren identisch. Die Zellen des GCAs benötigen $O(\log p)$ Bits zur Simulation (der Schreibbefehle).*

Beweis: Wie schon in der Herleitung des Theorems 2.2 festgestellt, ist die Simulation der Schreibbefehle das Problem. Wir nehmen im Folgenden an, dass Prozessor i, $1 \leq i \leq p$, einen Wert in die Speicherstelle m_i, $1 \leq m_i \leq m$, schreiben möchte. Der 1-armige GCA besitzt $p \cdot m$ Zellen $z_{i,j}$, $1 \leq i \leq p$, $1 \leq j \leq m$. Dabei simulieren die Zellen $z_{i,1}$, $1 \leq i \leq p$, die Prozessoren der PRAM und die Zellen $z_{1,j}$, $1 \leq j \leq m$, die Speicherstellen der PRAM. Dieser GCA ist in Abbildung 2.1 verdeutlicht. Für die Simulation eines Schreibbefehls auf die Speicherstelle j werden die Zellen $z_{1,j}$ bis $z_{p,j}$ benutzt. Ein Schreibbefehl von Prozessor i auf die Speicherstelle m_i wird wie folgt durch den GCA simuliert ($1 \leq i \leq p$, $1 \leq j \leq m$):

(1) Für alle i, j überprüft Zelle $z_{i,j}$, ob Prozessor i (entspricht $z_{i,1}$) auf Speicherzelle j (entspricht $z_{1,j}$) schreiben will, d. h. ob $m_i = j$ gilt. Falls ja, dann merkt sich Zelle $z_{i,j}$ den Wert i, ansonsten den Wert 0.

(2) Für alle j wird in den Zellen $z_{1,j}, \ldots, z_{p,j}$ der Prozessor ermittelt, der auf Speicherstelle j schreiben darf. In dem Falle, dass bei einem Schreibkonflikt immer der Prozessor mit der höchsten Nummer gewinnt, ist an dieser Stelle das Maximum der Inhalte von $z_{1,j}$ bis $z_{p,j}$ zu ermitteln. Das Ergebnis steht am Ende in $z_{1,j}$. Sei dieses Prozessor $p_{1,j}$, $1 \leq p_{1,j} \leq p$.

(3) Für alle j: Zelle $z_{j,1}$ liest den Wert aus Prozessor $p_{1,j}$.

[4] $p + m \in O(\max\{p, m\})$
[5] siehe im Kapitel 4
[6] slowdown

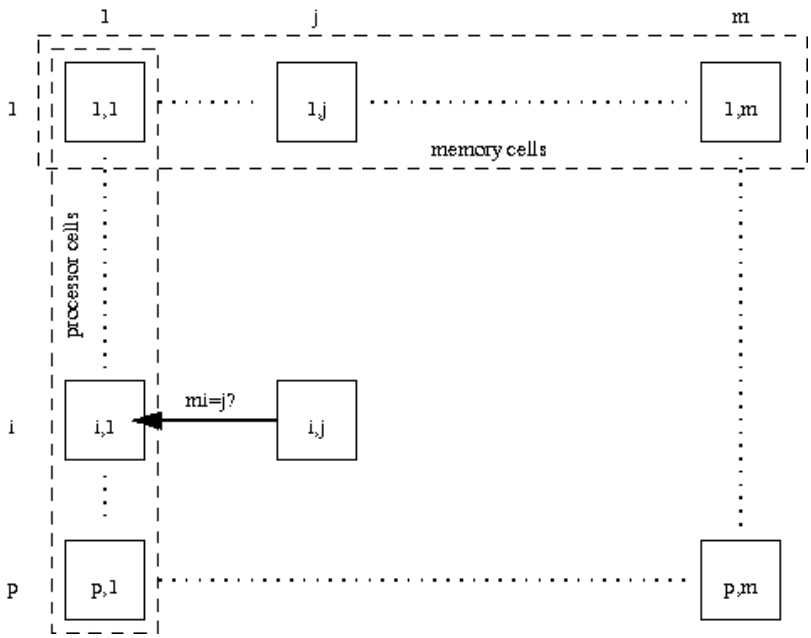

Abbildung 2.1: 1-armiger GCA zur Simulation des konkurrierenden Schreibens

Schritt (1) kann von allen Zellen parallel ausgeführt werden und benötigt somit $O(1)$ Generationen. Schritt 2 ist in $O(\log p)$ Generationen möglich [10] und Schritt (3) benötigt $O(1)$ Generationen. $\diamond$

Betrachtet man den Schritt (1) der Simulation im Beweis zu Theorem 2.3, dann stellt man fest, dass die Speicherzelle $z_{j,1}$ im Falle eines p-armigen GCAs in einer Generation alle Schreibwünsche ermitteln kann. Existiert im Befehlssatz des GCAs zusätzlich auch noch eine p-stellige Maximumsfunktion, dann können die Schritte (1), (2) und (3) im obigen Beweis von $p+m$ Zellen in $O(1)$ Generationen ausgeführt werden kann. Mit einer Fallunterscheidung wie im Beweis zu Theorem 2.3 ergibt sich dann:

Theorem 2.4 *Jedes Problem, das in t Schritten auf einer Priority CRCW PRAM mit p Prozessoren unter der Benutzung von m Speicherstellen gelöst wird, kann auf einem p-armigen GCA mit $\max\{p, m\}$ Zellen in $O(t)$ Generationen gelöst werden. Der Befehlssatz von m Zellen des GCAs muss dazu die p-stellige Maximumsfunktion unterstützen. Ansonsten ist der Befehlssatz der Zellen des GCAs, bis auf Schreibbefehle, mit dem Befehlssatz der PRAM Prozessoren identisch.*

Mit Theorem 2.4 wissen wir, dass eine Simulation eines Schrittes einer PRAM ohne Zeitverlust auf einem GCA möglich ist. Der Preis dafür ist die Benutzung von p-armigen Zellen.

Der beste uns bisher bekannte Slowdown bei der Benutzung eines 1-armigen GCAs ist $O(\log p)$ (Theorem 2.3). Geht man davon aus, dass n-stellige Eingaben für den GCA in den ersten n Zellen liegen[7], kann man zeigen, dass dieser Wert nicht mehr verbessert werden kann.

Theorem 2.5 *Ein Schritt einer EREW PRAM mit p Prozessoren, kann im Allgemeinen nicht in weniger als $\Omega(\log p)$ Generationen von einem 1-armigen GCA simuliert werden.*

Beweis: (siehe Lemma 2.3 in [28]) Gegeben seien p Zahlen $z_1, \ldots, z_p \in \{0, 1\}$ in den Speicherstellen 1 bis p.

[7] bei der PRAM liegen sie in den ersten n Speicherstellen

Zusätzlich wissen wir, dass $z_i = 1$ für maximal ein $i \in \{1, \ldots, p\}$ gilt. Dann kann eine EREW PRAM mit p Prozessoren in $O(1)$ Schritten das Ergebnis der Funktion $OR(z_1, \ldots, z_p)$ in das Register 1 schreiben [28]. Wir zeigen nun, dass auf einem 1-armigen GCA mindestens $\Omega(\log p)$ Generationen nötig sind diese Funktion zu berechnen. Dabei gehen wir ebenfalls davon aus, dass die Eingabe in den Zellen 1 bis p liegt und die Ausgabe in Zelle 1 erfolgen soll. Im Folgenden bezeichnet eine Zeichenkette $w \in \{0,1\}^p$ den Inhalt der ersten p Zellen.

Wir definieren, dass eine Zelle j von der Eingabezelle i in Generation t *abhängt*, wenn der Zustand von j bei der Eingabe 0^p in Generation t anders ist, als bei der Eingabe $0^{i-1}10^{p-i}$. Mit vollständiger Induktion läßt sich nun zeigen, dass in Generation t jede Zelle von maximal 2^t Eingabezellen abhängt: In einer Generation kann eine Zelle nur maximal von **einer** Zelle etwas lesen und nur bei sich selbst schreiben. Somit ist eine Zelle im Schritt t von den Eingabezellen abhängig von denen sie im Schritt $t-1$ abhängig war und zusätzlich dazu noch von den Eingabezellen von denen die Zelle von der gelesen wurde im Schritt $t-1$ abhängig war, also insgesamt maximal $2 \cdot 2^{t-1} = 2^t$ Stück. (Bei EW wäre die Zelle auch noch von allen Zellen abhängig, die auf sie schreiben dürfen!). Sei T die Generation zu der das Ergebnis der Berechnung von $OR(z_1, \ldots, z_p)$ in der Zelle 1 steht. Zum Zeitpunkt T muss Zelle 1 von allen Eingabezelle abhängen. Somit folgt $T \in \Omega(\log p)$. Dieser Wert ist **unabhängig** von der Mächtigkeit des Befehlssatzes der einzelnen Zellen sowie der Anzahl der Zellen! $\diamond$

Das Ergebnis gilt auch für Priority CRCW PRAM, Arbritrary CRCW PRAM und Common CRCW PRAM, da sie bzgl. der Berechnungsfähigkeit mächtiger sind als EREW PRAM.

Corollary 2.6 *Ein Schritt einer Priority (Arbitrary, Common) CRCW PRAM mit p Prozessoren, kann im Allgemeinen nicht in weniger als $\Omega(\log p)$ Generationen von einem 1-armigen GCA simuliert werden.*

Für mehrarmige GCA kann ein entsprechendes Ergebnis erzielt werden. Dazu beweisen wir, dass auf einem a-armigen GCA für die Berechnung eines Wertes, der von n Eingaben in den Zellen 1 bis n abhängt, mindestens $\Omega(\log_{a+1} n)$ Schritte nötig sind:

Lemma 2.7 *Gegeben ist ein a-armiger GCA. Es soll ein Wert berechnet und in Zelle z_{n+1} abgespeichert werden, der von n Eingaben abhängt. Die Eingaben liegen in den Zellen 1 bis n. Dann sind dafür mindestens $\Omega(\log_{a+1} n)$ Schritte nötig.*

Beweis: Am Start der Berechnung hängt der Inhalt der Zelle z_{n+1} von keiner anderen Zelle ab. Da der GCA a-armig ist, hängt der Inhalt von z_{n+1} am Ende der ersten Generation maximal von a anderen Zellen und dem eigenen Startinhalt ab, also von maximal $a+1$ Zellen. Am Ende von Schritt t, $t > 1$, hängt der Inhalt von den Zellen ab, von dem er schon nach $t-1$ Schritten abhing und zusätzlich von den a Zellen auf die im Schritt t zugegriffen wurde. Man erhält also für die maximale Anzahl x_t der Zellen von denen z_{n+1} am Ende von Generation t abhängig ist, die Gleichung $x_{t+1} = x_t + a \cdot x_t = (1+a) \cdot x_t$, $t \geq 1$ und $x_1 = a+1$. Daraus ergibt sich $x_t = (1+a)^t$.

Da die Eingabe auf n Zellen verteilt ist, ergibt sich aus $(1+a)^t \geq n$ somit die Schranke $\Omega(\log_{a+1} n)$. $\diamond$

Das erzielte Ergebnis ist unabhängig von der Anzahl der Zellen und dem Befehlssatz des GCAs. Daraus ergibt sich:

Theorem 2.8 *Ein Schritt einer CREW PRAM mit p Prozessoren, kann im Allgemeinen nicht in weniger als $\Omega(\log_{a+1} p)$ Generationen von einem a-armigen GCA simuliert werden, unabhängig vom Befehlssatz und der Anzahl der Zellen des GCAs.*

In Theorem 2.3 werden $p \cdot m$ Zellen zur Simulation benötigt. Diese Anzahl kann ohne Zeitverlust auf $O(m+p)$ gedrückt werden. Allerdings unter der Voraussetzung, dass die Register des GCAs mindestens m Bits Länge haben. Die Idee dabei ist die folgende: Wir bilden einen ausgeglichenen Binärbaum mit p Zellprozessoren als Blätter. Ein solcher Baum besteht somit aus $2p-1 = O(p)$ Zellen. Wir nennen den Wurzelprozessor des Baums r. Zusätzlich existieren m Speicherzellen. Jede Zelle besitzt ein Register M mit m Bits. Ein Zellprozessor setzt bei einem Schreibwunsch auf Speicherzelle j das j-te Bit von M. Mittels üblicher Baumtechnik kann nun in $O(\log p)$ Generationen im Register M von r berechnet werden, auf welche Speicherzellen geschrieben werden soll (i-tes Bit in M auf r ist genau dann gesetzt wenn auf Zelle i geschrieben werden soll). Die m Speicherzellen können nun mittels Baumsuche beginnend mit Zelle

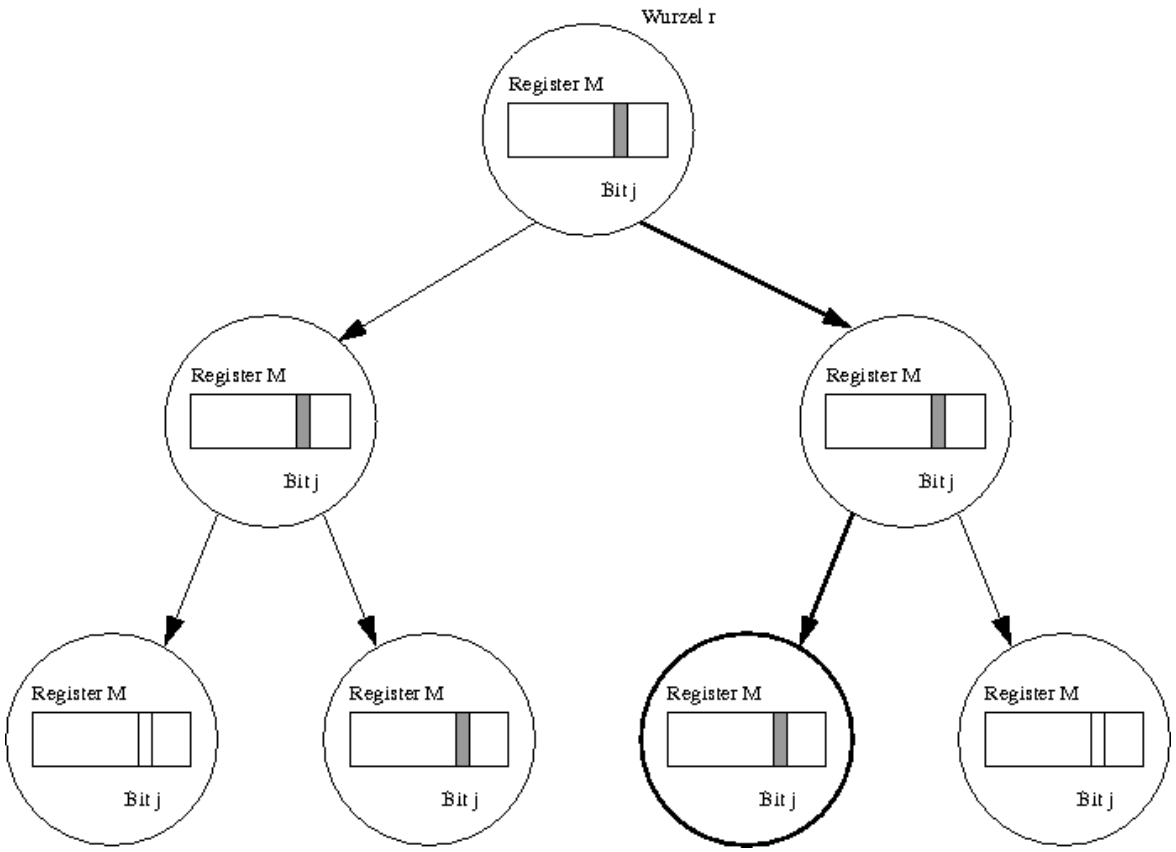

Abbildung 2.2: Prozessorbaum zur effizienten Bestimmung konkurrierenden Schreibens.

r in $O(\log p)$ Generationen die Prozessorzelle ermitteln von der sie lesen müssen. Dabei liest die Speicherzelle j zunächst Register M der Wurzel r. Sofern dort Bit j nicht gesetzt ist, muss sie nichts tun. Ansonsten liest sie das Register M im rechten Kind der Wurzel, das die Wurzel des Teilbaums mit der Hälfte der Prozessoren mit höherer Priorität bildet. Ist dort Bit j nicht gesetzt, macht sie im linken Teilbaum weiter. Ist im rechten Kind Bit j gesetzt, macht sie im rechten Teilbaum weiter. Dies geschieht solange, bis ein Blatt erreicht wird, nämlich der Prozessor mit höchster Priorität, der Speicherzelle j schreiben will. Die Suche ist in Abbildung 2.2 illustriert. Somit ergibt sich:

Theorem 2.9 *Jedes Problem, das in t Schritten auf einer Priority CRCW PRAM mit p Prozessoren unter der Benutzung von m Speicherstellen gelöst wird, kann auf einem 1-armigen GCA mit $O(\max\{p, m\})$ Zellen in $O(t \cdot \log p)$ Generationen gelöst werden. Der Befehlssatz der Zellen des GCAs ist dabei, bis auf Schreibbefehle, mit dem Befehlssatz der PRAM Prozessoren identisch. Die Zellen des GCAs benötigen $O(m)$ Bits zur Simulation (der Schreibbefehle).*

Mapping CRCW auf GCA

In Theorem 2.9 wird ein Slowdown von $O(\log p)$ bei der Simulation von Priority CRCW PRAMs durch 1-armige GCAs erreicht. Auf Grund des Korollars 2.6 ist dieses Ergebnis optimal bzgl. der Laufzeit für 1-armige GCAs. Diese Feststellung gilt für die drei Varianten Priority, Arbitrary und Common CRCW PRAM, da Theorem 2.9 für das mächtigste der drei Modelle, die Priority PRAM, gezeigt wurde und da Theorem 2.5 sogar für EREW PRAMs gilt.

Für die Simulation von Priority (Arbitrary, Common) CRCW PRAMs durch mehrarmige GCA kann das Verfahren aus Theorem 2.4 angewendet werden. Dort wird ein konstanter Slowdown erreicht. Ein besseres Ergebnis kann durch eine Schritt für Schritt Simulation nicht erzielt werden.

Mapping CREW PRAM auf GCA

Das Theorem 2.5 sagt aus, dass ein Schritt einer EREW PRAM mit p Prozessoren i. A. nicht schneller als in $O(\log p)$ Generationen auf einem 1-armigen GCA simuliert werden kann. Im Theorem 2.3 wird dieser Wert für die Simulation einer CRCW PRAM erreicht. Das dortige Verfahren ist also optimal bzgl. der Laufzeit für 1-armige GCAs.

Für die Simulation von CREW PRAMs durch mehrarmige GCA wird in Theorem 2.4 ein konstanter Slowdown erreicht. Ein besseres Ergebnis kann durch eine Schritt für Schritt Simulation nicht erzielt werden.

Mapping CROW PRAM auf GCA

CROW PRAMs mit p Prozessoren entsprechen einem 1-armigen GCA mit p Zellen. Im CROW Modell ist jeder Prozessor Besitzer von m_i, $1 \leq i \leq p$, Speicherstellen. Nur auf diesen m_i Speicherstellen darf der Prozessor i schreiben. Bei der Simulation der PRAM werden diese m_i Speicherstellen dem lokalen Speicher der Zelle i zugeordnet. Somit folgt:

Theorem 2.10 *Jedes Problem, das in t Schritten auf einer CROW PRAM mit p Prozessoren unter Benutzung von m Speicherzellen gelöst wird, kann auf einem 1-armigen GCA mit p Zellen in $O(t)$ Generationen gelöst werden.*

Mapping EROW PRAM auf GCA

Es gilt das in Kapitel 2.3.1 gesagte. Also

Theorem 2.11 *Jedes Problem, das in t Schritten auf einer EROW PRAM mit p Prozessoren unter Benutzung von m Speicherzellen gelöst wird, kann auf einem 1-armigen GCA mit p Zellen in $O(t)$ Generationen gelöst werden.*

2.3.2 GCA und Prozessornetzwerke

Praxisrelevanter als PRAMs sind Prozessornetzwerke. Die Prozessoren in einem Prozessornetzwerk besitzen keinen gemeinsamen Speicher. Wollen verschiedene Prozessoren im Netzwerk miteinander Daten austauschen, müssen sie über ein Verbindungsnetzwerk miteinander kommunizieren. Prozessornetzwerke werden daher meist als ungerichtete Graphen $G = (V, E)$ modelliert. Die Knoten des Graphen sind die Prozessoren, die Kanten modellieren die Verbindung zwischen den Prozessoren. In einem Schritt können zwei Prozessoren v_1, $v_2 \in V$ nur dann miteinander kommunizieren, wenn $\{v_1, v_2\} \in E$ gilt. Die Berechnungsmächtigkeit der einzelnen Prozessoren aus V ist im Allgemeinen nicht beschränkt, d. h. in einem Schritt kann jede berechenbare Funktion berechnet werden. Ein Schritt im Prozessornetzwerk kann daher einfach durch einen GCA mit $|V|$ Zellen simuliert werden. Dabei simuliert Zelle v den Prozessor v wie folgt:

(1) Lese die von den Nachbarn geschickten Nachrichten.

(2) Führe die Berechnungen durch, die v durchgeführt hat.

(3) Speichere die von v gesendeten Daten.

In Schritt (1) werden die gesendeten Nachrichten der direkten Nachbarn des Knoten v in G gelesen. Der Inhalt der Nachricht ist dabei im Zustand der Nachbarzelle kodiert. Ist d der maximale Knotengrad in G, dann benötigt man an dieser Stelle einen d-armigen GCA. In Schritt (2) führt die Zelle v die internen Berechnungen des Prozessors v aus

und in Schritt (3) werden die von Prozessor v an die Nachbarn gesendeten Daten (im Zustand der Zelle) gespeichert. Eine einfache Folgerung daraus ist:

Theorem 2.12 *Jedes Problem, das in Zeit t auf dem Prozessornetzwerk $G = (V, E)$ gelöst wird, kann auf einem d-armigen GCA mit $|V|$ Zellen in $O(t)$ Generationen gelöst werden. Dabei ist d der maximale Grad eines Knotens in G.*

2.3.3 CROW zwischen NC^1 und AC^1

Unter der Annahme, dass die Zellen des GCAs einen Befehlssatz wie die Prozessoren einer PRAM besitzen, entspricht eine CROW PRAM einem 1-armigen GCA. In diesem Kapitel ordnen wir die Klasse der Sprachen, die von einer CROW PRAM mit polynomiell vielen Prozessoren in logarithmischer Zeit entschieden werden, kurz CROW($\log$), in die Hierarchie der Komplexitätsklassen zwischen NC^1 und AC^1 ein.

Die Klasse NC^k besteht genau aus den Sprachen, die von uniformen[8] Schaltkreisen mit polynomieller Größe, $\log^k$ Tiefe und Gattern mit beschränktem Eingangsgrad entschieden werden können. Die Klasse AC^k besteht genau aus den Sprachen, die von uniformen Schaltkreisen mit polynomieller Größe, $\log^k$ Tiefe und Gattern mit unbeschränktem Eingangsgrad entschieden werden können. Weiterhin wird NC:= $\bigcup_{i\in\mathbb{N}} \mathrm{NC}^i$ und AC:= $\bigcup_{i\in\mathbb{N}} \mathrm{AC}^i$ definiert. Aus diesen Definitionen und der Einsicht, dass sich ein Gatter mit Eingangsgrad n durch einen Schaltkreis polynomieller Größe und Tiefe $\log n$ mit Gattern mit konstantem Eingangsgrad simulieren lässt, folgt: NC$^i \subseteq$ NC^{i+1}, AC$^i \subseteq$ AC^{i+1}, AC$^i \subseteq$ NC^{i+1} und AC=NC. Da ein Schaltkreis von polynomieller Größe und $\log^k$ Tiefe in polynomieller Zeit auf einer Turingmaschine ausgewertet werden kann, folgt $AC \subseteq P$. Ob AC=P gilt ist unbekannt.

Ziel dieses Kapitels ist es die folgenden Beziehungen zu erläutern:

$$\mathrm{NC}^1 \subseteq DLOG \subseteq LogDCFL = CROW(\log) \subseteq LogCFL \subseteq \mathrm{AC}^1$$

Für die Klassen NC^1 und AC^1 am Rande dieser Hierarchie existieren neben der oben angegebenen Definition noch weitere äquivalente Charakterisierungen. Für NC^1:

- durch uniforme alternierende Turingmaschinen in logarithmischer Zeit [31],
- durch polynomiell große Programme über endlichen Monoiden [1],
- durch polynomiell große Branching Programme [1]
- durch Schaltkreise mit polynomieller Weite und konstanter Tiefe SC^0 (Zitat noch nicht gefunden)
- durch Formeln mit bestimmten Eigenschaften [2].

Für AC^1:

- durch CRCW PRAMs in logarithmischer Zeit und mit polynomiell vielen Prozessoren
- durch uniforme alternierende Turingmaschinen mit logarithmischem Platz und logarithmisch vielen Zustandswechseln zwischen universellen und existentiellen Zuständen.

Kommen wir nun zu den Klassen $DLOG$, $LogDCFL$ und $LogCFL$. $DLOG$ ist die Klasse von Sprachen, die von deterministischen Turingmaschinen mit logarithmischen Bandbedarf akzeptiert wird. Sie entspricht der von Schaltkreisen mit logarithmischer Tiefe und polynomieller Weite akzeptierten Klasse von Sprachen SC^1. Aus der oben zitierten Charakterisierung von NC^1 durch SC^0, der für alle $i \in \mathbb{N}$ aus der Definition folgenden Beziehung SC$^i \subseteq$ SC^{i+1} folgt somit NC$^1 \subseteq DLOG$. Über die Echtheit der Beziehung ist nichts bekannt. $LogDCFL$ ist die Klasse

[8] uniform: der Schaltkreis für die Problemgröße n muss bei Eingabe n, von einer Turingmaschine, die bestimmte Bedingungen erfüllt, berechnet werden können [31]

von Sprachen, die mittels logarithmischen Platz auf kontextfreie Sprachen reduziert werden können. Analog dazu ist $LogDCFL$ die Klasse von Sprachen die mittels logarithmischen Platz auf deterministisch kontextfreie Sprachen reduziert werden können. Aus der aus der Chomsky-Hierarchie bekannten Beziehung $DCFL \subset CFL$ folgt sofort $LogDCFL \subseteq LogCFL$. Ob sich die Echtheit der Inklusion auch vererbt ist nicht bekannt[9]. In [23] wird gezeigt, dass $LogCFL$ die Menge von Sprachen ist, die von uniformen Schaltkreisen mit polynomieller Größe und logarithmischer Tiefe mit OR-Gattern von beliebigem Eingangsgrad und AND-Gattern von maximalem Eingangsgrad 2, entschieden werden kann. Daraus folgt sofort $LogCFL \subseteq \mathrm{AC}^1$. Die Beziehung $CROW(\log) = LogDCFL$ wird in [7] gezeigt.

Die Einordnung der Klasse CROW($\log^k$) [10] in die polynomielle Hierarchie ist eine interessante Fragestellung, die aus Zeitgründen hier nicht weiter untersucht wurde. Erste Ergebnisse dazu sind in [26] zu finden.

[9] vermutlich nicht

[10] Klasse der Sprachen, die von einer CROW PRAM mit polynomiell vielen Prozessoren in $\log^k$ Zeit entschieden werden.

Kapitel 3

Algorithmen für den GCA

Inzwischen wurden für den GCA in vielen Veröffentlichungen [8, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 25] parallele Algorithmen vorgestellt. Dabei handelt es sich allerdings im Allgemeinen um Implementierungen, die die Funktionsweise des GCAs erklären sollen und nicht um speziell optimierte Algorithmen. Vorgestellt werden dort Algorithmen für die folgenden Problemgebiete: Sortieren, Graphen, Matrixmultiplikation, Kryptographie, Vektoroperationen, FFT, Mehrkörperprobleme und Listranking.

In den folgenden Tabellen 3.1, 3.2 und 3.3 werden Algorithmen für drei verschiedene Probleme gelistet: das Sortieren von n Zahlen, das Finden einer Zusammenhangskomponente eines ungerichteten Graphen mit n Knoten und die Multiplikation von zwei $n \times n$ Matrizen. Zwei dieser Probleme (Zusammenhangskomponente, Sortieren) wurden gewählt, da für sie spezialisierte GCA bzw. CROW PRAM Algorithmen publiziert wurden. Die Matrixmultiplikation wurde gewählt, da es sich dabei um ein nicht triviales und häufig in mathematischen Anwendungen zur Problemlösung benutztes Problem handelt. In der Tabelle werden CROW Implementierungen beim GCA aufgeführt, da sie direkt auf dem GCA implementiert werden können (siehe Kapitel 2.3.1). Wenn nichts anderes angegeben wird, wird ein 1-armiger GCA angenommen.

In der Tabelle wird mit T die Worst-Case Laufzeit, mit P die Anzahl der benutzten Prozessoren bzw. Zellen und mit W die von dem Algorithmus verursachte Arbeit angegeben. Die verursachte Arbeit eines Algorithmus ist $\sum_p T_p$ wobei T_p die Anzahl der von Prozessor p ausgeführten Schritte ist. Im Falle des sequentiellen Algorithmus ist also $W = T$. Alle Werte sind bzgl. der O-Notation angegeben.

Bei allen drei GCA Algorithmen konnte die Laufzeit des schnellsten CREW PRAM Algorithmus erreicht werden.

Tabelle 3.1: Vergleich - Problem Zusammenhangskomponente finden.

Problem	GCA	CREW	SEQ
Zusammenhangskomp.	$T = \log^2 n$	$T = \log^2 n$	$T = n^2$
von $G = (V, E)$, $n = \|V\|$	$P = n^2$	$P = \frac{n^2}{\log^2 n}$	$P = 1$
Eingabe: Adjazensmatrix	$W = n^2 \log^2 n$	$W = n^2$	$W = n^2$
	[21]	[4]	[32]
	Hirschbergs Alg.	Hirschbergs Alg.	
	$T = n \log n$		
	$P = n$		
	$W = n^2 \log n$		
	[20]		
	Hirschbergs Alg.		

Tabelle 3.2: Vergleich - Problem Sortieren.

Problem	GCA	CREW	SEQ
Sortieren	$T = \log n$	$T = \log n$	$T = n \log n$
Eingabe: n Zahlen	$P = n \log n$	$P = n$	$P = 1$
	$W = n \log^2 n$	$W = n \log n$	$W = n \log n$
	[27]	[5]	
	Mergesort	Mergesort	Mergesort
	$T = \log^2 n$		
	$P = n$		
	$W = n \log^2 n$		
	[16, 14, 10, 11]		
	Bitonisches S.		

Tabelle 3.3: Vergleich - Problem Matrixmultiplikation.

Problem	GCA	CREW	SEQ
Matrixmultiplikation	$T = \log n$	$T = \log n$	$T = n^{2.38}$
Eingabe:	$P = n^3$	$P = \frac{n^{2.38}}{\log n}$	$P = 1$
zwei $n \times n$ Matrizen	$W = n^3 \log n$	$W = n^{2.38}$	$W = T$
	[10]	[30]	[6]
	Standard	Strassen	Strassen

Allerdings konnte bei keiner GCA Implementierung bisher ein arbeitsoptimaler Algorithmus mit schneller Laufzeit erzielt werden. Hier sind noch weitere Untersuchungen nötig.

Beim Vergleich von CREW PRAM Algorithmen mit GCA Algorithmen ist aber zu bedenken, dass die Menge m des benutzten gemeinsamen Speichers bei PRAM-Algorithmen keine Rolle spielt, d. h., unabhängig von der Größe des benutzten globalen Speichers wird davon ausgegangen, dass ein Speicherzugriff konstante Zeit benötigt. Da auch Speicher bei seiner Realisierung eine räumliche Ausdehnung besitzt und die maximale Übertragungsgeschwindigkeit der Daten durch die Lichtgeschwindigkeit begrenzt ist, ist es vernünftig anzunehmen, dass ein Speicherzugriff proportional zu $m^{1/3}$ Zeit benötigt. In der Arbeit von Bilardi und Preparata [3] wird der Einfluss dieser Annahme diskutiert.

Einschränkend muss aber erwähnt werden, dass der GCA die Möglichkeit eines globalen Lesezugriffs auf jede beliebige Zelle in $O(1)$ Generationen besitzt. Dies ist eine ebenso unrealistische Annahme wie der lesende und schreibende Speicherzugriff in $O(1)$ der PRAM.

Da ein GCA keinen gemeinsamen Speicher besitzt, bevorzugt die $O(1)$ Annahme für den Speicherzugriff das PRAM Modell. Bei einer Simulation von PRAM Algorithmen hat man die Möglichkeit für die Speicherstellen zusätzliche Zellen zur Verfügung zu stellen oder durch die die Prozessoren simulierenden Zellen den Speicher verwalten zu lassen. Im letzteren Fall muss dann bei einer gleichmäßigen Verteilung der Speicherstellen jede Zelle m/p Speicherstellen simulieren. Im Fall $m > p$ führt dieses zu Leistungsverlusten [21]. Im ersten Fall ist, falls $m > p$, die Anzahl der benutzten Zellen höher und das Produkt aus Zellenanzahl und Laufzeit schlechter [21] als beim PRAM Algorithmus.

Die Möglichkeiten im Algorithmenentwurf, die sich durch den Einsatz von mehrarmigen GCA ergeben, wurden außerhalb dieser Arbeit bisher noch nicht untersucht.

3.1 Anmerkungen zur Optimalität

Bei der Kostenbetrachtung von Algorithmen auf PRAMs spielt die Anzahl m der benutzten globalen Speicherstellen im Allgemeinen keine Rolle. Ein wichtiges Maß bei der Beurteilung von PRAM-Algorithmen ist, neben der Laufzeit, die Anzahl p der benutzten Prozessoren. Im Gegensatz dazu existiert beim GCA kein globaler Speicher, d. h. jede Speicherstelle muss *irgendwie* durch eine Zelle simuliert werden. Setzt man voraus, dass die Zellen eines GCAs nicht beliebig groß sein dürfen[1] und dieses auch für die Speicherstellen der PRAM gelten soll, dann können maximal konstant viele Speicherstellen der PRAM von einer Zelle des GCAs simuliert werden. Somit erhält man auf dem GCA $\theta(p + m) = \theta(\max\{p, m\})$ als Zellkosten eines p Prozessor, m Speicherstellen PRAM-Algorithmus.

Diese Erkenntnis spielt bei der Umsetzung von optimalen PRAM Algorithmen eine Rolle. Beispielsweise benötigt der optimale PRAM-Algorithmus für das Listranking-Problem im Worst-Case bei einer n-elementigen Liste $\theta(n/\log n)$ Prozessoren, $\theta(\log n)$ Zeit und $\theta(n)$ Speicherstellen. Die Anzahl der Speicherstellen kann nicht mehr verringert werden, da schon per Definition die Eingabe für dieses Problems auf n Speicherstellen gegeben ist. Auf einem GCA benötigt man daher für einen Listranking-Algorithmus auf jeden Fall $\Omega(n)$ Zellen. Als untere Schranke für die Anzahl der Generationen von Listranking auf einem GCA folgt $\Omega(\log n)$ aus Lemma 2.7. Ein **optimaler** Listranking Algorithmus für den GCA benötigt also $\Omega(n)$ Zellen und $\Omega(\log n)$ Generationen. Durch die Umsetzung eines **einfachen** n Prozessor $O(\log n)$ Zeit PRAM Algorithmus, lässt sich ein **optimaler** n Zellen und $O(\log n)$ Generationen Algorithmus auf dem GCA erzielen. Eine Betrachtung wie im Beispiel des Problems Listranking ist für Probleme möglich, für die $m > p$ gilt. Ein weiteres Beispiel hierfür ist das Problem des Findens von Zusammenhangskomponenten eines ungerichteten Graphen in [21].

[1] beispielsweise $O(\log n)$ bits

Kapitel 4

Anhang: PRAM-Befehlssatz

Wir nehmen die Standarddefinition des PRAM-Modells aus [17, 24] an. Im Folgenden bezeichnen wir mit L_x einen lokalen Speicherplatz eines Prozessor, mit G_x einen globalen Speicherplatz im gemeinsamer Speicher und x, y, z Konstanten. Ein üblicher Befehlssatz einer PRAM sieht wie folgt aus (siehe auch [26]):

(1) Konstanten: L_x :=Konstante, L_x :=Eingabegröße, L_x :=Prozessornummer

(2) Schreiben: $G_{L_x} := L_y$

(3) Lesen: $L_x := G_{L_y}$

(4) Lokale Zuweisung: $L_x := L_y$

(5) Bedingte Zuweisung: if $L_x > 0$ then *Zuweisung*

(6) Binäre Operationen: $L_x := L_y \circ L_z$

(7) Sprünge: Goto $Marke_x$, if $L_x > 0$ goto $Marke_y$

(8) Sonstiges: if $L_x > 0$ then Halt, if $L_y > 0$ then TueNichts.

Literaturverzeichnis

[1] Barrington, D. Bounded width polynomial size branching programs recognize exactly NC1. Journal of Computer and System Science, 38(1):154–164, 1989.

[2] Barrington, D., Immerman, N., Straubing, H. On Uniformity within NC1. Journal of Computer and System Science, 41(3):274–306, 1990.

[3] Bilardi, G., Preparata, F. P. Horizons of Parallel Computation. Journal of Parallel and Distributed Computing, 27(2):172–182, 1995.

[4] Chin, F. Y., Lam, J., Chen, I. Efficient Parallel Algorithms for some Graph Problems. Communications of the ACM **25**,9, 1982.

[5] Cole, R. Parallel Merge Sort. SIAM Journal on Computing 17:770-785, 1988.

[6] Coppersmith, D., Winograd, S. Matrix Multiplication via Arithmetic Progressions. Journal of Symbolic Computation, 9(3):251–270, 1990.

[7] Dymond, P. W., Ruzzo, W. L. Parallel RAMs with Owned Global Memory and Deterministic Context-Free Language Recognition (Extended Abstract) Proceedings of the 3th International Colloquium of Automata, Languages and Programming, LNCS 226, 95–104, 1986.

[8] Ehrt, C. Globaler Zellularautomat: Parallele Algorithmen. Diplomarbeit, Technische Universität Darmstadt, 2005.

[9] Fortune, S., Wyllie, J.: Parallelism in random access machines. Proceedings of 10th STOC,114-118, 1978.

[10] Heenes, W. Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten. Darmstädter Dissertationen D17, Technische Universität Darmstadt, 2006

[11] Heenes, W., Jendrsczok, J., Hoffmann, R. Eine massiv parallele Rechnerarchitektur für das GCA Modell. 20. PARS-Workshop, Lübeck, 2005.

[12] Hoffmann, R., Heenes, W., Halbach, M. Implementation of the Massively Parallel Model GCA. Parelec, Dresden, 2004.

[13] Hoffmann, R., Völkmann, K.P., Waldschmidt, S.: Global Cellular Automata GCA: An Universal Extension of the CA Model. Proceedings of the ACRI Conference, 2000.

[14] Hoffmann, R., Völkmann, K.P., Heenes, W.: Globaler Zellularautomat (GCA): Ein neues massivparalleles Berechnungsmodell. 17th PARS Workshop, 2001.

[15] Hoffmann, R., Völkmann, K.-P., Waldschmidt, S., Heenes, W. GCA: Global Cellular Automata. A Flexible Parallel Model. Proceedings of the 6th International Conference of Parallel Computing Technologies, LNCS 2127, 66–73, 2001.

[16] Hoffmann, R., Völkmann, K.P., Heenes, W.: GCA: A massively parallel Modell. International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing(WMPP), 2003.

[17] Jájá, J. An Introduction to Parallel Algorithms. Addison-Wesley, 1992.

[18] Jendrsczok, J., Ediger, P., Hoffmann, R. A Scalable Configurable Architecture for the Massively Parallel GCA Model. 10th Workshop on Advances in Parallel and Distributed Computational Models (APDCM/IPDPS), 2008.

[19] Jendrsczok, J., Hoffmann, R., Keller, J.: Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton. To appear in International Journal of Foundations of Computer Science

[20] Jendrsczok, J., Hoffmann, R., Keller, J. Hirschberg's Algorithm on a GCA and Its Parallel Hardware Implementation. Proceedings of the 13th International Euro-Par Conference, LNCS 4641, 815–824, 2007.

[21] Jendrsczok, J., Hoffmann, R., Keller, J. Implementing Hirschberg's PRAM-Algorithm for Connected Component on a Global Cellular Automaton. Proceedings of the 21th International Parallel and Distributed Processing Symposium, 1–8, 2007.

[22] Jendrsczok, J., Ediger, P., Hoffmann, R.: Language and Machine Support for the Massively Parallel GCA Model. Technical Report RA-1-2007, Technische Universität Darmstadt, FB Informatik, 2007.

[23] Johnson, D.S. A catalog of complexity classes. Chapter 2, Handbook of Theoretical Computer Science, Volume A, Elsevier, 1990.

[24] Karp, R.M., Ramachandra, V. Parallel Algorithms for shared-memory machines. Chapter 17 in Algorithms and Complexity, volume A of Handbook of Theoretical Computer Science, Elsvier, 1990.

[25] Koch, A. Aufwandsvergleich zur Implementierung von PRAM-Algorithmen in einem GCA. Bachelorarbeit an der Fernuniversität in Hagen im Lehrgebiet Parallelität und VLSI, 2008.

[26] Lange, K.-J., Niedermeier, R. Data-independences of parallel random access machines. Proceedings of the 13th Conference in Foundations of Software Technology and Theoretical Computer Science, LNCS 761, 104–113, 1993.

[27] Lin, D. C., Dymond, P. W., Deng, X. Parallel Merge Sort on Concurrent-Read Owner-Write PRAM. Proceedings of the 3rd International Euro-Par Conference, LNCS 1300, 379–383, 1997

[28] Nisam, N. CREW PRAMs and decision trees. SIAM Journal on Computing, 20(6):999-1007, 1991.

[29] Mahajan, M. Polynomial Size Log Depth Circuits: Between NC1 and AC1 Computational Complexity Column 91, 2007.

[30] Pan, V. Y. Complexity of Parallel Matrix Computations. Theoretical Compututer Science, 54:65–85, 1987.

[31] W. L. Ruzzo On uniform circuit complexity. Journal of Computer and System Science, 22(3):365–383, 1981.

[32] Tarjan, R. Depth-first search and linear graph algorithms. SIAM Journal on Computing. 1(2): 46-160, 1972.