Parallele Algorithmen
Sommersemester 2012
FU Berlin

Kapitel 1

Einleitung

1.1 Chronik der parallelen Algorithmenentwicklung

Parallele Verfahren finden nicht nur in der Algorithmik und Informatik allgemein Anwendung, sondern sind im täglichen Leben zu finden. Sobald mehrere Personen oder Maschinen parallel an der Lösung eines Problems oder Erstellung eines Gegenstandes (z.B. Autobau, Konstruktion von Gebäuden, etc.) arbeiten, ist dies ein paralleles Verfahren.
In der Informatik basierte lange Zeit jeder Computer auf der Von-Neumann-Architektur, d.h. es gab eine einzelne CPU. Damit waren sequentielles Arbeiten und somit auch sequentielle Algorithmen erzwungen. Parallelität gab es nur innerhalb der CPU durch verschiedene Schaltkreise, die nebeneinander arbeiten.
In Algorithmen und Programmierung ist die Idee paralleler Algorithmen schon seit den 1960er Jahren vorhanden und z.B. in den Büchern von Knuth zu finden.
In den 1980er und frühen 1990er Jahren wurde viel in dem Gebiet der parallelen Algorithmen und Architekturen geforscht und gearbeitet. Damals kam die Idee auf, anwendungsspezifische (special-purpose) parallele Architekturen zu entwickeln. Dazu wurden mehrere Prozessoren auf Gitterpunkten oder Kreisen angeordnet und auf bestimmte Art und Weise untereinander verbunden.

Beispiel (Hyperwürfel von Prozessoren): Die Prozessoren befinden sich auf den Gitterpunkten und sind mit den Nachbarn nach Vorschrift eines d-dimensionalen Würfels verbunden:

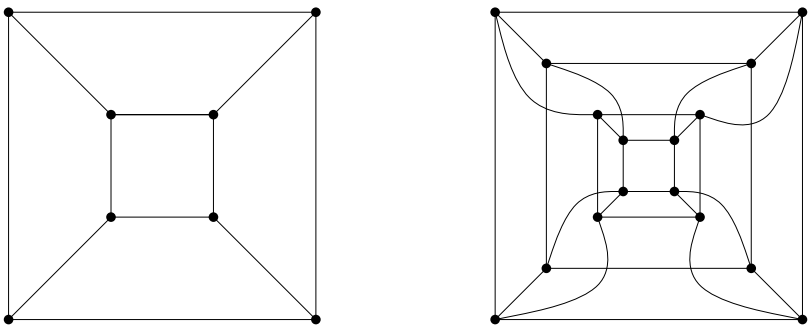


Abbildung 1.1: Drei- und vierdimensionaler Hyperwürfel

Beispiel (Connection Machines): Die Connection Machine CM1 bestand aus 65536 1-Bit Prozessoren, die in der Art eines 16-dimensionalen

Würfels angeordnet waren. Die Software war an der funktionalen Sprache LISP ausgerichtet.

Um parallele Algorithmen analysieren und vergleichen zu können wurde ein paralleles Berechnungsmodell entwickelt. Dieses theoretische Modell ist eine Weiterentwicklung der RAM und wird als parallele RAM (PRAM) bezeichnet.
Eine parallele Registermaschine (PRAM) besteht aus einer Menge von Prozessoren, einem gemeinsamen Speicher und einem Befehlssatz. Die Prozessoren kommunizieren über den gemeinsamen Speicher miteinander.

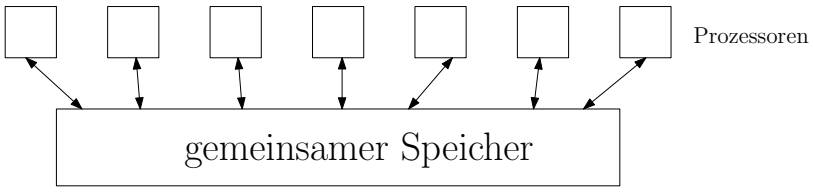


Abbildung 1.2: Idee der parallelen Registermaschine

An der PRAM als Berechnungsmodell gab es die Kritik, dass die Anzahl der Prozessoren abhängig von der Eingabegröße sein darf und die PRAM somit kein realistisches Modell ist. Allerdings gilt das Gleiche bei der Registermaschine in Bezug auf den Speicherplatz. Trotz der Kritik wird die PRAM das Berechnungsmodell für die Analyse der Algorithmen sein, die in dieser Vorlesung vorgestellt werden, da sie die Möglichkeit einer größtmöglichen Parallelisierung eines Problems liefert. Eine Simulation mit weniger Prozessoren ist dann ggf. leicht zu bewerkstelligen. Außerdem ist sie als Modell für komplexe theoretische Betrachtungen sinnvoll und kann auch als Benutzerschnittstelle für parallele Programmierung dienen.

In den 1990er und 2000er Jahren wurde die Forschung auf dem Gebiet der parallelen Algorithmen und Architekturen fast eingestellt. Dies hatte mehrere Gründe. Aus theoretischer Sicht waren für viele Standardprobleme parallele Algorithmen entwickelt worden. In der Praxis wurden mehr Fortschritte durch die Beschleunigung der CPUs durch höhere Taktraten erreicht als durch die Entwicklung paralleler Architekturen. Außerdem war die bisherige Software historisch bedingt auf sequentielle Rechner ausgerichtet. Das Entwickeln und Erstellen paralleler Software war wesentlich schwieriger und teurer.

Heutzutage ist fast jeder Rechner ein Parallelrechner mit mehreren Kernen (cores). Graphikkarten sind "special purpose hardware“ und arbeiten parallel. Die Firma NVIDIA entwickelte die Programmiersprache CUDA mit

der parallele Algorithmen programmiert und auf Graphikkarten ausgeführt werden können.

In dieser Vorlesung wird CILK für die Implementierung paralleler Algorithmen verwendet, welches auf C bzw. C++ basiert.

1.2 Drei Beispiele paralleler Algorithmen

1.2.1 Berechnung der Summe von n Zahlen.

Gegeben: n Zahlen $a_1, \ldots, a_n$
Berechne: $\sum_{i=1}^{n} a_i$

Werden die n Zahlen sequentiell aufaddiert wird $\Theta(n)$ Zeit benötigt. Angenommen, es stehen hinreichend viele Prozessoren zu Verfügung, die parallel arbeiten können, wie schnell kann das Problem dann gelöst werden? Wird die Summe in einer Art Baumstruktur berechnet, wobei auf jeder Ebene des Baumes die Operationen parallel ausgeführt werden, so kann die Summe von n Zahlen in $O(\log n)$ Zeit mit $\frac{n}{2}$ Prozessoren berechnet werden.

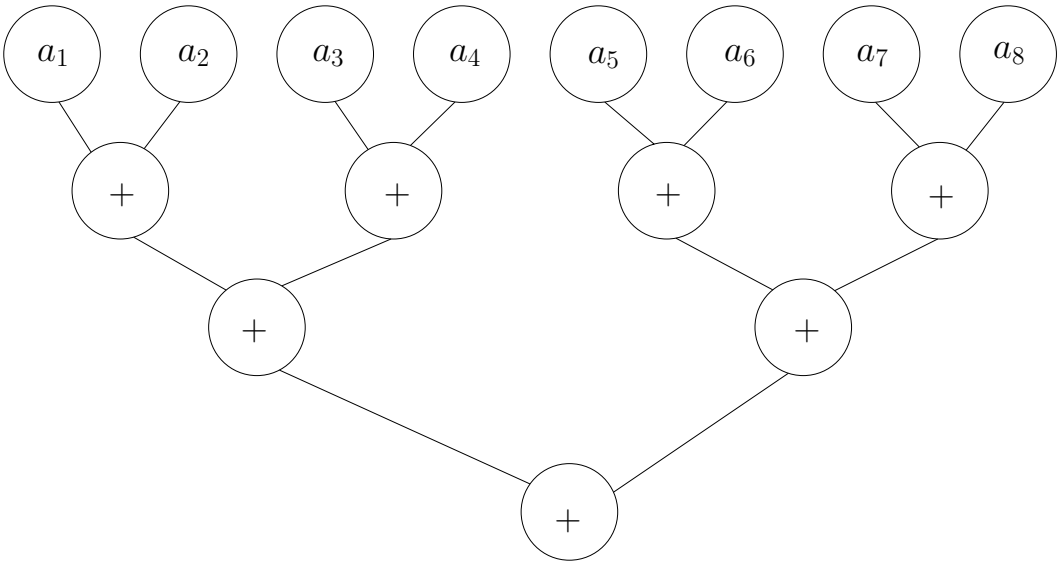


Abbildung 1.3: Schema zur parallelen Berechnung der Summe von n Zahlen.

Der Gesamtaufwand eines parallelen Algorithmus berechnet sich als das Produkt der Laufzeit und der Anzahl der Prozessoren. In diesem Fall wäre der Gesamtaufwand also $O(n \log n)$. Dies kann verbessert werden, indem nicht $\Theta(n)$ sondern $\Theta\left(\frac{n}{\log n}\right)$ viele Prozessoren verwendet werden. Die Folge der

Zahlen wird zunächst in Blöcke der Größe $\Theta(\log n)$ unterteilt und die Summe jedes Blocks wird von einem Prozessor sequentiell berechnet. Dies braucht insgesamt $\Theta(\log n)$ Zeit für alle Summen, da die Prozessoren parallel arbeiten. Dann wird die Summe der $\Theta\left(\frac{n}{\log n}\right)$ Zahlen wie oben beschrieben parallel berechnet. Die Laufzeit ist $\Theta(\log n)$ und damit ist der Gesamtaufwand $\Theta\left(\frac{n}{\log n}\log n\right) = \Theta(n)$.
Algorithmus 1 beschreibt das vorgestellten Verfahren mit Hilfe von Pseudocode. Das Schlüsselwort **par** bezeichnet die parallele Ausführung der Anweisungen in der zugehörigen Schleife.

Algorithmus 1 Summe der Zahlen $A[0], \ldots, A[n-1]$

```
1: for h := 0 to log n do
2:   for i := 0 step 2^{h+1} to n − 2^{h+1} par do
3:     A[i] := A[i] + A[i + 2^h];
4:   end for
5: end for
6: return A[0];
```

Um die Summe von n Zahlen zu berechnen, kann auch ein divide-&-conquer Ansatz verwendet werden, der parallelisiert werden kann. In CILK und in unserem Pseudocode wird dazu der Befehl **spawn** vor den rekusiven Aufruf der Funktion geschrieben.

Algorithmus 2 Summe der Elemente in A mit divide-&-conquer.

```
function add (A, l, r)
if l = r then
  return A[l];
else
  if l < r then
    m := ⌊(l+r)/2⌋;
    x :=spawn add (A, l, m);
    y :=spawn add (A, m + 1, r);
    return x + y;
  else
    return 0;
  end if
end if
```

Laufzeit (bei hinreichend vielen Prozessoren):
Sei $n = r - l + 1$.

$$\begin{aligned} T(1) &= c_1 \\ T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_2, \end{aligned}$$

für Konstanten c_1 und c_2.
Wegen der Parallelität wird das Maximum der Laufzeiten der beiden rekursiven Aufrufe betrachtet und nicht wie sonst die Summe der Laufzeiten.
Die Lösung der Rekursionsgleichung und somit die Laufzeit der Funktion **add** ist $T(n) = O(\log n)$ mit $O(n)$ Prozessoren.

Bemerkung: Der Algorithmus kann nicht nur für die Addition verwendet werden, sondern funktioniert für beliebige assoziative Operationen.

1.2.2 Berechnung von Präfixsummen

Als nächstes Beispiel betrachten wir die Berechnung der Präfixsummen von einer Folge von n Zahlen.
Gegeben: n Zahlen $a_1, \ldots, a_n$
Berechne: n Zahlen $s_1, \ldots, s_n$ mit $s_k = \sum_{i=1}^{k} a_i$

Beispiel:

a :	2	5	6	1	2	3	5	4
s :	2	7	13	14	16	19	24	28

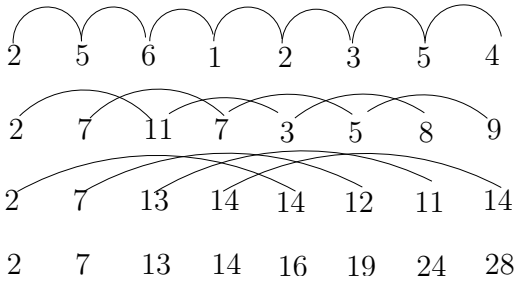
Ein einfacher sequentieller Algorithmus arbeitet wie folgt:

Algorithmus 3 Sequentielles Berechnen der Präfixsummen.

```
s_0 := 0
for i:=1 to n do
    s_i := s_{i-1} + a_i;
end for
```

Laufzeit: Jede Schleifeniteration benötigt $\Theta(1)$ Zeit, somit ist die Gesamtlaufzeit $\Theta(n)$.
Algorithmus 3 lässt sich nicht einfach durch das Hinzufügen der Anweisung **par** parallelisieren, da die jeweiligen Ergebnisse der Berechnung in der Schleife nicht voneinander unabhängig sind.

Eine Möglichkeit eines parallelen Verfahrens zur Berechnung der Präfixsummen ist durch das folgende Schema gegeben:



Im ersten Schritt wird die Summe zweier in der Folge benachbarter Zahlen berechnet und an der Stelle der 2. Zahl gespeichert. In den darauf folgenden Schritten werden jeweils die Summen zweier Zahlen mit Abstand zwei, vier, acht, etc. berechnet, bis im letzten Schritt die Summen zweier Zahlen mit Abstand $\frac{n}{2}$ berechnet werden. Diese bilden dann das Ergebnis der Präfixsummenfolge.

Algorithmus 4 Paralleles Berechnen der Präfixsummen.

```
h := 1;
for i := 1 to n par do
  s_i := a_i;
end for
while h ≤ n/2 do
  for i := h + 1 to n par do
    t_i := s_i + s_{i−h};
  end for
  h := h · 2;
  for i := 1 to n par do
    s_i := t_i;
  end for
end while
```

Laufzeit: Da die Anweisungen in den for-Schleifen parallel ausgeführt werden, und zusammen $\Theta(1)$ Zeit benötigen, ist die Gesamtlaufzeit proportional zu der Anzahl der Schleifendurchläufe der while-Schleife. Diese wird $\Theta(\log n)$ mal durchlaufen, da h bei jedem Durchlauf verdoppelt wird. Die Laufzeit des parallelen Algorithmus' zur Berechnung der Präfixsummen von n Zahlen ist somit $\Theta(\log n)$ mit n Prozessoren.

divide & conquer-Ansatz für Präfixsummen

Als nächstes betrachten wir einen divide & conquer-Ansatz für die Lösung der Präfixsummen.

Algorithmus 5 Parallele Berechnung der Präfixsummen mit dem divide & conquer-Ansatz.

function praef $(a_1, \ldots, a_n)$
if $n = 1$ **then**
  $s_1 := a_1$;
else
  $s_1 \ldots s_{\lfloor \frac{n}{2} \rfloor}$:=**spawn praef**$(a_1 \ldots a_{\lfloor \frac{n}{2} \rfloor})$;
  $s_{\lfloor \frac{n}{2} \rfloor + 1} \ldots s_n$:=**spawn praef**$(a_{\lfloor \frac{n}{2} \rfloor + 1} \ldots a_n)$;
end if
for $i := \lfloor \frac{n}{2} \rfloor + 1$ **to** n **par do**
  $s_i := s_i + s_{\lfloor \frac{n}{2} \rfloor}$;
end for

Wie oben bereits erwähnt, erzeugt der Befehl **spawn** einen eigenen Thread für den nachfolgenden Befehl.
parallele Laufzeit:

$$\begin{aligned} T(1) &= c_1 \\ T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_2 \\ \Rightarrow T(n) &= O(\log n) \end{aligned}$$

für die Konstanten c_1 und c_2.
Der erste Summand hat keinen Faktor 2, da beide Funktionen rekursiv ausgeführt werden. Der zweite Summand c_2 kommt von der for-Schleife.

Anzahl der Prozessoren :
Für Probleme der Größe 1 benötigt man 1 Prozessor, für Probleme der Größe n ist es möglich, mit n Prozessoren auszukommen.
Beweis(Induktion über n):
I.A.: $n = 1$✓
I.V.: Die Aussage ist für Zahlen $< n$ richtig.
I.S.: Betrachte den else-Fall der if-Abfrage. Der erste rekursive Aufruf benötigt nach I.V. $\lfloor \frac{n}{2} \rfloor$ Prozessoren, der zweite $\lceil \frac{n}{2} \rceil$. In der Summe werden also n Prozessoren verwendet. Die for-Schleife benötigt sogar nur $\lceil \frac{n}{2} \rceil$ Prozessoren.

Bemerkungen:

- Das Zeit-Prozessoranzahl-Produkt ist beim sequentiellen Algorithmus mit $O = (n)$ besser als beim parallelen mit $\theta(n \log n)$, es ist aber möglich, eine Laufzeit von $O(\log n)$ mit $O(\frac{n}{\log n})$ Prozessoren zu erhalten.

- In der for-Schleife lesen $\frac{n}{2}$ Prozessoren den Wert $s_{\lfloor \frac{n}{2} \rfloor}$ gleichzeitig. Dieses concurrent read kann in Wirklichkeit nicht in $O(1)$ realisiert werden. Es ergibt sich also die Frage, ob man dies für theoretische Betrachtungen trotzdem erlauben soll.

- Der Algorithmus funktioniert nicht nur für die +-Operation, sondern auch für beliebige andere assoziative (zweistellige) Operationen.

- Dies ist eine wichtige Technik beim parallelen Algorithmen.

1.2.3 Addierer

Als drittes und letztes Beispiel wollen wir einen Schaltkreis zum Addieren von zwei Binärzahlen mit n Stellen $a_{n-1}, \ldots, a_0$ und $b_{n-1}, \ldots, b_0$ mit $a_i, b_i \in \{0, 1\}, i = 0, \ldots, n-1$ betrachten.

Schaltkreis(Schaltkreismodell):
Wir betrachten Schaltkreise, die aus folgenden Bausteinen bestehen:

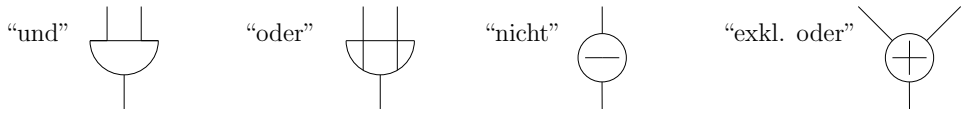


Abbildung 1.4: Grundbausteine von Schaltkreisen.

Aus diesen Grundbausteinen lässt sich ein Volladdierer zusammensetzen, den wir symbolisch wie folgt darstellen:

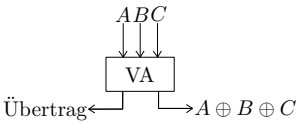


Abbildung 1.5: Symbolische Schreibweise eines Volladdierers.

Der Volladdierer kann aus den Gattern beispielsweise durch folgenden Schaltplan realisiert werden:

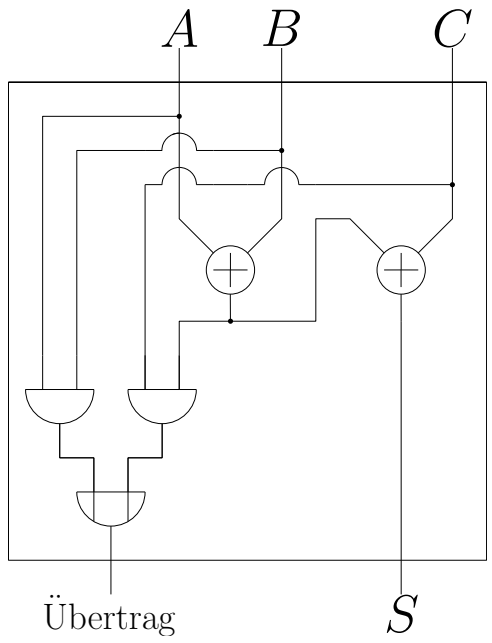


Abbildung 1.6: Schaltkreis eines Volladdierers im Detail.

Die Tiefe eines Schaltkreises ist die maximale Zahl der Gatter auf einem Weg von einem Eingang zu einem Ausgang und entspricht der parallelen Laufzeit. Die Gesamtzahl aller Gatter eines Schaltkreises nennt man seine Größe; sie entspricht dem Gesamtaufwand des Algorithmus.

Aus n Volladdierern kann man nun einen sogenannten Carry-Ripple-Adder zum Addieren zweier n-Bit-Zahlen zusammensetzen:

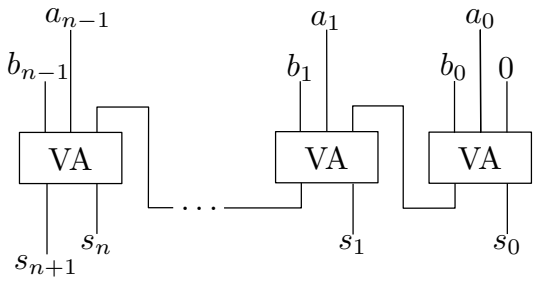


Abbildung 1.7: Carry-Ripple-Adder.

Dessen Größe ist mit $O(n)$ gut, die Tiefe von $\theta(n)$ ist dagegen schlecht.

Anmerkung:
Jeder n-Bit Addierer hat eine minimale Tiefe von $\Omega(\log n)$, da der Ausgang s_{n+1} immer von allen Ausgängen abhängt, d.h. er muss mit allen verbunden sein. Wenn man alle Gatter betrachtet, die von s_{n+1} aus erreichbar sind, ergibt sich ein Binärbaum, da jedes Gatter maximal zwei Eingänge hat (evtl. Knoten mit nur einem). Die Blätter entsprechen den $2n$ Eingängen des Schaltkreises. Daraus ergibt sich eine minimale Höhe des Baumes von $\log(2n)$.

Frage: Ist eine Tiefe $O(\log n)$ möglich?

Ja, dies ist durch Vorausberechnung der Überträge möglich und kann mit dem soganannten "Carry-Lookahead-Adder" realisiert werden. Um diesen zu konstruieren, definieren wir den Schaltkreis CLA_n. Führe dazu zunächst eine neue Notation ein.

Sei n eine Zweierpotenz. Dann bezeichne $\mathbf{S_{i,l}}$ für $l = 0, \ldots, \log n$ und $i = 0, \ldots, \frac{n}{2^l} - 1$ das Segment von $n-1, \ldots, 0$, das von $(i+1)2^l - 1$ bis $i2^l$ reicht:

$n-1$	$n-2$	$\ldots$	$(i+1)2^l - 1$	$\ldots$	$i2^l$	$\ldots$	2	1	0
$\ldots$			$S_{i,l}$ (i-tes Stück)			$\ldots$			

Der Schaltkreis CLA_n berechnet zwei Bits, die wir wie folgt definieren:
$\mathbf{g_{i,l}} = \mathbf{1}$, wenn das Segment $S_{i,l}$ von a und b einen Übertrag "erzeugt" (wenn man beide addiert ohne die anderen Segmente zu betrachten)
$\mathbf{p_{i,l}} = \mathbf{1}$, wenn das Segement $S_{i,l}$ einen Übertrag propagiert, d.h., wenn von rechts ein Übertrag in das Segment hineinkommt, dann wird links auch einer weitergegeben.
Es gilt:

$$\begin{aligned} g_{i,0} &= a_i \wedge b_i \\ p_{i,0} &= a_i \vee b_i \\ p_{i,l+1} &= p_{2i,l} \wedge p_{2i+1,l} \\ g_{i,l+1} &= g_{2i+1,l} \vee (g_{2i,l} \wedge p_{2i+1,l})(i+1)2^{l+1} - 1 \end{aligned}$$

Der CLA lässt sich nun wie folgt rekursiv aufbauen:

Für $n = 1$:

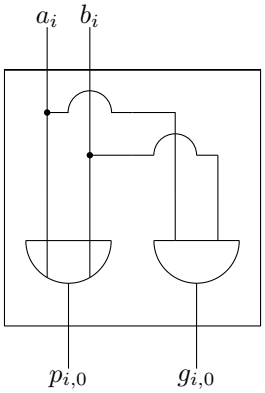


Abbildung 1.8: CLA für n=1.

Sonst ($n > 1$):

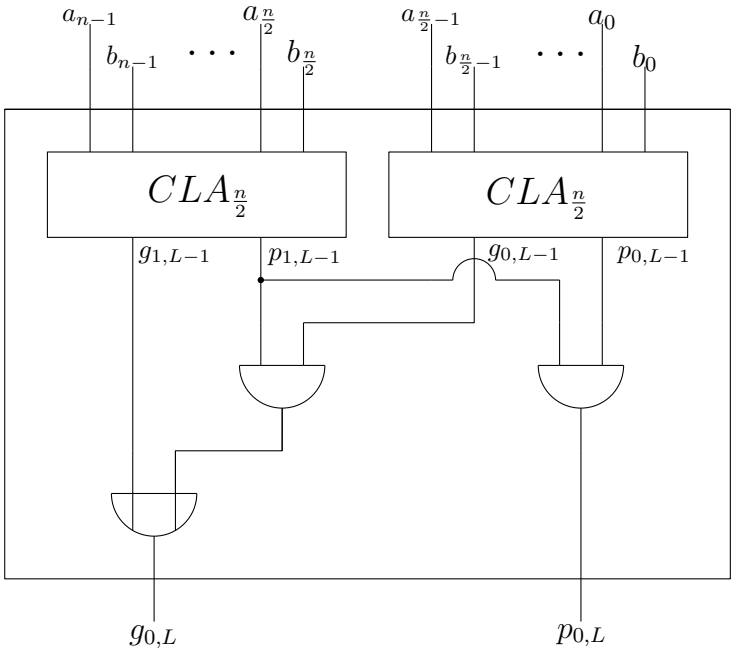


Abbildung 1.9: CLA für n=1.

Damit sind $p_{i,l}$ und $g_{i,l}$ für $l = 0, \ldots, L$ und $i = 0, \ldots, \frac{n}{2^l} - 1$ in Tiefe $O(\log n)$ berechenbar.

Mit Hilfe von $p_{i,l}$ und $g_{i,l}$ lassen sich nun die eigentlichen Überträge berechnen: Sei $\mathbf{C_{i,l}}$ der Übertrag am Ende vom Segment $S_{i,l}$ (bei der Gesamtaddition). Dann gilt:

$$\begin{aligned} c_{2i,l} &= g_{2i,l} \vee (c_{i-1,l+1} \wedge p_{2i,l}) \\ c_{0,L} &= g_{0,L} \\ c_{2i+1,l} &= c_{i,l+1} \end{aligned}$$

Die Rekursionsgleichungen können wir nun zur Formulierung des Pseudocodes für den Schaltkreis der tatsächlichen Überträge nutzen:

Algorithmus 6 Berechnung der eigentlichen Überträge im CLA-Addierer.

for $l := L - 1$ **to** 0 **do**
  for $i := 1$ **to** $\frac{n}{2^l}$ **par do**
    $c_{2i+1,l} = c_{i,l+1}$;
    $c_{2i,l} = g_{2i,l} \vee (c_{i-1,l+1} \wedge p_{2i,l})$;
  end for
end for

Die innere **for**-Schleife braucht $O(1)$ Zeit, da sie vollständig parallel ausgeführt wird. Die äußere **for**-Schleife braucht $O(\log n)$ Zeit. Also ergeben sich für die Überträge folgende Größen:

$$\begin{aligned} \text{Tiefe: } T(n) &= \log n \\ \text{Größe: } G(n) &= \sum_{l=0}^{L-1} \sum_{i=1}^{\frac{n}{2^l}} O(1) = O(n) \end{aligned}$$

Mit den Überträgen können wir nun die endgültige Summe berechnen. Dabei bezeichnet s_j die Summe an der Stelle j, mit $0 \leq j \leq n$, und $c_j = c_{i,l}$ den Übertrag an der Stelle j, mit $j = (2i + 1)2^l - 1$.

Algorithmus 7 Berechnung der Summe.

for $j := 0$ **to** $n - 1$ **par do**
  $s_j = a_j \oplus b_j \oplus c_j$;
end for

Hiermit werden alle Stellen der Summe, bis auf die Letzte (n-te Stelle), ausgerechnet. Sind wir ebenfalls an dieser Stelle interessiert, so kann diese einfach durch $s_n = c_n$ berechnet werden. Das berechnen der Summe benötigt $O(1)$ Zeit; der zugehörige Schaltkreis ist $O(n)$ groß.
Damit ergeben sich für die insgesamte Tiefe und Größe des CLA-Addierers:

$$\begin{aligned} \text{Tiefe: } T(n) &= \log n \\ \text{Größe: } G(n) &= O(n) \end{aligned}$$

1.3 Überblick zur Vorlesung

Die Vorlesung wird nach aktueller Planung folgende Themen ansprechen:

- Einleitung
- Rechnermodelle
- Elementare Techniken in parallelen Algorithmen
- Suchen und Sortieren
- Graphen
- Strings
- Arithmetik
- Parallele Komplexitätstheorie

1.4 Literatur

S. Aki The design and analysis of parallel algorithms

J. Ja' Ja' Introduction to parallel algorithms

F.T. Leighton Introduction to Parallel Algorithms and Architectures

Cormen et. al. Introduction to Algorithms, Kapitel 27, 3. Auflage

Kapitel 2

Berechnungsmodell

2.1 Klassifikation nach Flynn (1972)

Nach Flynn können wir Rechnerarchitekturen wie folgt in grundlegende Klassen einteilen:

{ single, multiple } instruction { single, multiple } data

Daraus ergeben sich die nachfolgenden vier Architekturen:

SISD Single Instruction, Single Data
Diese Klasse enthält alle sequentiellen Einprozessor-Rechnerarchitekturen, wie z.B. Einprozessor-PCs: Hier wird nacheinander eine Operation auf ein einzelnes Datum angewandt.

SIMD Single Instruction, Multiple Data
Diese Architektur enthält Architekturen, bei denen eine Operationen auf viele Daten gleichzeitig angewendet werden kann. Dies trifft zum Beispiel auf sog. Vektorrechner oder die parallelen Berechnungseinheiten einer GPU zu.

MISD Multiple Instruction, Single Data
Sehr unintuitive Klasse; hier werden mehrere Operationen auf dasselbe Datum angewendet. Diese Architektur findet man bei special-purpose-Systemen, z.B. bei redundanter Berechnung zur Erhöhung der Fehlertoleranz.

MIMD Multiple Instruction Multiple Data
MIMD beinhaltet den Aufbau von Rechnernetzen bzw. verteilten Systemen, in denen allgemeine Threads/Prozesse modelliert werden können. Hier können verschiedene Operationen auf viele Daten vollparallel berechnet werden.

In der Vorlesung werden wir ein MIMD-artiges theoretisches Modell für parallele Algorithmen, die **PRAM**, nutzen. Die **PRAM** (**P**arallel **R**andom **A**ccess **M**achine) besitzt beliebig viele Prozessoren, die jeweils auf einen unendlich großen Speicher zugreifen (lesen und schreiben) können. Dabei benötigt jeder Schritt genau eine Zeiteinheit, explizite Synchonisierung wird nicht benötigt.

Der Befehlssatz ist analog zu denen der normalenRegistermaschine, beinhaltet also lesen, schreiben, verschiede arithmetische und Vergleichsoperationen.

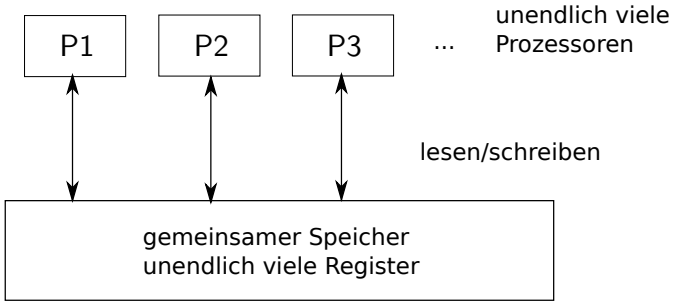


Abbildung 2.1: Schematische Darstellung einer PRAM

2.2 Lese- und Schreibzugriff

Zusätzlich zur Architektur können wir auch das Verhalten bzgl. parallelem Zugriff auf gleiche Register im gemeinsamen Speicher unterscheiden:

{ **E**xclusive, **C**oncurrent } **R**ead { **E**xclusive, **C**oncurrent } **W**rite

Also können wir vier Abstraktionen unterscheiden:

1. EREW
2. CREW
3. ERCW
4. CRCW

Bei der Abstraktion CRCW unterscheidet man drei Verhaltensweisen bei gleichzeitigem Schreibzugriff auf dasselbe Register:

beliebig (engl. *arbitrary*): Keine Kontrolle, wer schreibt.

übereinstimmend (engl. *common*): Schreiben nur dann erfolgreich, wenn alle dasselbe schreiben.

vorrangig (engl. *priority*): Prozessor mit kleinerem Index schreibt.

2.3 Ressourcen/Komplexitätsmaße

Im Gegensatz zur sequentiellen Registermaschine können bei der **PRAM** zusätzliche Ressourcen gemessen werden.
Man unterscheidet:

(parallele) Laufzeit bezeichnet die maximale Befehlsanzahl, die ein Prozessor ausführt.

Zahl d. Prozessoren die aktiv werden.

Speicherplatz bezeichnet die Anzahl der benutzen Register.

Gesamtaufwand (engl. *work*) ist die Summe der Befehle pro Prozessor und kann durch das Produkt Laufzeit · Zahl d. Prozessoren abgeschätzt werden.

Wie üblich wird der Ressourcenverbrauch als Funktion in der Größe der Eingabe angegeben. So betrachten wir als worst-case-Komplexität das Maximum und als durchschnittliche Komplexität den Mittelwert des ressourcenverbrauchs über alle Eingaben der Eingabegröße.
Als Funktionsnamen verwenden wir

$T(n)$	für	Zeit
$P(n)$	für	Prozessoren
$S(n)$	für	Speicher
$W(n)$	für	Aufwand