UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

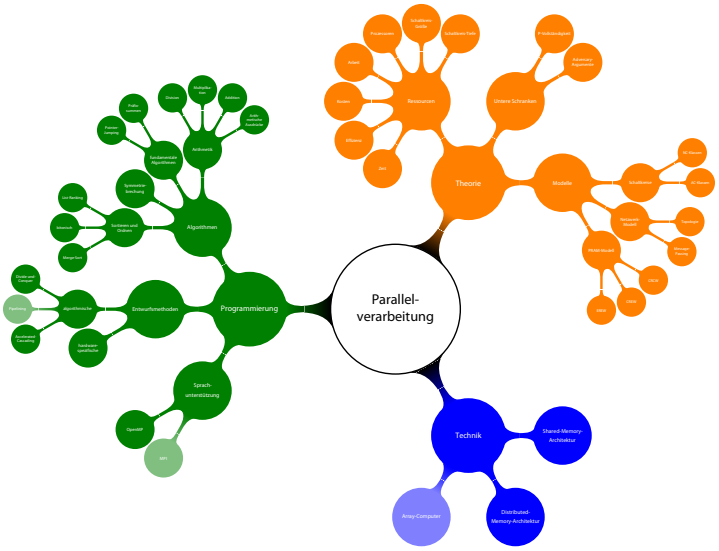
Vorlesungsskript

Parallelverarbeitung

CS3051, CS4502, Sommersemester 2018

Fassung vom 12. Juli 2018

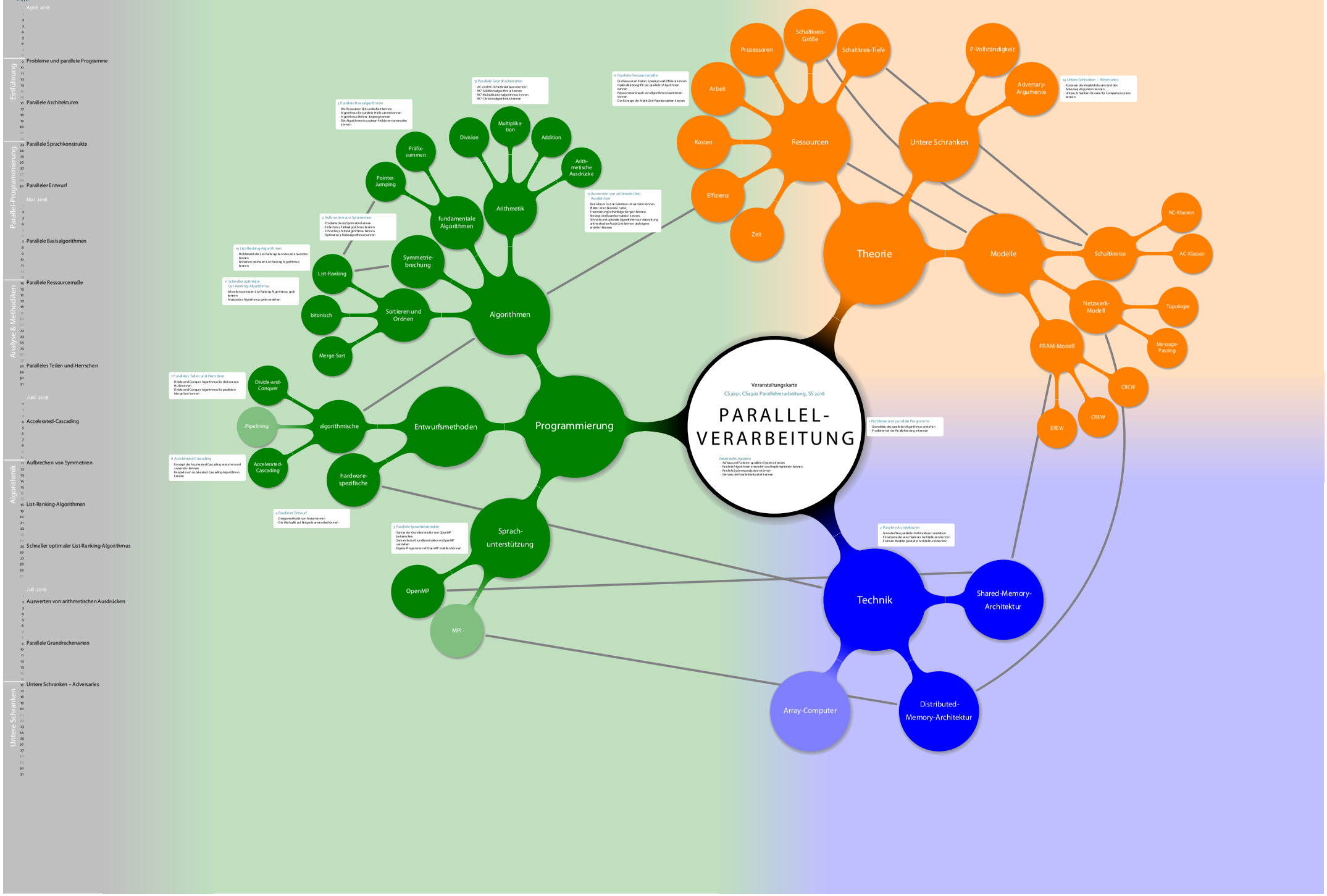
Prof. Dr. Till Tantau



UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK

TILL TANTAU
WWW.TCS.UNI-LUEBECK.DE

13. JULI 2018



Einführung
- April 2018
- Probleme und parallele Programme
- Parallele Architekturen

Parallel-Programmierung
- Parallele Sprachkonstrukte
- Paralleler Entwurf
- Mai 2018
- Parallele Basisalgorithmen

Analyse & Methodiken
- Parallele Ressourcenmaße
- Parallelles Teilen und Herrschen
- Juni 2018
- Accelerated-Cascading

Algorithmik
- Aufbrechen von Symmetrien
- List-Ranking-Algorithmen
- Schneller optimaler List-Ranking-Algorithmus

Untere Schranken
- Juli 2018
- Auswerten von arithmetischen Ausdrücken
- Parallele Grundrechenarten
- Untere Schranken – Adversaries

Inhaltsverzeichnis

Vorwort

Liebe Leserin, lieber Leser, willkommen in einer Welt, in der alles gleichzeitig passiert. Es ist eine besondere Welt mit ihren ganz eigenen Regeln; Dinge, die im Sequentiellen völlig trivial sind, müssen mittels abenteuerlich komplizierter Algorithmen beackert werden, dagegen sind andere Dinge plötzlich ganz fix zu lösen.

Um ein Gefühl für die Probleme zu bekommen, stellen Sie sich vor, Sie bekommen ein dickes, fettes Übungsblatt, das Sie in einer Woche bearbeiten müssen. Stellen Sie sich weiter vor, dass Sie aufgrund einer ganzen Verkettung von widrigen Umständen erst eine Stunde vor Abgabeschluss damit beginnen, das Blatt zu bearbeiten. Sie stellen schnell fest, dass Sie das Blatt alleine nicht werden lösen können. Die Lösung liegt aber auf der Hand: Arbeitsteilung! Hat das Blatt drei Teilaufgaben, so kann man die Arbeit logischerweise auf drei Leute gut verteilen. (Dies werden wir einen *Speedup* von 3 nennen.) Nehmen wir an, auch dies reicht noch nicht aus, da jede Aufgabe alleine schon über eine Stunde dauert. Sie probieren wieder, auf Arbeitsteilung zu setzen: Pro Aufgabe diskutieren nun vier Studierende, wie man sie lösen sollte. Wie Sie sicherlich aus eigener Erfahrung wissen, können Sie nicht erwarten, dass vier Leute eine Aufgabe vier Mal so schnell gelöst bekommen wie eine Person. Aber etwas schneller wird es schon gehen, sagen wir doppelt so schnell. (Wir werden davon reden, dass die *Effizienz* auf 50% sinkt.) Wenn Sie nun auch mit den mittlerweile zwölf Studierenden nicht rechtzeitig fertig werden, dann könnten Sie auf die Idee kommen, das ganze Semester an den Aufgaben arbeiten zu lassen. Dabei werden Sie aber wahrscheinlich feststellen, dass Sie hauptsächlich damit beschäftigt sein werden, Kaffee von A nach B zu transportieren und die Leute davon zu überzeugen versuchen, doch bitte weiterzuarbeiten. (Wir werden sagen, dass auch mit beliebig vielen fleißigen Helfern eine Zeit T^∞ nicht unterschritten werden kann und dass irgendwann mehr kommuniziert wird als gerechnet wird.) Moral: Bitte beginnen Sie rechtzeitig mit den Übungsblättern.

Ein paralleles System ist das elektronische Analogon zu einer Horde Studierende, die gemeinsam ein Übungsblatt zu lösen versuchen. Je nach Art des Übungsblatts gelingt dies mehr oder weniger leicht. Diese Vorlesung beschäftigt sich ausführlich mit der Frage, welche Probleme eben gerade gut parallelisierbar sind und welche weniger gut.

Bis vor etwa 10 Jahre war es eine eher akademische Frage, wie man Probleme parallelisiert: Parallele Systeme waren sündhaft teure Maschinen, die meist schon kurz nach ihrer Anschaffung veraltet waren. Software für durchschnittliche Benutzer zu parallelisieren, war reichlich sinnlos: Viel Aufwand und keinerlei Nutzen, die Programme wurden eher langsamer als vorher. Es gab ein Henne-Ei-Problem: Da es keine parallele Hardware gab, lohnte es sich nicht, parallele Programme zu schreiben – und da es keine parallelen Programme gab, lohnte es sich nicht, parallele Hardware zu bauen. Neuerdings hat sich nun aber mit den Multi-Core-Systemen die Sachlage gewandelt: Es lohnt sich, parallele Software auch tatsächlich zu schreiben. Genau dies werden Sie in Grundzügen in dieser Vorlesung lernen.

Soviel zum Inhalt der Veranstaltung; nun zu den Zielen. Diese lauten:

1. Aufbau und Funktion paralleler Systeme kennen
2. Parallele Algorithmen entwerfen und implementieren können
3. Parallele Systeme analysieren können
4. Grenzen der Parallelisierbarkeit kennen

Die Wörter »kennen« und »können« tauchen dort recht häufig auf. Um etwas wirklich zu können, reicht es nicht, davon gehört zu haben oder davon gelesen zu haben. Man muss es auch wirklich *getan* haben: Sie können sich tausend Fußballspiele im Fernsehen anschauen, sie sind deshalb noch kein guter Fußballspieler; sie können tausend Stunden World of Warcraft spielen, sie werden deshalb trotzdem keinen Frostblitz auf Ihren Professor geschleudert

bekommen. Deshalb steht bei dieser Veranstaltung der Übungsbetrieb mindestens gleichberechtigt neben der Vorlesung. Der Ablauf ist dabei folgender: In der Vorlesung werde ich Ihnen die Thematik vorstellen und Sie können schon mit dem Üben im Rahmen kleiner Miniübungen während der Vorlesung beginnen. Alle zwei Wochen gibt es ein Übungsblatt, das inhaltlich zu den Vorlesung gehört. Sie müssen sich die Übungsblätter aber nicht »alleine erkämpfen«. Vielmehr gibt es Tutorien, in denen Sie Aufgaben üben werden, die »so ähnlich« wie die Aufgaben auf den Übungsblättern sind. Sie werden feststellen, dass das Lösen der Übungsblätter mit diesen Vorbereitungen immer möglich sein wird und dies auch mit vertretbarem Aufwand.

Ich wünsche Ihnen viel Spaß mit dieser Veranstaltung.

Till Tantau

Teil I

Parallele Programme

Den Auftakt dieser Veranstaltung über Parallelverarbeitung soll eine kleine Einführung in die Programmierung von Parallelrechnern bilden. Etwas systematischer wäre es, zuerst die Theorie und die Algorithmik einzuführen und sich erst dann in die Niederungen der Programmiersprachen zu begeben. Systematischer, aber auch langweiliger – und die Langeweile und der Lernerfolg sind schon immer zerstritten gewesen.

In der Einführung zur Parallelen Programmen soll es primär um folgende Fragen gehen:

- Wie funktioniert Parallelisierung technisch?
- Wie unterstützen Programmiersprachen die Parallelisierung?
- Wie geht man vor, wenn man einen parallelen Algorithmus implementiert?

All diese Fragen haben viele mögliche Antworten und über die Frage, welche die richtige ist, wird genauso heftig gestritten wie über die Frage, ob man nun funktional oder doch lieber imperativ programmieren solle. Genau wie bei der Frage zu funktional versus imperativ gibt es natürlich in Wirklichkeit keine richtige Antwort: Shared-Memory ist weder besser noch schlechter als Message-Passing, es ist einfach eine Alternative mit ihren eigenen Vor- und Nachteilen.

1-1

Kapitel 1
Probleme und parallele Programme
Schneller durch mehr Kerne?

1-2

Lernziele dieses Kapitels

1. Grundidee des parallelen Algorithmus verstehen
2. Probleme bei der Parallelisierung erkennen

Inhalte dieses Kapitels

Worum es heute geht

Gewöhnlich beginnt die Erstellung von Software mit einer Problemstellung – typischerweise ausgeschrieben von einer Kundin, die auch nach der dritten Verhandlungsrunde nicht bereit ist, einen auch nur annähernd kostendeckenden Tagessatz zu bezahlen – und dann entwickeln Sie eine Lösung. Es gibt aber auch die andere Richtung: Ihre Entwicklungsabteilung hat eine (nach Ansicht der Softwareingenieure) ganz dolle Software erstellt und Vertrieb und Marketing müssen dann zusehen, wie sie diese »Solution« an die Frau und den Mann bringen, die gar nicht wussten, dass sie ein Problem hatten. (Merke: »Computer lösen Probleme, die man ohne sie nicht hätte.«)

In diesem kurzen einführenden Kapitel beginnen wir auch mit der Lösung: Parallelverarbeitung. Wenn Sie heutzutage einen Computer von der Stange kaufen, dann wird dieser zwei oder mehr Kerne haben, jeder wird viele Operationen auf mehrere Daten gleichzeitig ausführen können, Ihre Graphikkarte wird viele, viele Shadereinheiten haben und diese sind mittlerweile echte Rechenhengste. Parallele Hardware steht unter jedem Schreibtisch und steckt in jeder Hosentasche, wobei die Anzahl der zur Verfügung stehenden Recheneinheiten in Zukunft immer weiter steigen wird, eine Ende ist nicht in Sicht.

Was sind nun aber die Probleme, die wir mit Parallelverarbeitung lösen können? Diese Frage werden wir im Laufe der späteren Kapitel noch in epischer Breite beantworten; mit diesem Kapitel möchte ich lediglich ein gewisses Bewusstsein für die Möglichkeiten und Grenzen schaffen. Grob gesprochen lautet die Antwort: Viele Probleme lassen sich in völlig trivialer Weise parallelisieren, im Englischen gibt es hierfür den schönen Begriff »embarrassingly parallel«; andere Probleme lassen sich nur mit einem gewissen Aufwand parallelisieren und schließlich gibt es auch viele Probleme, die sich gar nicht parallelisieren lassen. Welches Problem in welche Kategorie gehört, ist oft leider nicht offensichtlich.

1.1 Die Lösung

1.1.1 Arten der Parallelverarbeitung

Grundsätzliche Arten von Parallelverarbeitung. 1-4

1. SISD: single instruction, single data
   »Eine Instruktion auf einen Datensatz anwenden«
   Beispiel: Normale CPU
2. SIMD: single instruction, multiple data
   »Eine Instruktion auf mehrere Datensätze anwenden«
   Beispiel: Graphikprozessor
3. MIMD: multiple instructions, multiple data
   »Unterschiedliche Instruktionen auf unterschiedliche Datensätze anwenden«
   Beispiel: Dual-Core-Prozessoren, große Parallelrechner

1.1.2 Mehr Prozessoren = Schneller?

Das Versprechen der Parallelverarbeitung. 1-5

- Die Kosten, Prozessoren schneller zu machen, steigen exponentiell.
- Die Kosten, die Anzahl der Prozessoren zu erhöhen, steigen linear.

Versprechen

Durch Parallelverarbeitung kann man Probleme billiger schnell lösen als durch schnellere Prozessoren.

1.2 Die Probleme

1.2.1 Erstes Fallbeispiel

Die Matrix-Vektor-Multiplikation. 1-6

Problem

Eingabe Eine $n \times n$ Matrix A und ein Vektor v.

Ausgabe $b = A \cdot v$.

Beispiel

$$\begin{pmatrix} 0 & 200 & 5 & -6 & 1 \\ 1 & 20 & 5 & 6 & 23 \\ 2 & 2 & 5 & -6 & 42 \\ 3 & 0{,}2 & 5 & 6 & 17 \\ 4 & 0{,}02 & 5 & -6 & 0 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 4 \\ 5 \\ 0 \\ -1 \end{pmatrix} = ?$$

✎ **Zur Übung** 1-7

Wie löst man das Problem, wenn man …

- …einen Prozessor hat?
- …zwei Prozessoren hat?
- …drei Prozessoren hat?
- …$n/2$ Prozessoren hat?
- …$n/2+1$ Prozessoren hat?
- …n Prozessoren hat?
- …$2n$ Prozessoren hat?
- …n^2 Prozessoren hat?
- …$2n^2$ Prozessoren hat?

Was bringen mehr Prozessoren? 1-8

Für das Problem Matrix-Vektor-Multiplikation hat die Verdoppelung der Prozessoranzahl folgenden Effekt:

- Für kleine Prozessorzahlen halbiert sich jedesmal die Rechenzeit.
- Für Prozessorzahlen über n gilt dies nicht mehr exakt, aber fast.
- Für Prozessorzahlen in der Nähe von n^2 verringert sich die Rechenzeit kaum.
- Für Prozessorzahlen über n^2 ändert sich die Rechenzeit nicht mehr.

1.2.2 Zweites Fallbeispiel

1-9

Erfüllende Belegungen zählen.

Problem

Eingabe Eine aussagenlogische Formel mit n Variablen.

Ausgabe Anzahl der erfüllenden Belegungen.

Beispiel

Wie viele erfüllende Belegungen hat $(x \vee (z \to y)) \wedge (u \to ((v \to y) \vee z))$?

✎ **Zur Diskussion**

Wie löst man das Problem, wenn man …

- … einen Prozessor hat?
- … zwei Prozessoren hat?
- … drei Prozessoren hat?
- … n Prozessoren hat?
- … 2^n Prozessoren hat?
- … 3^n Prozessoren hat?

1-10

Was bringen mehr Prozessoren?

Für das Zählen erfüllender Belegungen lässt sich bei einem naiven Algorithmus jeder zusätzliche Prozessor optimal nutzen bis es mehr als 2^n Prozessoren werden. Völlig unklar bleibt, ob man das Problem nicht auch ganz anders viel schneller lösen könnte.

1.2.3 Drittes Fallbeispiel

1-11

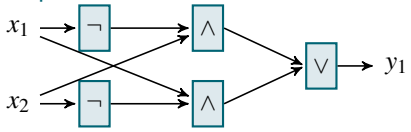
Schaltkreise auswerten.

Problem

Eingabe Eine Schaltkreis plus Belegungen der Eingabegatter, alles als String kodiert.

Ausgabe Werte der Ausgabegatter.

Beispiel



✎ **Zur Diskussion**

Wie löst man das Problem, wenn man …

- … einen Prozessor hat?
- … zwei Prozessoren hat?
- … drei Prozessoren hat?
- … n Prozessoren hat?

1-12

Was bringen mehr Prozessoren?

Stand der Forschung ist, dass sich das Schaltkreisauswertungsproblem nicht parallelisieren lässt.

Zusammenfassung dieses Kapitels

1-13

1. *Parallele Algorithmen* können Probleme schneller lösen als sequentielle Algorithmen.
2. Probleme haben häufig *parallele Anteile* und *sequentielle Anteile*.
3. Man sieht es den Problemen nicht immer an, wie gut sie parallelisierbar sind.

2-1

Kapitel 2
Parallele Architekturen

Quatschen lenkt von der Arbeit ab

Lernziele dieses Kapitels

1. Grundaufbau paralleler Architekturen verstehen
2. Einsatzzwecke verschiedener Architekturen kennen
3. Formale Modelle paralleler Architekturen kennen

2-2

Inhalte dieses Kapitels

Worum es heute geht

Bevor man parallele Programme schreiben kann, sollte man sich erstmal klar werden, auf was für einer Art Parallelrechner das Programm später laufen wird. Bei sequentiellen Programmen ist das weniger wichtig: Einen Quicksort programmiert man immer gleich, egal ob dieser später auf einer Smartcard, einem Supercomputer oder einer Waschmaschine laufen wird. Ganz anders bei parallelen Programmen: Mit einem komplexen parallelen Algorithmus für eine Gravitationssimulation braucht man den Shader-Einheiten einer Graphikkarte nicht zu kommen. Bei diesem Algorithmus müssen sich nämlich die Einheiten ständig ausführlichst beratschlagen, was den doch sehr eigenbrötlicherischen Shadern schwer fällt. In diesem Kapitel soll es um einige typische Arten paralleler Systeme gehen, wobei sich zu parallelen Architekturen noch beliebig viel mehr sagen ließe. Die Auswahl ist so getroffen worden, dass die in der Realität wichtigsten Systeme abgedeckt sind – in Theorie und Praxis.

Zu den verschiedenen Architekturen gehören auch verschiedene Maschinenmodelle. Solche Modelle sind ausgesprochen wichtig, wenn man allgemeine Aussagen der Art »Dieses Problem lässt sich oder lässt sich nicht gut parallelisieren.« Im Prinzip geht man beim Entwurf der Modelle genauso vor wie im sequenziellen Fall: Man beginnt mit einem realen parallelen Computer und modelliert mathematisch alles, was dort so passiert. Dann beginnt man, von Besonderheiten der Architektur zu abstrahieren. Beispielsweise ist es wohl eher unerheblich, ob nun 2 Gigabyte oder 4 Gigabyte Speicher zur Verfügung stehen. Ebenso ist es reichlich

unerheblich, ob es ein A20-Gate gibt. Resultat dieser Vereinfachungen ist idealerweise ein einfaches, elegantes mathematisches Modell mit der Eigenschaft, dass alle anderen sinnvollen Modelle ohne großen Zeit- oder Platzverlust simuliert werden können. Im Sequenziellen hat dieser Ansatz gut funktioniert: Es sind dabei das RAM-Modell und die Turing-Maschine herausgekommen; zwei sehr einfache Modelle, die beide einander simulieren können.

Im Parallelen ist die Sache leider komplexer. Es gibt kein Modell, von dem man behaupten könnte, es sei gleichzeitig einfach, elegant und universell einsetzbar. Ist das Modell schön einfach (wie das PRAM-Modell), so lässt es leider zu viele reale Details weg, in denen bekanntlich der Teufel steckt. Ist es geradezu übermächtig (wie das Netzwerkmodell, in dem die Kommunikation der Prozessoren komplett modelliert wird), so kann man kaum etwas Sinnvolles beweisen. Wir werden später auf das PRAM-Modell setzen, obwohl dieses wie schon gesagt »zu optimistisch« ist. Dies hat aber auch einen wichtigen Vorteil: Können wir zeigen, dass sich ein Problem *noch nicht mal auf einer* PRAM *schnell lösen lässt*, so gilt dies sicher auch für alle »weniger optimistischen« Modelle.

2.1 Grundbegriffe

2.1.1 Parallele Programme

2-4

Was ist ein paralleles Programm?

Grundsätzlich wollen wir mit Hilfe der Parallelverarbeitung *dieselben Probleme* lösen wie im Sequentiellen, wo wir für beliebige Eingabeworte

- auf einer Maschine (real oder nur mathematisch modelliert)
- ein Programm ausführen.

In der Parallelverarbeitung lösen wir ein Problem, indem für beliebige Eingabeworte

- mehreren Rechenwerken einer Maschine (real oder nur mathematisch modelliert)
- jeweils eigene lokale Programme ausgeführt werden (eventuell überall die gleichen).

Vereinbarung zur Sprechweise

Mit einem *parallelen Programm* bezeichnen wir die *Gesamtheit aller Anweisungen* zur Lösung eines Problems auf einer parallelen Maschine.

2.1.2 Kerne und Rechner

2-5

Unterschiede zwischen Kernen, Prozessoren und Rechnern.

Sprechweisen zum Aufbau von parallelen Computern

Der *Rechner* oder auch die *Maschine* (das Gerät als Ganzes) enthält einen oder mehrere *Prozessoren* (Chips), die einen oder mehrere *Kerne* (sequentiell arbeitende Recheneinheiten) beherbergen.

Vor der Einführung der Multi-Core-Systeme waren Kerne und Prozessoren synonym, *Prozessor* meinte dasselbe, was heute *Kern* heißt. Bei Graphikprozessoren heißen die Kerne hingegen traditionell *Shader*. Aus diesen Gründen ist eine einheitliche, abstrakte Sprachregelung sinnvoll:

▶ Definition

Die einzelnen, sequentiell arbeitenden Rechenwerke eines Parallelrechners nennen wir im Folgenden *Einheiten*.

2.1.3 Threads und Tasks

Unterschiede zwischen Programmen, Threads und Tasks

2-6

Was sind Threads?

Ein paralleles Programm spezifiziert verschiedene *potenziell gleichzeitig ausführbare Instruktionsfolgen*, die wir *Threads* nennen. Diese können auf Variablen und Speicherinhalte anderer Threads zugreifen.
Über die Zeit hinweg kann sich die Anzahl der gleichzeitig ausführbaren Aktivitäten ändern, also auch die Anzahl der Threads.

Was sind Tasks?

Eine *Task* ist die Ausführung eines kompletten Programms. Tasks können nicht auf Variablen oder den Speicher anderer Task zugreifen. Tasks und Prozesse sind folglich Synonyme.

2.1.4 Lokaler und globaler Speicher

Unterschiede zwischen lokalem und globalem Speicher.

2-7

Der lokale Speicher

Der *lokale Speicher* eines Thread ist ein Speicherbereich, auf den nur dieser Thread Zugriff hat.

Der globale Speicher

Auf den globalen Speicher haben alle Threads jederzeit gleichzeitig sowohl lesend wie schreibend Zugriff.

Subtilitäten beim Zugriff auf globalen Speicher.

2-8

✎ Zur Diskussion

Wie oft wird Stelle A durchlaufen?

```
// Global:
int counter = 0;

...
// In Thread 1
while (counter < 1000) {
  int temp = counter;
  temp = temp + 1;
  counter = temp;
}

...
// In Thread 2
while (counter < 1000) {
  // Stelle A
  int foo = counter;
  foo = foo + 1;
  counter = foo;
}
```

Zugriffskonflikte zweier Threads lassen sich verschieden regeln.

2-9

Lesezugriffs-Arten

- Exclusive read (*ER*)
- Concurrent read (*CR*)

Schreibzugriffs-Arten

- Owner write (*OW*): Nur eine bestimmte Einheit darf eine bestimmte globale Zelle beschreiben.
- Exlusive write (*EW*)

- Concurrent write (*CW*). Hier gibt es die Untervarianten
  - Common: Nur Schreiben identischer Werte ist erlaubt
  - Arbitrary: Ein beliebiger Wert wird geschrieben
  - Priority: Der Wert der Einheit mit der kleinsten Nummer wird geschrieben.

Sinnvolle Kombinationen sind EROW, EREW, CREW, CROW, CRCW-Common, CRCW-Arbitrary, CRCW-Priority.

2.2 Vektor-Rechner

2.2.1 Idee

2-10

Die einfachsten Arten von Parallelrechnern.

Ein *Vektor-Rechner* oder auch *Prozessor-Array* besteht aus vielen *identischen Einheiten*, die *identische Instruktionen* (also insbesondere identische Threads) ausführen und nicht miteinander kommunizieren (also insbesondere keinen globalen Speicher nutzen).

Vorteile

+ Sehr einfacher Aufbau.
+ Dadurch sehr große Anzahl an Einheiten möglich.

Nachteile

Durch die fehlende Kommunikation sind nur sehr spezielle Probleme lösbar (so genannte *embarrisingly parallel problems*).

2.2.2 Hardware und Modelle

2-11

Hardware-Beispiele von Vektor-Rechnern und ihre Modellierung.

Beispiele von Vektor-Rechnern

- Aktuelle Graphikprozessoren mit vielen Shader-Einheiten.
- Intels Streaming SIMD Extensions (SSE) oder Äquivalentes bei anderen Herstellern.

Modellierung

Man braucht kein abstraktes Modell für Vektorrechner: Man nehme einfach ein sequentielles Modell und wende es auf alle Wörter eines Tupels von Eingabeworten parallel an.

2.2.3 Programmierung

2-12

Wie programmiert man Vektor-Rechner?

Probleme bei der Programmierung von Vektor-Rechnern

Vektor-Rechner haben häufig kein vollständiges Instruktionssatz. Beispielsweise fehlen oft Alternativen (If-Then) und Sprünge. Deshalb kann man normalen Programmtext (beispielsweise in C oder Java) nicht für sie übersetzen.

Möglichkeiten der Programmierung von Vektor-Rechnern

- Man schreibt das Programm direkt in der Maschinensprache, die die Einheiten verstehen.
- Man benutzt eine Bibliothek, die gut implementierte Grundprogramme anbietet.
- Man benutzt einen speziellen Compiler, der eingeschränkten Hochsprachencode in Maschinensprache übersetzt.

2.3 Shared-Memory-Architekturen

2.3.1 Idee

Parallelrechner mit gemeinsamem Speicher.

2-13

In einem Parallelrechner mit *Shared-Memory-Architektur* arbeiten einige, typischerweise identische Einheiten, die eigene Threads ausführen und über den globalen Speicher kommunizieren. Manchmal ist der Speicher nur virtuell global (wird von der Hardware vorgegaukelt).

Vorteile

+ Praktisch und theoretisch gut verstanden
+ Leicht zu programmieren
+ Natürliche Erweiterung des Standard-Modells eines Rechners.

Nachteile

– Globaler Speicher ist technisch schwierig zu realisieren.
– Globaler Speicher ist langsam.
– Cache-Kohärenz ist schwer zu garantieren.

2.3.2 Hardware und Modelle

Hardware-Beispiele von Shared-Memory-Rechnern und ihre Modellierung.

2-14

Hardware-Beispiele von Shared-Memory-Rechnern

– Moderne Computer mit mehreren Prozessoren und/oder Kernen.
– Mayflower Parallelrechner.

Copyright Univesität zu Lübeck

Modellierung von Shared-Memory-Rechnern

– Das PRAM-Modell (dazu gleich mehr).
– Das Series-Parallel-Modell (dazu nächstes Mal mehr).

Modell eines Shared-Memory-Rechners: Die PRAM

2-15

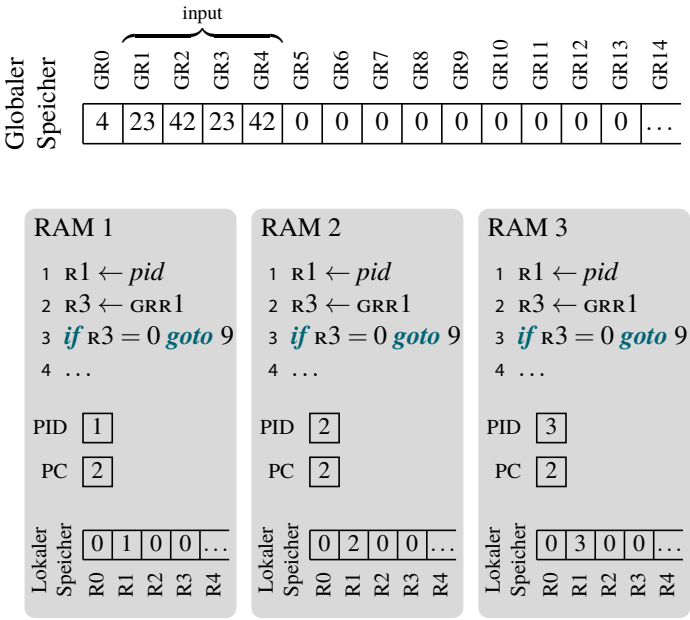
Die Abkürzung PRAM steht für Parallel Random Access Machine. Jede Einheit ist eine eigene RAM. Das bedeutet:

– Eine RAM hat Zugriff auf einen Satz Register, in denen Zahlen stehen.
– Eine RAM arbeitet ein festes Programm ab.

Neu im Vergleich zur einfachen RAM *ist:*

– Es gibt noch *globale Register*.
– Es gibt noch ein *nur lesbares Einheit-ID-Register (*PID*)*.
– Es gibt neue Befehle, um die globalen Register zu lesen und zu beschreiben.

Die Einheiten arbeiten synchron (exakt gleichgetaktet). Auf jeder Einheit läuft ein Thread, der durch ein für alle gleiches Programm beschrieben wird. Bei jeder Einheit ist anfangs nur die PID anders initialisiert. Ein- und Ausgabe liegen in bestimmten Bereichen des globalen Speichers.



2-16

Beispiel eines PRAM-Programms (in Pseudocode).
Folgendes Programm berechnet die Summe S der Zahlen im Array A im globalen Speicher.

```
1 n ← length(A)
2 for h ← 1 to log n seq do
3     if pid ≤ n/2^h then
4         global read a ← A[2 · pid]
5         global read b ← A[2 · pid − 1]
6         c ← a + b
7         global write A[pid] ← c
8 if pid = 1 then
9     global write S ← c
```

✎ **Zur Übung**
Welche Schreib-/Lese-Zugriffsart (EROW, EREW, ...) wird im Programm verwendet?

2.3.3 Programmierung

2-17

Möglichkeiten, Shared-Memory-Systeme zu programmieren.

- Das parallele Programm enthält explizite Anweisungen, die Threads erzeugen, koordinieren und beenden.
  Dazu wird eine Thread-Library benutzt, die vom System bereitgestellt wird.
  Beispiel: Beliebige Sprache mit `pthreads`
- Das parallele Programm enthält Hinweise an den Compiler, welche Teile parallelisierbar sind. Die Erzeugung der Threads übernimmt dann der Compiler.
  Beispiel: `C` mit `OpenMP`
- Das Programm benutzt eine Bibliothek, die große Operationen parallel auf große Datenmengen anwendet. Das Programm selbst ist nicht parallel.
  Beispiel: Beliebige Sprache mit `linpack`
- Das Programm ist in einer Sprache geschrieben, die speziell für Parallelisierung gemacht ist. Der Compiler kümmert sich dann um alles.
  Beispiel: `nesl`, `fortress`

2.3.4 Referenz: Syntax und Semantik der PRAM

Skript-Referenz

Das PRAM-Modell wird das »zentrale« Modell sein, welches wir im Rest der Veranstaltung benutzen werden. Aus diesem Grund ist im Folgenden (nur im Skript) für an Formalisierungen Interessierte eine genaue Beschreibung des Modells angegeben. *Es ist aber nicht nötig, diese genau zu kennen, Algorithmen und Beweise werden wir eher auf der Pseudo-Code-Ebene führen.* (Man beweist auch nicht die Eigenschaften eines Quicksorts anhand einer Implementation in Maschinensprache…) Es gibt verschiedene Arten, RAMs und PRAMs zu formalisieren, die folgende Formalisierung ist eher für Komplexitätsbetrachtungen geeignet, da es nur wenige Befehle gibt und »aufwendige« Berechnungen wie die Multiplikation explizit *nicht* als ein Befehl zur Verfügung stehen.

▶ Definition: Syntax eines PRAM-Programms

Ein PRAM-*Programm* besteht aus einer endlichen, durchnummerierten Folge von Befehlen, wobei die Zählung mit 1 beginnt. In Befehlen kommen *Registeradressierungen* vor, wovon es genau folgende vier Arten gibt (im Folgenden sei immer $i, j, k \in \mathbb{N}$):

1. Ri bezeichnet eine *direkte lokale Adressierung*
2. RRi bezeichnet eine *indirekte lokale Adressierung*
3. GRi bezeichnet eine *direkte globale Adressierung*
4. GRRi bezeichnet eine *indirekte globale Adressierung*

Folgende Befehle sind möglich:

- Die Transportbefehle
  1. R$i \leftarrow$ Rj
  2. R$i \leftarrow$ RRj
  3. R$i \leftarrow$ GRj
  4. R$i \leftarrow$ GRRj
  5. RR$i \leftarrow$ Rj
  6. GR$i \leftarrow$ Rj
  7. GRR$i \leftarrow$ Rj
- Die arithmetischen Befehle
  8. R$i \leftarrow k$
  9. R$i \leftarrow pid$
  10. R$i \leftarrow$ Rj + Rk
  11. R$i \leftarrow$ Rj − Rk
  12. R$i \leftarrow \lfloor \frac{1}{2}Rj \rfloor$
  13. R$i \leftarrow$ lso(Rj)
      (Hier steht »lso« für »least significant one«.)
- Die Sprungbefehle
  14. ***if*** R$i = 0$ ***goto*** k
  15. ***if*** R$i > 0$ ***goto*** k
  16. ***goto*** k

  Bei allen Sprungbefehlen muss k die Nummer einer Programmzeile sein.
- Die Stopp- und Idlebefehle
  17. ***idle while*** GRR$i = 0$
  18. ***stop***

  Der letzte Befehl eines Programms muss der Stoppbefehl sein.

Ein Beispiel eines PRAM-Programms wäre folgendes:

```
1  R0 ← pid
2  R3 ← GR0
3  R2 ← 1
4  R1 ← R0 + R2
5  R4 ← R1 − R3
6  if R4 > 0 goto 13
7  R5 ← GRR0
8  R6 ← GRR1
9  R7 ← R6 + R7
10 GRR0 ← R7
11 R2 ← R2 + R2
12 goto 4
13 stop
```

▶ Definition: Semantik eines PRAM-Programms

Sei P ein PRAM-Programm. Eine PRAM hat einen *globalen Speicher* sowie eine unbeschränkte Anzahl an Einheiten, die alle einen *lokalen Speicher* besitzen. Eine PRAM arbeitet getaktet, wobei mit Takt $t = 0$ begonnen wird. Der Inhalt des i-ten globalen Registers zum Zeitpunkt t ist eine natürliche Zahl, welche wir mit $\langle \text{GR}i \rangle_t$ bezeichnen. Der Inhalt des i-ten lokalen Registers der p-ten Einheit zum Zeitpunkt t bezeichnen wir mit $\langle \text{R}i \rangle_t^p$. Weiterhin verfügt jede Einheit über einen Programmschrittzähler, dessen Wert zum Zeitpunkt t mit $\langle \text{PC} \rangle_t^p$ bezeichnet wird.

Die Gesamtheit der Inhalte aller lokalen und globalen Register (dies sind abzählbar unendlich viele) bezeichnen wir als *Konfiguration*. Eine *Berechnung* ist eine Folge von Konfigurationen, die den weiter unten angegebenen Bedingungen genügt. Eine *Eingabe* für eine PRAM ist ein Vektor v von natürlichen Zahlen. Zu einer Eingabe gehört die *Anfangskonfiguration* bei der alle lokalen Register 0 sind, alle Programmschrittzähler 1 sind, das erste globale Register (GR0) die Dimension n von v enthält und die globalen Register GR1 bis GRn die Komponenten von v enthalten. Eine *Endkonfiguration* ist eine Konfigurationen, in der der Programmschrittzähler aller Einheiten eine Nummer enthält, für die der Programmbefehl »Stopp« lautet.

Die *Berechnungsrelation* gibt an, welche neue Konfiguration zum Zeitpunkt $t+1$ aus der Konfiguration zum Zeitpunkt t hervorgeht. Dazu führen alle Einheiten gleichzeitig den jeweiligen Befehl aus, der im PRAM-Programm P in der Zeile $\langle \text{PC} \rangle_t^p$ steht. Die Effekte dieser Befehle sind in der folgenden Tabelle angegeben (es sei $\text{lso}(0) = 0$ und $\text{lso}(x) = \min\{i \mid x \bmod 2^i \neq 0\}$ für $x \geq 1$):

Befehl	*Wirkung, wenn von Einheit p ausgeführt*
R$i \leftarrow$ Rj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p$
R$i \leftarrow$ RRj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R}\langle \text{R}j \rangle_t^p \rangle_t^p$
R$i \leftarrow$ GRj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{GR}j \rangle_t$
R$i \leftarrow$ GRRj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{GR}\langle \text{R}j \rangle_t^p \rangle_t$
RR$i \leftarrow$ Rj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}\langle \text{R}i \rangle_t^p \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p$
GR$i \leftarrow$ Rj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{GR}i \rangle_{t+1} = \langle \text{R}j \rangle_t^p$, aber siehe unten
GRR$i \leftarrow$ Rj	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{GR}\langle \text{R}i \rangle_t^p \rangle_{t+1} = \langle \text{R}j \rangle_t^p$, aber siehe unten
R$i \leftarrow k$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = k$
R$i \leftarrow pid$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = p$
R$i \leftarrow$ R$j +$ Rk	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p + \langle \text{R}k \rangle_t^p$
R$i \leftarrow$ R$j -$ Rk	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \max\{\langle \text{R}j \rangle_t^p - \langle \text{R}k \rangle_t^p, 0\}$
R$i \leftarrow \lfloor \frac{1}{2}Rj \rfloor$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \lfloor \langle \text{R}j \rangle_t^p / 2 \rfloor$
R$i \leftarrow$ lso(Rj)	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \text{lso}(\langle \text{R}j \rangle_t^p)$
goto k	$\langle \text{PC} \rangle_{t+1}^p = k$,
if R$i = 0$ ***goto*** k	$\langle \text{PC} \rangle_{t+1}^p = k$, falls $\langle \text{R}i \rangle_t^p = 0$, sonst $\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$
if R$i > 0$ ***goto*** k	$\langle \text{PC} \rangle_{t+1}^p = k$, falls $\langle \text{R}i \rangle_t^p > 0$, sonst $\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$
idle while GRR$i = 0$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, falls $\langle \text{GR}\langle \text{R}i \rangle_t^p \rangle_t > 0$, sonst keine Änderung
stop	keine Änderungen

Es kann vorkommen, dass mehrere Einheiten gleichzeitig auf ein globales Register schreibend zugreifen. Dann können Konflikte entstehen und der Wert von $\langle \text{GR}i \rangle_{t+1}$ ist nicht wohldefiniert, da unterschiedliche Angaben vorliegen können. Dieses Problem wird wie folgt aufgelöst:

1. Beim *Owner-Write-Modell* darf auf das Register GRi nur Einheit i schreiben. Die Nachfolgekonfiguration ist nicht definiert, falls eine Einheit sich nicht daran hält.
2. Beim *Exclusive-Write-Modell* darf in jedem Takt für jedes globale Register i nur genau eine Einheit $\langle \text{GR}i \rangle_{t+1}$ einen Wert zuweisen. Versuchen dies mehrere, so gibt es wiederum keine Nachfolgekonfiguration.
3. Beim *Arbitrary-Write-Modell* gibt es im Falle von unterschiedlichen Werten, die von verschiedenen Einheiten in dasselbe globale Register GRi geschrieben werde, für jeden Wert eine eigene Nachfolgekonfiguration, in der dieser Wert eingenommen wird (das Verhalten ist also nichtdeterministisch).
4. Beim *Common-Write-Modell* müssen alle Einheiten denselben Wert schreiben. Geschieht dies nicht, ist die Nachfolgekonfiguration wieder nicht definiert.
5. Beim *Priority-Write-Modell* sei P_i die Menge aller Einheitennummern von Einheiten, die versuchen, im Zeittakt t den Wert von $\langle \text{GR}i \rangle_{t+1}$ zu verändern. Sei $p_i = \min P_i$ und sei x_i der Wert, den p_i schreiben möchte. Dann ist $\langle \text{GR}i \rangle_{t+1} = x_i$.

Während eine Einheit den Idle-Befehl ausführt (und somit schläft) zählt er unter Umständen nicht zu den aktiven Prozessoren und liefert keinen Beitrag zur so genannten *Arbeit* des Systems.

2.4 Distributed-Memory-Architekturen

2.4.1 Idee

Super-Computer haben keinen globalen Speicher.

2-18

Bei einer *Distributed-Memory-Architektur* arbeiten viele Einheiten, die unterschiedliche Threads abarbeiten und über *Message-Passing* kommunizieren, da es *keinen globalen* Speicher gibt.

Vorteile

+ Skaliert sehr gut (lässt sich leicht fast beliebig erweitern).
+ Funktioniert auch mit heterogenen Einheiten gut.

Nachteile

Schwierig zu programmieren, da die Kommunikation »mitprogrammiert« werden muss.

2.4.2 Hardware und Modelle

Hardware-Beispiele von Distributed-Memory-Systemen und ihre Modellierung.

2-19

Hardware-Beispiele von Distributed-Memory-Systemen

Platz 1 der Top-500-Computerliste: Googeln wir einmal…

Modellierung von Distributed-Memory-Systemen als Graphen

Man modelliert das feste Kommunikationsnetzwerk durch einen Graphen. Die *Knoten* des Graphen sind die Einheiten. Die *Kanten* des Graphen sind Kommunikationskanäle. Die Einheiten arbeiten in der Regel *asynchron* und schicken sich gegenseitig Nachrichten.

Skript-Referenz

Die wichtigstens Parameter eines Netzwerks sind:

Parameter Durchmesser

Maximale Entfernung zwischen Knoten: Dies ist die Zeit, die eine Nachricht höchstens braucht. *Kleiner ist besser.*

Parameter Grad

Maximale Anzahl von Nachbarn eines Knoten: Je höher der Grad, desto mehr Kabel muss man ziehen. *Kleiner ist besser.*

Parameter Zusammenhänge

Minimale Anzahl von Knoten/Kanten, die man löschen muss, um das Netz in zwei Teile zu spalten: Je höher der Zusammenhang, desto eher lässt sich umrouten. *Größer ist besser.*

Zu den typischen Topologien, nach denen Netzwerke aufgebaut sein können, gehören:

- Ring
- 2D-Gitter
- 2D-Torus
- 3D-Gitter
- 3D-Torus
- Hyperwürfel
- Binärbaum

Die »Güte« dieser Topologien ist unterschiedlich. Für Graphen mit n Knoten und den unterschiedlichen Topologien gilt:

Topologie	Durchmesser	Grad	Zusammenhang
Ring	$n/2$	2	2
2D-Gitter	$2\sqrt{n}$	4	2
2D-Torus	$\sqrt{n}$	4	4
3D-Gitter	$3\sqrt[3]{n}$	6	3
3D-Torus	$\sqrt[3]{n}$	6	6
Hyperwürfel	$\log_2 n$	$\log_2 n$	$\log_2 n$
Binärbaum	$\log_2 n$	3	1

2.4.3 Programmierung

2-20

Programmierung mittels Message-Passing.
In Distributed-Memory-Systemen müssen sich die Einheiten mittels *Message-Passing* abstimmen. Das Message Passing Interface (MPI) ist eine Bibliothek, die dieses erleichtert.

Was MPI leistet
Es stellt Methoden zur Verfügung, Threads zu erstellen und automatisch auf Einheiten zu verteilen. Weiterhin stellt es Methoden zur Verfügung, um typische Kommunikationsaufgaben zu erledigen:

- Verteilung von Daten an alle Einheiten
- Sammeln von (Ergebnis-)Daten bei einer Einheit
- Schicken von Daten von Einheit zu Einheit
- Schicken von Daten von allen Einheiten zu allen Einheiten

In MPI geschriebene Programme können auch auf einem Shared-Memory-System ausgeführt werden, die Kommunikation geschieht dann viel schneller über den globalen Speicher.

2-21

Beispiel eines MPI-Programms

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char* argv[]) {
  int i;
  int id; /* Unit rank */
  int p;  /* Numer of units */

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);

  /* Do up to 1000 things in parallel */
  for (i=id; i < 1000; i+=p)
    do_something_for_thing (i);

  printf ("Process␣%d␣is␣done\n", id);
  fflush (stdout);
  MPI_finalize ();
  return 0;
}
```

Zusammenfassung dieses Kapitels

2-22

▶ **Grundbegriffe**

Einheit Abstrakter Begriffe für Core oder Prozesser. Arbeitet physikalisch parallel zu anderen Einheiten.

Thread Instruktionsfolge, die eine Einheit (potentiell) parallel zu anderen Threads auf anderen Einheiten ausführen kann.

Paralleles Programm Ein Programmtext, in dem explizit oder implizit mehrere Threads erzeugt werden.

▶ **Speicherorganisation bei parallelen Programmen**

Lokaler Speicher Speicher, auf den ein Thread exklusiven Zugriff hat.

Globaler Speicher Speicher, auf den alle Threads Zugriff haben.

- **EROW** Exclusive Read, Owner Write
- **EREW** Exclusive Read, Exclusive Write
- **CROW** Concurrent Read, Owner Write
- **CREW** Concurrent Read, Exclusive Write
- **CRCW** Concurrent Read, Concurrent Write

▶ Architekturen 1: Vektor-Rechner

- Idee: SIMD
- Programmierung: Spezialsprachen, Bibliotheken
- Beispiele: einfache Shader, SSE-Befehlserweiterungen

▶ Architekturen 2: Shared-Memory-Architekturen

- Idee: MIMD, globaler Speicher
- Mathematisches Modell: PRAM
- Programmierung: OpenMP
- Beispiele: Multi-Kern-Systeme

▶ Architekturen 3: Distributed-Memory-Architekturen

- Idee: MIMD, verteilter Speicher, Message-Passing
- Mathematisches Modell: Kommunikationsgraph
- Programmierung: MPI
- Beispiele: Super-Computer

Übungen zu diesem Kapitel

Übung 2.1 Einfache Programmanalyse, leicht

Gegeben sei ein globales Array A der geraden Länge n, sowie das folgende Programm:

```
1 global read x ← A[pid]
2 global read y ← A[pid + n/2]
3 z ← x + y
4 global write A[pid] ← z
5 global write A[pid + n/2] ← z
```

Beantworten Sie folgende Fragen in Bezug auf das Programm:

1. Was macht das Programm?
2. Welche Zugriffsart (wie zum Beispiel EREW, Priority-CRCW, etc.) benötigt das Programm? Begründen Sie Ihre Antwort.
3. Wie schnell lässt sich das Problem sequentiell lösen?

Übung 2.2 Ein PRAM-Programm für die Matrixtransposition erstellen, einfach

Wir wollen zu einer $n \times n$-Matrix A ihre transponierte Matrix berechnen.

1. Beschreiben Sie eine Parallelisierung mit $p \leq n$ Prozessoren für die Matrix-Transposition,
2. Geben Sie für Ihren Ansatz aus 1. ein PRAM-Programm in Pseudocode an (ein Beispiel hierzu finden Sie im Skript auf Seite 2-16).
3. Geben Sie die Laufzeit Ihres Programms in Abhängigkeit von der Eingabegröße und der Anzahl der Prozessoren an.

Übung 2.3 Ein PRAM-Programm für die Matrixmultiplikation erstellen, mittel

Sie haben mögliche Parallelisierungen für die Matrix-Vektor-Multiplikation kennengelernt. Nun soll eine Parallelisierung der Matrix-Matrix-Multiplikation entwickelt und als PRAM-Programm formuliert werden. Die Multiplikation einer quadratischen ganzzahligen $n \times n$ Matrix A mit einer $n \times n$ Matrix B ergibt eine $n \times n$ Matrix C mit folgenden Einträgen $C[i, j]$:

$$C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]$$

1. Beschreiben Sie eine Parallelisierung mit $p \leq n$ Prozessoren für die Matrix-Matrix-Multiplikation.
2. Geben Sie für Ihren Ansatz aus 1 ein PRAM-Programm in Pseudocode an, ein Beispiel hierzu finden Sie im auf Folie 2-16.
3. Geben Sie die Laufzeit Ihres Programms in Abhängigkeit von der Eingabegröße und der Anzahl der Prozessoren an.

3-1

Kapitel 3
Parallele Sprachkonstrukte

Wie sag' ich's meinem Compiler?

3-2

Lernziele dieses Kapitels

1. Syntax der Grundkonstrukte von OpenMP beherrschen
2. Semantik der Grundkonstrukte von OpenMP verstehen
3. Eigene Programme mit OpenMP erstellen können

Inhalte dieses Kapitels

Worum es heute geht

In diesem Kapitel soll es nun zum ersten Mal darum gehen, *tatsächlich* auf einem echten Parallelrechner zu programmieren. Dazu wird OpenMP vorgestellt, ein aktueller, sich immer stärker verbreitender Standard für die Programmierung von Shared-Memory-Architekturen (zur Erinnerung: das ist die Architektur von Rechnern, die sie beim Elektronik-Discounter Ihres Vertrauens bekommen).

Die Idee hinter OpenMP ist elegant: Anstatt eine ganz neue Sprache zu schaffen und damit die sowieso schon überarbeitete Programmiererzunft zu zwingen, alle ihre Programme neu zu schreiben, geht OpenMP von bestehendem Programmtext aus. Tatsächlich braucht ein Compiler OpenMP gar nicht zu kennen, er wird die Programme klaglos übersetzen können – diese werden eben nur sequentiell und damit langsamer sein. Die Anwesenheit von OpenMP äußert sich lediglich durch »Hinweise« an einen willigen Compiler, dass diese oder jene Schleife doch ganz toll parallelisierbar wäre, falls es nicht zu viele Umstände machte. Der Grundkonsens lautet »alles kann, nichts muss.«

In der ersten Version von OpenMP konnte man im Wesentlichen dem Compiler den Hinweis geben, dass eine Schleife parallelisierbar ist, das war's. Mit der Zeit sind dann natürlich syntaktische Erweiterungen, Sonderbehandlungen und immer obskurere Spezialfälle hinzugekommen, so dass der OpenMP-Standard in der Version 3.1 schon auf 354 Seiten angeschwollen ist. Dieses Kapitel stellt eine Destillat dieser 354 Seiten dar, mit dem sich schon ganz ansehnliche parallele Programme anrühren lassen.

3.1 Das Fork-Join-Modell

Das Fork-Join-Modell der Parallelverarbeitung. 3-4

Während ein paralleles Programm abläuft, kann sich die Anzahl der *aktuell parallel ausführbaren Aktivitäten* ändern:

- Anfangs gibt es nur sequentielle Aktivität wie Initialisierungen.
- Irgendwann lassen sich dann (prinzipiell) mehrere Arbeitspakete gleichzeitig erledigen.
  Beispiel: Für jede Zeilen einer Matrix muss etwas getan werden.
- Dann können in manchen Arbeitspaketen vielleicht wieder Sub-Arbeitspakete entstehen.
  Beispiel: In jeder Zeile lassen sich baumartig Summen bilden.
- Irgendwann sind die parallelen Aktivitäten zu Ende.
- Danach können dann wieder neue, andere Arbeitspakete entstehen.

Idee hinter dem Fork-Join-Modell

Das Fork-Join-Modell beschreibt, wo Arbeitspakete entstehen (*Fork*) und wo sie wieder beendet werden (*Join*). Das Modell beschreibt *nicht*, wie diese Arbeitspakete auf die vorhandenen Einheiten verteilt werden.

3.1.1 Theorie: Seriell-parallele Graphen

Theoretisches Fundament des Fork-Join-Modells: Seriell-parallele Graphen. 3-5

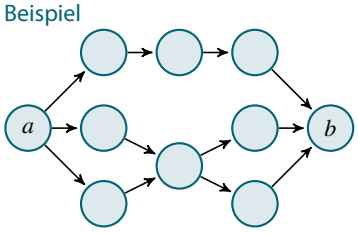
▶ Definition: Seriell-paralleler Graph

Seriell-parallele Graphen sind gerichtete Graphen G mit einer *Quelle* a und einer *Senke* b. Sie sind induktiv definiert:

1. Eine einzelne Kante von der Quelle zur Senke ist ein seriell-paralleler Graph.
2. Sind (G_1, a_1, b_1) und (G_2, a_2, b_2) seriell-parallele Graphen, so auch ihre *serielle Komposition*. Sie ergibt sich, indem man G_1 und G_2 zunächst disjunkt vereinigt und dann die Senke b_1 mit der Quelle a_2 identifiziert.
3. Sind (G_1, a_1, b_1) und (G_2, a_2, b_2) seriell-parallele Graphen, so auch ihre *parallele Komposition*. Sie ergibt sich, indem man G_1 und G_2 zunächst disjunkt vereinigt und dann einerseits die Quellen a_1 und a_2 und anderseits die Senken b_1 und b_2 identifiziert.

Beispiel eines seriell-parallelen Graphen. 3-6

Beispiel



3.1.2 Praxis: OpenMP

Praktische Umsetzung des Fork-Join-Modells: OpenMP. 3-7

In OpenMP spezifiziert man implizit den seriell-parallelen Graphen der Arbeitspakete: Ein so genanntes `parallel`-Pragma gibt eine Fork-Stelle an. Innerhalb eines solchen Pragmas kann es dann weitere, verschachtelte Fork-Stellen geben. Am Ende des Blocks des Pragmas findet implizit ein Join statt.
Achtung: Bei älteren OpenMP-Implementationen ist die Verschachtelung nicht möglich.

3.2 Grundideen von OpenMP

3.2.1 Pragmas

3-8

Was umfasst OpenMP?

OpenMP ist ein offenes System für die Programmierung von *Shared-Memory*-Multi-Prozessoren. Es stellt Möglichkeiten zur Verfügung, dem *Compiler* mitzuteilen, wie der seriell-parallele Graph der Arbeitspakete aussieht. Dies geschieht hauptsächlich mittels *Pragmas* und zu einem kleinen Teil über eine API. OpenMP ist im Prinzip sprachunabhängig, konkret gibt es Implementationen aber nur für *C*, *C++* und Fortran.

3-9

Was sind OpenMP-Pragmas?

Ein *Pragma* ist ein »pragmatischer Hinweis« an den Compiler. Die Syntax für alle OpenMP-Pragmas lautet:

```
...                 // Normaler C-Text
#pragma omp ...     // Hinweis an den Compiler, ...
{                   // ... der sich auf den
  ...               // folgenden Block bezieht
}
```

Versteht ein Compiler ein Pragma nicht, so *kann er es einfach ignorieren*. Das Programm *muss dann immernoch korrekt funktionieren*.

3-10

Was leistet die OpenMP-API?

Die OpenMP-API (Application Programming Interface) stellt einige wenige Funktionen zur Verfügung, um zur Laufzeit die Anzahl und Nummer des aktuellen Threads herauszufinden:

```
// Aktuelle Anzahl Threads im Team
int omp_get_num_threads(void);

// Aktuelle Thread-Nummer im Team
int omp_get_thread_num(void);

// Anzahl Einheiten ("Processors") auf der Maschine
int omp_get_num_procs(void);

// Setze Anzahl an Threads für neue Teams
void omp_set_num_threads(int num_threads);
```

Anders als bei MPI braucht man die OpenMP-API-Funktionen nur eher selten.

3.2.2 Thread-Erzeugung

3-11

Das Pragma zur Erstellung von Arbeitspaketen: `parallel`.

Syntax

```
#pragma omp parallel ...
{
  // Block
}
```

Auf `parallel` können noch *Konfigurationen* folgen wie:

- `if (...)`
- `num_threads (...)`

Für eine vollständige Liste siehe die OpenMP-Spezifikation.

Semantik

Das `parallels`-Konstrukt gibt an, dass der Block mehrere Arbeitspakete enthält. Der aktuelle Thread teilt sich (Fork) zu einem *Team*, das diese Arbeitspakete dann parallel abarbeitet. Pro Arbeitspaket gibt es genau einen Thread in dem Team und dies ändert sich auch nicht (es können aber neue Teams gebildet werden). Alle Threads des Teams arbeiten den Block einmal parallel ab.

Die genaue Anzahl an Arbeitspakete wird normalerweise automatisch gewählt, kann durch die Angabe von `num_threads` spezifiziert werden und ist 1, wenn eine durch `if` angegebene Bedingung nicht zutrifft. Am Ende des Blocks findet automatisch ein Join aller Threads des Teams statt.

Beispiel: Hallo Welt

3-12

Das folgende Programm druckt fünfmal »Hallo Welt.«, wenn es fünf Einheiten gibt.

```
#include <stdio.h>
#include <omp.h>

int main(){
  #pragma omp parallel num_threads(5)
  {
    printf("Hallo␣Welt.\n");
  }
}
```

Bemerkungen:

- Übersetzt wird dies mit `gcc -fopenmp -o example-3-12 example-3-12.c`.
- In älteren Implementation von OpenMP gibt es noch kein `num_threads`. Hier muss stattdessen die Funktion `omp_set_num_threads` vorher aufgerufen werden.
- In manchen Implementation druckt dies auch fünfmal »Hallo Welt.«, wenn es weniger als fünf Einheiten gibt.

Beispiel: Hallo Welt in C++

3-13

Das Programm kann man natürlich auch in C++ schreiben:

```
#include <iostream>
#include <omp.h>

int main(void){
  #pragma omp parallel num_threads(5)
  {
    std::cout << "Hallo␣Welt." << std::endl;
  }
}
```

Übersetzt wird dies mit `g++ -fopenmp -o example-3-13 example-3-13.c`.

Beispiel: Who am I?

3-14

Das folgende Programm gibt für jeden erzeugten Thread dessen Nummer aus (in irgendeiner Reihenfolge):

```
#include <stdio.h>
#include <omp.h>

int main(){
  #pragma omp parallel num_threads(omp_get_num_procs())
  {
    printf("Thread␣%d.\n", omp_get_thread_num());
  }
}
```

Beispiel: Paralleles Initialisieren

3-15

Das folgende Programm füllt Array-Element *i* mit der Nummer *i*.

```
#include <stdio.h>
#include <omp.h>

int array[10000];

int main(){
  #pragma omp parallel num_threads(13)
```

```
  {
    int i;   // Diese Variable ist thread-private
    for (i = omp_get_thread_num();
         i < 10000;
         i += omp_get_num_threads())
      array[i] = i;
  }
  // Hat alles funktioniert?
  int j;
  for (j = 0; j<10000; j++)
    if (array[j] != j)
      printf("Array-Index␣%d␣ist␣falsch,␣nämlich␣%d\n", j,
          array[j]);
}
```

3.3 Arbeitsteilung

3.3.1 Implizit: For-Schleifen

3-16

Aufteilung der Arbeit.

In einem Parallel-Abschnitt machen zunächst alle Threads dasselbe. Man muss dann mittels `omp_get_thread_num()` innerhalb des Block spezielle Arbeit für den aktuellen Thread auswählen. Für wichtige und typische Fälle, wie For-Schleifen, gibt es spezielle Pragmas, die dies stark erleichtern.

3-17

Das Pragma für parallele For-Schleifen.

Syntax

```
#pragma omp for ...
for (i = start; i < end; i++) {
  // Block
}
```

Die Schleife kann auch eine etwas allgemeinere Form haben (beispielsweise mit `i >= 0` als Abbruchbedingung und `i += x` als Inkrement). Wichtig ist, dass sich sofort berechnen lässt, wie viele Iterationen die Schleife umfasst. Deshalb ist auch ein **`break`** in der Schleife verboten.
Auf **`for`** können noch *Konfigurationen* folgen wie:

- `private (...)`
- `reduction (...:...)`

Für eine vollständige Liste siehe die OpenMP-Spezifikation.

Semantik

Das **`for`**-Konstrukt muss *innerhalb eines `parallel`-Konstrukts* stehen. Es sorgt dafür, dass das Team sich die Arbeit der For-Schleife aufteilt:

- Es wird zunächst ermittelt, wie viele Iterationen die Schleife hat.
- Wenn es n Threads im Team gibt, dann übernimmt jeder dieser Threads ein n-tel dieser Iterationen.

Am Ende der For-Schleife ist eine Barriere, es geht also erst weiter, wenn alle Threads mit der Schleife fertig sind. Es werden keine neuen Threads erzeugt oder gelöscht, es wird *nur Arbeit verteilt*.

Beispiele von parallelen Schleifen.

3-18

Beispiel: Parallele Schleifen

```
int a[10000];
int b[10000];

int main(){
  omp_set_num_threads (omp_get_num_procs ());
  #pragma omp parallel
  {
    int i;
    #pragma omp for
    for (i = 0; i < 10000; i++)
      a[i] = i;

    #pragma omp for
    for (i = 0; i < 10000-1; i+=2)
      b[i] = a[i] + a[i+1];
  }
}
```

Kleine Erleichterung beim Schreiben.

3-19

Man kann die `parallel`- und **`for`**-Pragmas zusammenfassen zu einem einzigen Pragma »`parallel for`«.

Beispiel: Parallele Schleifen

```
int a[10000];
int b[10000];

int main(){
  omp_set_num_threads (omp_get_num_procs ());
  int i;
  #pragma omp parallel for
  for (i = 0; i < 10000; i++)
    a[i] = i;

  #pragma omp parallel for
  for (i = 0; i < 10000-1; i+=2)
    b[i] = a[i] + a[i+1];
}
```

✎ Zur Diskussion

Warum ist obiger Programmtext etwas langsamer als der mit nur einem einzigen `parallel`-Abschnitt?

✎ Zur Übung

3-20

Eine einfache *Glättung* eines Arrays erhält man, indem man jedes Array-Element a_i ersetzt durch $(a_{i-1}+2a_i+a_{i+1})/4$ (Randelement behandelt man sinnvoll besonders). Die n-fache Glättung ergibt sich durch n-fache Anwendung der Glättungsoperation. Geben Sie den Programmtext einer Funktion

```
void n_smooth(int array[], int length, int n)
```

an, die die n-fach Glättung des Arrays berechnet. Natürlich soll dabei sinnvoll parallelisiert werden.

3-21

Scheduling-Strategien.

Normalerweise werden die Iterationen durch das For-Pragma in (fast) gleichgroße Blöcke aufgeteilt. Dauern aber manche Iterationen viel länger als andere, so ist dies nicht immer optimal. Mit der `schedule`-Konfiguration kann man andere Strategien angeben. Die wichtigsten sind:

- `schedule (static, x)`
  Es werden Blöcke von je x Iterationen pro Thread abgearbeitet. Der erste Thread erhält die ersten x Iterationen, der zweite die nächsten x und so weiter. Sind alle Threads versorgt und noch Blöcke übrig, erhält diese wieder erste Thread und so weiter.
- `schedule (dynamic, x)`
  Es werden wieder Blöcke von je x Iterationen pro Thread gebildet. Die Blöcke werden nun aber dynamisch an Threads verteilt: Immer, wenn ein Thread fertig ist, bekommt er den nächsten Block zugewiesen, der noch nicht bearbeitet wurde.

3.3.2 Explizit: Sections

3-22

Explizite Arbeitspakete.

Syntax

```
#pragma omp sections ...
{
  #prama omp section
  {
    ...
  }
  #prama omp section
  {
    ...
  }
  ...
}
```

Auf `sections` können *Konfigurationen* folgen wie:

- `private (...)`

Für eine vollständige Liste siehe die OpenMP-Spezifikation.

Semantik

Jede `section` wird genau einmal von einem der Threads des aktuellen Teams ausgeführt. Wenn weniger Threads als Abschnitte zur Verfügung stehen, werden einige Abschnitte nacheinander ausgeführt. Die Reihenfolge der Ausführung ist implementationsabhängig. Am Ende der `sections` gibt es eine implizite Barriere.

3-23

Beispiel von expliziten Arbeitspaketen.

Beispiel: Explizite parallele Blöcke

```
#include <stdio.h>
#include <omp.h>

int main(){
  #pragma omp parallel sections
  {
    #pragma omp section
    { printf("1\n"); }
    #pragma omp section
    { printf("2\n"); }
    #pragma omp section
    { printf("3\n"); }
    #pragma omp section
    { printf("4\n"); }
    #pragma omp section
```

```
    { printf("5\n"); }
  }
}
```

3.3.3 Koordination

Kennzeichnung kritischer Bereiche.

3-24

Syntax

```
#pragma omp critical ...
{
  ...
}
```

Auf `critical` kann ein Name folgen. Folgt keiner, so wird ein eindeutiger, interner Name vergeben.

Semantik

Erreicht ein Thread einen kritischen Bereich, so betritt er ihn nur, wenn kein anderer Thread einen kritischen Bereich gleichen Namens ausführt. Anderenfalls wartet er.

Beispiel eines kritischen Bereichs.

3-25

Beispiel: Koordiniertes Erhöhen einer gemeinsamen Variable

```
int i;
int sum_uncritical = 0;
int sum_critical = 0;

#pragma omp parallel for
for (i = 1; i <= 1000; i++) {
  sum_uncritical += i;

  #pragma omp critical
  sum_critical += i;
}
printf("sum_critical = %d, sum_uncritical = %d\n",
    sum_critical, sum_uncritical);
```

Für die Erhöhung einer Variable so wie im obigen Beispiel ist ein kritischer Bereich etwas mit Kanonen auf Spatzen schießen. Besser ist das folgende `atomic`-Pragma oder das später beschriebene `reduction`-Pragma.

Skript-Referenz

Syntax

```
#pragma omp atomic
var += something;
```

Semantik

Es wird garantiert, dass der Zyklus »Lesen der Variable, Verändern der Variable, Schreiben der Variable« ohne Unterbrechung abläuft. Dies ist flotter als dasselbe als kritischer Bereich.

Einmalige Ausführung.

3-26

Syntax

```
#pragma omp single
{
  ...
}
```

Semantik

Der Block wird genau einmalig von genau einem Thread im Team ausgeführt.

Dies ist beispielsweise zur Ausgabe von Meldungen sehr nützlich.

3.4 Privatisierung von Variablen

3.4.1 Private versus Shared-Variablen

3-27

Was sind private und geteilte (shared) Variablen?

Eine *geteilte Variable* (shared variable) kann von allen Threads eines Teams gleichermaßen gelesen und geschrieben werden. Fast alle Variablen sind standardmäßig geteilt:

- Alle zu Beginn des Teams bereits vorhandene Variablen.
- Alle Werte auf dem Heap, egal wann sie erzeugt wurden.

Bei einer *privaten Variable* (private variable) hat jeder Thread eines Team eine *lokale Kopie*. Die Werte der Kopien anderer Threads sind nicht sichtbar. Standardmäßig sind privat:

- Die Laufvariable einer Schleife bei einem `for`-Pragma.
- Alle innerhalb eines Arbeitspaket neu deklarierten Variablen.

3-28

Geteilte Variablen wirklich teilen.

Syntax

```
#pragma omp flush (var)
```

Semantik

Der Wert der Variable in den Caches wird in den globalen Speicher geschrieben.

3-29

Variablen privatisieren.

Man kann bei vielen Pragmas als Konfiguration noch Folgendes angeben:

```
#pragma omp ... private (var1, var2, ...)
{
  ...
}
```

Dies hat folgende Effekte:

- Die angegebenen Variablen müssen bereits existieren.
- Für jede angegebene Variable und jeden Thread wird eine lokale Variable erzeugt.
- Der initiale Wert der privaten Variablen ist aber *nicht* der Wert, den sie vorher hatte, sondern er ist undefiniert.
- Nach dem Block ist die Variable wieder geteilt.
- Ihr Wert nachher ist *ebenfalls undefiniert*.

Skript-Referenz

Besondere Initialisierung privater Variablen.

- Benutzt man statt `private` den Befehl `firstprivate`, so ist der *initiale Wert* der privaten Variable in allen Threads doch der Wert der ursprünglichen Variable.
- Benutzt man statt `private` den Befehl `lastprivate`, so ist der Wert der geteilten Variable *nach dem Block* der logisch letzte Wert der privaten Variable. (So, als wäre alles sequentiell ausgeführt worden.)
- Man kann auch beides angeben.

3-30

Beispiel einer privaten Variable.

Beispiel: Matrix initialisieren

```
float a[1000][1000];

int main () {
  // Init a
  int i, j;
  #pragma omp parallel for private (j)
  for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
      a[i][j] = 42;
}
```

✎ **Zur Diskussion**

Was würde passieren, wenn man `private (j)` weglassen würde?

3.4.2 Reduktionen

Summen und Produkte bilden. 3-31

Man kann bei vielen Pragmas als Konfiguration noch Folgendes angeben:

```
#pragma omp ... reduction (+:var)       // oder (*:var)
{
  ...
}
```

Dies hat folgende Effekte:

- Die angegebenen Variable wird privat und mit 0 oder, für die Multiplikation, mit 1 initialisiert.
- Am Ende des parallelen Blocks werden die Werte der privaten Variablen aller Threads aufaddiert oder aufmultipliziert.
- Dieser Wert wird wiederum auf den ursprünglichen Wert der Variable aufaddiert oder aufmultipliziert.

Beispiel einer Reduktion. 3-32

Beispiel: Schnelle Summe

```
int i;
int sum = 500;

#pragma omp parallel for reduction(+:sum)
for (i = 0; i < 1000; i++)
  sum += i;

printf("sum = %d\n", sum);
```

Zusammenfassung dieses Kapitels

Funktionsweise von OpenMP 3-33

- OpenMP beschreibt die seriell-parallele Struktur eines Programms mittels Hinweisen an den Compiler.
- OpenMP benutzt Pragmas, um Stellen anzudeuten, wo Code *prinzipiell* parallel ausgeführt werden könnte.
- Ignoriert ein Compiler alle Pragmas, so muss das Programm immernoch funktionieren.

Private und Shared-Variablen

- Eine *private* Variable ist für jeden Thread in Kopie vorhanden, *Shared-Variablen* hingegen nicht.
- Privat sind nur innerhalb eines `omp parallel` Bereichs deklarierte Variablen und solche, die explizit als `private` markiert wurden.

Die wichtigsten Pragmas

- `omp parallel`
  Gibt vor einem Block an, dass der Block von mehreren Threads ausgeführt werden soll.
- `omp for`
  Gibt an, dass das nachfolgende For-Statement parallelisiert vom aktuellen Team ausgeführt werden soll.
- `omp parallel for`
  Kombination beider obiger Pragmas.
- `omp sections` und `omp section`
  Spezifikation von unabhängigen Code-Bereichen, die parallel ausgeführt werden können.
- `omp once`
  Gibt an, dass ein Code-Block genau einmal ausgeführt werden soll.
- `omp flush` und `omp criticial`
  Pragmas zur Synchronisation von Variablen.

Zum Weiterlesen

[1] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 3.1,* www.openmp.org, Zugriff am 16. April 2012

Übungen zu diesem Kapitel

Übung 3.1 OpenMP-Programme für Vektoroperationen entwickeln, einfach

Die Addition zweier Vektoren gleicher Dimension ergibt sich aus der komponenentenweisen Addition ihrer Einträge. Das Skalarprodukt zweier Vektoren ergibt sich aus der komponentenweisen Multiplikation und dem nachfolgenden Aufsummieren der Produkte. In dieser Aufgabe sollen sequentielle und parallele Methoden in C und OpenMP für diese Vektoroperationen entwickelt werden:

1. Schreiben Sie eine *sequentielle* Methode, welche zwei Vektoren `a` und `b` addiert und das Ergebnis in `c` speichert.
2. Verwenden Sie OpenMP, um Ihre Methode aus Teil 1 direkt zu parallelisieren. Bestimmen Sie dazu zuerst die Anzahl der Threads, die im parallelen Abschnitt verwendet werden sollen und erweitern Sie danach Ihr Programm durch geeignete Pragmas.
3. Schreiben Sie eine parallele Methode, die das Skalarprodukt zweier Vektoren berechnet. Für die Bildung der Summe können Sie die Reduction-Klausel verwenden.

Übung 3.2 OpenMP-Programme für die Matrixaddition entwickeln, einfach

Die Addition zweier Matrizen ergibt sich durch komponentenweise Addition. In dieser Aufgabe sollen sequentielle und parallele Methoden in C und OpenMP für die Matrixaddition entwickelt werden:

1. Schreiben Sie eine sequentielle Methode für die Matrixaddition.
2. Parallelisieren Sie Ihre Methode durch OpenMP-Pragmas.

Übung 3.3 For-Schleifen auf Parallelisierbarkeit untersuchen, mittel

Verwenden Sie OpenMP-Pragmas, um die folgenden For-Schleifen parallel auszuführen, oder begründen Sie, warum eine direkte Parallelisierung nicht möglich ist:

1.
```
for(i = 0; i < (int) sqrt(x); i++) {
  a[i] = 2.3 * i;
  if (i < 10) b[i] = a[i];
}
```

2.
```
flag = 0;
for(i = 0; (i < n) & (!flag); i++) {
  a[i] = 2.3 * i;
  if(a[i] < b[i]) flag = 1;
}
```

3.
```
for(i = 0; i < n; i++)
  a[i] = foo(i);
```

Übung 3.4 Ein OpenMP-Programm für die Summenberechnung programmieren, mittel

In der Vorlesung haben Sie ein PRAM-Programm für die parallele Summenberechnung kennengelernt. Verwenden Sie diesen Ansatz und entwickeln Sie eine C-Methode mit Signatur `int sum(int a[N])`, welche die Zahlen in *a* parallel addiert. Achten Sie darauf, dass Sie parallele Bereiche nicht innerhalb (großer) for-Schleifen verwenden, da ein iteriertes Erstellen und Löschen von Threads das Programm verlangsamen kann. Die parallele Berechnung von Summen ist in OpenMP zwar direkt über Reduce-Klauseln implementiert, in dieser Aufgabe sollen Sie aber einmal explizit die parallele Summenbildung nachvollziehen und programmieren. In späteren Übungen werden wir zum Beispiel komplexere summenartige Berechnungen kennenlernen, die nicht mehr direkt von OpenMP zur Verfügnung gestellt werden.

Übung 3.5 For-Schleifen auf Parallelisierbarkeit untersuchen, einfach

Verwenden Sie OpenMP-Pragmas, um die folgenden For-Schleifen parallel auszuführen, oder begründen Sie, warum eine direkte Parallelisierung nicht möglich ist:

1.
```
for(i = 0; i < n; i++) {
  a[i] = a[2*i] + a[2*i+1];
}
```

2.
```
dotp = 0;
for (i = 0; i < n; i++) {
  dotp += a[i] * b[i];
}
```

3.
```
for(i = k; i < 2*k; i++) {
  a[i] = a[i] + a[i-k];
}
```

Übung 3.6 Ein OpenMP-Programm für die parallele Matrixmultiplikation programmieren, einfach

Im Tutorium zum ersten Übungszettel haben Sie ein PRAM-Programm für die Matrixmultiplikation mit $p \leq n$ Prozessoren erstellt. Dieses Verfahren soll nun in C mit OpenMP implementiert und getestet werden. Gehen Sie dabei wie folgt vor:

1. Erweitern Sie Ihr Programm zu Aufgabe 1.3 um eine Methode für die parallele Matrixmultiplikation mit $p \leq n$ Prozessoren. Beachten Sie, dass nur die Berechnung der Multiplikation, nicht aber das Einlesen und Auslesen der Dateien, parallel implementiert werden soll.
2. Messen Sie die Laufzeiten für das reine Multiplizieren bei konstanter Matrixgröße, aber verschiedener Anzahl von Threads. Vergleichen Sie die Laufzeit dieser Implementierung mit der vom ersten Übungszettel.

Übung 3.7 Matrixaddition ohne verschachtelte Parallel-Pragmas programmieren, einfach

In einer vorherigen Aufgabe haben Sie ein sequentielles Programm durch Hinzufügen von OpenMP-Pragmas parallelisiert. Gegebenenfalls haben Sie dabei auch Parallel-Pragmas verschachtelt verwendet. Da die OpenMP-Implementierung auf dem `t12` verschachtelte Parallel-Pragmas nicht sinnvoll verwendet (bei geschachtelten Pragmas werden keine neuen Threads erzeugt), soll in dieser Aufgabe die Matrixaddition mit einem einzigen Parallel-Pragma so implementiert werden, dass bis zu n^2 Threads parallel eingesetzt werden können.

Übung 3.8 Ein OpenMP-Programm für die parallele Vektorfaltung programmieren, leicht

Sei a ein n-dimensionaler und b ein m-dimensionaler Vektor. Die Faltung $a * b$ von a und b ist ein $(n+m-1)$-dimensionaler Vektor c mit

$$c[i] = \sum_{k=\max\{0,\,i-(n-1)\}}^{\min\{m-1,\,i\}} a[i-k] \cdot b[k].$$

1. Die Faltung zweier Vektoren soll nun algorithmisch gelöst werden. Entwickeln Sie eine sinnvolle Parallelisierung mit $p < n$ Prozessoren für diese Aufgabe und schreiben Sie eine Methode, die diese mit OpenMP-Pragmas umsetzt.
2. Schätzen Sie die Laufzeit Ihrer Lösung in Abhängigkeit von p, n und m ab.

Übung 3.9 Ein OpenMP-Programm für die parallele Matrixfaltung programmieren, mittel

In der Bildverarbeitung verbergen sich hinter Filtern zur Glättung oder für viele andere Effekte oft die Faltung eines Bildes (als Pixelmatrix) mit einer den Filter definierenden Faltungsmatrix. Sei A eine $n \times n$-Matrix und B eine $m \times m$-Matrix mit $m \leq n$. Die Faltung $A * B = C$ ist eine $(n-m+1) \times (n-m+1)$-Matrix mit

$$C[i,j] = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} A[i+k, j+l] \cdot B[k,l].$$

1. Die Faltung zweier Matrizen soll nun algorithmisch gelöst werden. Entwickeln Sie eine sinnvolle Parallelisierung mit $p \leq n$ Prozessoren für diese Aufgabe und schreiben Sie eine Methode, die diese mit OpenMP-Pragmas umsetzt.
2. Schätzen Sie die Laufzeit Ihrer Lösung in Abhängigkeit von p, n und m ab.

Übung 3.10 Ein OpenMP-Programm für die parallele Matrix-Vektor-Multiplikation entwickeln, einfach

In der ersten Vorlesung haben Sie als Fallbeispiel die Matrix-Vektor-Multiplikation kennengelernt. Die Multiplikation einer ganzzahligen $n \times n$ Matrix A mit einem Vektor v der Länge n ergibt einen weiteren Vektor b der Länge n mit folgenden Einträgen:

$$b[i] = \sum_{j=1}^{n} A[i,j] \cdot v[j]$$

In dieser Aufgabe soll ein Programm für die Matrix-Vektor-Multiplikation entwickelt werden, das bis zu n^2 Threads sinnvoll parallel einsetzt. Bearbeiten Sie dazu die folgenden Teilaufgaben:

1. Geben Sie ein PRAM-Programm in Pseudocode an, welches die Matrix-Vektor-Multiplikation in Zeit $O(\log n)$ berechnet. Sammeln Sie dazu zuerst die Zwischenergebnisse der Multiplikationen und berechnen danach die n Summen für die Einträge von b. Die Berechnungen der Summen können Sie analog zur Folie 2-15 aus der Vorlesung durchführen.
2. Implementieren Sie Ihren Ansatz als C-Programm mit OpenMP. Zur parallelen Berechnung der Summen können Sie in OpenMP die Reduction-Klausel verwenden.

4-1

Kapitel 4
Paralleler Entwurf

Aufteilung, Kommunikation, Ballung, Arbeitsverteilung

Lernziele dieses Kapitels

1. Designmethodik von Foster kennen
2. Die Methodik auf Beispiele anwenden können

4-2

Inhalte dieses Kapitels

Worum es heute geht

In der Theorie sah alles völlig klar aus: Zur parallelen Berechnung des Maximums eines Arrays kann man eine schönen, ausgeglichenen Binärbaum über die Daten legen und dann die Ebenen parallel bearbeiten. Dies führt zu einer phantastischen Laufzeit von $O(\log n)$ – was ziemlich gut ist verglichen mit $O(n)$ bei sequentiellen Programmen. In späteren Kapiteln wird mit etwas Theoretiker-Voodoo sogar eine Laufzeit von $O(\log \log n)$ bei minimaler Arbeit möglich sein.

Wie ernüchternd ist dann die Praxis, wenn man die Algorithmen frohgemut implementiert und ausprobiert. Die Programme brauchen dann nämlich auch bei eher großen Eingabearrays deutlich *mehr* Zeit als die gute alte Schleife. Die Gründe hierfür sind vielfältig: Prozessoren sind *sehr* schnell, wenn sie linear durch den Speicher joggen. Die Aufteilung der Arbeit in Threads, deren Verwaltung, Synchronisation, Cache-Kohärenzherstellung und Inter-Boardkommunikation halten die Kerne hingegen ziemlich von der Arbeit ab.

Folgender Vergleich zeigt das Problem auf: Stellen Sie sich vor, Sie sollen als Finanzchef im 100-seitigen Bilanzbericht Ihrer Firma den größten Einzelposten finden. Wenn Sie das Ihren Assistenten alleine machen, dann dauert das vielleicht eine Stunde. Wenn Sie jedoch Ihren gesamten Stab der Finanzabteilung damit beauftragen, dann muss von den Mitarbeitern vielleicht jeder nur zwei Seiten bearbeiten. Real werden Sie aber vermutlich Wochen auf das Ergebnis warten, nämlich bis der letzte beauftragte Mitarbeiter wieder aus dem Urlaub zurück ist.

In diesem Kapitel soll es darum gehen, wie man parallele Algorithmen so entwirft, dass die Parallelisierung tatsächlich etwas bringt: Durch eine geschickte Aufteilung und eine geschickte Ballung der Arbeit kann dafür gesorgt werden, dass die Einheiten nicht nur Quatschen, sondern auch etwas Produktives tun.

4.1 Fosters Methodik

4-4 Overhead und dessen Vermeidung bei parallelen Programmen.

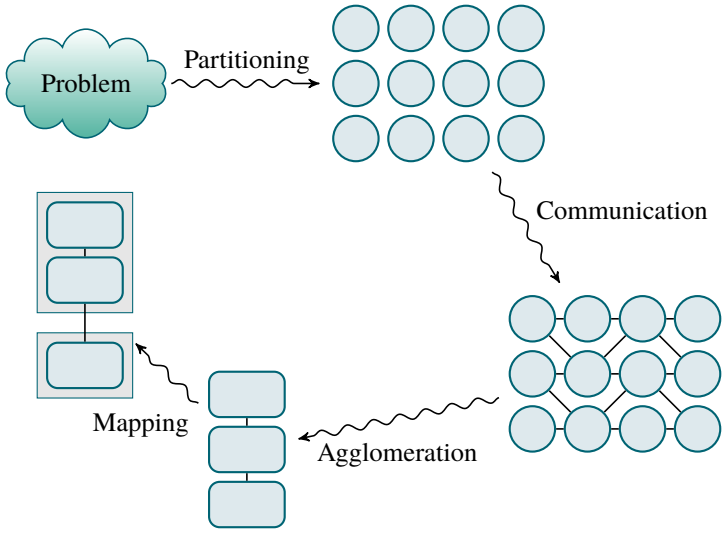
Overhead bei parallelen Programmen

Einheiten müssen implizit oder explizit kommunizieren, was viel Zeit kosten kann. Ebenso kann die Aufteilung und Verteilung von Arbeitspaketen viel Zeit kosten. Im schlimmsten Fall kann ein paralleles Programm langsamer sein als ein sequentielles.

Idee hinter der Methodik von Foster

Man sollte parallele Programme so anlegen, dass der Overhead minimiert wird. Dazu sollte die *Notwendigkeit für Kommunikation* minimiert werden. Ebenso sollte die *Verwaltung der Arbeitspakete* so einfach wie möglich gehalten werden.

4-5 Fosters Methodik als Bild.



4.1.1 Partitioning

4-6 Erster Schritt: Aufteilung des Problems in Teilprobleme

Partitioning

Ziel des Partitioning

Beim *Partitioning* versucht man, möglichst viele *Daten* und *Berechnungen* zu finden, die sich *prinzipiell parallelisieren* lassen.

Es geht darum, was *prinzipiell möglich wäre*. Ob das später tatsächlich parallelisiert wird, ist noch nicht gesagt.

Paritioning in OpenMP

Sowohl die Sections- wie die For-Pragmas teilen dem Compiler mit, dass hier eine Partitionierung möglich ist.

4.1.2 Communication

Zweiter Schritt: Bestimmung der Kommunikationsstellen. 4-7
Communication

Ziel der Kommunikations-Analyse

Man bestimmt, welche Kommunikation für die Bearbeitung der einzelnen Arbeitspakete nötig ist. Dabei unterscheidet man zwischen:

- *Lokaler Kommunikation* für die Bearbeitung von wenigen Arbeitspaketen, die auch noch »nahe beieinander« liegen.
- *Globaler Kommunikation*, bei der Ergebnisse in großem Maßstab kommuniziert oder synchronisiert werden müssen.

Kommunikation in OpenMP

Bei privaten Variablen ist keine Kommunikation möglich oder erwünscht. Geteilte (shared) Variablen dienen hingegen der Kommunikation. Jede implizite Barriere ist ein Kommunikationsstelle.

✎ Zur Diskussion

Im letzten Kapitel wurde die Funktion `n_smooth` vorgestellt, die die n-fache Glättung eines Arrays berechnet.
Welche lokale und welche globale Kommunikation ist bei diesem Problem nötig?

4.1.3 Agglomeration

Dritter Schritt: Ballung von Arbeitspaketen. 4-8
Agglomeration

Ziele der Ballung

Ballung bedeutet, mehrere kleine Arbeitspakete zu einem Paket zusammenzufassen. Dadurch soll der Verwaltungsaufwand für die Arbeitspakete minimiert werden. Es soll auch die Kommunikation minimiert werden, da Einheiten *innerhalb* eines Arbeitspakets sich nur »mit sich selbst« koordinieren müssen. Die agglomerierten Pakete sollten alle ähnlich groß sein.

Agglomeration in OpenMP

- Eine Angabe wie `if (n>1000)` bei einem parallelen Konstrukt dient der Ballung.
- Ebenso eine Angabe wie `num_threads (4)`.
- Man kann darauf verzichten, »kleine« Schleifen mit einem For-Pragma zu kennzeichnen, obwohl dies möglich wäre.
- Man kann die Reihenfolge von Schleifen abändern, so dass agglomerierte Daten nacheinander abgearbeitet werden.

✎ Zur Übung 4-9

Ein Filter für ein Bild berechnet für jedes Pixel des Bildes den geeignet gewichteten Durchschnitt der Pixel in einem bestimmten Abstand.
Nehmen wir an, es sollen vier verschiedene Filter (beispielsweise ein Weichzeichner, ein Kantendetektor, eine Erosion und ein Schärfer) nacheinander auf ein Bild der Größe 1000 mal 1000 Pixel angewandt werden.

1. Benennen Sie zwei sinnvolle Agglomerationsstrategien.
2. Welche Vor- und Nachteile haben diese?

4.1.4 Mapping

4-10

Vierter Schritt: Verteilung von Arbeit an Einheiten.

Mapping

Ziele des Mapping

Arbeitspakete müssen zur Laufzeit tatsächlich freien Einheiten zugeordnet werden. Alle verfügbaren Einheiten sollten ständig möglichst gleichmäßig ausgelastet sein.

Mapping in OpenMP

Im Allgemeinen geschieht das Mapping automatisch. Bei einem For-Pragma kann aber die `schedule`-Option angegeben werden. Bei einem dynamischen Schedule werden Einheiten gleichmäßiger mit Arbeitspaketen versorgt als bei einem statischen – was aber Verwaltungsaufwand erzeugt.

4.2 Fallbeispiele

4.2.1 Maximums-Bestimmung

4-11

Bestimmung des Maximums.

Problem

Bestimmung des Maximums eines Arrays.

Partitionierung

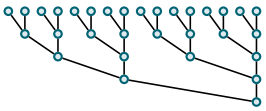
Je zwei Array-Elemente formen ein Mini-Paket. Es gibt $\log n$ Runden, in jeder Runde halbiert sich die Anzahl der Pakete.

Beispiel

Kommunikation

Jedes Paket der nächsten Ebene benötigt die Ergebnisse von genau zwei Paketen der Ebene darüber.

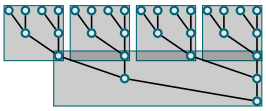
Beispiel



Ballung

- Ballung im Anfangsbereich zu großen Blöcken
- Ballung der Schlussberechnung zu einem Block

Beispiel



Mapping

Die geballten oberen Pakete werden statisch auf die verfügbaren Einheiten verteilt. Nach einer Barriere kommt das letzte Paket auf die Master-Einheit.

4.2.2 Windkanal

Luftströmungen hinter einem Flugzeug.

4-12

Authored by US federal government, public domain

Simulation einer Luftströmung.

4-13

Problembeschreibung

Wir wollen die Strömung von Luft oder Wasser um ein Objekt simulieren. Für jeden Raumpunkt kann man einen *Strömungsvektor* angeben. Differentialgleichungen beschreiben nun das Verhalten der Strömungsvektoren über die Zeit hinweg.

Numerische Lösung

- Wir teilen den Raum in *kleine Würfel* ein.
- Wir teilen die Zeit in *kleine Intervalle* ein.
- Für jeden Würfel speichern wir, ob dort ein Teil des Objekts ist und, falls nicht, einen Strömungsvektor.
- Der Strömungsvektor nach einem Zeitschritt hängt *nur von den Eigenschaften der umliegenden Würfel* ab.

Programmentwurf nach Foster

4-14

Partitioning

Arbeitspakete sind Würfel in einem Zeitintervall.

Kommunikation

Die Abhängigkeiten zwischen den Arbeitspaketen bilden ein vierdimensionales Gitter.

Agglomeration

Eine Ballung über die Zeit hinweg ist nicht möglich, da Ergebnisse später ja von früheren Schritten abhängen. Wenn die Anzahl der Einheiten deutlich kleiner ist als die Anzahl Würfel pro Raumrichtung, so ist aber eine Ballung mehrerer Raumebenen in einem Zeitabschnitt sinnvoll.

Mapping

Die geballten Arbeitspakete werden statisch auf Einheiten verteilt mit einer Barriere nach jedem Zeitschritt.

4.2.3 Gravitations-Simulation

4-15

Die Sombrero-Galaxie M104

Authored by NASA, public domain

4-16

Simulation eines Sternen-Systems.

Problembeschreibung

Die Bewegung der Sterne unserer Galaxis sollen simuliert werden. Für jeden Stern ist seine Masse, Position und aktuelle Bewegung gespeichert. Die newtonschen Differentialgleichungen beschreiben nun die Bewegung der Sterne über die Zeit hinweg.

Numerische Lösung

Wir teilen die Zeit in *kleine Intervalle* ein. In jedem Zeitschritt lässt sich die Gesamtanziehungskraft aller anderen Sterne auf jeden Stern berechnen. Dann bewegen sich alle Sterne ein kleines Stück.

4-17

Programmentwurf nach Foster

Partitioning

Eine einfache Aufteilung ist »ein Arbeitspaket pro Stern und Zeitintervall«. Aber: Prinzipiell hängt dann jedes Arbeitspaket von *allen anderen* Arbeitspaketen einen Schritt früher ab. Da die Gravitation über die Entfernung schnell abfällt, lässt sich der Einfluss größerer Sterngruppen auf weiter entfernte Sterne als *Masseschwerpunkt* beschreiben. Folglich ist es sinnvoll, zusätzlich zu den Stern-Arbeitspaketen noch Arbeitspakete für *größere Raumbereiche* zu haben.

Kommunikation

Die Berechnungen für ein Stern-Arbeitspaket hängen nun von folgenden Arbeitspaketen einen Schritt früher ab:

- Den Sternen im eigenen Raumbereich und den angrenzenden Raumbereichen.
- Dem Masseschwerpunkt der anderen Raumbereiche.

Für die Neuberechnung der Masseschwerpunkte der Raumbereiche sind die die Positionen der Sterne im aktuellen Bereich wichtig und die Wanderung der Sterne; sie können den Raumbereich wechseln.

Agglomeration

- Die Sterne in einem Raumbereich sollten geballt werden.
- Räumlich benachbarte Raumbereichen könnten geballt werden.
- Da Sterne wandern, können Raumbereiche leer werden. Deshalb muss die Ballung gegebenenfalls *dynamisch angepasst werden*.

Mapping

- Werden die geballten Raumbereiche dynamisch in etwa gleich groß gehalten, so können sie statisch den Einheiten pro Zeitschritt zugeordnet werden.
- Alternativ können auch die Raumbereiche konstant gehalten werden und dafür das Mapping dynamisch erfolgen.

Zusammenfassung dieses Kapitels

Forsters Methodik

Fosters Methodik gibt vier Schritte an, nach denen man parallele Programme *entwirft*. Diese sind:

1. Partitioning
2. Communication
3. Agglomeration
4. Mapping

Ziel ist es, die *Kommunikation zu minimieren.*

Unterstützung der Methodik durch OpenMP

Die Schritt nach Foster werden von OpenMP wie folgt unterstützt:

- Partitioning: Sections- und For-Pragmas.
- Communication: Globale Variablen und implizite Barrieren.
- Agglomeration: `if` und `num_threads` Bedingungen und Schleifenreihenfolgen.
- Mapping: `schedule` Hinweise.

5-1

Kapitel 5
Parallele Basisalgorithmen
Präfixsumme hilft immer

5-2

Lernziele dieses Kapitels

1. Die Ressourcen Zeit und Arbeit kennen
2. Algorithmus für parallele Präfixsumme kennen
3. Algorithmus Pointer-Jumping kennen
4. Die Algorithmen in anderen Problemen anwenden können

Inhalte dieses Kapitels

Worum es heute geht

Es gibt Probleme, die so grundlegend sind, dass sie immer und immer wieder auftauchen. Zwei Standardprobleme sind Suchen und Sortieren – ständig sind Computer (genau wie Menschen) damit beschäftigt, etwas zu suchen und Ordnung zu schaffen (wobei sie weitaus erfolgreicher den zweiten Hauptsatz der Thermodynamik bekämpfen, als Menschen dies im Allgemeinen gelingt).

In der Parallelverarbeitung gibt es auch zwei Standardprobleme, die immer und immer wieder auftauchen: Die Bestimmung von Präfixsummen und die Bestimmung der Wurzeln eines Waldes. Auf den ersten Blick scheinen diese Probleme etwas exotisch, im Sequentiellen spielen sie jedenfalls nur eine eher untergeordnete Rolle. Dies täuscht aber, der Punkt ist lediglich, dass diese Probleme im Sequentiellen in der Regel nicht *explizit* gelöst werden.

Dazu ein kleines Beispiel: Sie führen eine elektronische Einkaufsliste mit beim Einkaufen im Bio-Discounter Ihrer Wahl, auf der Sie erstandene Produkte von der Liste streichen können. Ein Algorithmus soll Ihre Einkaufsliste »komprimieren«, also nur noch die fehlenden Objekte im Speicher halten. Ihre Einkaufsliste ist ein Array von Strings, ein zweiter Array A gibt für jedes Objekt an, ob Sie es schon besorgt haben (kodiert als 0 oder 1).

Ein sequentieller Algorithmus, der die Dinge ja per Definition »nacheinander« angehen muss, sucht nach dem ersten Objekt, für das $A[i]$ eine 1 ist, und schiebt dieses an die erste Position. Das nächste Objekt mit $A[i] = 1$ kommt an Position 2, das nächste an Position 3 und so weiter. Ein paralleler Algorithmus wird nun versuchen, alle Objekte gleichzeitig an den richtigen Platz zu schieben. Wohin aber mit dem zweiundvierzigsten Objekt? Die korrekte Stelle hat der sequentielle Algorithmus implizit in einem Zähler gehalten. Der parallele Algorithmus braucht die Position »explizit« und – das ist der entscheidende Punkte – diese Stelle ist gerade $A[1]+A[2]+\cdots+A[42]$. Die Präfixsummen sagen also dem parallelen Algorithmus »wo er arbeiten muss«.

5.1 Vorschau: Zeit und Arbeit

Zeitmaße bei parallelen Algorithmen.

5-4

Zur Lösung eines Problems sei ein paralleler Algorithmus gegeben, der *mit einer variablen Anzahl an Einheiten arbeiten kann*.

▶ Definition

Die Rechenzeit des Algorithmus bei *Eingabe* x und *genau* p *Einheiten* bezeichnen wir mit $T_p(x)$. Wie üblich bezeichnet dann $T_p(n) = \max_{x\in\Sigma^n} T_p(x)$ die maximale Rechenzeit des Algorithmus bei Eingaben der Länge n.

▶ Definition

Die Rechenzeit eines *parallelen Algorithmus, der beliebig viele Einheiten benutzen darf,* bezeichnen wir mit $T_\infty(n)$, also $T_\infty(n) = \inf_{p\to\infty} T_p(n)$.

Arbeit = Stromverbrauch

5-5

▶ Definition: Arbeit

Die *Arbeit* $W(x)$ eines Algorithmus bei Eingabe x ergibt sich wie folgt:

1. Für jeden Zeitschritt wird gemessen, wie viele Einheiten tatsächlich arbeiten (nicht in einem Idle-Modus sind).
2. Diese Zahlen werden über die gesamte Berechnung hinweg aufsummiert.

Die *Arbeit* $W(n)$ bei Eingaben der Länge n ist dann die maximale Arbeit bei Eingaben der Länge n, also $W(n) = \max_{x\in\Sigma^n} W(x)$.

Beispiel

Beim baumartigen Algorithmus zur Maximumsbestimmung ist die Arbeit $W(n) = O(n)$, da sich die Arbeit pro Schritt jeweils halbiert und $\sum_{i=1}^{\log n} n/2^i < n$.

✎ Zur Übung

5-6

Wie lauten $T_\infty(n)$ und $W_p(n)$ bei folgendem Programm?

```
void matrix_vector_product (int A[][], int b[],
                            int c[], int n)
{
  // Calculate the product of the n times n matrix A
  // and the length-n vector b and store the result in c

  #pragma omp parallel for
  for (int i = 0; i<n; i++) {
    int sum = 0;
    for (int j = 0; j<n; j++)
      sum += A[i][j] * b[j];
    c[i] = sum;
  }
}
```

5.2 Präfix-Summen

5.2.1 Problemstellung

5-7 Ein Problem, das in vielen Anwendungen auftaucht.

Das Präfix-Summen-Problem

Eingabe Array X von n Zahlen.

Ausgabe Array S von n Zahlen, wobei $S[i]$ die Summe der ersten i Zahlen in X ist.

Beispiel

Eingabe $X = (3,4,1,2,3)$

Ausgabe $S = (3,7,8,10,13)$.

5-8 ✎ Zur Übung

Wir kennen bereits einen Algorithmus für die Summenbildung mit Laufzeit $O(\log n)$ und Arbeit $O(n)$. Hieraus können wir einen parallelen Algorithmus für das Präfix-Summen-Problem bauen: Wir berechnen einfach parallel alle $S[i]$, indem wir mittels des Summenbildungsalgorithmus die Summe der ersten i Zahlen in X berechnen.
Wie viele Einheiten braucht der Algorithmus? Was ist die Laufzeit? Was ist die Arbeit?

5.2.2 Algorithmus

5-9 Ein arbeitsscheuer Algorithmus für das Präfix-Summen-Problem in Pseudocode

```
1  for j ∈ {1,...,n} par do // Input
2      B_0[j] ← X[j]
3
4  // Berechnung von immer längeren Teilsummen
5  for i ← 1 to log n seq do
6      for j ∈ {1,...,n/2^i} par do
7          B_i[j] ← B_{i-1}[2j-1] + B_{i-1}[2j]
8
9  C_{log n}[1] ← B_{log n}[1]
10
11 // Berechnung der finalen Partialsummen; C_i[0] sei immer 0
12 for i ← log n - 1 to 0 seq do
13     for j ∈ {1,...,n/2^i} par do
14         if j ist gerade then
15             C_i[j] ← C_{i+1}[j/2]
16         else
17             C_i[j] ← C_{i+1}[(j-1)/2] + B_i[j]
18
19 for j ∈ {1,...,n} par do // Output
20     S[j] ← C_0[j]
```

Der Algorithmus in Aktion.

5-10

1

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$									
$B_2[j]$									
$C_3[j] = B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j] = C_0[j]$	0								

2

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$									
$C_3[j] = B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j] = C_0[j]$	0								

3

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j] = B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j] = C_0[j]$	0								

4

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j] = B_3[j]$		9							
$C_2[j]$	0								
$C_1[j]$	0								
$S[j] = C_0[j]$	0								

5

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j] = B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0								
$S[j] = C_0[j]$	0								

6

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j] = B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0	3	6	7	9				
$S[j] = C_0[j]$	0								

7

j	0	1	2	3	4	5	6	7	8
$X[j] = B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j] = B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0	3	6	7	9				
$S[j] = C_0[j]$	0	1	3	3	6	7	7	7	9

5.2.3 Analyse

Analyse des Algorithmus

5-11

▶ Satz

Der Algorithmus berechnet die Präfix-Summen in Zeit $O(\log n)$ und Arbeit $O(n)$.

Beweis. Zur Korrektheit siehe Übungsaufgabe 5.6.
Die Laufzeit erhält man sofort dadurch, dass sequentiell lediglich zwei Schleifen jeweils $\log n$ oft durchlaufen werden. Die behauptete Arbeit ergibt sich daraus, dass in der Schleife

in Zeilen 5 bis 7 die Arbeit $n\sum_{i=1}^{\log n} 1/2^i < n$ geleistet wird und genauso in der Schleife zwischen 12 und 17. □

5.2.4 Erweiterungen

5-12 Es müssen nicht Präfix-Summen sein.

Der Algorithmus funktioniert offenbar auch, wenn die Präfix-*Produkte* statt Präfix-*Summen* berechnen wollen. Allgemein funktioniert er für alle *assoziativen* Operationen.

✎ Zur Diskussion

Für welche weiteren assoziativen binären Operationen könnte das Präfix-Summen-Problem interessant sein?

5.3 Pointer-Jumping

5.3.1 Problemstellung

5-13 Ein zweites Problem, das in vielen Anwendungen auftaucht.

Das Wurzelproblem

Eingabe Gerichteter Wald.

Ausgabe Für jeden Knoten die Wurzel des Baumes, in dem er liegt.

✎ Zur Diskussion

Sollten die Kanten zur Wurzel hin gerichtet oder weg gerichtet sein?

5.3.2 Algorithmus

5-14 Das Problem wird mittels Pointer-Jumping gelöst.

```
1 for i ∈ {1,...,n} par do
2     while S[i] ≠ S[S[i]] do
3         S[i] ← S[S[i]]
```

5.3.3 Analyse

5-15 Analyse des Pointer-Jumping-Algorithmus.

▶ Satz

Der Pointer-Jumping-Algorithmus löst das Wurzelproblem in Zeit $O(\log h)$ und mit Arbeit $O(n \log h)$, wobei h die Höhe des Waldes ist (Länge des längsten Pfades).

Beweis. Man zeigt per Induktion, dass sich in jeder Iteration die Höhe aller Bäume halbiert. Dies liegt daran, dass der Weg von einem Knoten zur Wurzel nach einem Jump-Schritt nur noch halb so lang ist, da jeder zweite Knoten weggelassen wurde.
Hieraus ergibt sich eine Laufzeit von $\log h$ und damit eine Arbeit von $O(n \log h)$. □

5.3.4 Erweiterungen

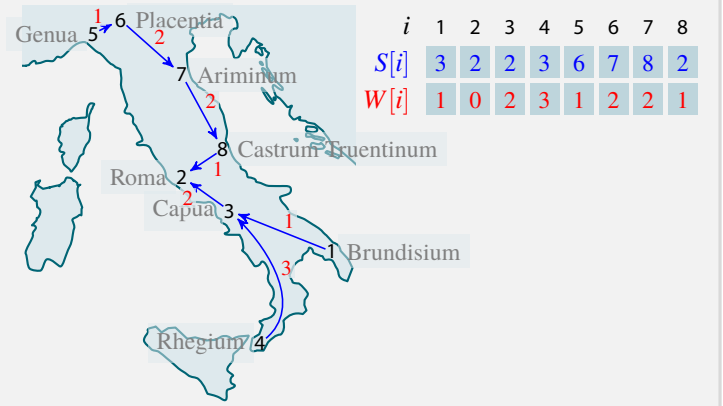
Man kann mehr berechnen als nur die Wurzel. 5-16

Während des Pointer-Jumpings kann man auch gleich noch die Entfernung das Knotens von der Wurzel bestimmen. Allgemeiner kann jede Kante eine Zahl oder einen Wert erhalten und wie bei den Präfix-Summen kann man mittels Pointer-Jumping Pfad-Summen bestimmen.

```
1 for i ∈ {1,...,n} par do
2     while S[i] ≠ S[S[i]] do
3         W[i] ← W[i] ⊗ W[S[i]]
4         S[i] ← S[S[i]]
```

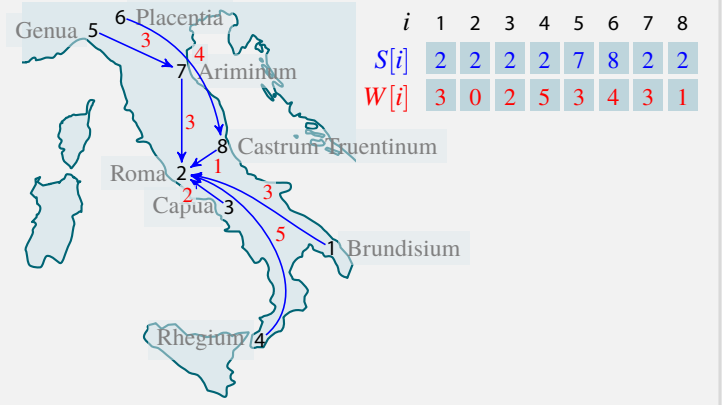
Beispiel: Alle Wege führen nach Rom. 5-17

1



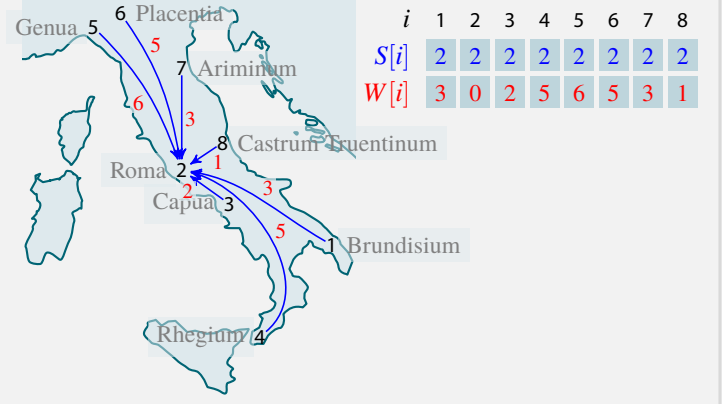
i	1	2	3	4	5	6	7	8
$S[i]$	3	2	2	3	6	7	8	2
$W[i]$	1	0	2	3	1	2	2	1

2



i	1	2	3	4	5	6	7	8
$S[i]$	2	2	2	2	7	8	2	2
$W[i]$	3	0	2	5	3	4	3	1

3



i	1	2	3	4	5	6	7	8
$S[i]$	2	2	2	2	2	2	2	2
$W[i]$	3	0	2	5	6	5	3	1

Zusammenfassung dieses Kapitels

5-18

▶ Arbeit und Zeit bei parallelen Programmen

- $T_p(n)$ ist die Zeit, die ein paralleler Algorithmus bei p Einheiten und Eingaben der Länge n maximal braucht.
- $T_\infty(n) = \inf_{p\to\infty} T_p(n)$.
- $W(n)$ ist die Summe über alle Zeitschritte hinweg der Anzahlen an Einheiten, die in dem jeweiligen Zeitschritt nicht »idle« sind.

▶ Präfixsummen

- Beim Präfixsummenproblem ist die Eingabe ein Zahlen-Array A, die Ausgabe der Array B mit $B[i] = \sum_{j=1}^{i} A[j]$.
- Das Präfixsummenproblem kann in Zeit $O(\log n)$ und Arbeit $O(n)$ gelöst werden.

▶ Pointer-Jumping

- Beim Pointer-Jumping ist die Eingabe ein Wald; die Ausgabe für jeden Knoten die Wurzel des Teilbaumes, in dem der Knoten liegt.
- Der Algorithmus führt einfach wiederholt den Befehl $S[i] \leftarrow S[S[i]]$ aus.
- Pointer-Jumping benötigt maximal Zeit $O(\log n)$ und Arbeit $O(n \log n)$.

Übungen zu diesem Kapitel

Übung 5.1 Pointer-Jumping, schwer

Im Speicher einer PRAM sei ein Wald mit n Knoten gespeichert. Dieser ist durch eine Liste $P(1), \ldots, P(n)$ repräsentiert, wobei $P(i)$ den Elternknoten des Knotens i angibt.

1. Geben Sie einen Algorithmus an, der die Anzahl der Bäume im Wald bestimmt.
2. Geben Sie einen Algorithmus an, der die Anzahl der Blätter im Wald bestimmt.
3. Geben Sie einen Algorithmus an, der die Größe des größten Baumes im Wald bestimmt. Die Größe eines Baumes ist dabei die Anzahl seiner Knoten. Der Algorithmus soll eine Laufzeit von $O(\log^2 n)$ und eine Arbeit von $O(n \log n)$ haben.
   Tipp: Diese Teilaufgabe ist nicht ganz einfach. Sie dürfen bei dieser Aufgabe davon ausgehen, dass die Sortierung von n Zahlen in Zeit von $O(\log^2 n)$ und unter Arbeitsaufwand von $O(n \log n)$ bewerkstelligt werden kann.

Übung 5.2 Sequentielles Pointer-Jumping, schwer

Nachdem in der Vorlesung ein paralleler Pointer-Jumping-Algorithmus vorgestellt wurde, soll nun Pointer-Jumping sequentiell realisiert werden. Erarbeiten Sie die Idee eines sequentiellen Algorithmus, der das Wurzelproblem in Zeit $O(n)$ löst.

Übung 5.3 Array nach Farben sortieren, mittel

Gegeben sei ein n-elementiges Array, dessen Elemente eine der Farben Rot, Grün oder Blau haben. Geben Sie einen Algorithmus an, der das Array derart sortiert, dass zuerst die roten dann die grünen und zuletzt die blauen Elemente im Array kommen. Die Laufzeit des Algorithmus soll $O(\log n)$ und die benötigte Arbeit $O(n)$ betragen.

Übung 5.4 Sortierung mit Präfix-Summen, schwer

Im Speicher einer PRAM ist eine Folge $Z(1), \ldots, Z(n)$ von Zeigern auf n Objekte gespeichert und zusätzlich eine Folge $B(1), \ldots, B(n)$, die jedem Objekt den Wert 0 oder 1 zuordnet. Die Objekte sollen jetzt anhand der B-Werte sortiert werden, das heißt in der Ergebnisliste stehen zuerst alle Zeiger $Z(i)$ mit $B(i) = 0$ und dann die mit $B(i) = 1$. Die Sortierung soll dabei *stabil* sein: Objekte mit gleichem B-Wert sollen nach dem Sortieren in derselben Reihenfolge stehen wie vorher. Entwickeln Sie die Idee eines parallelen Algorithmus mit Laufzeit $O(\log n)$ und Arbeit $O(n)$, der die Objekte wie beschrieben sortiert.

Übung 5.5 Sortierung mit Präfix-Summe implementieren, schwer

Verallgemeinern Sie Ihren Algorithmus aus Aufgabe 5.4, indem die B-Werte der Objekte einen Wert von 1 bis l annehmen können. Dabei ist l fest und nicht Teil der Eingabe. Geben Sie Ihren Algorithmus in Pseudocode an. Dabei können Sie annehmen, dass Ihnen der Präfix-Summen-Algorithmus in einer Methode PrefixSum(A) zur Verfügung steht. Die Laufzeit soll weiterhin $O(\log n)$ und die Arbeit $O(n)$ bleiben.

Übung 5.6 Beweis der Korrektheit des Präfixsummenalgorithmus, mittel

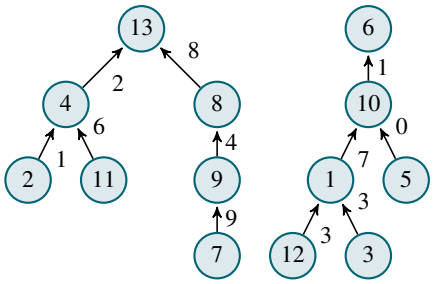
Zeigen Sie, dass der Präfixsummenalgorithmus von Seite 5-9 korrekt ist.

Tipps: Dies ist eine einfache, aber etwas fummelige Induktion, in der man sich klar machen muss, welche Teilsummen die $B_i[j]$ und $C_i[j]$ jeweils darstellen.

Übung 5.7 Pointer-Jumping nachvollziehen, leicht

Es soll Pointer-Jumping benutzt werden, um in einem Wald mit gewichteten Kanten für jeden Knoten das maximale Gewicht auf dem Pfad von der Wurzel des Knotens zur Wurzel zu berechnen.

Geben Sie dazu zu folgendem Wald den Inhalt der Arrays S und W zum Anfang und nach jedem »Jump-Schritt« an:



Übung 5.8 Segmentierte Präfix-Summen-Problem, mittel

Wir betrachten eine Variante des Präfix-Summen-Problems, das sogenannte *segmentierte Präfix-Summen-Problem*. Gegeben ist eine Liste $x = (x_1, \ldots, x_n)$ von Elementen aus einer Menge S und ein assoziativer Operator $\otimes\colon S \times S \to S$. Weiterhin gibt es eine Bitfolge $b = (b_1, \ldots, b_n)$ mit $b_1 = 1$. Ausgegeben wird eine Liste $y = (y_1, \ldots, y_n)$ von Elementen aus S. Die Bitfolge b definiert eine Unterteilung der Listen x und y in Segmente. Ein neues Segment beginnt immer dann, wenn $b_i = 1$ gilt; bei $b_i = 0$ wird das aktuelle Segment fortgesetzt. Das segmentierte Präfix-Summen-Problem besteht nun darin, die Präfix-Summen getrennt für jedes Segment zu berechnen.

Im folgenden Beispiel wird die Liste x durch die Bitfolge b in vier Segmente eingeteilt:

$b =$	1	0	0	1	0	1	1	0	0	0	0
$x =$	1	2	3	4	5	6	7	8	9	10	11
$y =$	1	3	6	4	9	6	7	15	24	34	45

1. Wir definieren den Operator $\widehat{\otimes}$ auf Paaren $(a,z),(a',z') \in \{0,1\} \times S$ wie folgt:

$$(a,z) \mathbin{\widehat{\otimes}} (a',z') = \begin{cases} (a, z \otimes z'), & \text{falls } a' = 0, \text{ und} \\ (1, z'), & \text{falls } a' = 1. \end{cases}$$

   Zeigen Sie, dass $\widehat{\otimes}$ assoziativ ist.
2. Geben Sie einen Algorithmus an, der das segmentierte Präfix-Summen-Problem einer n-elementigen Liste auf einer EREW-PRAM in Zeit $O(\log n)$ löst.

Übung 5.9 Präfix-Summe implementieren und testen, mittel

In dieser Aufgabe sollen sequentielle und parallele C-Methoden für das Präfix-Summen-Problem erstellt und deren Laufzeiten untersucht werden. Bearbeiten Sie dazu die folgenden Teilaufgaben:

1. Erstellen Sie eine *sequentielle* C-Methode, welche die Präfix-Summen für ein Feld X von ganzzahligen Werten berechnet und diese in einem Feld S speichert. Als assoziativer Operator für die Verknüpfung der Zahlen kann die Addition verwendet werden.
2. Für die parallele Berechnung der Präfix-Summen beschreibt der Algorithmus der Vorlesung bereits eine Partitionierung des Problems in Teilsummen und die baumartige Kommunikation bei der Berechnung der Teilsummen. Beschreiben Sie ausgehend von diesem Ansatz eine sinnvolle Agglomerationsstrategie für die Berechnung der Präfix-Summen.

3. Verwenden Sie Ihre Strategie und implementieren Sie eine *parallele* C-Methode für die Berechnung der Präfix-Summen. Für das Mapping auf die verwendeten Einheiten können Sie geeignete OpenMP-Pragmas verwenden.
4. Vergleichen Sie die Laufzeiten Ihrer sequentiellen und parallelen Methoden für verschiedene Eingabegrößen und verschiedene Anzahlen von Einheiten. Stellen Sie Ihre Ergebnisse in einem Diagramm dar.

Übung 5.10 Pointer-Jumping implementieren und testen, mittel

In dieser Aufgabe sollen sequentielle und parallele C-Methoden für Pointer-Jumping erstellt und deren Laufzeiten untersucht werden. Bearbeiten Sie dazu die folgenden Teilaufgaben:

1. Erstellen Sie eine *sequentielle* C-Methode, welche das Wurzelproblem löst und dabei für jeden Knoten die Entfernung zu dessen Wurzelknoten berechnet. Die Eingabe besteht dabei aus einem Feld S positiver Werte, welches einen gerichteten Wald kodiert ($S[i] = j$ bedeutet, dass es eine Kante vom Knoten mit Index i zum Knoten mit Index j gibt), und einem Feld W positiver Werte, welches die Längen der einzelnen Kanten speichert.
2. Der Algorithmus für das Pointer-Jumping beschreibt implizit eine Partitionierung in »Jump-Schritte« und die Abhängigkeiten zwischen diesen. Beschreiben Sie nun eine geeignete Stategie zur Ballung der »Jump-Schritte«.
3. Implementieren Sie Ihre Strategie als C-Methode. Für das Mapping können Sie OpenMP-Pragmas verwenden.
4. Vergleichen Sie die Laufzeiten Ihrer sequentiellen und parallelen Methoden für verschiedene Eingabegrößen und verschiedene Anzahlen von Einheiten. Stellen Sie Ihre Ergebnisse in einem Diagramm dar.

Teil II

Algorithmen-Analyse und -Methodiken

Wenn Sie einen Theoretiker in Verlegenheit bringen wollen, dann fragen Sie ihn doch mal Folgendes: »Wie schnell kann man zwei Matrizen multiplizieren?« Vor vierzig Jahren wäre die Antwort einfach gewesen: »Na, $O(n^3)$ und das ist sicherlich optimal.« Mit einer gewissen Verblüffung hat die Theoretische Informatik dann das Ergebnis von Volker Strassen zur Kenntnis genommen, dass es auch in Zeit $O(n^{\log_2 7})$ geht. Heute würde der befragte Theoretiker wohl antworten »$O(n^{2+\text{Forschungskonstante}})$«. Richtig ärgern können Sie ihn dann, wenn Sie entgegnen »Also, auf meinem Parallelrechner geht das in Zeit $O(\log n)$.«

In der Welt der Parallelverarbeitung geht es recht flott zu. Probleme, die man normalerweise mit viel Mühe in quadratischer oder kubischer Zeit gelöst bekommt, lassen sich in logarithmischer Zeit oder sogar noch schneller lösen. Wir wollen nun etwas genauer klären, wann ein Problem »parallel in Zeit XY lösbar ist«. Dazu werden wir in klassischer Theoretikermanier ein Ressourcemaß einführen, wobei wir natürlich mit der *Zeit* beginnen. Die Aussage »man kann Matrizen parallel in Zeit $O(\log n)$ multiplizieren« lässt sich dann präziser (und wie so oft bei präzisen Formulierungen: weniger verständlich) fassen als »Es gibt ein PRAM-Programm mit $T^\infty \in O(\log n)$, das das Produkt von beliebigen Matrizen berechnet.« Nun hat sich im Verlauf der vorigen Kapitel aber schon angedeutet, dass der Geschwindigkeitsvorteil der Parallelverarbeitung teuer erkauft ist. Parallele Rechner mögen schnell sein, in der Regel sind sie in Bezug auf Stromverbrauch und »Miete« ziemlich verschwenderisch. Wir werden deshalb besondere Ressourcemaße einführen, die solche Dinge wie Stromverbrauch messen – solche Maße machen im Sequentiellen interessanteweise keinen besonderen Sinn. Die wichtigste Definition wird die des heiligen Grals der Parallelverarbeitung sein: schnelle arbeitsoptimale Algorithmen.

Für die Konstruktion von schnellen arbeitsoptimalen Algorithmen gibt es einige Reihe von Methodiken, von denen wir zwei genauer betrachten wollen: das gute alte *divide et impera* in der parallelen Form und Accelerated-Cascading. Anders als die Entwurfsprinzipien nach Foster sind dies Methodiken, die gewissermaßen »eine Ebene höher« ansetzen: Hier werden Algorithmen optimiert, lange bevor es um die Aufteilung der Arbeit auf konkrete Einheiten geht.

6-1

Kapitel 6
Parallele Ressourcemaße
Lohnt sich der ganze Aufwand?

6-2

Lernziele dieses Kapitels

1. Die Ressourcen Kosten, Speedup und Effizienz kennen
2. Optimalitätsbegriffe bei parallelen Algorithmen kennen
3. Ressourceverbrauch von Algorithmen bestimmen können
4. Das Konzept der Arbeit-Zeit-Repräsentation kennen

Inhalte dieses Kapitels

Worum es heute geht

Frage: *Welche drei Ressourcen sind bei sequentiellen Programmen am wichtigsten?* Antwort: *Zeit, Zeit und Zeit.* Wenn man möchte, kann man als vierte noch den Arbeitsplatz hinzufügen; aber eigentlich sind die ersten drei Ressourcen doch wichtiger. Bei parallelen Programmen liegt die Sachlage kompliziert. Wir werden viele Probleme kennen lernen, die sich auf einem parallelen System theoretisch rasend schnell lösen lassen. Eine Laufzeit von $O(\log n)$ ist da schon langsam – und bekanntermaßen wächst der Logarithmus ausgesprochen gemütlich mit der Eingabegröße.

Trotzdem werden uns diese Algorithmen nicht befriedigen: Sie mögen schnell sein, sie gehen aber oft sehr verschwenderisch mit der zur Verfügung stehenden Rechenkraft um. Dann tritt folgender Effekt ein: Sie haben sich einen niegelnagelneuen Rechner mit zwei Quad-Core-Prozessoren unter Ihrem Schreibtisch gestellt und erwarten nun, dass dieser Probleme achtmal schneller lösen wird als ein Rechner mit nur einem Prozessor. Vielleicht auch nur sieben- oder sogar nur sechsmal schneller – schließlich müssen die Einheiten etwas mehr kommunizieren. Sie wären zu Recht beleidigt, wenn aber das System mit seinen acht Kernen das Problem *langsamer* löst als eine einzelne Einheit. Genau das passiert aber schnell bei parallelen Algorithmen: Diese sind oft so verschwenderisch, dass sie 20- oder 30-mal mehr an Rechnungen insgesamt durchführen.

Von *guten* parallelen Algorithmen dürfen wir verlangen, dass dieser Effekt nicht auftritt – wir werden sie (etwas verwirrlicherweise) *optimale* Algorithmen nennen. Um das genau zu definieren benötigen wir aber erstmal etwas (recht einfache) Theorie und einige neue (recht intuitive) Ressourcemaße wie *Speedup* und *Effizienz*.

6.1 Ressourcen und Maße

6.1.1 Wiederholung: Zeit

Zeitmaße bei parallelen und sequentiellen Algorithmen. 6-4

▶ Definition
Die Zeitkomplexität des *schnellsten (in der Regel unbekannten) sequentiellen Algorithmus* zur Lösung eines Problems bezeichnen wir mit $T^*(n)$.

▶ Definition
Die Rechenzeit eines parallelen Algorithmus Q bei *Eingabe* x und *genau* p *Einheiten* bezeichnen wir mit $T_p^Q(x)$, wobei wir Q auch weglassen, wenn dies aus dem Kontext klar ist. Wie üblich bezeichnet dann $T_p(n) = \max_{x\in\Sigma^n} T_p(x)$ die maximale Rechenzeit des Algorithmus bei Eingaben der Länge n.

▶ Definition
Die Rechenzeit eines *parallelen Algorithmus, der beliebig viele Einheiten benutzen darf,* bezeichnen wir mit $T_\infty(n)$, also $T_\infty(n) = \inf_{p\to\infty} T_p(n)$.

6.1.2 Speedup und Effizienz

Maße der Güte von parallelen Algorithmen. 6-5

▶ Definition: Speedup
Der *Speedup* ist das Verhältnis

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

▶ Definition: Effizienz
Die *Effizienz* ist

$$E_p(n) = \frac{T^*(n)}{p \cdot T_p(n)}.$$

✎ Zur Diskussion
Welchen Unterschied würde es machen, wenn man definieren würde $S'_p(n) = T_1(n)/T_p(n)$ und $E'_p(n) = T_1(n)/(pT_p(n))$?

6.1.3 Kosten und Arbeit

Kosten versus Arbeit 6-6

▶ Definition: Kosten
Die *Kosten* eines Algorithmus sind das Produkt

$$C_p(n) = p \cdot T_p(n).$$

▶ Definition: Arbeit
Die *Arbeit* $W_p(x)$ bei Eingabe x ergibt sich wie folgt:

1. Für jeden Zeitschritt wird gemessen, wie viele Einheiten tatsächlich arbeiten (nicht in einem Idle-Modus sind).
2. Diese Zahlen werden über die gesamte Berechnung hinweg aufsummiert.

Die *Arbeit* $W_p(n)$ bei Eingaben der Länge n ist $\max_{x\in\Sigma^n} W_p(x)$. Weiter sei $W_\infty(n) = \inf_{p\to\infty} W_p(n)$.

Merke
Wenn man einen Parallelrechner mietet, so sind die Kosten *die Miete* und die Arbeit der *Stromverbrauch*.

6-7

✎ Zur Übung

Bestimmen Sie Theta-Klassen für $T^*(n)$, $T^\infty(n)$, $W_p(n)$, $S_p(n)$, $E_p(n)$ und $C_p(n)$ für folgende Probleme/Algorithmen:

- Maximumsbestimmung
- Präfixsumme
- Pointer-Jumping
- 2D-Weichzeichner mit einer 5×5-Faltungsmatrix

6-8

Zusammenhang zwischen Kosten und Zeit

▶ Satz

Sei ein PRAM*-Programm Q für p Einheiten geben und eine Einheitenanzahl $p' \leq p$. Dann gibt es ein* PRAM*-Programm Q', für das gilt*

$$T_{p'}^{Q'}(n) = O\left(\frac{C_p^Q(n)}{p'}\right).$$

Beweis. Es gilt $C_p^Q(n) = p \cdot T_p^Q(n)$. Die Behauptung ist also $T_{p'}^{Q'}(n) = O(T_p^Q(n) \cdot p/p')$. Das Programm Q' darf also um den Faktor p/p' langsamer sein als Q. Dies ist aber einfach zu bewerkstelligen: Jede der p' Einheiten simuliert sequentiell $\lceil p/p' \rceil$ Einheiten von Q. Dann dauert die Simulation *eines Schrittes von Q* gerade $O(p/p')$ *Schritt von Q'*. □

6.1.4 Optimalität

6-9

Was sind die »besten« parallelen Algorithmen?

▶ Definition

Ein paralleler Algorithmus heißt *optimal*, wenn für ihn gilt $W_\infty(n) = \Theta\big(T^*(n)\big)$.
Dabei ist die Rechenzeit *egal (!)*.

Beispiel

Der sequentielle Algorithmus zur Berechnung des Matrix-Vektor-Produkts ist, als entarteter paralleler Algorithmus aufgefasst, optimal.

Beispiel

Der parallele Algorithmus zur Berechnung des Matrix-Vektor-Produkts ist optimal, da die Arbeit nur $O(n^2)$ ist.

6.2 Arbeit-Zeit-Repräsentation

6.2.1 Konzept

6-10

Wie sag ich es meinem Computer?

Aufschreibe-Schwierigkeiten bei parallelen Algorithmen

Will man PRAM-Programme »in Maschinensprache« aufschreiben, dann

- muss man für jede Einheit angeben, was sie genau machen soll,
- muss man darauf achten, dass das Timing stimmt, und
- muss man sich mit vielen Indizes herumschlagen.

```
1 n ← length(A)
2 for h ← 1 to log n seq do
3     if pid ≤ n/2^h then
4         global read a ← A[2 · pid]
5         global read b ← A[2 · pid − 1]
6         c ← a + b
7         global write A[pid] ← c
8 if pid = 1 then
9     global write S ← c
```

Idee

Wir schreiben Programme in der so genannten *Arbeit-Zeit-Repräsentation* auf. Dazu geben wir für jeden Zeitpunkt an, was alles parallel *passieren könnte*. Die Verteilung auf die Einheiten wird vom Compiler übernommen. Die `parallel for` und `sections` Statements von OpenMP sind *genau die gewünschten Angaben, was parallel passieren könnte*.

Formalisierung der Arbeit-Zeit-Repräsentation

6-11

Formalisierung

Formal ist die Arbeit-Zeit-Repräsentation eine Einschränkung daran, wie ein PRAM-Programm auf den globalen Speicher zugreifen kann. Formal würde man zunächst alle Zugriffe auf den globalen Speicher und das PID-Register *verbieten*. Dann führt man spezielle Befehle neu ein, die `parallel for` und `sections` von OpenMP entsprechen.

Schreibweise

Im Pseudocode schreiben wir ***par do*** statt `parallel for`.

Beispiel einer WT-Repräsentation

6-12

```
1 for h ← 1 to log n seq do
2     for i ∈ {1, 2, 3, . . . , n/2^h} par do
3         A[i] ← A[2i − 1] + A[2i]
4 S ← A[1]
```

- Für jeden Par-Do-Befehl werden eigentlich $n/2^h$ Einheiten benötigt.
- Stehen so viele zur Verfügung, führt einfach jeder den Par-Do-Körper für ein anderes i aus.
- Stehen nur $p < n/2^h$ Einheiten zur Verfügung, so kümmert sich jede dieser Einheiten *nacheinander* um den Par-Do-Körper für einen Block von $(n/2^h)/p$ vielen unterschiedlichen i.

6.2.2 Beschleunigungssatz

Arbeit und Zeit von Programmen in WT-Repräsentation

6-13

▶ Definition

Der *Zeitbedarf* $T^Q(n)$ und die *Arbeit* $W^Q(n)$ eines Programms Q in WT-Repräsentation sind die maximale Rechenzeit und Arbeit bei Eingaben der Länge n, wenn alle Par-Do-Befehle parallel ausgeführt werden.

▶ Satz

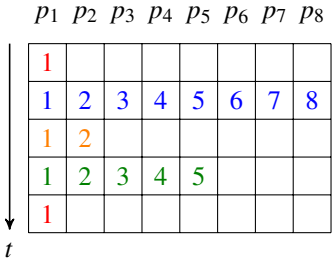
Sei ein Programm Q in WT-Repräsentation gegeben und p Einheiten. Dann gibt es ein PRAM*-Programm Q' mit*

$$T_p^{Q'}(n) \leq \left\lfloor \frac{W^Q(n)}{p} \right\rfloor + T^Q(n).$$

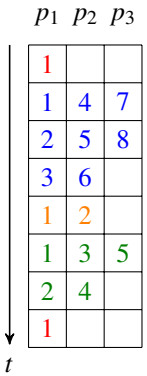
Merke
Für kleine p gilt $T_p(n) \approx W(n)/p$, falls $W(n) \gg T(n)$.

Beweis. Die PRAM arbeitet wie folgt: Sie arbeitet das Programm sequentiell ab bis zum ersten Par-Do-Befehl. Bei diesem möge eine Variable v gerade x unterschiedliche Werte annehmen.
Falls $p \leq x$ (falls also zu wenig Einheiten zur Verfügung stehen), so beginnen alle Einheiten parallel zu arbeiten. Jeder von ihnen führt sequentiell den Körper des Par-Do-Befehls für $\lceil x/p \rceil$ unterschiedliche Werte aus.
Falls $p > x$ (falls also mehr als genügend Einheiten zur Verfügung stehen), so führen die ersten x Einheiten parallel jeder einmal den Körper des Par-Do-Befehls aus (natürlich jeder für anderen Wert von v). Die restlichen Einheiten machen zunächst nichts, sie werden aber vorrätig gehalten für eventuelle verschachtelte Par-Do-Befehle.
Als Beispiel sei die Simulation eines Programms Q durch Q' angegeben:

Programm Q



Programm Q' für $p = 3$



Sei nun W^t die Menge an Arbeit, die Q in Zeitschritt t leistet. Dann gilt $W(n) = \sum_{t=1}^{T(n)} W^t$ und

$$T'_p(n) \leq \sum_{t=1}^{T(n)} \lceil W^t/p \rceil \leq \sum_{t=1}^{T(n)} \left(\lfloor W^t/p \rfloor + 1 \right) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + T(n).$$

□

6.2.3 Fallstricke

6-14

Wie viel wird hier gearbeitet?

```
1 for i ∈ {1,...,n} par do
2     while S[i] ≠ S[S[i]] do
3         S[i] ← S[S[i]]
```

Offenbar werden (in der Regel) einige Einheiten früher fertig als andere. Sie warten alle an der impliziten Barriere am Ende der For-Schleife.

✎ Zur Diskussion
Der Compiler möchte den Einheiten im Idle-Modus Arbeit verschaffen. Was ist hier anders als bei einem normalen Par-Do? Wie könnte der Compiler mit dem Problem umgehen?

Zusammenfassung dieses Kapitels

▶ Ressourcemaße bei parallelen Programmen

- $T^*(n)$ Rechenzeit des schnellsten sequentiellen Algorithmus bei Eingabelänge n.
- $T_p^Q(n)$ Rechenzeit des parallelen Algorithmus Q bei p Einheiten.
- $E_p^Q(n)$ Effizienz von Q, definiert als $T^*(n)/(p \cdot T_p(n))$. Beginnt bei 1, tendiert gegen 0 für p gegen Unendlich.
- $S_p^Q(n)$ Speedup von Q, definiert als $T^*(n)/T_p(n)$. Beginnt zwischen 0 und 1, steigt dann maximal linear, tendiert gegen einen Grenzwert für p gegen Unendlich.
- $C_p^Q(n)$ Kosten von Q, definiert als $T_p^Q(n) \cdot p$.
- $W_p^Q(n)$ Arbeit von Q

▶ Optimalität

Ein *optimaler* Algorithmus *arbeitet* nicht mehr als der beste sequentielle Algorithmus. Seine Rechenzeit *ist egal*.

▶ Arbeit-Zeit-Repräsentation

Bei der WT-Repräsentation wird angegeben, was *prinzipiell parallel* ausgeführt werden kann. Es gilt:

$$T_p(n) \leq W(n)/p + T(n).$$

Übungen zu diesem Kapitel

Übung 6.1 Programmanalyse, leicht

Gegeben sei das folgende Programm, welches $p = n$ Einheiten benutzt:

```
1  for i ← 1 to n seq do
2      if pid ist gerade und pid < n then
3          global read a ← A[pid]
4          global read b ← A[pid + 1]
5          if a > b then
6              global write A[pid] ← b
7              global write A[pid + 1] ← a
8      // Sync
9      if pid ist ungerade und pid < n then
10         global read a ← A[pid]
11         global read b ← A[pid + 1]
12         if a > b then
13             global write A[pid] ← b
14             global write A[pid + 1] ← a
15     // Sync
```

Wir nehmen an, dass Indizierungen bei 1 beginnen und dass die Einheiten an den angegebenen Stellen synchronisiert werden (wenn also ein Prozessor einen If-Block nicht ausführt, dann wartet er genau so lange, wie die anderen Einheiten zur Ausführung brauchen).

Beantworten Sie folgende Fragen in Bezug auf das Programm:

1. Was macht das Programm?
2. Welche Zugriffsart (wie zum Beispiel EREW, Priority-CRCW, etc.) benötigt das Programm? Begründen Sie Ihre Antwort.
3. Wie lautet $T^*(n)$?
4. Berechnen Sie für $p = n$ die Arbeit, die Kosten, die Effizienz und den Speedup des Programms (grob, O-Klasse reicht).
5. Ist der Algorithmus optimal? Begründen Sie Ihre Antwort.

7-1

Kapitel 7
Paralleles Teilen und Herrschen

Teilbar = parallelisierbar

7-2

Lernziele dieses Kapitels

1. Divide-and-Conquer Algorithmus für die konvexe Hülle kennen
2. Divide-and-Conquer Algorithmus für parallelen Merge-Sort kennen

Inhalte dieses Kapitels

Worum es heute geht

Wie jeder Informatikfeldherr weiß, lassen sich mit dem antiken »Teile und herrsche« viele Probleme in den Griff bekommen. Jedoch: Erstens ist die Redewendung gar nicht antik (siehe unten) und zweitens ist im Parallelen das *Teilen* oft leicht, das *Herrschen* (sprich: Zusammenfügen) weniger. Wie dies doch geht, darum soll es heute gehen.

Aus *de.wikipedia.org/wiki/Divide_et_impera*

> *Divide et impera* (lateinisch für »Teile und herrsche«) ist eine Redewendung und steht für das Prinzip, unter Gegnern Zwietracht und Uneinigkeit zu säen, um so in der Machtausübung ungestört zu bleiben. Es ist angeblich ein Ausspruch des französischen Königs Ludwigs XI. (*diviser pour régner*), die lateinische Version geht vermutlich auf die Renaissance zurück. Heinrich Heine schrieb am 12. Januar 1842 aus Paris: »König Philipp hat die Maxime seines mazedonischen Namensgenossen, das ›Trenne und herrsche‹, bis zum schädlichsten Übermaß ausgeübt.« Der gemeinte Namensgenosse war Philipp II. (359–336 vor Christus), der den größten Teil des Balkans beherrschte. Die Redewendung ist wahrscheinlich nicht antik, wenngleich die damit bezeichnete Strategie sehr alt ist und zum Beispiel in der römischen Außenpolitik ohne Zweifel wiederzuerkennen ist. Sunzi (um 500 vor Christus) beschreibt »Teile und herrsche« als eine Strategie der chinesischen Kriegskunst. [...]
>
> Goethe formuliert in »Sprichwörtliches« einen Gegenvorschlag:
>
> > Entzwei und gebiete! Tüchtig Wort. –
> > Verein und leite! Besserer Hort.

7.1 Konvexe Hüllen

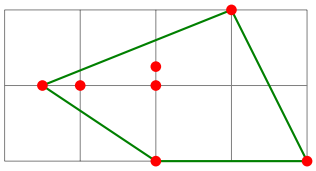
7.1.1 Problemstellung

Das Konvexe-Hülle-Problem in 2D. 7-4

Problem

Eingabe Eine *Menge von Punkten* in der Ebene.

Output Die *konvexe Hülle* der Punkte (kleinstes Polygon, das alle Punkte enthält).



Eingabe

Ausgabe

Dieses Problem ist wichtig, wenn man *maximale Entfernungen* berechnen möchte oder testen möchte, ob ein Punkt *innerhalb* einer Punktwolke liegt.

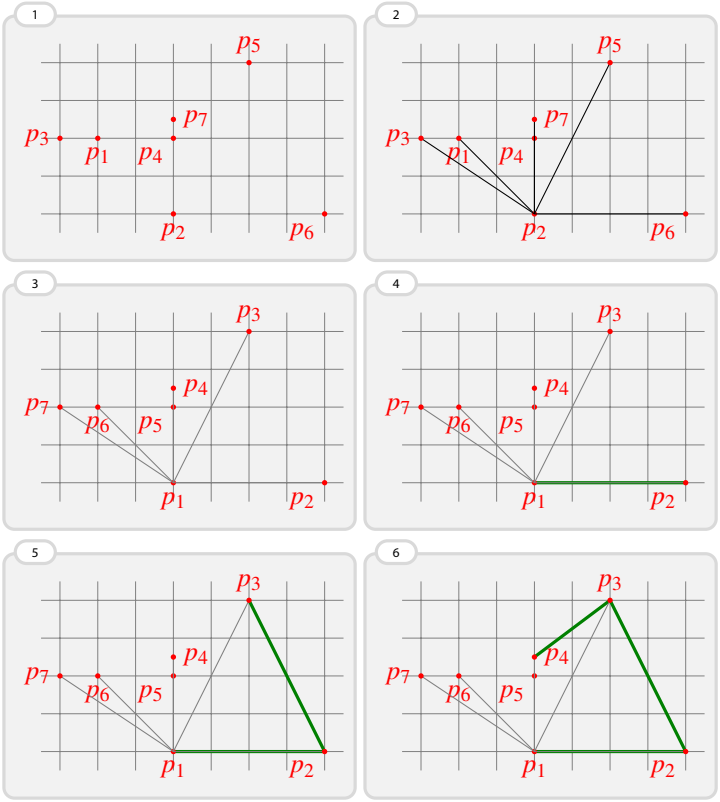
7.1.2 Sequentieller Algorithmus

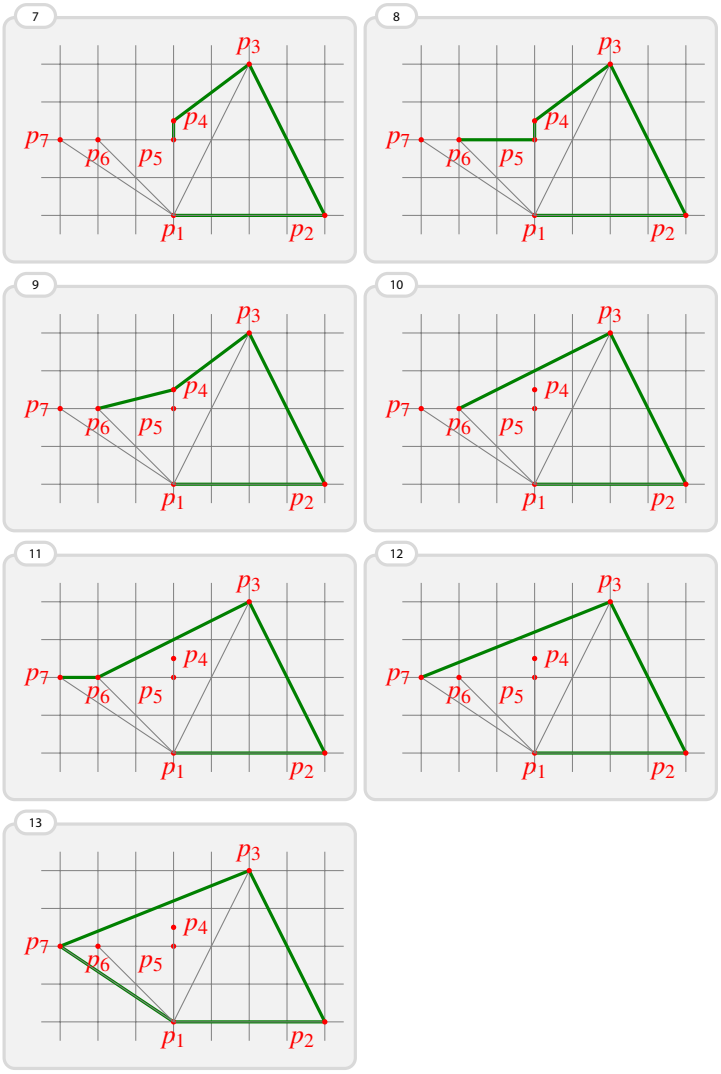
Grahams Algorithmus für das Problem. 7-5

Graham's Scan berechnet *sequentiell* konvexe Hüllen in Zeit $O(n \log n)$.
Er funktioniert wie folgt:

1. Bestimme den Punkt mit der kleinsten y-Koordinate.
2. Erklären diesen Punkt zum Ursprung.
3. Sortiere die Punkte nach ihrem Polarwinkel (beispielsweise mittels Merge-Sort).
4. Für jeden Punkt tue nun folgendes:
   - 4.1 Füge den Punkt zur konvexen Hülle hinzu.
   - 4.2 Ziehe an der Hüllenschnur.

Graham's scan in action. 7-6





7.1.3 Paralleler Algorithmus

7-7

Ziel: Ein paralleler Algorithmus für die konvexe Hülle.

Graham's Scan ist »sehr sequentiell«. Für einen (vorzugsweise optimalen) parallelen Algorithmus müssen wir uns etwas Neues einfallen lassen. Idee: Divide-and-Conquer.
Divide-and-Conquer besteht aus drei Phasen:

1. Teile die Eingabe in mehrere gleichgroße Teile.
2. Löse das Problem rekursiv auf den Teilen.
3. Füge die Teile zu einem Ganzen zusammen.

Offenbar ist der zweite Schritt leicht zu parallelisieren, die Schwierigkeiten liegen beim ersten und letzten Schritt.

7-8

Ein paralleler Algorithmus für die konvexe Hülle.

Der Algorithmus beginnt mit einer Vorverarbeitung:

1. Wir sortieren die Punkte nach ihren x-Koordinaten. Dies geht in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$, wie wir später sehen werden.
2. Wir beschränken uns auf das Problem, die obere konvexe Hülle zu finden.

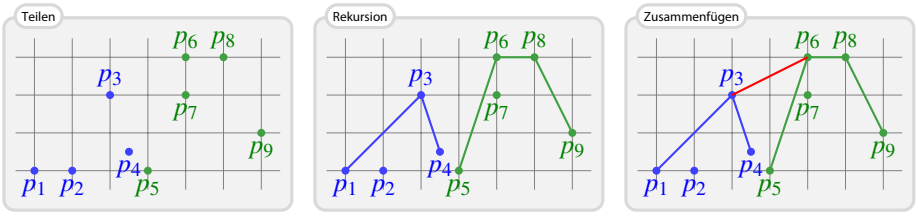
Der eigentliche D&C-Teil geht nun wie folgt:

1. Teile die Punkte auf in die erste und die zweite Hälfte bezüglich den x-Koordinaten.
2. Berechne rekursiv die konvexen Hüllen der Teile.
3. Zum Zusammenfügen berechne die beiden Punkte auf den konvexen Hüllen, »unterhalb« derer alle Punkte liegen.

Die Phasen des parallelen Algorithmus.

7-9

1. Teile die Punkte in die erste und die zweite Hälfte bezüglich den x-Koordinaten.
2. Berechne rekursiv die konvexen Hüllen der Teile.
3. Zum Zusammenfügen berechne die beiden Punkte p^0 und p^1 auf den konvexen Hüllen, »unterhalb« derer Verbindung alle Punkte liegen.



7.1.4 Analyse

Die Komplexität des Algorithmus.

7-10

▶ Lemma

Das Punktepaar (p^0, p^1) kann in Zeit $O(\log n)$ sequentiell berechnet werden.

Beweis. Übungsaufgabe 7.1. □

▶ Satz

Der parallele Algorithmus berechnet konvexe Hüllen in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$.

Da, wie wir später sehen werden, das Sortieren $O(\log^2 n)$ dauert und Arbeit $O(n \log n)$ verursacht, müssen wir nur die Zeit und die Arbeit des Divide-and-Conquer-Teils berechnen.

Beweis des Satzes – Die Laufzeit.

7-11

Analyse der Laufzeit. Für die benötigte Rechenzeit lässt sich folgende Rekursionsformel aufstellen:

$$T(n) = T(n/2) + c \cdot \log n,$$

wobei $T(n/2)$ die in der Rekursion verbrachte Zeit ist und $c \cdot \log n$ die Zeit zur Berechnung der Punkte p^0 und p^1. Das Umkopieren der Listen im Speicher geschieht in Zeit $O(1)$.
Die Lösung der Gleichung mittels des Master-Theorems (siehe die Vorlesung »Algorithmendesign«) liefert sofort $T(n) = O(\log^2 n)$. □

Beweis des Satzes – Die Arbeit.

7-12

Analyse der Arbeit. Die Hauptarbeit fällt beim Zusammenfügen der zwei Listen an, nachdem die Rekursion fertig ist. Diese Arbeit ist einmal $O(\log n)$ für die Berechnung der Punkte p^0 und p^1 und dann nochmal $O(n)$, um die Teillisten im Speicher nebeneinander zu kopieren. Dies ergibt die Formel:

$$W(n) = 2W(n/2) + O(n).$$

Dies ist die gleiche Rekursionsformel wie bei Merge-Sort und hat bekanntermaßen die Lösung $W(n) = O(n \log n)$. □

7.2 Verschmelzen

7.2.1 Problemstellung

7-13

Sortieren

Das Sortierproblem

Eingabe Liste von Zahlen (oder allgemein, Objekten)

Ausgabe Sortierte Liste

7-14

✎ Zur Übung

Geben Sie einen parallelen Sortier-Algorithmus an, der in Zeit $O(\log n)$ und mit Arbeit $O(n^2)$ auskommt.

Tipp: Bestimmen Sie für jedes Element parallel, wo es in der sortierten Liste auftaucht. Diese Position ist gleich der Anzahl der Element, die kleiner als es selbst sind.

7-15

Verschmelzen

Um das Sortierproblem mit weniger Arbeit zu lösen, können wir Merge-Sort verwenden. Offenbar ist nun das Problem, das *Verschmelzen* zu parallelisieren.

Das Verschmelzungsproblem

Eingabe Zwei sortierte Listen von Zahlen

Ausgabe Verschmolzene sortierte Liste

7.2.2 Einfacher paralleler Algorithmus

7-16

Eine Definition, die beim einfachen parallelen Verschmelzungs-Algorithmus hilfreich ist.

▶ Definition

Für eine sortierte Liste A von Zahlen und eine Zahl x ist der *Rang* von x in A die Stelle, an die x in A einsortiert werden müsste.

Beispiel

Der Rang von 15 in der Liste $(1,3,9,14,20,30)$ ist 5.

Offenbar kann der Rang eines Elementes sequentiell in Zeit $O(\log n)$ mittels binärer Suche ermittelt werden.

7-17

✎ Zur Übung

Gegeben seien eine sortierte Liste A und zwei Elemente x_1 und x_2 mit $x_1 < x_2$.

- Wie groß sind parallele Zeit und Arbeit, die Ränge von x_1 und x_2 zu bestimmen?
- Seien r_1 und r_2 die Ränge. An welchen Stellen landen dann x_1 und x_2, wenn man sie *beide* einfügt?

✎ Zur Übung

Wie die vorherige Aufgabe, nur mit drei Zahlen $x_1 < x_2 < x_3$.

7-18

Der einfache parallele Verschmelzungsalgorithmus.

Einfacher Verschmelzungsalgorithmus

Gegeben seien zwei sortierte Listen A und B.

1. Bestimme parallel für jedes a_i seinen Rang in B.
2. Bestimme gleichzeitig parallel für jedes b_i seinen Rang in A.
3. Platziere parallel alle a_i und b_i im verschmolzenen Array nach folgender Vorschrift:
   - Ist r_i der Rang von a_i in B, so platziere a_i an die Stelle $r_i + i - 1$.
   - Ist s_i der Rang von b_i in A, so platziere b_i an die Stelle $s_i + i - 1$.

Beispielablauf des Verschmelzungsalgorithmus. 7-19

Die Schritte des Algorithmus für die Eingaben

$$A = (10,20,30,40,50,60,70,80,90)$$
$$B = (1,2,3,41,42,43,81,82,100)$$

1. Die Ränge werden bestimmt. Die Ränge der a_i in B und der b_i in A lauten jeweils:

$$r = (4,4,4,4,7,7,7,7,9)$$
$$s = (1,1,1,5,5,5,9,9,10)$$

2. Die a_i und b_i werden deshalb wie folgt platziert:

Position r_i+i-1	4	5	6	7	11	12	13	14	17
Wert a_i	10	20	30	40	50	60	70	80	90

Position s_i+i-1	1	2	3	8	9	10	15	16	18
Wert b_i	1	2	3	41	42	43	81	82	100

3. Das Resultat ist

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Wert	1	2	3	10	20	30	40	41	42	43	50	60	70	80	81	82	90	100

Die Komplexität des einfachen Verschmelzungsalgorithmus. 7-20

▶ Satz

Der einfache Verschmelzungsalgorithmus benötigt Zeit $O(\log n)$ und Arbeit $O(n \log n)$.

Beweis. Es wird parallel n mal etwas getan, was $O(\log n)$ lange dauert. □

Eine einfache, aber noch nicht ausreichende Folgerung. 7-21

▶ Folgerung

Das Sortierproblem kann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log^2 n)$ gelöst werden.

Beweis. Wieder lassen sich zwei Rekursionsgleichungen aufstellen. Auf jeder Rekursionsstufe wird $O(\log n)$ lange gearbeitet und das macht $O(n \log n)$ Arbeit. Dies liefert:

$$T(n) = T(n/2) + O(\log n),$$
$$W(n) = 2W(n/2) + O(n \log n).$$

Das Master-Theorem liefert $T(n) = O(\log^2 n)$ und $W(n) = O(n \log^2 n)$. □

7.2.3 Ausblick: Optimaler Algorithmus

Wie bekommen wir einen optimalen parallelen Algorithmus? 7-22

Wir sind nicht optimal, da die Arbeit $O(n \log n)$ statt $O(n)$ ist. Wir müssen also Arbeit einsparen. Der Trick ist, *weniger zu parallelisieren*. In der Vorlesung zum *Accelerated Cascading* werden wir sehen, wie dies geht. Mit diesem verbesserten Algorithmus werden wir dann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ sortieren können.

Zusammenfassung dieses Kapitels

▶ Divide-and-Conquer im Parallelen

Wie im Sequentiellen besteht Divide-and-Conquer im Parallelen aus drei Schritten:

1. Teilen
2. Rekursion
3. Zusammenfügen

Die Rekursion wird »automatisch parallelisiert«, die anderen Teil sind oft schwieriger.
Ist die Laufzeit eines sequentiellen Divide-and-Conquer-Algorithmus

$$T_1(n) = aT_1(n/b) + f(n),$$

so gilt dann für die parallelisierte Variante

$$T(n) = T(n/b) + f(n),$$
$$W(n) = aT(n/b) + f(n).$$

▶ Satz

Man kann die konvexe Hülle von n Punkten in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ berechnen.

▶ Satz

Man kann zwei sortierte Listen der Längen n in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ verschmelzen.

Übungen zu diesem Kapitel

Übung 7.1 Schnelles Finden einer Tangente, schwer

Beweisen Sie das Lemma von Seite 7-10.

Tipps: Führen Sie gleichzeitig auf beiden Teilhüllen eine binäre Suche durch. In jedem Schritt sollten Sie in beiden Teilhüllen jeweils die Hälfte der Punkte eliminieren. Um zu ermitteln, welche Punkte eleminierte werden können, berechnen Sie Tangenten an die aktuellen Punkte und die Teilhüllen. Berechnen Sie, wo diese Tangenten sich treffen und wo dieser Schnittpunkt relativ zu den aktuellen Punkten auf der Tangente liegt (links oder rechts).

Übung 7.2 Anordnung von Punkten bestimmen, leicht

Geben Sie ein möglichst einfaches und numerisch stabiles Verfahren an um festzustellen, ob ein Punkt p links oder rechts von der durch die Punkte q_1 und q_2 gehenden Gerade liegt.

Übung 7.3 Programmodifikation, mittel

Bei den Algorithmen zum Verschmelzen zweier Listen und beim Partitionierungsalgorithmus hatten wir vorausgesetzt, dass alle Elemente paarweise verschieden sind. Wir betrachten nun Listen, die auch gleiche Elemente enthalten dürfen.

Zeigen Sie, wie man mit Hilfe eines Vorverarbeitungsschrittes und eines Nachbearbeitungsschrittes die Algorithmen auch für solche Listen verwenden kann. Geben Sie für die Vorverarbeitung und die Nachbearbeitung je ein Verfahren an, das konstante Zeit und lineare Arbeit benötigt.

Tipps: Addieren Sie kleine Störwerte auf die Elemente.

Übung 7.4 Kenngrößen in Wäldern, schwer

Gegeben sei ein Wald mit n Knoten. Dieser ist durch eine Liste $P(1), \ldots, P(n)$ repräsentiert, wobei $P(i)$ den Elternknoten des Knotens i angibt.

1. Geben Sie einen Algorithmus an, der die Anzahl der Bäume im Wald bestimmt. Formalisieren Sie den Algorithmus in Arbeit-Zeit-Repräsentation und bestimmen Sie $T^*(n)$, $T^\infty(n)$, $W(n)$, $E_p(n)$ und $C_p(n)$.
2. Geben Sie einen Algorithmus an, der die Anzahl der Blätter im Wald bestimmt. Formalisieren Sie den Algorithmus in Arbeit-Zeit-Repräsentation und bestimmen Sie $T^*(n)$, $T^\infty(n)$, $W(n)$, $E_p(n)$ und $C_p(n)$.
3. Geben Sie einen Algorithmus an, der die Größe des größten Baumes im Wald bestimmt. Die Größe eines Baumes ist dabei die Anzahl seiner Knoten. Der Algorithmus soll eine Laufzeit von $O(\log^2 n)$ und eine Arbeit von $O(n \log n)$ haben. Sie dürfen davon ausgehen, dass man n Zahlen in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ sortieren kann.

Übung 7.5 Medianbestimmung, schwer

Gegeben ist eine Liste S aus n Zahlen. Der Median von S ist der Wert m, der nach sortieren der Liste an Stelle $\lfloor \frac{n}{2} \rfloor$ steht. Anders ausgedrückt: Der Median von S ist der $\lfloor \frac{n}{2} \rfloor$-kleinste Eintrag in S.

1. Entwickeln Sie einen sequenziellen Algorithmus, der den Median einer Liste bestimmt und mit möglichst wenig Zeit auskommt.
2. Entwickeln Sie einen sequenziellen Algorithmus, der den Median einer Liste bestimmt und mit einer möglichst kleinen *durchschnittlichen* Laufzeit auskommt.
   Tipp: Nutzen Sie einen Divide-and-Conquer-Ansatz ähnlich zu Quicksort.
3. Analysieren Sie für beide Algorithmen die Laufzeiten im Best-Case, Average-Case und Worst-Case.
4. Diskutieren Sie, wie gut sich die beiden Algorithmen parallelisieren lassen.

Übung 7.6 Quicksort parallelisieren, mittel

Parallelisieren Sie Quicksort, indem Sie die folgenden zwei Teilaufgaben bearbeiten:

1. Beschreiben Sie den sequentiellen Quicksort. Gehen Sie auf das Teilen und das Verschmelzen ein und erklären Sie jeweils wie die Aufgaben parallelisiert werden können.
2. Entwerfen Sie daraus einen parallelen Quicksort und schreiben sie ihn in Arbeit-Zeit-Repräsentation auf.

Übung 7.7 Parallelen Quicksort analysieren, schwer

Analysieren Sie den Ressourcenbedarf des in Übung 7.6 entwickelten Algorithmus. Gehen Sie dazu wie folgt vor:

1. Analysieren Sie Zeit und Arbeit ihres Quicksorts im Worst-Case.
2. Analysieren Sie Zeit und Arbeit ihres Quicksorts wahlweise im Best-Case oder (etwas schwieriger) im Average-Case.
3. Berechnen Sie $T^*(n)$, $T^\infty(n)$, $W(n)$, $E_p(n)$ und $C_p(n)$.

Übung 7.8 Parallelen Quicksort implementieren, mittel

Schreiben Sie ein OpenMP-Programm das Ihren parallelen Quicksort aus Übung 7.6 implementiert und testen Sie es.

Übung 7.9 Verschmelzungsalgorithmen erweitern, mittel

Bei den Algorithmen zum Verschmelzen zweier Listen und beim Partitionierungsalgorithmus in der Vorlesung hatten wir vorausgesetzt, dass alle Elemente paarweise verschieden sind. Wir betrachten nun Listen, die auch gleiche Elemente enthalten dürfen.

Zeigen Sie, wie man mit Hilfe eines Vorverarbeitungsschrittes und eines Nachbearbeitungsschrittes die Algorithmen auch für solche Listen verwenden kann. Geben Sie für die Vorverarbeitung und die Nachbearbeitung je ein Verfahren an, das konstante Zeit und lineare Arbeit benötigt. Hierbei dürfen Sie davon ausgehen, dass es sich bei den Elementen in der Liste um ganze Zahlen handelt.

Tipps: Addieren Sie kleine Störwerte auf die Elemente.

8-1

Kapitel 8
Accelerated-Cascading
Schnell und gut

8-2

Lernziele dieses Kapitels

1. Konzept des Accelerated-Cascading verstehen und anwenden können
2. Beispiele von Accelerated-Cascading-Algorithmen kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn man naiv an den Entwurf paralleler Algorithmen herangeht, so parallelisiert man »alles und jedes«. Dies führt leider schnell dazu, dass die Einheiten ganz eifrig rechnen, die insgesamt geleistete Arbeit aber uferlos steigt. Dabei ist schon eine parallele Arbeit, die auch nur um einen logarithmischen Faktor größer ist als die optimale sequentielle Laufzeit, nicht akzeptabel: Für eine realitische Eingabegröße wie $n = 1000$ werden wir zu recht die Nase über einen Algorithmus rümpfen, der auf einem System mit zwei Quad-Core-Prozessoren langsamer ist als ein ganz einfacher sequentieller Algorithmus.

Oberstes Gebot bei parallelen Algorithmen muss sein, die Arbeit niedrig zu halten. Was aber tun, wenn durch die Parallelisierung die Arbeit steigt? Weniger parallelsieren! »Accelerated-Cascading« ist der vornehme Fachausdruck hierfür, letztendlich geht es aber einfach darum, weniger zu arbeiten, indem man weniger parallelisiert. Natürlich wollen wir das Kind nicht mit dem Bade ausschütten und zu radikal »sequentialisieren«. Der Trick ist, gerade so viel weniger zu parallelisieren, dass die Geschwindigkeit nicht wächst, die Arbeit aber merklich fällt. Dieser Balanceakt gelingt nicht immer, aber meistens.

Das erste Beispiel, an dem Accelerated-Cascading heute demonstriert wird, ist auf den ersten Blick schon fast beleidigend einfach: Die Bestimmung des Maximums von n Zahlen. Trivialerweise ist dieses Problem sequentiell in Zeit $O(n)$ lösbar und parallel natürlich in Zeit $O(\log n)$ und Arbeit $O(n)$. Die hohe Kunst ist nun, es *deutlich schneller* hinzubekommen. Verblüffenderweise kann man das Maximum von n Zahlen sogar in *konstanter (!) Zeit* berechnen, dies geht aber massiv auf Kosten der Arbeit, welche auf $O(n^2)$ hochschnellt. Mittels Accelerated-Cascasing und weiteren Raffinessen werden wir dies ausbalancieren und eine Laufzeit von $O(\log\log n)$ bei einer Arbeit von $O(n)$ hinbekommen.

8.1 Vorbereitung: Das Maximumproblem

8.1.1 Problemstellung

Die Hauptproblemstellung für heute. 8-4

Das Maximumproblem

Eingabe Eine Liste A von n Zahlen.

Ausgabe Das Maximum von A.

Wie schnell lässt sich das Problem lösen?

1. Offenbar ist $T^*(n) = n$.
2. Es gibt einen EREW-Algorithmus mit $T(n) = O(\log n)$ und $W(n) = O(n)$.
3. Es gibt einen CRCW-Algorithmus mit $T(n) = O(1)$ und $W(n) = O(n^2)$.
4. *Unser Ziel:* Ein CRCW-Algorithmus mit $T(n) = O(\log\log n)$ und $W(n) = O(n)$.

8.1.2 Schneller, verschwenderischer Algorithmus

Ein schneller CRCW-Algorithmus für das Maximumproblem. 8-5
Ziel und Vorbereitung.

Ziel

Ein CRCW-Algorithmus, der das Maximumproblem in Zeit $O(1)$ löst.

Hierzu ist eine kleine Vorüberlegung nützlich.

▶ Satz

Das Maximumproblem kann in Zeit $O(1)$ und Arbeit $O(n)$ gelöst werden, wenn alle Zahlen nur 0 oder 1 sind.

Beweis.

```
1  input A[1], ..., A[n]
2  m ← 0
3  for i ∈ {1,...,n} par do
4      if A[i] = 1 then
5          m ← 1
6  output m
```

□

Ein schneller CRCW-Algorithmus für das Maximumproblem. 8-6
Der Algorithmus.

```
1  input A[1], ..., A[n]
2  // Vorbereitung
3  for i ∈ {1,...,n} par do
4      for j ∈ {1,...,n} par do
5          if A[j] > A[i] then B[i,j] ← 1 else B[i,j] ← 0
6
7  // Zeilenmaxima
8  for i ∈ {1,...,n} par do
9      M[i] ← max{B[i,1],...,B[i,n]}
10
11 // Gesamtmaximum
12 for i ∈ {1,...,n} par do
13     if M[i] = 0 then
14         m ← i
15 output m
```

8-7 Ein schneller CRCW-Algorithmus für das Maximumproblem.
Die Analyse.

▶ Satz
Der CRCW*-Algorithmus berechnet das Maximum von n Zahlen in Zeit* $O(1)$ *und Arbeit* $O(n^2)$.

Beweis. Die Zeit ist konstant, da nur konstant viele sequentielle Schritte auszuführen sind (nämlich 3 oder 4, je nach Rechnung). Es werden nur $O(n^2)$ Einheiten benutzt. Da diese nur eine konstante Anzahl Schritte arbeiten, kann die Arbeit auch nur $O(n^2)$ sein. □

8.1.3 Schneller, fast optimaler Algorithmus

8-8 Ein schneller, fast optimaler CRCW-Algorithmus.
Ziel und Vorbereitung.

Ziel
Ein optimaler CRCW-Algorithmus, der das Maximumproblem möglichst schnell löst.

Zur Erinnerung:

- »Optimal« bedeutet hier »in Arbeit $O(n)$«.
- Wir können das Problem optimal in Zeit $O(\log n)$ auf einer EREW-PRAM lösen.

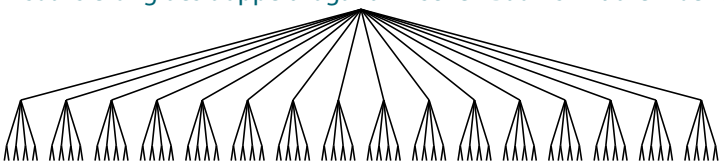
Wir suchen deshalb einen Algorithmus, der schneller als $O(\log n)$ ist. Wir werden in einem späteren Kapitel sehen, dass man nicht schneller als $O(\log\log n)$ werden kann. Also setzen wir uns $O(\log\log n)$ *als Ziel*.

8-9 Hilfsmittel: Doppelt-logarithmische Bäume.

▶ Definition: Doppelt-logarithmischer Baum
Ein *doppelt-logarithmischer Baum* ist wie folgt aufgebaut:

- Alle Blätter liegen auf einer Höhe, genannt *Ebene 0*.
- Ihre Eltern bilden die Ebene 1, deren Eltern die Ebene 2, und so weiter.
- Jeder Knoten auf Ebene 1 hat $2 = 2^1 = 2^{2^0}$ Kinder.
- Jeder Knoten auf Ebene 2 hat $4 = 2^2 = 2^{2^1}$ Kinder.
- Jeder Knoten auf Ebene 3 hat $16 = 2^4 = 2^{2^2}$ Kinder.
- Jeder Knoten auf Ebene 4 hat $256 = 2^8 = 2^{2^3}$ Kinder.
- Jeder Knoten auf Ebene 5 hat $65536 = 2^{16} = 2^{2^4}$ Kinder.
- Jeder Knoten auf Ebene i hat $2^{2^{i-1}}$ Kinder.

8-10 Visualisierung des doppelt-logarithmischen Baums mit drei Ebenen.



8-11 Tiefe und Blätteranzahl von doppelt-logarithmischen Bäumen.

✎ Zur Übung
Wie viele Blätter hat ein doppelt-logarithmischer Baum mit

1. einer Ebene?
2. zwei Ebenen?
3. drei Ebenen?
4. vier Ebenen?
5. fünf Ebenen?
6. i Ebenen?

Folgern Sie: Hat ein doppelt-logarithmischer Baum n Blätter, so ist seine Tiefe $\log(1+\log n) \leq 1+\log\log n$.
(Daher der Name.)

Maximale Anzahl von Kindern in doppelt-logarithmischen Bäumen. 8-12

▶ Lemma
In einem doppelt-logarithmischen Baum mit n Blättern hat die Wurzel $\sqrt{2n}$ Kinder.

Beweis. Die Wurzel liegt auf der Ebene der Höhe des Baumes, also auf Ebene $\log(1+\log n)$. Knoten auf Höhe i haben $2^{2^{i-1}}$ Kinder. Also hat die Wurzel

$$2^{2^{\log(1+\log n)-1}} = 2^{2^{\log(1+\log n)}/2} = 2^{(1+\log n)/2} = (2n)^{1/2} = \sqrt{2n}.$$

Kinder. □

Ein schneller, fast optimaler CRCW-Algorithmus. 8-13
Der Algorithmus.

Nehmen wir an, dass n die Größe eines doppelt-logarithmischen Baumes ist (also $n = 2^{2^k-1}$ für ein k). Wir »stellen uns vor«, dass ein Baum über die Eingaben gelegt wird.

```
1 input A[1], ..., A[n]
2 for i ← 1 to log(1 + log n) seq do
3     for alle Knoten k auf Ebene i par do
4         Berechne das Maximum der Kinder von k mit dem schnellen Algorithmus
5         Speichere das Maximum im Knoten k
6 output Wert an der Wurzel
```

Ein schneller, fast optimaler CRCW-Algorithmus. 8-14
Die Analyse.

▶ Satz
Der neue CRCW*-Algorithmus berechnet das Maximum von n Zahlen in Zeit $O(\log\log n)$ und Arbeit $O(n\log\log n)$.*

Beweis. Für die Zeit: Es gibt $O(\log\log n)$ serielle Schleifendurchläufe. In jedem Schleifendurchlauf wird nur konstant lange gerechnet, also ist die Laufzeit $O(\log\log n)$.
Für die Arbeit: Wir zeigen, dass in jedem Durchlauf höchstens $O(n)$ gearbeitet wird. In jedem Durchlauf werden die Wurzeln von x Teilbäumen der Größe n/x behandelt. Diese Wurzeln haben $\sqrt{2n/x}$ Kinder. Also ist die Arbeit $O\big(x(\sqrt{2n/x})^2\big) = O(2n)$. □

8.2 Methodik: Accelerated Cascading

8.2.1 Idee

Ziel des Accelerated Cascading: Gut und schnell. 8-15
Wir haben einen schnellen, aber nicht optimalen Algorithmus gegeben. Weiterhin haben wir einen langsamen, aber optimalen Algorithmus gegeben. Ziel von *Accelerated Cascading* ist, sie zu vereinigen und einen schnellen optimalen Algorithmus zu erhalten.

Allgemeine Vorgehensweise

- Teile die Eingabe in viele kleine Teile auf.
- Wende den langsamen Algorithmus auf die kleinen Teile parallel an.
  Dies ist langsam, aber bei kleinen Eingaben nicht schlimm.
- Wende den schnellen Algorithmus auf die Ergebnisse an.
  Dies kostet viel Arbeit, aber es gibt nur noch wenige Eingaben.

8.2.2 Beispiel: Maximumproblem

8-16

Beispiel eines Accelerated Cascading
Ein schneller optimaler CRCW-Algorithmus.

Ziel
Ein CRCW-Algorithmus, der das Maximumproblem in Zeit $O(\log\log n)$ und Arbeit $O(n)$ löst.

Zur Erinnerung:

- Wir haben einen schnellen, aber nicht optimalen Algorithmus, der Zeit $O(\log\log n)$ und Arbeit $O(n\log\log n)$ benötigt.
- Wir haben einen langsamen, aber optimalen Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n)$.

8-17

Beispiel eines Accelerated Cascading
Ein schneller optimaler CRCW-Algorithmus.

Algorithmus zur Maximumsbildung

1. Teile die Eingabe in Blöcke der Größe $\log\log n$ auf.
2. Bestimme für alle Blöcke parallel das Maximum.
3. Wende den schnellen Algorithmus auf die $n/\log\log n$ Ergebnisse an.

Beispiel: Beispiel für $n = 1.000.000.000$

1. Teile die Eingabe in Blöcke der Größe $\log_2\log_2 n \approx \log_2 30 \approx 5$ auf.
2. Bestimme für alle Blöcke parallel das Maximum.
3. Wende den schnellen Algorithmus auf die $1.000.000.000/5 = 200.000.000$ Ergebnisse an.

Analyse
Die ersten beiden Schritte dauern $O(\log\log\log n)$ und machen $O(n)$ Arbeit. Der letzte Schritt dauert höchstens $O(\log\log n)$ und die Arbeit ist $O(n/\log\log n \cdot \log\log n) = O(n)$.

8.2.3 Beispiel: Verschmelzung

8-18

Beispiel eines Accelerated Cascading
Ein schneller optimaler Verschmelzungsalgorithmus.

Ziel
Ein Verschmelzungsalgorithmus, der zwei sortierte Listen in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmilzt.

Zur Erinnerung: Es war einfach, einen Algorithmus anzugeben, der in Zeit $O(\log n)$ und Arbeit $O(n\log n)$ zwei Listen A und B verschmilzt: Suche mit binärer Suche für jedes Element a_i seine Position r_i in B und platziere a_i und Stelle $r_i + i - 1$, umgekehrt mit den b_i. Wir können sequentiell in Zeit $O(n)$ und Arbeit $O(n)$ zwei Listen verschmelzen.

8-19

Die Ideen beim optimalen parallelen Algorithmus
Anstatt alle Positionen parallel zu bestimmen, tun wir dies nur für jede $(\log n)$-te Zahl. Dazu unterteilen wir A in Intervalle der Länge $\log n$. Nennen wir die Intervalle $A_1, A_2, \ldots, A_{n/\log n}$. Dann bestimmen wir für *das erste Elemente* jedes Intervalls den Rang dieses Elements in B. Dies unterteilt B in $n/\log n$ Intervalle $B_1, \ldots, B_{n/\log n}$. Wären alle B_i von der Größe $O(\log n)$, so könnten wir parallel jedes A_i mit seinem B_i sequentiell verschmelzen und hätten den gewünschten Algorithmus. Da die B_i aber im Allgemeinen zu groß sind, wiederholen wir den Algorithmus nun nocheinmal für jedes Paar (A_i, B_i), aber mit vertauschten Rollen. Dies liefert Subblöcke $A_{i,j}$ und $B_{i,j}$, die Größe höchstens $O(\log n)$ haben, und deshalb alle gemeinsam in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmolzen werden können.

Beispiel einer Verschmelzung

8-20

Aufteilung des ersten Arrays.

Die Eingaben seien

i	1	2	3	4	5	6	7	8	9	10
$A[i]$	1	3	4	7	9	15	17	19	22	24
r_i										
$B[i]$	2	5	8	12	13	14	16	20	21	25

Wir bestimmen nun parallel lediglich die Ränge der *Blockanfänge*.

i	1	2	3	4	5	6	7	8	9	10
$A[i]$	1	3	4	7	9	15	17	19	22	24
r_i	1			3			8			10
$B[i]$	2	5	8	12	13	14	16	20	21	25

Wir müssen nun noch verschmelzen:

1. $A_1 = (1,3,4)$ mit $B_1 = (2,5)$.
2. $A_2 = (7,9,15)$ mit $B_2 = (8,12,13,14,16)$.
3. $A_3 = (17,19,22)$ mit $B_3 = (20,21)$.
4. $A_4 = (24)$ mit $B_4 = (25)$.

Dies ist einfach, außer bei B_2.

Beispiel einer Verschmelzung

8-21

Aufteilung der Blöcke im zweiten Array.

Für die »zu großen« B_i-Blöcke wiederholen wir das Spiel (hier nur für den einzigen zu großen Block gezeigt):

i	1	2	3	4	5
$A_2[i]$	7	9	15		
$B_2[i]$	8	12	13	14	16
s_i					

Wir bestimmen nun parallel wieder lediglich die Ränge der *Subblockanfänge*.

i	1	2	3	4	5
$A_2[i]$	7	9	15		
$B_2[i]$	8	12	13	14	16
s_i	2			3	

Wir müssen also für A_2 und B_2 nun noch verschmelzen:

1. $A_{2,1} = (9)$ mit $B_{2,1} = (8,12,13)$.
2. $A_{2,2} = (15)$ mit $B_{2,2} = (14,16)$.

Nun enthalten sicherlich alle $A_{i,j}$ und $B_{i,j}$ höchstens $\log n$ Elemente.

Die Komplexität des optimalen Verschmelzungsalgorithmus.

8-22

▶ Satz

Der optimale Verschmelzungsalgorithmus benötigt Zeit $O(\log n)$ und Arbeit $O(n)$.

▶ Folgerung

Das Sortierproblem kann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ gelöst werden.

Zusammenfassung dieses Kapitels

8-23

1. Man kann das *Maximum von n Zahlen* bestimmen in
   - *Zeit $O(\log n)$ und Arbeit $O(n)$* auf einer EREW-PRAM.
   - *Zeit $O(1)$ und Arbeit $O(n^2)$* auf einer CRCW-PRAM.
   - *Zeit $O(\log \log n)$ und Arbeit $O(n)$* auf einer CRCW-PRAM.
2. Accelerated Cascading bedeutet, einen schnellen und einen optimalen Algorithmus zu einem *schnellen, optimalen* Algorithmus zu verschmelzen.
3. Man kann zwei sortierte Listen der Länge n in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmelzen.

Übungen zu diesem Kapitel

Herr Lenz ist Manager bei der *Bunte-Bilder AG*, einer großen Beratungsfirma. Derzeit führt Herr Lenz die Planung mehrerer Projekte durch, die alle am 1. Juni beginnen werden. Jedes Projekt besteht aus mehreren Arbeitspaketen. Für jedes Arbeitspaket hat er sich in einer Tabelle aufgeschrieben, wie viele Tage das Paket vermutlich dauern wird und zu welchem Projekt es gehört:

Paketnummer	1	2	3	4	5	6	7	8	...
Arbeitsaufwand	1d	10d	5d	1d	9d	2d	4d	1d	...
Projekt	IBM	SAP	IBM	IBM	SAP	Bund	SAP	Bund	...

Herr Lenz möchte nun wissen, wann welches Arbeitspaket anfangen kann. Beispielsweise können die ersten beiden Pakete am 1. Juni starten, das dritte kann am 2. Juni starten, das vierte am 7. Juni, das fünfte am 11. Juni, das sechste am 1. Juni und so weiter.

Bei den folgenden Aufgaben geht es nun darum, dieses Problem mit einer PRAM in Zeit $O(\log n)$ und Arbeit $O(n)$ zu lösen, wenn n die Anzahl der Pakete ist und es höchstens $\log n$ unterschiedliche Projekte gibt. Dazu soll im Speicher der PRAM ein Array A mit den Arbeitsaufwänden (also $A[1] = 1$ und $A[2] = 10$) gespeichert sein und ein Array P mit Projektnummern (die Projekte seien also durchnummeriert von 1 bis $\log n$, dabei steht dann 1 für die IBM, 2 für SAP und so weiter).

Übung 8.1 Umwandlung in Vektordarstellung, leicht

Jedes Arbeitspaket kann als Vektor mit $\log n$ Elementen aufgefasst werden, der an Stelle $P[i]$ den Eintrag $A[i]$ hat und ansonsten nur Nullen. Geben Sie einen Algorithmus an, der zu jedem Paket einen solchen Vektor anlegt. Wie groß sind Zeitbedarf und Arbeit Ihres Algorithmus? (Sie dürfen voraussetzen, dass der Speicher mit Nullen initialisiert ist.)

Übung 8.2 Präfixsummen bei Vektoren, mittel

Geben Sie den Code eines Algorithmus an, der auf der Liste der Vektoren die Präfixsummen berechnet. Die Summe zweier Vektoren ist dabei komponentenweise zu verstehen. Der Algorithmus soll eine Laufzeit von $O(\log n)$ besitzen. Beantworten Sie folgende Fragen:

1. Wie groß ist seine Arbeit?
2. Welcher Zusammenhang besteht zwischen den Präfixsummen der Vektoren und den eigentlich gesuchten Anfangszeiten der Pakete?

Übung 8.3 Accelerated-Cascading, schwer

Um die Arbeit des Algorithmus zu verringern, soll nun Accelerated-Cascading verwendet werden. Dazu unterteilen wir die Liste der Pakete in $n/\log n$ Blöcke einer Länge von jeweils $\log n$. Für jeden Block lässt sich nun *ein* Vektor mit $\log n$ Einträgen bestimmen, der die Summe der aller Vektoren des Blockes beinhaltet.

Geben Sie einen Algorithmus an (eine Beschreibung genügt, Code ist aber auch in Ordnung), der zu jedem Block einen solchen Vektor bestimmt und anschließend auf alle diese $n/\log n$ Vektoren den Präfixsummen-Algorithmus aus der vorherigen Aufgabe anwendet. Die Laufzeit soll $O(\log n)$ betragen und die Arbeit $O(n)$.

Welcher Zusammenhang besteht zwischen den hier errechneten Präfixsummen der $n/\log n$ Vektoren und den eigentlich gesuchten Präfixsummen?

Übung 8.4 Bestimmung der fehlenden Werte, schwer

Erweitern Sie den Algorithmus aus der vorherigen Aufgabe, so dass nun die von Herrn Lenz gesuchten Werte in einer Laufzeit von $O(\log n)$ und Arbeit von $O(n)$ berechnet werden.

Übung 8.5 Sortieren von wenigen Werten, mittel

Beschreiben Sie einen PRAM-Algorithmus, der einen Array von n Zahlen, die alle in der Menge $\{1, \ldots, \log n\}$ liegen, in Zeit $O(\log n)$ und Arbeit $O(n)$ sortiert.

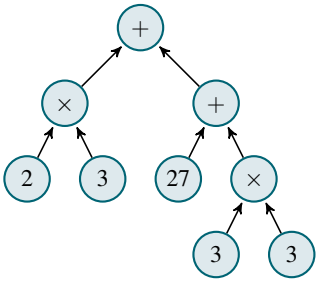
Tipp: Benutzen Sie Übung 8.4, wobei Sie den Arbeitsaufwand immer auf 1 setzen.

Teil III

Parallele Algorithmen

Manche Probleme lassen sich ganz einfach parallelisieren. Schon fast trivial ist das Problem, zwei Vektoren zu addieren: Hier kann man einfach alle Addition unabhängig voneinander und damit parallel durchführen. Eine Spur schwieriger ist das Problem, die Summe von n Zahlen zu berechnen: Hier wird man baumartig vorgehen und zunächst parallel die Summe von jeder Zahl an einer ungerade Position und der neben ihr stehenden Zahl an der nächsthöheren Position; dies verringert die Anzahl der zu addierenden Zahlen um die Hälfte, weshalb man nach $O(\log n)$ Schritten das Ergebnis in seinen virtuellen Händen halten kann.

Einige Probleme sind kniffliger. Das wohl schönste Beispiel ist das Auswerten eines arithmetischen Baumes wie dem folgenden, welcher selbstverständlich zu 42 auswertet:



Wie wertet man einen solchen Baum parallel aus? Wenn er zufälligweise schön ausgeglichen ist, dann kann man sich gut von den Blättern her in Richtung Wurzel vorarbeiten. Bei obigem Baum und noch schlimmer bei den entarteten Bäumen, die aus den Horner-Schema entstehen, geht dies nicht. Es ist zunächst nicht einmal klar, ob sich dieses Problem *überhaupt* parallelisieren lässt.

In diesem Teil werden wir viel Zeit und Energie darauf verwenden, dieses Problem optimal zu parallelisieren. Dazu beginnen wir mit dem im Sequentiellen völlig trivialen Problem, eine Liste mit drei Farben zu färben (wenn Sie nicht sehen, was das auch nur entfernt mit dem Auswerten eines arithmetischen Baumes zu tun hat – warten Sie's ab). Darauf aufbauend gehen wir List-Ranking an, das ebenfalls im Sequentiellen triviale Problem, für jedes Element einer verketteten Liste seine Position in der Liste zu bestimmen. Kombiniert man dann List-Ranking mit parallelen Euler-Touren und erweitert man die Auswertungsbäume um Linearformen so, voilà, hat man den gewünschten Algorithmus.

Wo wir schon beim Rechnen dabei sind, werden wir auch gleich die Grundrechenarten etwas genauer anschauen. Zwei Zahlen zu multiplizieren fällt einem modernen Prozessor nicht schwer, allerdings nur, wenn die Zahlen nur größer als vielleicht 64 Bit sind. Will man zwei zehntausendstellige Zahlen multiplizieren, dann sieht die Sache schon anders aus. Insbesondere ist unklar, wie man das schnell parallel hinbekommt. Noch kniffeliger ist die Division, das Schulverfahren ist offenbar hochgradig sequentiell und man muss tief in die numerische Trickkiste greifen.

9-1

Kapitel 9
Aufbrechen von Symmetrien

Alle Tiere sind gleich, aber manche sind gleicher als andere

9-2

Lernziele dieses Kapitels

1. Problematik der Symmetrie kennen
2. Einfachen 3-Färbealgorithmus kennen
3. Schnellen 3-Färbealgorithmus kennen
4. Optimalen 3-Färbealgorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Heute wird es bunt; wir werden munter Knoten in Kreisen und Listen anpinseln, wobei wir aus haushaltstechnischen Gründen mit den Farben sparsam umgehen müssen. Ziel dabei ist weniger eine besonders ästhetische Bemalung einer Liste als vielmehr das Finden von *unabhängigen Mengen*. Zur Erinnerung: Bei einer *gültigen Färbung* eines Graphen haben durch Kanten verbundene Knoten unterschiedliche Farben. Folglich bilden alle Knoten derselben Farbe eine *unabhängige Menge*, was sie wiederum dazu prädestiniert, unabhängig (sprich: parallel) verarbeitet zu werden.

Das Färben von Graphen ist eine Wissenschaft für sich. Der berühmte Vier-Farben-Satz besagt, dass sich jeder planare Graph mit vier Farben färben lässt. Anderseits wissen Sie auch, dass die Frage, ob sich ein Graph mit drei Farben färben lässt, NP-vollständig ist. Schließlich ist es wiederum leicht, einen Graphen mit zwei Farben zu färben, falls dies möglich ist (durch eine Breitensuche). Wir wollen heute eine besonders einfache Art von Graphen färben: Listen (eine andere Bezeichnung für gerichtete Pfade in diesem Kontext). Offenbar lässt sich jede Liste mit zwei Farben (zum Beispiel »schwarz« und »weiß«, auch wenn dies nicht gerade Paradebeispiele von »Farben« sind) färben, nämlich immer schön abwechselnd.

Die Kunst ist nun aber, Listen parallel zu färben. Da Listen prinzipbedingt »kreuz und quer« im Speicher liegen, gibt es da ein kleines Problem: Sollte das Listenelement an Speicherstelle 234255 nun schwarz oder weiß gefärbt werden? Das Vorgängerelement liegt übrigens an Speicherstelle 1024 und der Nachfolger an Stelle 65500. Offenbar »sieht man einer Speicherstelle nicht an«, ob sie nun schwarz oder weiß werden soll.

Mit etwas List und Tücke werden wir dieses Problem aber lösen: Zunächst schrauben wir unsere Ansprüche herunter und versuchen lediglich, eine Färbung mit einer »kleinen Anzahl« an Farben zu berechnen. Das ist dann ja schonmal etwas. In weiteren Schritten werden wir die Farbanzahl dann immer weiter reduzieren, bis wir nur noch drei Farben brauchen. Damit werden wir es für dieses Kapitel gut sein lassen; um auf zwei Farben zu kommen, werden wir noch eine Menge weiterer List und Tücke brauchen.

9.1 Motivation

Das Problem der Symmetriebrechung. 9-4

Problemstellung

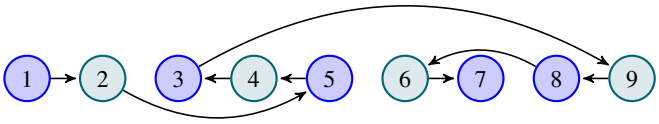
Eingabe Eine durch Zeiger verkettete Liste oder ein Ring von Objekten.

Ausgabe Ein gültige Färbung der Objekte.

Eine *gültige Färbung* bedeutet, dass durch einen Zeiger verbundene Objekte unterschiedlich gefärbt sind.
Offenbar kann man eine Liste immer mit zwei Farben gültig färben, einen Ring immer mit drei Farben.

Eine Liste im Speicher und als Bild. 9-5

Visualisierung einer gültig gefärbten Liste



Speicherung der Liste

Position i	1	2	3	4	5	6	7	8	9
Nachfolger $S[i]$	2	5	9	3	4	7	–	6	8
Farbe $c[i]$	0	1	0	1	0	1	0	0	1

Warum Färbealgorithmen wichtig sind. 9-6

Gleich gefärbte Objekte stehen *nie nebeneinander*. Dies ist unter Umständen beim *Parallelisieren* wichtig: Man erzeugt zuerst eine Färbung, dann bearbeitet man alle gleich gefärbten Objekte gleichzeitig – dabei kommt man sich nicht »ins Gehege«. Man spricht statt von Färben auch von *Symmetriebrechung*: In der ungefärbten Liste sieht lokal »alles gleich aus«, in der gefärbten nicht mehr.
Anwendungen werden wir in den nächsten Wochen kennen lernen.

Das Produkt vieler Matrizen. 9-7

Problemstellung

Eingabe Verkettete Liste der Länge n von (4×4)-Matrizen.

Ausgabe Produkt der Matrizen in der durch die Verkettung gegebenen Reihenfolge.

Sie sollten in der Lage sein, folgende Frage beantworten zu können:

Lernkontrolle
Mit welcher Methode kann man dieses Problem in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ lösen?

Eine Anwendung von Färbungen. 9-8

Ziel
Wir wollen die Arbeit auf $O(n)$ zu drücken.

Idee
Wiederhole Folgendes:

1. Jede zweite Einheit in der Liste berechnet das Produkt seiner Matrix und der Matrix des Nachfolgers.
2. Lösche den Nachfolger aus der Liste.

Problem
Einer Einheit sieht man nicht an, ob sie an einer »ungeraden Stelle« in der Liste auftaucht.

Lösung (grobe Idee)
Wir erstellen eine legale Färbung und bearbeiten dann alle mit einer der Farben gefärbten Matrizen.

9.2 Färbealgorithmen

9.2.1 Einfacher Algorithmus

9-9 Die genaue Problemstellung.

Allgemeine Problemstellung für heute

Eingabe Ein gerichteter Zyklus $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Dieser ist gegeben durch einen Array S, wobei $S[i]$ der Nachfolger von i ist.

Ausgabe Ein gültige Färbung der Objekte, also eine Funktion $c\colon V \to \mathbb{N}$ mit $c[i] \neq c[S[i]]$.

- Unser Ziel ist natürlich, mit möglichst wenigen Farben auszukommen (idealerweise nur 3).
- Offenbar gilt $T^*(n) = O(n)$.

9-10 Ein ganz einfacher Algorithmus.

```
1 for i ∈ {1,...,n} par do
2     c[i] ← i
```

- Dieser Algorithmus liefert eine gültige Färbung.
- Er ist sehr schnell und optimal.
- Er braucht viel zu viele Farben.

9-11 Ein Algorithmus zur Farbreduktion: *ReduceColors*.

```
1 for i ∈ {1,...,n} par do
2     k ← Position des niederwertigsten Bits (Zählung beginnt bei 0), wo sich c[i] und
          c[S[i]] unterscheiden
3     b ← der Wert des k-ten Bits von c[i]
4     c[i] ← 2k + b
```

Beispiel

	Farbe (dezimal)	Farbe (binär)
$c[i]$	36	100100
$c[S[i]]$	8	1000

Wir erhalten $k = 2$ und $b = 1$ und $2k + b = 5$. Die Farbe von $c[i]$ würde also von 36 zu 5 geändert.

9-12 Der Algorithmus erhält gültige Färbungen.

▶ Lemma

Der Algorithmus ReduceColors werde mit einer Liste gestartet, die mit den Farben $0, 1, \ldots, C-1$ *gültig gefärbt ist. Dann ist die Liste hinterher immernoch gültig gefärbt und es werden höchstens* $2\lceil \log_2 C \rceil + 1$ *Farben verwendet.*

Beweis. Die behauptete Anzahl der Farben ergibt sich daraus, dass $2k + b \leq 2\lceil \log_2 C \rceil + 1$ gilt. Für die Korrektheit siehe Übung 9.1. □

9-13 Eine subtile Fragestellung.

✎ Zur Diskussion

Wie lauten $T(n)$ und $W(n)$ bei einer Anwendung von *ReduceColors*?

9-14 Zusammenfassung zum einfachen Färbungsalgorithmus.

▶ Satz

Wenden wir ReduceColors einmal an, so erhalten wir eine gültige $O(\log n)$*-Färbung. Die Rechenzeit ist* $O(1)$ *und die Arbeit ist* $O(n)$.

▶ Satz

Wenden wir ReduceColors zweimal an, so erhalten wir eine gültige $O(\log \log n)$*-Färbung. Die Rechenzeit ist* $O(1)$ *und die Arbeit ist* $O(n)$.

9.2.2 Schneller Algorithmus

Ein schneller, einfacher Färbungsalgorithmus. 9-15

Mit Hilfe das einfachen *ReduceColors*-Algorithmus können wir in Zeit $O(1)$ die Farbenzahl von C auf $2\lceil \log_2 C \rceil + 1$ verringern. Starten wir mit n Farben und wenden wir den Algorithmus

1. Einmal an, so werden noch $O(\log n)$ Farben verwendet.
2. Zweimal an, so werden noch $O(\log\log n)$ Farben verwendet.
3. Dreimal an, so werden noch $O(\log\log\log n)$ Farben verwendet.

Nach $O(\log^*(n))$ Anwendungen hat man nur noch 6 Farben.

▶ **Lemma**

Wenden wir ReduceColors $O(\log^* n)$ mal an, so erhalten wir eine gültige 6-Färbung.
Die Rechenzeit ist $O(\log^* n)$ und die Arbeit ist $O(n \log^* n)$.

Von sechs Farben zu drei Farben. 9-16

- Wir wollen nun noch die Farbenanzahl von sechs auf drei reduzieren.
- Dazu benutzen wird folgenden *KnockOutColor* Algorithmus:

```
1 input Färbung c und spezielle Farbe X > 2
2 for i ∈ {1,...,n} par do
3     if c[i] = X then
4         h ← eine Farbe aus {0,1,2}, die weder der Vorgänger noch der Nachfolger
              von i hat
5         c[i] ← h
```

- Rufen wir nacheinander *KnockOutColor*(3), dann *KnockOutColor*(4) und schließlich *KnockOutColor*(5) auf, so haben wir eine gültige 3-Färbung.

✎ **Zur Übung** 9-17

Machen Sie sich die Funktionsweise von *KnockOutColor* an einem Beispiel klar. Wie groß sind $T(n)$ und $W(n)$?

Zusammenfassung zum schnellen, einfachen Färbungsalgorithmus. 9-18

▶ **Satz**

Wenden wir ReduceColors $O(\log^* n)$ mal an und dann dreimal KnockOutColor, so erhalten wir eine gültige 3-Färbung.
Die Rechenzeit ist $O(\log^* n)$ und die Arbeit ist $O(n \log^* n)$.

9.2.3 Optimaler Algorithmus

Ein optimaler Algorithmus. 9-19

Die Arbeit $O(n \log^* n)$ ist zwar nicht optimal, *aber für alle (!) praktischen Zwecke ist* $\log^* n \leq 6$. Ein optimaler Algorithmus fängt mit den Aufrufen von *KnockOutColor* früher an:

```
1 call ReduceColors()
2 Sortiere alle Knoten nach ihrer Farbe
3 for i ← 3 to 2⌈log n⌉ seq do
4     call KnockOutColor(i)
```

Zusammenfassung zum optimalen Färbungsalgorithmus. 9-21

▶ **Satz**

Wenden wir ReduceColors einmal an, sortieren wir die Knoten nach ihrer Farbe und wenden dann $O(\log n)$ mal KnockOutColor an, so erhalten wir eine gültige 3-Färbung.
Die Rechenzeit ist $O(\log n)$ und die Arbeit ist $O(n)$.

Beweisskizze. Die Korrektheit sollte klar sein. Die Sortierung kann in Zeit $O(\log n)$ und Arbeit $O(n)$ erfolgen, siehe Übung 8.5. Durch die Sortierung ist sichergestellt, dass in jedem Schleifendurchlauf nur so viele Einheiten arbeiten müssen, wie Knoten der aktuellen Farbe vorhanden sind. □

Zusammenfassung dieses Kapitels

9-22

▶ Symmetriebrechung
Symmetriebrechung bedeutet in der Parallelverarbeitung, eine Datenstruktur so vorzuverarbeiten, dass *parallel verarbeitbare Teile* entstehen. Bei *Arrays* ist dies in der Regel trivial (man nehme jedes zweite Element). Bei *verketteten Listen* errechnet man *3-Färbungen*. Bei *allgemeinen Graphen* ist Färben hingegen *sehr schwierig*.

▶ Satz
Man kann einen Ring mit $\log\log n$ Farben in *Zeit* $O(1)$ und *Arbeit* $O(n)$ färben.

▶ Satz
Man kann einen Ring mit *3* Farben in *Zeit* $O(\log^* n)$ und *Arbeit* $O(n \log^* n)$ färben.

▶ Satz
Man kann einen Ring mit *3* Farben in *Zeit* $O(\log n)$ und *Arbeit* $O(n)$ färben.

Übungen zu diesem Kapitel

Übung 9.1 Beweis der Korrektheit von ReduceColors, mittel

Geben Sie einen detaillierten Beweis des Lemmas von Folie 9-12 an.

Tipps: Man führe einen Widerspruchsbeweis: Wären am Ende des Algorithmus $c[i]$ und $c[S[i]]$ gleich, so müssten sie auch schon vorher gleich gewesen sein.

Übung 9.2 Algorithmenanalyse, leicht

Professor Schlauhmayer hat sich eine Beschleunigung des schnellen optimalen 3-Färbealgorithmus von Seite 9.19 ausgedacht. Sein neuer Algorithmus lautet wie folgt:

```
1 ReduceColors()
2 ReduceColors()
3 Sortiere alle Knoten nach ihrer Farbe
4 for i ← 3 to 2⌈log log n⌉ + 4 seq do
5     KnockOutColor(i)
```

Der Professor behauptet, dass dieser Algorithmus eine 3-Färbung in Zeit $O(\log\log n)$ und Arbeit $O(n)$ berechnet. Was stimmt hieran nicht? Begründen Sie Ihre Antwort.

Übung 9.3 Einen Kompaktifizierungsalgorithmus entwerfen, schwer

Ein Programm benutzt einen Memory-Heap zur Speicherung einer Datenstruktur. Der Memory-Heap besteht aus einem Array M von Zellen, die von 1 bis n nummeriert sind. Jede Zelle i enthält entweder den Wert *leer* oder sie enthält ein Tripel bestehend aus einer reellen Zahl R_i sowie zwei Zeigern p_i und q_i. Die Zeiger p_i und q_i enthalten die Nummer einer nichtleeren Memory-Zelle oder den Wert 0.

Sie sollen einen in Zeit $O(\log n)$ und Arbeit $O(n)$ arbeitenden Algorithmus entwerfen, der den Memory-Heap kompaktifiziert. Dies bedeutet, dass nach Ablauf Ihres Algorithmus die nichtleeren Zellen am Anfang des Memory-Heaps stehen und die Werte und Pointer korrekt angepasst wurden. Dazu soll für jeden Index i einer nichtleeren Zelle im alten Memory-Heap der Inhalt dieser Zelle an eine neue Stelle i' kopiert werden und die Zeiger p_i und q_i sollen zu p'_i und q'_i verändert werden.

Hier ein Beispiel: Der Memory-Heap habe anfangs den Inhalt

Zellenummer i:	1	2	3	4	5	6
R_i:	5	π	`leer`	$-3{,}2$	`leer`	100
p_i:	0	1	–	0	–	4
q_i:	6	6	–	0	–	1

Dann könnte er nach der Kompaktifizierung folgenden Inhalt haben:

Zellenummer i:	1	2	3	4	5	6
R_i:	5	π	$-3{,}2$	100	`leer`	`leer`
p_i:	0	1	0	3		
q_i:	4	4	0	1		

Tipp: Sie dürfen Hilfsarrays verwenden. Führen Sie eine Präfixsumme über ein Array durch, der eine 1 für jede nichtleere Zelle hat.

Übung 9.4 Färbungsalgorithmus implementieren, mittel

Implementieren Sie den schnellen parallelen Färbungsalgorithmus mit OpenMP. Zur Erinnerung: Bei Eingabe einer Liste mit n Elementen produziert der Algorithmus zuerst eine Färbung mit n Farben. Danach wird die Anzahl der Farben durch $(\log^* n)$-faches Anwenden der Methode *ReduceColors* auf eine konstante Anzahl von Farben verringert. Zum Abschluss wird mithilfe des *KnockOutColor* Algorithmus eine 3-Färbung berechnet.

10-1

Kapitel 10
List-Ranking-Algorithmen

1, 2, 3, . . .

10-2

Lernziele dieses Kapitels

1. Problematik des List-Rankings kennen und anwenden können
2. Einfachen optimalen List-Ranking Algorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Die verkettete Liste gehört unzweifelhaft zu den grundlegendsten Datenstrukturen der Informatik. Man begegnet ihr in solidem, äußerst bodenständigem Code einer industriellen Oberflächenprogrammierung, in den tiefsten Eingeweiden des Betriebssystem innerhalb des guru-gehacktem Assemblercodes des Scheduler, aber auch in der hohen Theorie der reinen Lehre der funktionalen Typtheorie.

Eines der Wesensmerkmale einer Liste ist, dass ihre Elemente in einer *Reihenfolge* stehen. Es gibt eben ein erstes Element, ein zweites, ein drittes und so weiter; und diese Reihenfolge ist im Allgmeinen äußerst wichtig. Nun die »einfache« Problemstellung: Gegeben eine Liste im Speicher, bestimme für jedes Element der Liste, das wievielte es ist (dies nennt man auch den *Rang* des Listenelements). Dieses Problem ist sequentiell sehr einfach mit einem Einzeiler zu lösen (etwa `while(c){c.rank=i++; c=c.next;}`). Jedoch ist diese Schleife »sehr« sequentiell und man wird sich zum Parallelisieren etwas Neues ausdenken müssen.

Wenn man im Parallelen nicht mehr weiter weiß, dann helfen bekanntlich immer Präfixsumme oder Pointer-Jumping. So auch hier: Eine Liste ist ja ein entarteter Baum und folglich kann man auf sie Pointer-Jumping anwenden. Damit lässt sich List-Ranking in Zeit $O(\log n)$ bewerkstelligen, jedoch mit einer Arbeit von $O(n \log n)$. Dies ist alles andere als optimal. In der heutigen Vorlesung soll es darum gehen, diese Arbeit zu »drücken«. Dazu kombinieren wir Pointer-Jumping und die Färbealgorithmen aus dem letzten Kapitel mittels Accelerated-Cascading.

Diese Kombination von Algorithmen ist schon eher komplex, obwohl es sich um »einfache« List-Ranking-Algorithmen handelt. Die Algorithmen sind auch schon ziemlich gut und schnell, aber eben nicht perfekt. Um auch noch das letzte Quäntchen Geschwindigkeit herauszupressen, müssen wir in nächsten Kapitel noch *wesentlich* tiefer in die Trickkiste greifen.

10.1 Motivation

10.1.1 Ein Anwendungsbeispiel

Wiederholung: Das Produkt vieler Matrizen. 10-4

Problemstellung

Eingabe Verkette Liste der Länge n von (4×4)-Matrizen.

Ausgabe Produkt der Matrizen in der durch die Verkettung gegebenen Reihenfolge.

Lösungsidee 1

Berechne das Produkt durch Pointer-Jumping. Die Rechenzeit ist dabei $O(\log n)$, die Arbeit ist $O(n \log n)$.

Lösungsidee 2

- Bestimmen für jede Matrix ihren Rang in der Liste.
- Schreibe jede Matrix in einen Array an die Stelle ihres Rangs.
- Bilde das Produkt der Elemente im Array (oder sogar Präfixsumme).

Falls wir den ersten Schritt in Zeit $O(\log n)$ und Arbeit $O(n)$ hinbekommen, so auch die restlichen Schritte.

10.1.2 Das Ranking-Problem

Das Ranking-Problem 10-5

▶ **Definition**

<only@1>

Eingabe Verkettete Liste der Länge n, gegeben durch die Nachfolger $S[i]$ und Vorgänger $P[i]$ für jeden Knoten in der Liste.

Ausgabe Abstand $R[i]$ des Knotens vom Ende der Liste (der Rang ist dann $n - R[i]$).

Ziele

1. Ein Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n \log n)$. (Einfach.)
2. Ein Algorithmus mit Zeit $O(\log^2 n)$ und Arbeit $O(n)$. (Mittels Färbungen.)
3. Ein Algorithmus mit Zeit $O(\log n \log\log n)$ und Arbeit $O(n)$. (Mittels Färbungen und Accelerated-Cascading.)
4. Ein Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n)$. (Mittels Färbungen, Accelerated-Cascading, Pipelining, amortisierter Analyse, Potentialfunktionen, ...)

10.2 Einfaches Ranking

Ein einfacher Ranking-Algorithmus: Pointer-Jumping. 10-6

```
1 for i ∈ {1,...,n} par do
2     if S[i] ≠ i then
3         R[i] ← 1
4     else
5         R[i] ← 0
6 for i ∈ {1,...,n} par do
7     while S[i] ≠ i
8         R[i] ← R[i] + R[S[i]]
9         S[i] ← S[S[i]]
```

Merke

Der Wert $R[i]$ speichert, wie weit $S[i]$ entfernt ist.

▶ **Satz**

Durch Pointer-Jumping kann man den Rang aller Elemente in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ bestimmen.

10.3 Optimales Ranking

10.3.1 Idee

10-7

Die Ideen hinter einem optimalen Ranking-Algorithmus.

Wiederhole Folgendes:

1. Streiche *ein Viertel der Elemente der Liste*. Diese werden später wieder eingefügt.
2. Zur Identifizierung der streichbaren Elemente benutzen wird *Färbungen*.

10.3.2 Ausklinken eines Elementes

10-8

Wir wollen ein Element loswerden.

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt, und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Ein spezielles Element x ist gegeben.

Ziel:

- Das Element x soll aus der Liste temporär entfernt werden.
- Dazu soll der Vorgänger von x auf den Nachfolger von x zeigen und sein R-Wert soll entsprechend angepasst werden.

10-9

Die Algorithmen zum Ausklinken.

Algorithmus *Suspend*(x).

```
1 Speichere die Werte S[x], P[x] und R[x] lokal am Knoten x
2 S[P[x]] ← S[x]
3 P[S[x]] ← P[x]
4 R[P[x]] ← R[P[x]] + R[x]
```

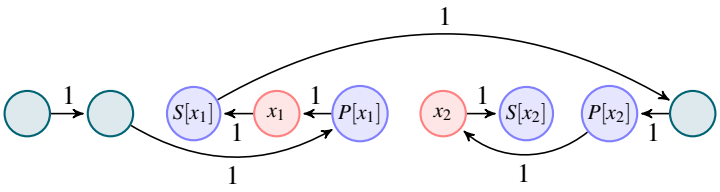
Algorithmus *Reinstall*(x).

```
1 Seien S_orig, P_orig und R_orig die gespeicherten Werte am Knoten x
2 R[x] ← R[S_orig] + R_orig
3 S[x] ← S[S_orig]
```

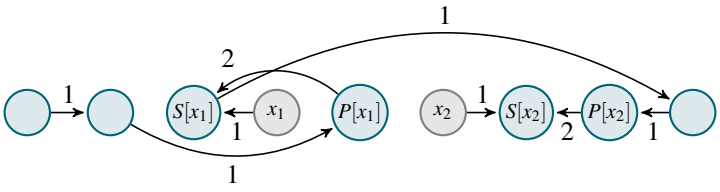
10-10

Beispiel eines Suspends.

Vor dem Suspend von x_1 und x_2



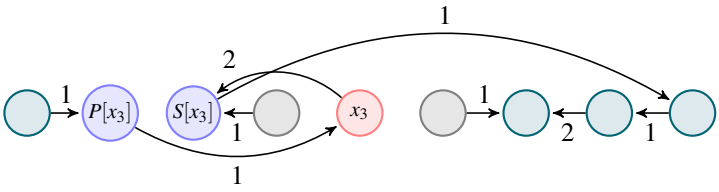
Nach dem Suspend von x_1 und x_2



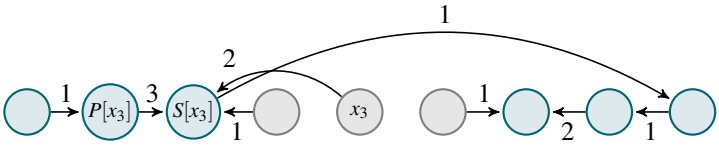
Beispiel eines weiteren Suspends.

10-11

Vor dem Suspend von x_3



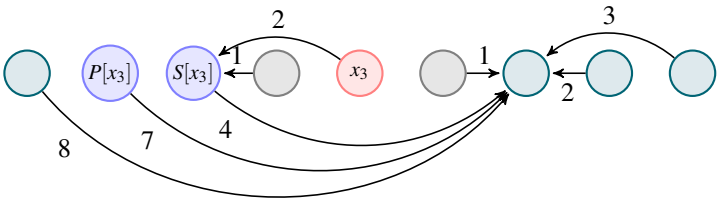
Nach dem Suspend von x_3



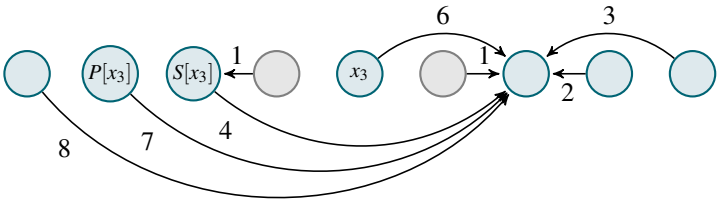
Beispiel eines Reinstall von x_3.

10-12

Die Liste vor einem Reinstall einige Schritte später



Nach dem Reinstall von x_3.



10.3.3 Unabhängige Mengen

Wir wollen viele Elemente loswerden.

10-13

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt, und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Eine Menge X von Elementen.

Ziel:

- Alle Elemente in X sollen parallel aus der Liste temporär entfernt werden.

Wann können wir Elemente parallel ausklinken?

10-14

- Wir können nicht zwei benachbarte Elemente gleichzeitig ausklinken.
- Deshalb muss X eine *unabhängige Menge* sein. Das heißt, kein Element in X ist Nachfolger eines Elementes in X.
- Damit stellt sich das Problem, eine möglichst große unabhängige Menge zu bestimmen.

10-15

Wie bestimmt man unabhängige Mengen?

Idee 1

Man bestimmt eine Färbung. Dann bilden alle Elemente mit der gleichen Farbe eine unabhängige Menge.

Problem

Selbst bei einer 3-Färbung können wir nicht sicher sein, dass eine konkrete Farbe oft vorkommt.

Beispiel

Wir bestimmen eine 3-Färbung und wollen dann alle, sagen wir, *roten* Elemente ausklinken. Dann könnte es passieren, dass es *gar keine roten Elemente gibt*.

10-16

Der Trick mit den lokalen Minima.

▶ **Definition**

Ein Knoten heißt *lokales Minimum*, wenn die Nummer seiner Farbe kleiner ist als die Farben beider seiner Nachbarn.

▶ **Satz**

Die lokalen Minima einer Färbung von n Knoten mit k Farben bilden eine unabhängige Menge der Größe mindestens $n/(2k-2)-1$.

Beweis. Übungsaufgabe 10.4. □

10.3.4 Algorithmus

10-17

Der Algorithmus Repeated-Suspend-Minima

Reduktionsteil

```
1  n_0 ← n
2  k ← 0
3  while n_k > 1 do
4      Färbe die Liste mit 3 Farben.
5      for i ∈ {1,...,n_k} par do
6          if i ist lokales Farbminimum then
7              call Suspend(i)
8      Kompaktifiziere die Liste
9      k ← k + 1
10     n_k ← Länge der kompaktifizierten Liste
```

10-18

Der Algorithmus Repeated-Suspend-Minima

Expansionsteil

```
1  for j ← k downto 1 seq do
2      Mache die j-te Kompaktifizierungen rückgängig
3      for i ∈ {1,...,n_k} par do
4          if i wurde bei diesem Schritt ausgeklinkt then
5              call Reinstall(i)
```

Performance des Algorithmus

10-19

▶ Satz
Der Algorithmus berechnet ein Ranking in Zeit $O(\log^2 n)$ und Arbeit $O(n)$.

Beweis. Die Arbeit ist $O(n)$:
- Die Arbeit in jedem While-Schleifendurchlauf ist $O(n_k)$.
- In jedem While-Schleifendurchlauf sinkt die Größe n_k um mindestens ein Viertel.
- Insbesondere sinkt die Größe in drei Durchläufen um mehr als die Hälfte.

Die Zeit ist $O(\log^2 n)$:
- Jeder While-Schleifendurchlauf dauert $O(\log n_k) \subseteq O(\log n)$.
- Es gibt nur $O(\log n)$ Schleifendurchläufe. □

10.4 Accelerated-Cascading

10.4.1 Idee

Die Idee hinter einem schnelleren optimalen Ranking-Algorithmus.

10-20

Wir haben:
1. Einen schnellen, nicht optimalen List-Ranking-Algorithmus (Pointer-Jumping).
2. Einen langsamen, optimalen List-Ranking-Algorithmus (Repeated-Suspend-Minima).

Nun bietet sich *Accelerated-Cascading* an: Die Arbeit von Pointer-Jumping »ist tolerabel«, wenn die Liste nur Länge $n/\log n$ hat. Dann ist die Arbeit $\Theta\left(\frac{n}{\log n} \log \frac{n}{\log n}\right) = \Theta(n)$. Wir hören also mit Repeated-Suspend-Minima auf, sobald diese Listenlänge erreicht ist.

10.4.2 Algorithmus

Der Algorithmus mit Accelerated-Cascading.

10-21

```
1  n_0 ← n
2  k ← 0
3  while n_k > n/log_2 n do
4      Färbe die Liste mit 3 Farben.
5      for i ∈ {1,...,n_k} par do
6          if i ist lokales Farbminimum then
7              call Suspend(i)
8      Kompaktifiziere die Liste
9      k ← k+1
10     n_k ← Länge der kompaktifizierten Liste
11 call PointerJump() auf die Restliste
12 for j ← k downto 1 seq do
13     Mache die j-te Kompaktifizierungen rückgängig
14     for i ∈ {1,...,n_k} par do
15         if i wurde bei diesem Schritt ausgeklinkt then
16             call Reinstall(i)
```

Performance des Algorithmus

10-22

▶ Satz
Der Algorithmus berechnet ein Ranking in Zeit $O(\log n \log\log n)$ und Arbeit $O(n)$.

Beweis. Die Arbeit ist $O(n)$:
- Der Beitrag von Repeated-Suspend-Minima ist weiterhin $O(n)$.
- Das Pointer-Jumping wird auf $n/\log n$ Elemente angewandt und macht somit $O\left(\frac{n}{\log n} \log \frac{n}{\log n}\right) = O(n)$ Arbeit.

Die Zeit ist $O(\log n \log\log n)$:
- Jeder While-Schleifendurchlauf dauert wieder $O(\log n)$.
- Nach $3\log\log n$ Runden gibt es höchstens $n/2^{\log\log n} = n/\log n$ Knoten. □

Zusammenfassung dieses Kapitels

10-23

▶ Invariante der List-Ranking-Algorithmen
Der Wert $R[i]$ gibt immer an, wie weit $S[i]$ in der Originalliste entfernt war.

▶ Repeated-Suspend-Minima-Algorithmus
Man klinkt so lange Knoten aus, die bestimmt wurden als lokale Minima von Färbungen, bis nurnoch ein Knoten übrig ist. Dann macht man alles wieder rückgängig.

▶ Zeit und Arbeit von List-Ranking-Algorithmen
Das List-Ranking-Problem kann gelöst werden:

1. In Zeit $O(\log n)$ und Arbeit $O(n \log n)$ mittels Pointer-Jumping.
2. In Zeit $O(\log^2 n)$ und Arbeit $O(n)$ mittels Repeated-Suspend-Minima.
3. In Zeit $O(\log n \log\log n)$ und Arbeit $O(n)$ mittels Accelerated-Cascading der obigen beiden Algorithmen.

Übungen zu diesem Kapitel

Übung 10.1 Eingabeformate ändern, leicht

Im Speicher einer PRAM liegt eine einfachverkettete Liste mit n Elementen vor, das heißt es gibt für jedes Element i einen Zeiger $S(i)$ auf den Nachfolger von i.

Geben Sie jeweils einen Algorithmus für folgende Probleme an, die jeweils in Zeit $O(1)$ und Arbeit $O(n)$ arbeiten (oder besser):

1. Umwandlung der Eingabe in eine doppeltverkette Liste. Dazu muss zu jedem Element i der Vorgängen $P(i)$ bestimmte werden.
2. Umwandlung der Liste in einen Ring, indem das erste Element zum Nachfolger des letzten Elements gemacht wird.

Übung 10.2 Modifikation der Suspend-Methode, leicht

Der Algorithmus *suspend*(x) setzt voraus, dass x weder das erste noch das letzte Element der Liste ist. Modifizieren Sie den Algorithmus so, dass er auch in diesen beiden Fällen korrekt arbeitet.

Wir vereinbaren, dass der Vorgängerzeiger des ersten Elements sowie der Nachfolgerzeiger des letzten Elements auf das jeweilige Element selbst zeigt. Weiterhin gibt es eine Variable *anfang*, die einen Zeiger auf das erste Element der Liste enthält. Stellen Sie sicher, dass das auch nach Ausführung Ihres Algorithmus noch der Fall ist.

Übung 10.3 Beweis des einfachen Lokale-Minima-Lemmas, mittel

Beweisen Sie: Eine Liste der Länge n sei mit k Farben gültig gefärbt. Dann bilden die lokalen Minima der Farben eine unabhängige Menge der Größe mindestens $(n/2k) - 1$.

Tipp: Zeigen Sie, dass je $2k$ aufeinander folgende Listenelemente mindestens ein lokales Minimum enthalten müssen. Die -1 wird für den Rand gebraucht.

Übung 10.4 Beweis des Lokale-Minima-Lemmas, mittel

Beweisen Sie: Eine Liste der Länge n sei mit k Farben gültig gefärbt. Dann bilden die lokalen Minima der Farben eine unabhängige Menge der Größe mindestens $n/(2k-2) - 1$.

Tipp: Gehen Sie ähnlich vor wie in Übung 10.3. Zeigen Sie, dass je $2k-2$ aufeinander folgende Listenelemente die Farbe 0 enthalten müssen.

Übung 10.5 Schneller Algorithmus für Erreichbarkeit in Mengen von Kreisen, mittel

Als Eingabe sei ein Graph gegeben, der die Vereinigung von disjunkten gerichteten Kreisen ist, sowie zwei Knoten s und t. Wie üblich ist der Graph durch ein Array S gegeben, wobei $S(i)$ der Nachfolgeknoten von i ist. Geben Sie einen Algorithmus an, der entscheidet, ob s und t im selben Kreis liegen. Der Algorithmus soll in Zeit $O(\log n)$ arbeiten, die Arbeit ist egal.

Tipp: Machen Sie t zum Nachfolger von sich selbst und führen Sie dann Pointer-Jumping durch.

Übung 10.6 Arbeitsoptimalen Algorithmus finden, schwer

Geben sie einen Algorithmus für dasselbe Problem wie in der vorigen Aufgabe an mit einer Laufzeit von $O(\log n \log\log n)$ und Arbeit $O(n)$. Es genügt eine grobe Beschreibung des Algorithmus und jeweils ein kurzes Argument, weshalb die Arbeit und die Laufzeit die behaupteten sind.

11-1

Kapitel 11
Schneller optimaler List-Ranking-Algorithmus

Wenn es unbedingt schnell gehen muss

Lernziele dieses Kapitels

1. Schnellen optimalen List-Ranking Algorithmus grob kennen
2. Analyse des Algorithmus grob verstehen

11-2

Inhalte dieses Kapitels

Worum es heute geht

Dieses Kapitel ist nichts für Leute mit schwachen Nerven und dies gleich aus mehreren Gründen:

1. Es geht recht blutig zu: süße kleine Schafe werden von bösen roten Wölfen gefressen. (In alten Fassungen dieser Vorlesung gab es statt der »Wölfe« etwas tierlieber »Hirten«. Aber auch bei den Hirten hatte nach jeder Runde jeder Hirte ein Schaf weniger, was nie zufriedenstellend begründet werden konnte.)
2. Es geht algorithmisch komplex zu: der Algorithmus besteht aus mehreren verwobenen Ideen, wobei wir alles, was wir bisher gelernt haben, in verschiedener Weise einsetzen werden.
3. Es wird analysetechnisch schwierig: Es ist schon nicht offensichtlich, dass der Algorithmus korrekt arbeitet – aber man kann sich mit etwas gutem Willen davon überzeugen. Richtig schwierig ist es zu zeigen, dass der Algorithmus tatsächlich auch arbeitsoptimal ist. Wir werden dazu eine amortisierte Analyse durchführen müssen, bei der zu allem Überfluss auch noch die so genannte *Kostenverteilung* nicht schön gleichmäßig ist, sondern eher die Komplexität des Bundeshaushalts erreicht.

Nehmen wir an, Sie haben starke Nerven. Was bringt dann alle die Mühe? Sie werden sich an einem *arbeitsoptimalen* List-Ranking-Algorithmus mit einer Laufzeit von $O(\log n)$ erfreuen können. Ein solcher Algorithmus ist in der Tat eine erfreuliche Sache: Ab sofort können Sie in Ihren Algorithmen nicht nur schreiben »… und dann einmal Präfixsumme anwenden, was weder die Arbeit noch die Zeit erhöht, und dann…«, sondern nun auch »… und dann einmal List-Ranking anwenden, was weder die Arbeit noch die Zeit erhöht, und dann…«.

11.1 Das Ziel

11-5

Die Problemstellung und was wir schon wissen.

Eingabe Verkettete Liste der Länge n, gegeben durch die Nachfolger $S[i]$ und Vorgänger $P[i]$ für jeden Knoten in der Liste.

Ausgabe Abstand $R[i]$ des Knotens vom Ende der Liste (der Rang ist dann $n - R[i]$).

- Lösbar in *Zeit* $O(\log n)$ und *Arbeit* $O(n \log n)$ (Pointer-Jumping).
- Lösbar in *Zeit* $O(\log n \log\log n)$ und *Arbeit* $O(n)$ (iteriertes Färben und Ausklinken).

Ziel

Algorithmus mit *Zeit* $O(\log n)$ und *Arbeit* $O(n)$.

11.2 Die Lösung

11.2.1 Idee

11-6

Die grobe Idee.

Wir beginnen ähnlich wie beim letzten Mal:

- Wir führen eine *Vorverarbeitung* durch.
- Dabei wird die Liste von n Knoten auf $n/\log n$ *Knoten geschrumpft*.
- Dazu werden wiederholt *unabhängige Mengen* von Knoten »ausgeklinkt«.
- Sobald die Länge der Liste auf $n/\log n$ geschrumpft ist, können wir Pointer-Jumping durchführen und die Knoten wieder einfügen.

Neu ist nun:

- Wir *garantieren*, dass die Liste schnell schrumpft.
- Dazu klinken wir Knoten nicht »irgendwo« aus, sondern etwas kontrollierter.

11-7

Zur Erinnerung: Ausklinken.

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Ein spezielles Element x ist gegeben.

Ziel:

- Das Element x soll aus der Liste temporär entfernt werden.
- Dazu soll der Vorgänger von x auf den Nachfolger von x zeigen und sein R-Wert soll entsprechend angepasst werden.
- Da wir nicht zwei benachbarte Elemente gleichzeitig ausklinken können, muss die Menge X der ausgeklinkten Elemente eine *unabhängige Menge* sein.

11.2.2 Die Blockbildung

11-8

Wir teilen die Liste in Blöcke ein.

Idee der Blockbildung

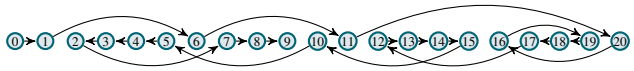
- Um mehr Kontrolle über das Ausklinken zu erhalten, teilen wir die Liste in Blöcke B_j der Länge $\log n$ auf.
- Diese Aufteilung erfolgt in Bezug auf die Reihenfolge, wie die Daten im Speicher stehen, nicht in Bezug auf die Listenreihenfolge.
- Die Position eines Knotens innerhalb eines Blocks nennen wir seine *Höhe*.

Grobes Ziel

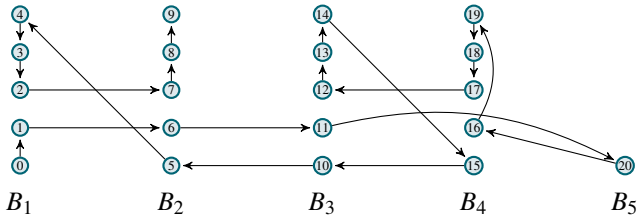
- Wir wollen (idealerweise) in jedem Schritt aus jedem Block ein Element ausklinken.
- Dies soll in Zeit $O(1)$ und Arbeit $O(n/\log n)$ geschehen.
- Dann sind nach $O(\log n)$ Schritten nur noch $O(n/\log n)$ Knoten vorhanden.

Visualisierung der Ausgangssituation. 11-9

Die Liste im Speicher



Die Liste in Blöcke der Größe $\lceil \log_2 20 \rceil = 5$ aufgeteilt.



Der Fokus in den Blöcke. 11-10

▶ Definition

- Für jeden Block B_j bezeichnet f_j den aktuellen *Fokus*.
- Der Fokus wir in der Regel in jedem Schritt hochgesetzt, er kann aber auch gleich bleiben.
- Wenn der Block leer ist, so ist der Fokus nicht definiert.

11.2.3 Knotenzustände

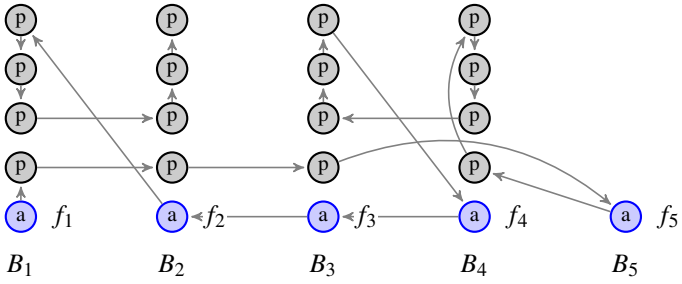
Woran gerade gearbeitet wird. 11-11

Jeder Knoten hat immer genau einen der folgenden Zustände:

active	Gerade fokussierter Knoten, den man am liebsten loswerden würde (blau).
passive	Kommt erst später dran (schwarz).
suspended	Schon ausgeklinkt. Wird nicht mehr beachtet. (grau).
isolated	Isolierter Knoten (orange).
wolf	Ein Wolfsknoten (rot).
sheep	Ein Schafsknoten (grün).

Da Knoten heute recht komplexe Gebilde sind, werden wir ab jetzt die *Objektnotation* verwenden und $i.successor$ statt $S[i]$ schreiben und $i.state$ für den Zustand und so weiter.

Visualisierung der Ausgangssituation 11-12



Ideen, wie wir Knoten loswerden. 11-13

Idee

1. Wir wollen in jeder Runde möglichst alle aktive Knoten loswerden (durch Ausklinken).
2. Danach wollen wir die Fokusse um 1 erhöhen und wiederholen dann Schritt 1.

Zur Diskussion

1. Warum können wir nicht einfach in jeder Runde alle aktiven Knoten ausklinken?
2. Was könnte man stattdessen tun?

11.2.4 Wölfe und Schafe

11-14

Wie finden wir Knoten zum Ausklinken?

Ziel und Problem

Ziel Wir wollen in jeder Runde möglichst viele aktiven Knoten loswerden (durch Ausklinken).

Problem Diese Knoten bilden im Allgemeinen keine unabhängige Menge.

Idee 1

- Wir ermitteln eine 3-Färbung der Knoten.
- Dies induziert (zum Beispiel durch die Farbminima) eine unabhängige Menge.
- Diese können wir ausklinken.

Problem: Die Rechenzeit von $O(\log n)$ ist viel zu hoch, denn wir haben nur $O(1)$ Zeit.

11-15

Eine neue Idee, um schneller Knoten zum Ausklinken zu finden.

Idee 2

- Wir ermitteln in Zeit $O(1)$ eine $\log\log n$-Färbung der Knoten (zweimal *ReduceColors*).
- Dies induziert (zum Beispiel durch die Farbminima) eine unabhängige Menge.
- Diese können wir ausklinken.

Vorteil: Die Rechenzeit ist nur $O(1)$.
Problem: Wir entfernen nur einen Anteil von $O(1/\log\log n)$ Knoten.

11-16

Wie man mehr Knoten los wird.

Idee 3

- Nehmen wir an, wir haben eine unabhängige Menge ermittelt.
- Es ist aber nur jeder $(\log\log n)$-te Knoten in der Menge.
- Dann bilden aber die *Nachfolger* dieser Knoten wieder eine unabhängige Menge. Diese können wir also danach entfernen.
- Und ebenso deren Nachfolger. Diese können wir dann danach entfernen.

Vorteil: Wenn alles gut läuft, dann kann in jedem Schritt jeder $(\log\log n)$-te Knoten ausgeklinkt werden.
Problem: Spätestens nach $\log\log n$ Runden bilden die Nachfolger keine unabhängige Menge mehr.

11-17

Mit Pipelining zum Sieg.

Idee 4

- Die vorherige Idee war gut, auch wenn sie nicht ganz funktionierte.
- Das Problem ist, dass gegen Ende viele Knoten nichts mehr tun, da sie schon ausgeklinkt wurden.
- Deshalb benutzen wir *Pipelining*, um diesen Knoten Arbeit zu verschaffen.

Problem: Jetzt wird es unübersichtlich.

11-18

Einteilung in Wölfe und Schafe.

Wie wir die aktiven Knoten aufteilen

In jedem Schritt wechseln die *aktiven Knoten* ihren Zustand in einen der folgenden:

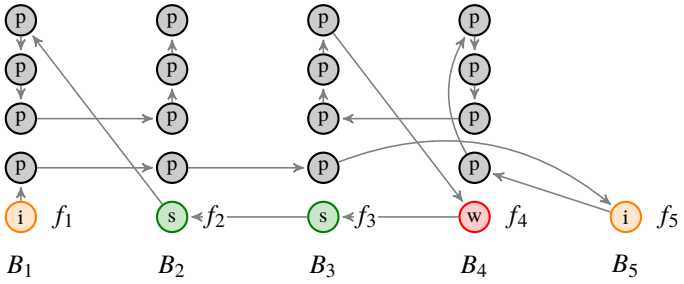
isolated Knoten, bei denen weder der Vorgänger noch der Nachfolger aktiv ist.

wolf Lokale Farbminima einer $(\log\log n)$-Färbung der aktiven Knoten.

sheep Alle anderen aktiven Knoten.

Einteilung der Knoten in isolierte Knoten, Schafe und Wölfe

11-19



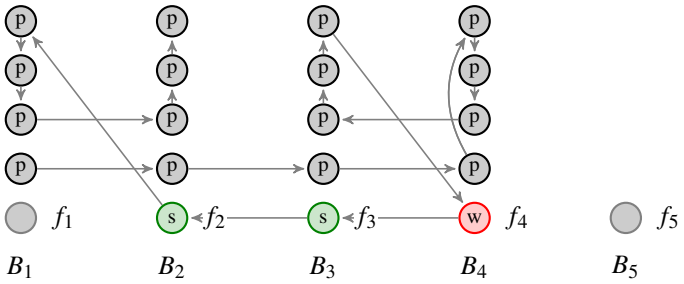
Beobachtungen zu Wölfen und Schafen.

11-20

- Wir können die Einteilung in isolierte Knoten, Wölfe und Schafe in Zeit $O(1)$ durchführen.
- Wir mögen isolierte Knoten, da man sie sofort ausklinken kann.
- Wölfe bilden eine unabhängige Menge. Deshalb bilden auch die den Wölfen direkt folgenden Schafe eine unabhängige Menge.

Die isolierten Knoten werden suspendiert.

11-21



Was Wölfe und Schafe tun.

11-22

Was Wölfe tun:

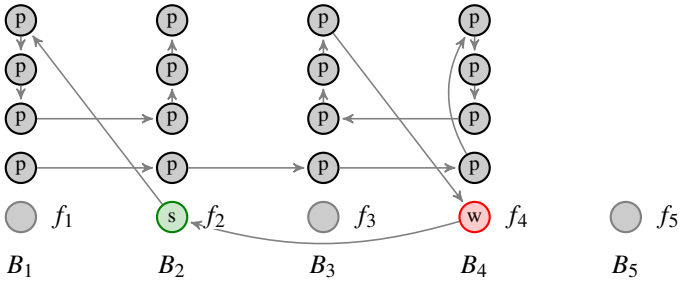
- In jeder Runde *fressen* die Wölfe die Schafe, die ihnen direkt folgen.
- Hat ein Wolf alle seine Schafe gefressen, ist er kein Wolf mehr und wechselt in den Zustand aktiv zurück.
- Er kann dann als isolierter Knoten oder als Schaf oder als Wolf wiedergeboren werden.
- Wölfe behalten den Fokus.

Was Schafe tun:

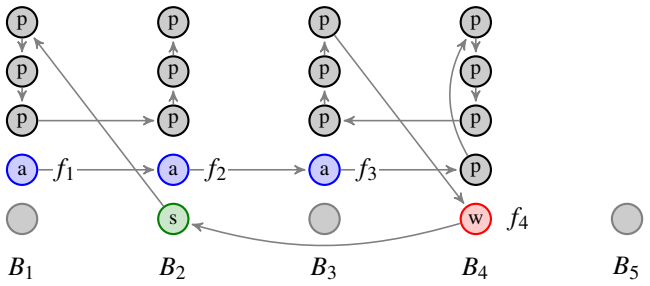
- Schafe warten geduldig darauf, gefressen zu werden.
- Schafe behalten *nicht* den Fokus, dieser wird einfach hochgeschoben.

Wölfe fressen ein Schaf durch Suspendierung.

11-23



11-24 Die Fokusse rücken vor und Knoten werden aktiv.



11.2.5 Algorithmus

11-25 Der ganze Algorithmus.

- Der Algorithmus verläuft in Runden.
- In jeder Runde geschehen folgende Dinge nacheinander:
  1. Die aktiven Knoten werden in isolierte Knoten, Wölfe und Schafe eingeteilt.
  2. Isolierte Knoten werden entfernt und Wölfe fressen ihre Schafe. Dadurch verlieren sie vielleicht ihren Status als Wolf und werden wieder aktive Knoten.
  3. Der Fokus wird aktualisiert: Bei Schafen und suspendierten Knoten wird der Fokus erhöht und neu fokussierte Knoten wechseln von passiv auf aktiv.
- Nach $O(\log n)$ Runden stoppen wir und wenden Pointer-Jumping an.

11-26 Die Einteilung im Detail.

```
1  call ReduceColors () für die aktiven Knoten
2  call ReduceColors () für die aktiven Knoten
3  for alle Blocknummern j par do // das heißt j ∈ {1,...,n/log n}
4      if f_j.state = active und f_j ist isoliert then
5          f_j.state ← isolated
6      else if f_j.state = active und f_j.color ist Farbminimum then
7          f_j.state ← wolf
8      else
9          f_j.state ← sheep
```

(Tatsächlich machen wir auch den ersten und letzten Knoten einer Kette zum Wolf und den zweiten Knoten einer Kette zum Schaf.)

11-27 Das große Fressen im Detail.

```
1  // Fressen
2  for alle Blocknummern j par do
3      if f_j.state = wolf
4          s ← f_j.successor
5          call Suspend(s)
6          if s ist letztes Schaf then
7              f_j.state ← active
8
9  // Aufräumen isolierter Knoten
10 for alle Blocknummern j par do
11     if f_j.state = isolated then
12         call Suspend(f_j)
```

Vorrücken des Fokus.

11-28

```
1 for alle Blocknummern j par do
2     if f_j.state ∈ {suspended, sheep} then
3         // Kann weitergehen
4         f_j ← f_j + 1
5         if f_j ist über die Blockgrenze hinausgelaufen then
6             ignoriere f_j und Block j im Folgenden
7         else
8             f_j.state ← active
```

Zusammenfassende Beobachtungen zu den Zuständen.

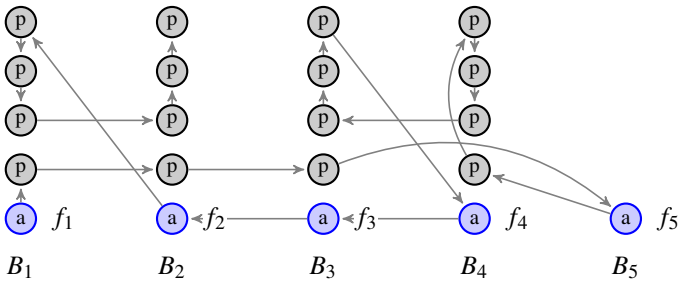
11-29

- Jeder Knoten ist immer in einem der Zustände »suspended«, »active«, »passive«, »isolated«, »wolf« oder »sheep«.
- Der Fokus liegt am Anfang einer Runde immer auf Wölfen und auf aktiven Knoten.
- *Unterhalb* des Fokus finden sich in einem Block nur Schafe und ausgeklinkte Knoten.
- *Oberhalb* des Fokus finden sich in einem Block nur passive Knoten.

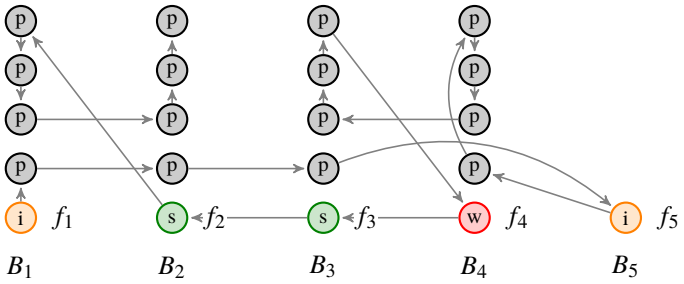
Der Algorithmus in Aktion.

11-30

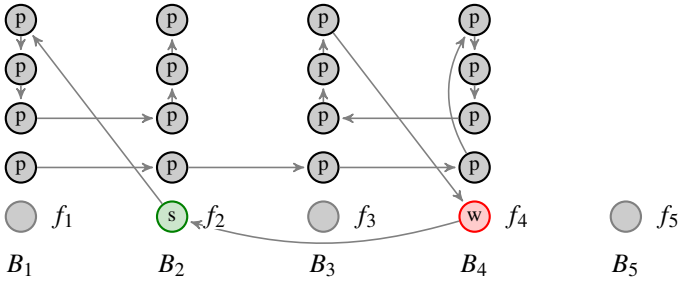
Ausgangssituation:



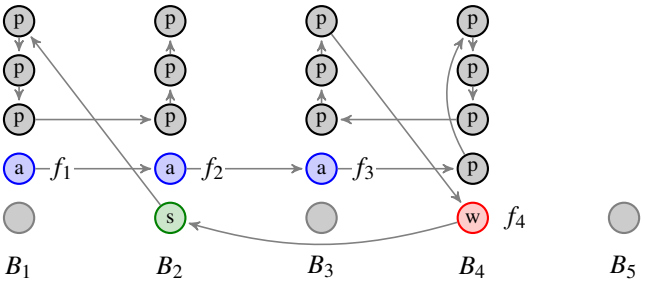
...Einteilung der aktiven Knoten in isolierte Knoten, Schafe und Wölfe...



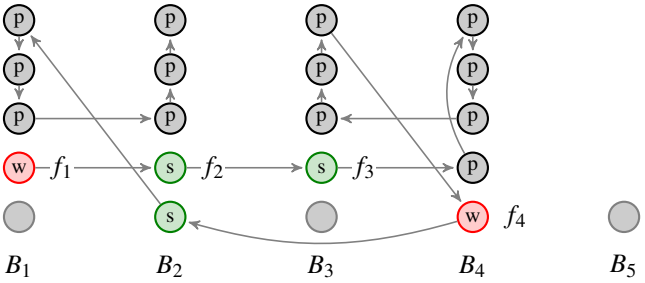
... Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert...



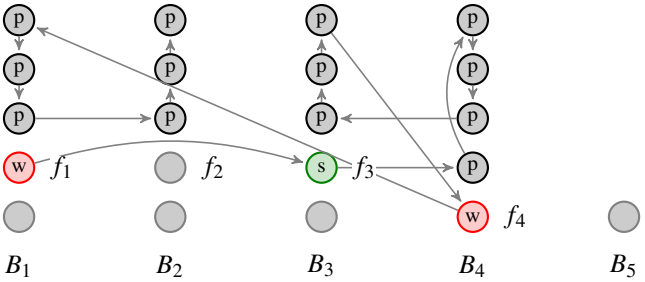
...Die Fokusse werden vorgerückt und Knoten aktiv gemacht ...



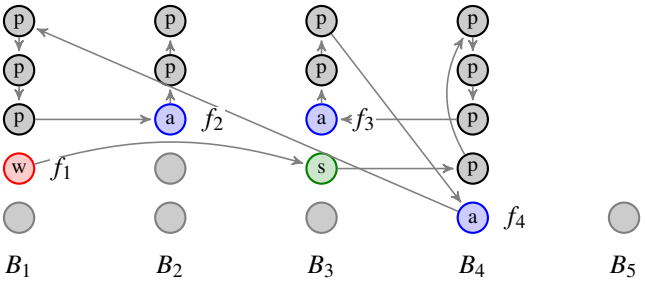
...Einteilung der aktiven Knoten in isolierte Knoten, Schafe und Wölfe...



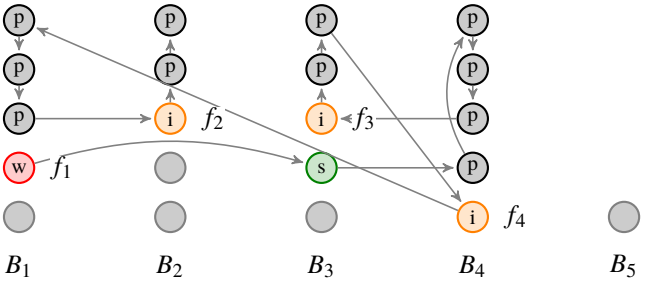
...Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert...



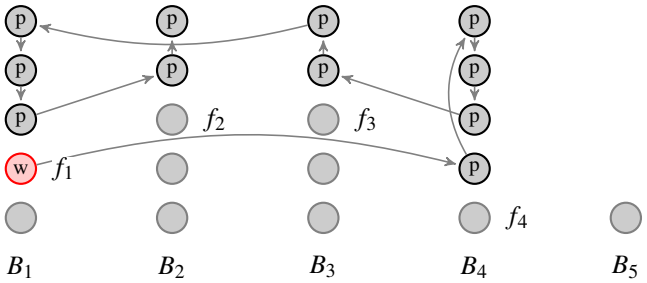
...Die Fokusse werden vorgerückt und Knoten aktiv gemacht, ein Wolf ohne Schafe wird als aktiver Knoten wiedergeboren...



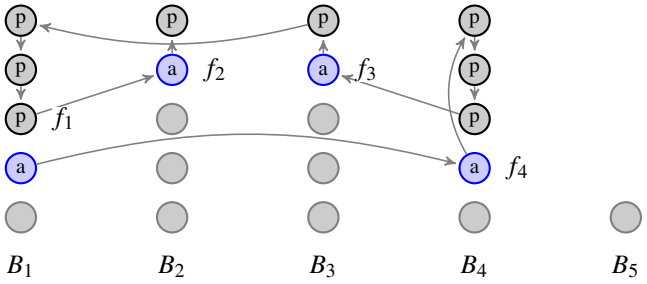
...Alle aktiven Knoten sind isoliert...



...Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert...



...Die Fokusse werden vorgerückt und Knoten aktiv gemacht, ein Wolf ohne Schafe wird als aktiver Knoten wiedergeboren.



11.3 Die Analyse

11.3.1 Plan

Die Laufzeit und die Arbeit. 11-31

Unser Ziel ist es nun, folgenden Satz zu beweisen:

▶ **Satz**

Der Algorithmus berechnet List-Rankings in Zeit $O(\log n)$ und Arbeit $O(n)$.

Beweisplan

- Wir analysieren zuerst die Zeit und Arbeit einer Runde.
- Dann zeigen wir, dass nach $O(\log n)$ Runden die Anzahl der Knoten, die nicht im Zustand suspended sind, nur noch $O(n/\log n)$ ist.

Die Laufzeit für eine Runde. 11-32

Hier nochmal, was alles in einer Runde passiert:

1. Die aktiven Knoten werden in isolierte Knoten, Wölfe und Schafe eingeteilt.
2. Isolierte Knoten werden entfernt und Wölfe fressen ihre Schafe.
3. Der Fokus wird aktualisiert: Bei ehemals aktive Knoten, die jetzt keine Wölfe sind, wird der Fokus erhöht und neu fokussierte Knoten wechseln von passiv auf aktiv.

▶ **Lemma**

Jede Runde kann in Zeit $O(1)$ und mit Arbeit $O(n/\log n)$ durchgeführt werden.

Beweis. In jeder Runde passieren nur Dinge, die $O(1)$ lange dauern und es arbeiten nur $O(n/\log n)$ Prozessoren. □

Was zu tun bleibt. 11-33

- Wir tun x mal etwas, was $O(1)$ lange dauert und Arbeit $O(n/\log n)$ verursacht.
- Falls also $x = O(\log n)$ ist, so brauchen wir $O(\log n)$ Schritte und die Arbeit ist $O(n)$.

Was zu zeigen bleibt.

Nach $O(\log n)$ Runden gibt es nur noch $O(n/\log n)$ Knoten, die nicht suspendiert wurden.

Probleme hierbei

- Es ist *nicht* der Fall, dass in jeder Runde alle Fokus-Zeiger hochgesetzt werden.
- Es ist *nicht* der Fall, dass alle Fokus-Zeiger in $O(\log n)$ Runden überhaupt das Blockende erreichen.

11.3.2 Nochmal Wölfe und Schafe

11-34

Eine neue Definition von Wölfen und Schafen.

Für die folgende Analyse ist es wichtig, dass Wölfe folgende Eigenschaft haben:

> Der Wolf soll mindestens so hoch wie alle seine Schafe sein.

Diese Bedingung ist normalerweise nicht garantiert, wir können sie aber durch folgenden Tricks erzwingen:

1. Wenn die aktiven Knoten in Wölfe und Schafe eingeteilt werden, werden auch solche Knoten Wölfe (statt Schafen), die ein lokales Höhenmaximum bilden.
2. Gleiche Höhen werden durch leichtes »Schütteln« ausgeschlossen (wie bei Übung 7.3).
3. Die Herde eines Wolfs ist nun die Menge aller Vorgänger und Nachfolger bis zum nächsten lokalen Höhenminimum oder bis zum nächsten Wolf.

11.3.3 Kostenanalyse

11-35

Die groben Ideen hinter der Kostenanalyse.

Sei $q = 1/\log\log n$.

- Wir machen Schulden bei einer Bank, und zwar gerade $\frac{n}{q\log n}$.
- Wir verteilen die Schulden (ungleichmäßig) auf die Knoten, die nicht suspended sind, jeder Knoten bekommt aber mindestens die Schulden $\frac{1}{2}(1-q)^{\log n}$.
- In jeder Runde zahlen wir die Schulden der in der Runde ausgeklinkten Knoten zurück.
- Dies wird jede Runde die Gesamtschulden um den Faktor mindestens $1-q/4$ senken.
- Nach $6\log n$ Runden sind unsere Schulden dann gesunken auf

$$\frac{n}{q\log n}(1-q/4)^{6\log n} \leq \frac{n}{\log n}(1-q)^{\log n}.$$

- Dann kann es nur noch $2n/\log n$ Knoten geben, denn sonst müssten die Schulden ja höher sein.

11-36

Die Verteilung der Kosten und die Blockschulden.

- Jeder nicht ausgeklinkte Knoten bekommt die Schulden $(1-q)^h$ aufgebrummt, wobei h die Höhe des Knotens ist.
- Schafe bekommen allerdings nur die halben Schulden.
- Die *Schulden eines Blocks* sind die Schulden aller passiven Knoten plus, falls es in dem Block einen Wolf gibt, die Schulden des Wolfs und seiner Herde.

Beobachtungen

- Jeder nicht ausgeklinkte Knoten bekommt mindestens den Wert $\frac{1}{2}(1-q)^{\log n}$ aufgebrummt, da $\log n$ die Blockhöhe ist.
- Blockschulden sind anfangs höchstens $\sum_{i=0}^{\log n}(1-q)^i \leq 1/q$.
- Also sind die Gesamtschulden anfangs höchstens $n/(q\log n)$.

11-37

Die Rückzahlungen bei isolierten Knoten.

Betrachten wir nun, was passiert, wenn ein *isolierter Knoten* gelöscht wird:

- Die Schulden seines Blocks waren *vorher*

$$\sum_{i=h}^{\log n-1}(1-q)^i.$$

- Die Schulden seines Blocks sind *hinterher*

$$\sum_{i=h+1}^{\log n-1}(1-q)^i \leq (1-q)\sum_{i=h}^{\log n-1}(1-q)^i.$$

- Die Kosten des Blocks sind also um den Faktor $1-q < 1-q/4$ gesunken.

Die Rückzahlungen bei Herdenbildung.

11-38

Betrachten wir nun, was passiert, wenn eine *Schafherde gebildet* wird:

- Betrachten wir die Gesamtschulden aller an der Herde beteiligten Blöcke.
- Die Gesamtschulden aller Blöcke *vorher* sind

$$Q := \sum_{j=1}^{k} \sum_{i=h_j}^{\log n-1} (1-q)^i.$$

  Hierbei ist h_j die Höhe des j-ten Schafs und h_1 die Höhe des Wolfs.
- Die Gesamtschulden aller Blöcke *nachher* sind

$$Q - \sum_{j=2}^{k} \frac{1}{2}(1-q)^{h_j}$$

- Es gilt

$$\begin{aligned} Q &= \sum_{j=1}^{k} \sum_{i=h_j}^{\log n-1} (1-q)^i \\ &\leq \sum_{j=1}^{k} \frac{1}{q}(1-q)^{h_j} \\ &\leq \sum_{j=2}^{k} \frac{2}{q}(1-q)^{h_j}. \end{aligned}$$

- Andererseits reduzierten sich die Kosten von Q auf

$$Q - \sum_{j=2}^{k} \frac{1}{2}(1-q)^{h_j} \leq Q - Q\frac{q}{4}.$$

Die Rückzahlungen beim Schaffressen.

11-39

Betrachten wir nun, was passiert, wenn ein *Schaf gefressen* wird:

- Die Schulden des Blocks waren *vorher*

$$\underbrace{\sum_{i=h}^{\log n-1} (1-q)^i}_{\text{passive Knoten}} + \underbrace{\frac{1}{2}\sum_{j=2}^{k}(1-q)^{h_j}}_{\text{Schafherde}}.$$

- Seien h_j die Höhen der Schafe und sei h_2 die Höhe des gefressenen Schafs. Wir dürfen annehmen, dass h_2 minimal ist, sonst verteilen wir die Gewichte um.
- Die Schulden des Blocks sind *hinterher*

$$\sum_{i=h}^{\log n-1} (1-q)^i + \frac{1}{2}\sum_{j=3}^{k}(1-q)^{h_j}.$$

- Es gilt $\sum_{i=h}^{\log n-1}(1-q)^i \leq \frac{1}{q}(1-q)^h$ und $\sum_{j=3}^{k}(1-q)^{h_j} \leq k(1-q)^{h_2} \leq \frac{1}{q}(1-q)^{h_2}$. Hier haben wir benutzt, dass jede Schafherde Größe höchstens $\log\log n = 1/q$ hat.
- Die Größe vorher ist also höchstens $\frac{3}{2q}(1-q)^{h_2}$ und reduziert sich um $\frac{1}{2}(1-q)^{h_2}$, also um den Faktor $1-q/3$.

Zusammenfassung dieses Kapitels

▶ Satz

Das List-Ranking-Problem lässt sich in *Zeit* $O(\log n)$ und *Arbeit* $O(n)$ lösen.

▶ Wesentliche algorithmische Ideen

- Man teilt die Eingabe in Blöcke der Größe $\log n$ auf.
- In jeder Runde wird eine Menge von Knoten gefärbt in Zeit $O(1)$.
- Hierdurch entstehen nur kleine unabhängige Mengen, aber dafür lange Ketten von nacheinander ausklinkbaren Knoten.
- Lange Ketten werden abgearbeitet, während »oberhalb« der Ketten schon weitergearbeitet wird.

▶ Wesentliche analysetechnische Ideen

- Es ist sofort klar, dass der Algorithmus in Zeit $O(\log n)$ läuft.
- Das Problem ist zu zeigen, dass er nur Arbeit $O(n)$ braucht.
- Dazu wird mittels einer *amortisierten Analyse* gezeigt, dass ein Potential nach $6 \log n$ Runden so weit gefallen ist, dass es nur noch $O(n \log n)$ Knoten geben kann.

12-1

Kapitel 12
Auswerten von arithmetischen Ausdrücken

Gemeinschaftliches Rechnen

12-2

Lernziele dieses Kapitels

1. Einen Baum in eine Eulertour umwandeln können.
2. Blätter eines Baumes in eine Traversierungsreihenfolge bringen können.
3. Konzept der Baumkontraktion kennen
4. Schnelle und optimale Algorithmen zur Auswertung arithmetischer Ausdrücke kennen und eigene erstellen können

Inhalte dieses Kapitels

Worum es heute geht

Über verschlungene Seitenpfade (Sie erinnern sich hoffentlich noch an bunte Listen, Italienkarten und schaffressende Wölfe) haben wir uns dem zentralen Problem dieses Teils der Veranstaltung genähert: Dem Auswerten arithmetischer Bäume.

Die Problematik liegt, wie schon früher angedeutet, darin, dass arithmetische Ausdrücke nicht schön symmetrisch sein müssen, sondern beliebig degenerieren können. Sind sie vollständig degeneriert (also im Wesentlichen Listen), so ist die Sache auch wieder übersichtlicher, wirklich problematisch sind die Bäume, die nicht Fisch und nicht Fleisch sind – weder einigermaßen ausgeglichene Bäume noch Listen. Um solche Ausdrücke zu verarbeiten, werden wir zwei »Tricks« benutzen: Erstens besorgen wir uns eine Inorder-Traversierung der Blätter, welche es uns erlauben wird, jedes »zweite« Blatt parallel zu verarbeiten.

Der zweite Trick ist, »schon mal anzufangen« mit dem Auswerten von Teilbäumen, auch wenn nicht alle nötigen Informationen vorliegen. Dies geht grob wie folgt: Ein Additionsknoten habe als linkes Kind ein Blatt mit dem Wert x. Das rechte Kind sei ebenfalls ein Additionsknoten, der als linkes Kind ebenfalls ein Blatt (mit dem Wert y) habe und als rechtes einen großen Teilbaum. Auf den Wert des großen Teilbaums wird man eventuell länger warten müssen; den Effekt des Additionsknoten und seines rechten Kindes kann aber leicht »schon mal« zusammengefasst werden zu einer einzigen Addition mit $x + y$. Ebenso kann man natürlich zwei Multiplikationen zu einer zusammenfassen. Kommen Additionen und Multiplikationen gemischt vor, so muss man etwas trickreicher vorgehen, aber auch dies ist möglich.

Wir werden heute nur Additionen und Multiplikationen betrachten. In den Übungsaufgaben wird angedeutet, wie man auch weitere Operationen integrieren kann; allerdings wird das nötige »trickreiche Vorgehen« dabei immer komplizierter.

Für den ersten Trick, die Inorder-Traversierung der Blätter, brauchen wir eine *Euler-Tour*

des Baumes. Diese gehen natürlich auf Leonhard Euler zurück. Über diesen weiß Wikipedia folgende interessante Dinge zu berichten (beachten Sie beispielsweise sein Alter zum Zeitpunkt seiner Berufung zum Professor und überlegen Sie, was Sie in diesem Alter getan haben):

Autor Petar Marjanovix, GNU Free Documentation License

Aus *de.wikipedia.org/wiki/Leonhard_Euler*

1707 wurde Leonhard Euler in der Deutschschweiz als der älteste Sohn des Pfarrers Paul Euler und Margarethe Bruckner geboren. Er besuchte das Gymnasium am Münsterplatz in Basel und nahm gleichzeitig Privatunterricht beim Mathematiker Johannes Burckhardt. Ab 1720 studierte er an der Universität Basel und hörte hier Vorlesungen von Johann Bernoulli. 1723 erlangte er durch einen Vergleich der Newtonschen und Kartesischen Philosophie in lateinischer Sprache die Magisterwürde. Seinen Plan, auch Theologie zu studieren, gab er 1725 auf. Am 17. Mai 1727 berief ihn Daniel Bernoulli an die Universität Sankt Petersburg. Er erbte die Professur des 1726 verstorbenen Nikolaus II Bernoulli. Hier traf er auf Christian Goldbach. 1730 erhielt Euler die Professur für Physik und trat schließlich 1733 die Nachfolge von Daniel Bernoulli als Professor für Mathematik an. Er bekam in den folgenden Jahren immer stärkere Probleme mit seinem Augenlicht und war ab 1740 halbseitig blind.

1741 wurde er von Friedrich dem Großen an die Berliner Akademie berufen. Euler korrespondierte und verglich seine Theorien weiterhin mit Christian Goldbach. Nach 25 Jahren in Berlin kehrte er 1766 zurück nach St. Petersburg. An seine Tätigkeit und sein damaliges Wohnhaus in Berlin erinnert eine Gedenktafel an der Behrenstraße 22/23, das heutige Haus der Bayerischen Landesvertretung in Berlin. Im St. Petersburg der Zarin Katharina der Großen wurde ihm an der Akademie der Wissenschaften ein ehrenvoller Empfang bereitet. Er arbeitete wie in der ersten Sankt Petersburger Periode in der Kunstkamera und lebte in einem von Katharina der Großen geschenkten Palais mit seinem Sohn Johann Albrecht direkt an der Newa.

1771 erblindete er vollständig. Trotzdem entstand fast die Hälfte seines Lebenswerks in der zweiten Petersburger Zeit. Hilfe erhielt er dabei von seinem Sekretär Niklaus Fuß, der nach seinem Tod als erster eine Würdigung verfasste, und seinen Söhnen Johann Albrecht, Karl und Christoph. 1783 starb er an einer Hirnblutung. Trotz seiner Forderung wurde er nie Präsident der Universität, dieses Amt besetzte meist einer der Liebhaber Katharinas, aber sein Einfluss in der Universität war fast dem des Präsidenten ebenbürtig.

Euler war extrem produktiv: Insgesamt gibt es 866 Publikationen von ihm. Ein großer Teil der heutigen mathematischen Symbolik geht auf Euler zurück (zum Beispiel e, π, i, Summenzeichen $\sum$, $f(x)$ als Darstellung für eine Funktion). 1744 gibt er ein Lehrbuch der Variationsrechnung heraus. Euler kann auch als der eigentliche Begründer der Analysis angesehen werden. 1748 publiziert er das Grundlagenwerk *Introductio in analysin infinitorum*, in dem zum ersten Mal der Begriff der Funktion die zentrale Rolle spielt.

In den Werken *Institutiones calculi differentialis* (1765) und *Institutiones calculi integralis* (1768–1770) beschäftigt er sich außer mit der Differential- und Integralrechnung unter anderem mit Differenzengleichungen, elliptischen Integralen sowie auch mit der Theorie der Gamma- und Betafunktion. Andere Arbeiten setzen sich mit Zahlentheorie, Algebra (zum Beispiel *Vollständige Anleitung zur Algebra*, 1770), angewandter Mathematik (zum Beispiel *Mechanica, sive motus scientia analytica exposita*, 1736 und *Theoria motus corporum solidorum seu rigidorum*, 1765) und sogar mit der Anwendung mathematischer Methoden in den Sozial- und Wirtschaftswissenschaften auseinander (zum Beispiel Rentenrechnung, Lotterien, Lebenserwartung).

In der Mechanik arbeitete er auf den Gebieten der Hydrodynamik (Eulersche Bewegungsgleichungen, Turbinengleichung) und der Kreiseltheorie (Eulersche Kreiselgleichungen). Die erste analytische Beschreibung der Knickung eines mit einer Druckkraft belasteten Stabes geht auf Euler zurück; er begründete damit die Stabilitätstheorie. In der Optik veröffentlichte er Werke zur Wellentheorie des Lichts und zur Berechnung von optischen Linsen zur Vermeidung von Farbfehlern.

Seine 1736 veröffentlichte Arbeit *Solutio problematis ad geometriam situs pertinentis* beschäftigt sich mit dem Königsberger Brückenproblem und gilt als eine der ersten Arbeiten auf dem Gebiet der Graphentheorie.

Über seinen wenig rezipierten Beitrag zur mathematischen Musiktheorie (*Tentamen novae theoriae musicae*, 1739), bemerkte ein Biograph: »für die Musiker zu anspruchsvolle Mathematik, für die Mathematiker zu musikalisch.«

1745 übersetzte Leonhard Euler das Werk des Engländers Benjamin Robins *New principles of gunnery* ins Deutsche, das im selben Jahre in Berlin unter dem Titel *Neue Grundsätze der Artillerie - enthaltend die Bestimmungen der Gewalt des Pulvers nebst einer Untersuchung über*

den Unterschied des Widerstands der Luft in schnellen und langsamen Bewegungen aus dem Englischen des Herrn Benjamin Robins übersetzt und mit den nötigen Erläuterungen und vielen Anmerkungen versehen. Das Buch beschäftigt sich mit der so genannten inneren Ballistik und – als Hauptthema – mit der äußeren Ballistik. Seit Galilei hatten die Artilleristen die Flugbahn der Geschosse als Parabeln angesehen, indem sie den Luftwiderstand wegen der »Dünnheit« der Luft glaubten vernachlässigen zu dürfen. Robins hat als einer der ersten wertvolle Experimente ausgeführt und gezeigt, dass dem nicht so ist; dass im Gegenteil die Flugbahn durch den Einfluss des Luftwiderstandes wesentlich abgeändert werde. Somit wurde dank Robins und Eulers Mithilfe »das erste Lehrbuch der Ballistik« geschaffen. Da solch ein Lehrbuch einer Armee einen Vorteil verschaffte, wurde es 1777 wieder ins Englische und 1783 ins Französische übersetzt. In Frankreich wurde es sogar als offizielles Lehrbuch in den Militärschulen eingeführt, sodass sogar Napoléon Bonaparte es (als Leutnant) studieren musste.

Besondere Bedeutung in der breiten Öffentlichkeit erlangte seine populärwissenschaftliche Schrift *Lettres à une princesse d'Allemagne* von 1768, in der er in Form von Briefen an die Prinzessin von Anhalt-Dessau, einer Nichte Friedrichs des Großen, die Grundzüge der Physik, der Astronomie, der Mathematik, der Philosophie und der Theologie vermittelt.

Zeitgenossen Eulers waren unter anderen Christian Goldbach, Jean le Rond d'Alembert, Alexis-Claude Clairaut, Johann Heinrich Lambert und einige Mitglieder der Familie Bernoulli.

12.1 Euler-Touren

Die Brücken von Königsberg 12-4

Definition von Eulertouren. 12-5

Unknown author, Public Domain

▶ Definition

Eine *Eulertour* durch einen Graphen ist eine Folge von Knoten, so dass

1. je zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind,
2. der letzte und der erste Knoten in der Folge durch eine Kante verbunden sind,
3. jede Kante des Graphen genau einmal besucht wird.

Bekanntermaßen gilt folgender Satz:

▶ Satz

Ein zusammenhängender Graph hat genau dann eine Eulertour, wenn jeder Knoten einen geraden Grad (Anzahl Nachbarn) hat.

12.1.1 Sortieren von Blättern

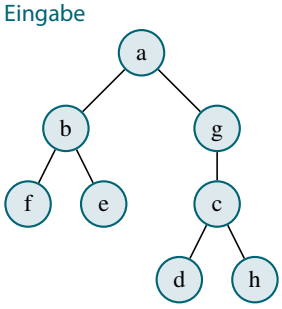
Ziel: Reihenfolge der Blätter bei einer Inorder-Traversierung. 12-6

Der Problem INORDER-RANKING

Eingabe Ein gerichteter Baum mit einer Reihenfolge auf den Kindern jedes Knotens.

Ausgabe Die Blätter in der Reihenfolge einer Inorder-Traversierung.

Eingabe



Ausgabe

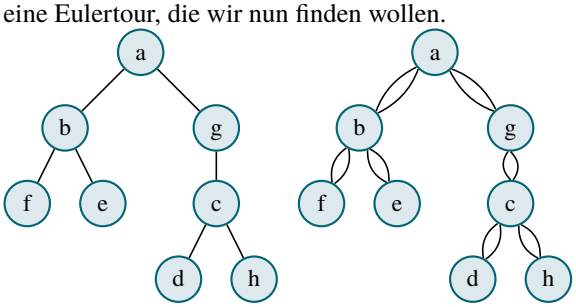
f, e, d, h

12.1.2 Vom Baum zur Eulertour

12-7

Wir können Bäume in Eulertouren verwandeln.

Gegeben sei ein Baum, dargestellt als Adjazenzliste. Wir ersetzen nun jede Kante durch zwei Kanten. Im resultierenden Graph hat jeder Knoten einen geraden Grad. Also gibt es eine Eulertour, die wir nun finden wollen.



12-8

Wie findet man die Eulertour?

Algorithmus

1. Eingabe sei die Adjazenzliste des Baumes.
2. Lege folgende Reihenfolge auf den Nachbarn jedes Knotens fest:
   Erst die Kinder von links nach rechts, dann der Elternknoten.
3. Erzeuge daraus eine Liste der Kanten für die Eulertour.
4. Verkette *die Kanten* wie folgt:
   - Sei (u, v) eine Kante.
   - Dann ist u ein Nachbar von v. Sei u' der »nächste« Nachbar von v.
   - Dann ist der Nachfolger der Kante (u, v) die Kante (v, u').

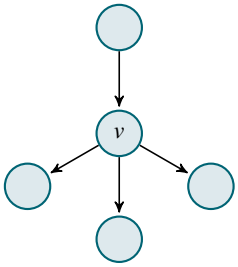
▶ Satz

Zu einem durch eine Adjazenzliste gegebenen Baum kann man in Zeit $O(1)$ und Arbeit $O(n)$ eine Eulertour berechnen.

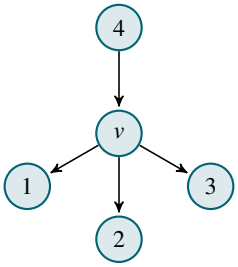
12-9

Der Algorithmus lokal für einen Knoten v.

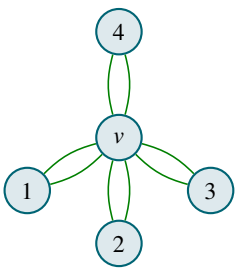
1 Der Knoten v in der Ausgangssituation



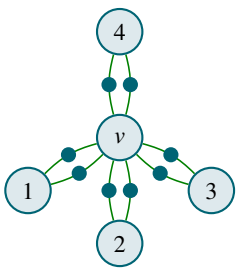
2 Eine Reihenfolge für die Nachbarn von v



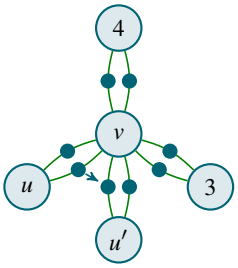
3 Erzeuge Kanten eines Graphen



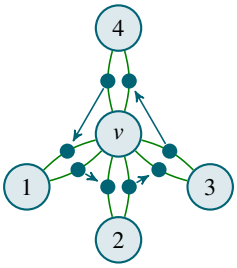
4.1 Diese Kanten sind die Knoten (!) der Liste



4.2 Die Kante zwischen (u, v) und (v, u')

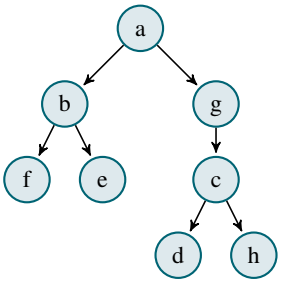


4.3 Alle lokalen Kanten

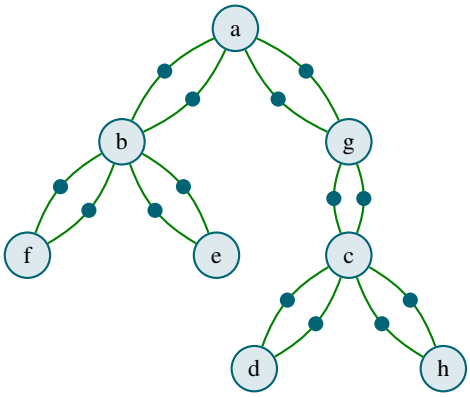


Der Algorithmus für einen ganzen Baum.

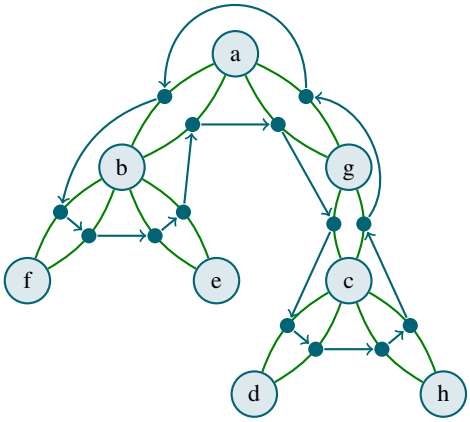
Beispiel: Der Ausgangsbaum



Beispiel: Die Knoten der Eulertour als Liste



Beispiel: Die Verkettung der Eulertour als Liste



12.1.3 Von der Eulertour zum Ranking

12-11

Von der Eulertour zur Sortierung der Blätter.

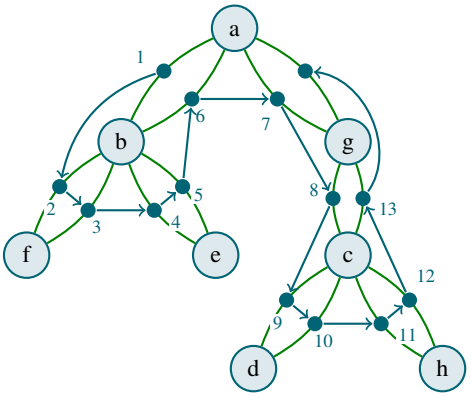
Unser Hauptziel ist immernoch, die Blätter eines Baumes in ihre Inorder-Reihenfolge zu bringen.
Dazu gehen wir wie folgt vor:

1. Wir berechnen eine Eulertour.
2. Wir *brechen diese am Ende auf* durch Löschen der letzten Kante.
3. Wir wenden einen *Ranking-Algorithmus* auf die Kanten an.
4. Wir *ziehen die Liste gerade*.
5. Wir löschen alle Kanten, die *nicht zu Blättern führen*.
6. Wir wenden *Präfixsummen* an, um jedem Blatt eine Position zuzuweisen.

12-12

Die Schritte des Algorithmus.

Beispiel: Das Ranking der aufgebrochenen Liste



Beispiel: Die geradegezogene Liste im Speicher

Rang	1	2	3	4	5	6	7	8	9	10	11	12	13
Kante von	a	b	f	b	e	b	a	g	c	d	c	h	c
nach	b	*f*	b	*e*	b	a	g	c	*d*	c	*h*	c	g

12-13

Wie schnell geht das Sortieren der Blätter?

▶ Satz

Die Inorder-Traversierungsreihenfolge eines Baumes, der als Adjazenzliste gegeben ist, kann in Zeit $O(\log n)$ und Arbeit $O(n)$ berechnet werden.

Beweis. Alle Schritte des Algorithmus benötigen entweder Zeit $O(1)$ oder $O(\log n)$ und alle verursachen Arbeit von höchstens $O(n)$. □

12.2 Arithmetische Ausdrücke

12.2.1 Problemstellung

Worum es geht.

12-14

Wir wollen *arithmetische Ausdrücke* ausrechnen.

- Beispiele sind $3+4$ oder $5\cdot(6+10)$.
- Es kommen nur Additionen und Multiplikationen vor.
- Dies Ausdrücke können beliebig komplex sein.

Wir wollen sie *parallel* auswerten. Die Addition und die Multiplikation von zwei Zahlen gilt als Elementaroperation, dauert also $O(1)$.

Die genaue Problemstellung.

12-15

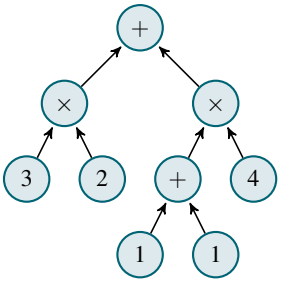
Problemstellung

Eingabe Die Adjazenzliste eines binären Baumes, dessen innere Knoten mit den Symbolen $+$ und $\times$ beschriftet sind und dessen Blätter mit ganzen Zahlen beschriftet sind.

Ausgabe Wert des Baumes.

Beispiel

Eingabe:



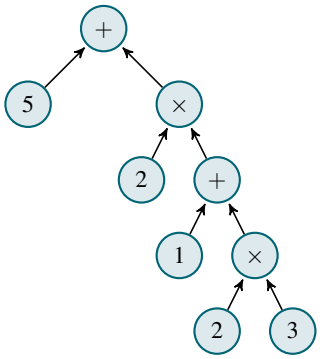
Ausgabe: 14

12.2.2 Die Problematik

Die Problematik im allgemeinen Fall.

12-16

Besonders problematisch scheint der Fall eines degenerierten Baumes. Hier kann man in jedem Schritt immer nur für einen Knoten den Wert sofort ausrechnen. Die entsprechenden arithmetischen Ausdrücke kommen sehr häufig vor (zum Beispiel im Horner-Schema).

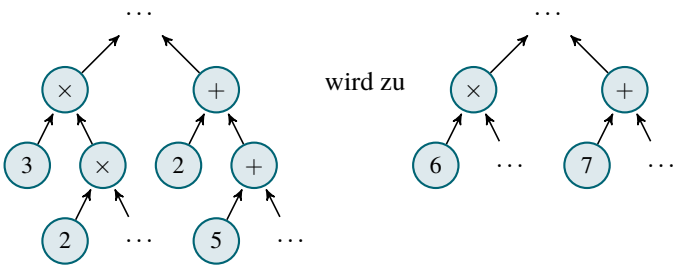


12.2.3 Lösungsidee

12-17

Lösungsidee: Schon Mal anfangen. . .

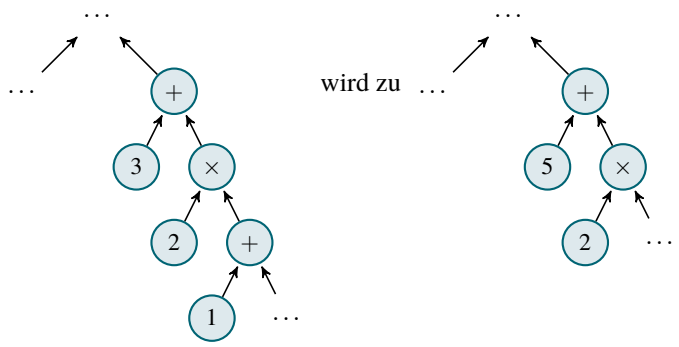
Wir können den Wert von Knoten »in der Mitte« nicht komplett auswerten; wir können aber »schon mal anfangen«. Dazu fassen wir die »Wirkung« mehrerer Knoten zusammen.



12-18

Neues Problem: Mischung von Addition und Multiplikation

Kommen Multiplikationen und Additionen durcheinander, so ist unklar, wie man »schon mal anfangen« kann. Der Trick ist, dass man *mehrere* gemischte Multiplikationen und Additionen zu *einer* Multiplikation gefolgt von *einer* Addition zusammenfassen kann.



12-19

Grobe Lösungsidee.

In jeder Runde werden wir parallel die Hälfte aller Blätter entfernen. Dazu wird an jedem zweiten Blatt »schon mal angefangen zu rechnen«. Dies verkleinert den Baum in jedem Schritt um die Hälfte. Nach $\log n$ Schritten haben wir dann das Ergebnis.

12.2.4 Linearformen als Kantengewichte

12-20

Was ist eine Linearform?

▶ Definition

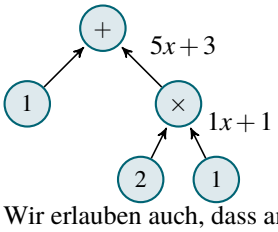
Eine *Linearform* ist eine Funktion der Form $f(x) = a \cdot x + b$ für zwei Konstanten a und b.

Linearformen beschreiben genau den gemeinsamen Effekt mehrerer Additionen und Multiplikationen. Wir werden Linearformen einfach schreiben als $ax + b$. Man kann sie sogar noch kürzer schreiben als (a, b).
Die Linearform $1x + 0$ ist offenbar die Identität.

12-21

Was machen wir mit den Linearform?

Wir schreiben Linearformen an *Kanten* des Baumes. Die *Bedeutung* einer Linearformen an einer Kante ist: Wenn das Ergebnis des Teilbaumes unterhalb der Kante »hochgereicht« wird, dann muss dabei die Linearformen angewandt werden.



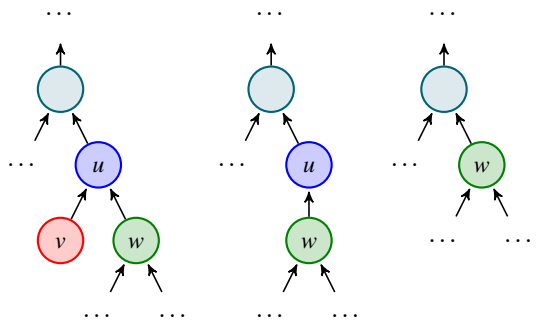
Wir erlauben auch, dass an einer Kante keine Linearform steht. Wen dies stört, der schreibt an solche Kanten $1x + 0$.

12.2.5 Die Rake-Operation

Die *Rake*-Operation. 12-22

Ziel ist es nun, ein Blatt »loszuwerden«. Das Blatt darf auch mitten im Baum stehen und die Kanten sind schon mit Linearformen belegt. Das Entfernen des Blattes geschieht in zwei Schritten.

1. Das Blatt wird entfernt.
2. Der Eltern-Knoten wird entfernt.

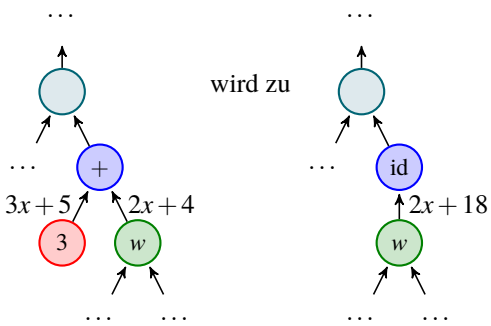


Details der *Rake*-Operation. 12-23

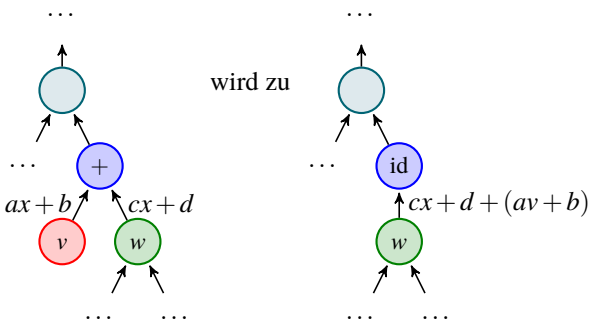
Teil 1.1: Blatt löschen bei einer Addition.

Wir wollen das Blatt v löschen und der Elternknoten u ist ein Additionsknoten.

Beispiel



Allgemeine Umwandlung

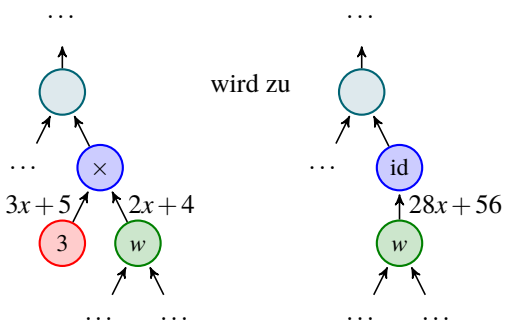


Details der *Rake*-Operation. 12-24

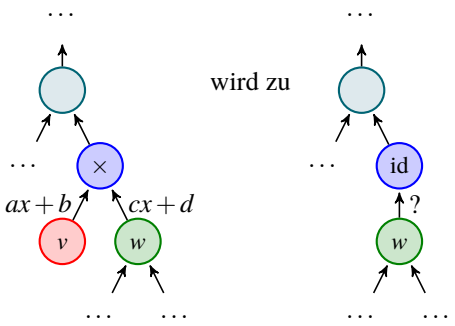
Teil 1.2: Blatt löschen bei einer Multiplikation.

Wir wollen das Blatt v löschen und der Elternknoten u ist ein Multiplikationsknoten.

Beispiel



✎ Zur Übung



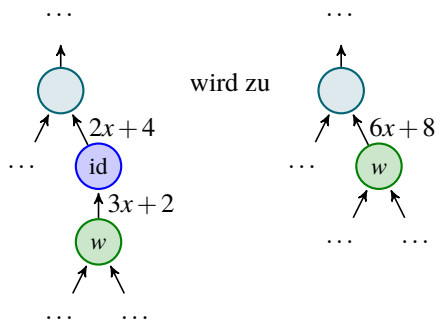
12-25

Details der *Rake*-Operation.

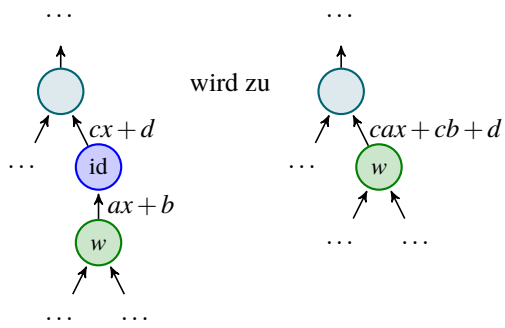
Teil 2: Löschen des Knotens mit nur einem Kind.

Wir wollen den alten Elternknoten löschen, der jetzt nur noch ein Kind hat.

Beispiel



Allgemeine Umwandlung



12-26

Zur Laufzeit und Korrektheit des Algorithmus

▶ Satz

Der Algorithmus Rake wandelt einen arithmetischen Baum in Zeit $O(1)$ und Arbeit $O(1)$ um, ohne seinen Wert zu ändern.

12.2.6 Algorithmus

Der Algorithmus – wir setzen jetzt alles zusammen. 12-27

Algorithmus *Evaluate*

1. Lies einen arithmetischen Baum als Eingabe, gegeben als Adjazenzliste.
2. Erstelle mittels der Euler-Tour-Technik einen Blätter-Array, in dem die Blätter nacheinander in ihrer Traversierungsreihenfolge im Baum stehen.
3. Wenn der Baum nur noch aus der Wurzel besteht, so steht dort das Ergebnis. Sonst wiederhole:
   - 3.1 Betrachte nur die Blätter mit *gerader Nummer*.
   - 3.2 Teile diese (konzeptionell) in die Mengen L und R der Blätter auf, die *linke Kinder* und die *rechte Kinder* sind.
   - 3.3 Führe *Rake* parallel für alle Elemente von L aus.
   - 3.4 Führe *Rake* parallel für alle Elemente von R aus.
   - 3.5 Streiche (konzeptionell) alle Blätter mit gerader Nummer aus dem Array.

Zur Laufzeit und Korrektheit des Algorithmus 12-28

▶ Satz
Der Algorithmus Evaluate wertet arithmetische Ausdrücke in Zeit $O(\log n)$ und Arbeit $O(n)$ aus.

Beweis.

- Die *Laufzeit ist* $O(\log n)$, da die Vorverarbeitung Zeit $O(\log n)$ braucht und die Hauptschleife $\log n$ mal durchlaufen wird und in der Schleife immer nur Zeit $O(1)$ benötigt wird.
- Die *Arbeit ist* $O(n)$, da die Vorverarbeitung $O(n)$ Arbeit macht und in der Hauptschleife jedes Blatt nur einmal betrachtet wird.
- *Der Algorithmus ist korrekt,* da sichergestellt ist, dass sich je zwei *Rake*-Operationen nicht »in die Quere kommen«. □

Zusammenfassung dieses Kapitels

1. Man kann einen Baum in eine Eulertour umwandeln, indem man jede Kante verdoppelt. 12-29
2. Man kann mittels der Eulertour die *Inorder-Reihenfolge* der Blätter bestimmen.
3. Arithmetische Ausdrücke lassen sich in Zeit $O(\log n)$ und Arbeit $O(n)$ auswerten.
4. Bei nicht ausgeglichenen Bäumen muss man mittels Linearformen die »Berechnungen schon mal in der Mitte beginnen«, obwohl nicht alle Daten vorliegen.

Übungen zu diesem Kapitel

Übung 12.1 Auswertung von monotonen booleschen Formeln, leicht, mit Lösung

Eine *monotone boolesche Formel ohne Variablen* kann man als Baum auffassen, an dessen Blättern die Konstanten `true` und `false` stehen und dessen innere Knoten mit den Junktoren $\wedge$ und $\vee$ beschriftet sind.

Beschreiben Sie, wie ein Algorithmus für die Auswertung solcher Formeln funktioniert, der eine Laufzeit von $O(\log n)$ und Arbeit $O(n)$ hat. Der Baum sei wie üblich als Adjazenzliste gegeben. Der Detailgrad Ihrer Beschreibung des Algorithmus sollte sich am Skript zur Vorlesung über arithmetische Ausdrücke orientieren.

Übung 12.2 Auswertung von booleschen Formeln, mittel

Eine allgemeine boolesche Formel ohne Variablen enthält neben den Und- und Oder-Junktoren auch noch Negationsknoten mit nur einem Kind.

Beschreiben Sie, wie Sie Ihren Algorithmus aus der vorherigen Aufgabe erweitern können, so dass er auch mit solchen Formeln umgehen kann (und noch dieselbe Laufzeit und Arbeit aufweist). Beachten Sie, dass Negationsgatter in langen Ketten auftreten können (es gibt verschiedene Arten, dieses Problem zu lösen).

Übung 12.3 Auswertung für Addition und Maximum, mittel

Geben Sie einen Algorithmus an, der einen arithmetischen Ausdruck auf den rationalen Zahlen über den binären Operationen Addition $(+)$ und Maximumsbildung (max) auswertet. Der Ausdruck ist dabei als Binärbaum mit n Blättern gegeben und der Algorithmus soll eine Laufzeit von $O(\log n)$ besitzen.

Tipp: Benutzen Sie statt der Linearformen $a \cdot x + b$ als Kantenbeschriftung Ausdrücke der Form $\max(a + x, b)$.

Übung 12.4 Auswertung von verallgemeinerten Formeln, schwer

Lösen Sie die vorherige Übungsaufgabe für arithmetische Ausdrücke auf den positiven rationalen Zahlen über den Operationen $+$, $\times$ und max.

Tipp: Überlegen Sie sich zunächst einen geeigneten Ersatz für die Linearformen.

13-1

Kapitel 13
Parallele Grundrechenarten

Computer rechnen nicht so, wie wir es in der Schule gelernt haben

Lernziele dieses Kapitels

1. AC- und NC-Schaltkreisklassen kennen
2. NC^1-Additionsalgorithmus kennen
3. NC^1-Multiplikationsalgorithmus kennen
4. NC^2-Divisionsalgorithmus kennen

13-2

Inhalte dieses Kapitels

Worum es heute geht

Schaltkreise sind die einfachste Form paralleler »Programme«. In der Tat lässt sich kaum leugnen, dass Schaltkreise massiv parallel arbeiten, denn alle Gatter arbeiten gleichzeitig. Den Unterschied zwischen Theorie und Praxis von Schaltkreisen erkennt man aber schon am Namen: Schalt*kreise* heißen elektrotechnisch so, weil da Elektronen »im Kreis« fließen; Schaltkreise heißen mathematisch so, obwohl sie definiert sind als besondere Graphen, die kreis*frei* sind. Trotzdem ist die mathematische Modellierung von Schaltkreisen als Graphen nicht sonderlich weit weg von der Wirklichkeit. Die Umwandlung in reale Hardware geht ohne größere Zeit- oder Platzverluste: Ein Und-Gatter ist auch in Hardware ein Und-Gatter, schlimmstenfalls eine kleine Ersatzschaltung aus mehreren NAND- oder NOR-Gattern.

In dem Film *Die Matrix* haben Sie gelernt »time is always against us«. Bei Schaltkreisen entspricht der Zeit die *Tiefe* des Schaltkreises, denn so lange müssen wir warten, bis ein Ergebnis vorliegt. Deshalb wird unser Hauptziel sein, *flache* Schaltkreise zu finden. Dazu definieren wir – wie sollte es in der Komplexitätstheorie anders sein – ein paar Klassen, die die Tiefe von Schaltkreisen messen, und sortieren dann Probleme in diese Klassen ein.

Mit diesen Klassenbegriffen bewaffnet können wir uns dann daran wagen, die Komplexität der Grundrechenarten zu untersuchen. Zwei Zahlen zu multiplizieren oder gar zu dividieren ist eine Kunst für sich. Beide Operationen sind so grundlegend, dass die Menschheit sich wirklich *viele* Gedanken darüber gemacht hat, wie man diese beiden Operationen möglichst effizient hinbekommt. Um so verblüffender erscheint es, dass die Komplexität der Division

aus komplexitätstheoretischer Sicht erst im Jahr 2001 endgültig geklärt werden konnte (das Dividieren ist vollständig für TC^0, eine Klasse zwischen AC^0 und NC^1).

Dabei erscheinen beide Operationen zunächst ganz einfach, schließlich lernt jedes Kind in der Schule schon recht früh, wie dies geht. Die »Schulmethoden« haben aber den Nachteil, dass sie (a) »sehr sequentiell« sind, insbesondere die Methode zur Division, und (b) quadratischen Aufwand (in der Länge der Zahlen) haben, was bei sehr großen Zahlen – wie sie zum Beispiel in der Kryptologie oft auftreten – zu viel erscheint.

Ist man nicht an einer Parallelisierung interessiert, dann bietet der Algorithmus von Anatolij Alexejewitsch Karazuba eine schnellere Methode. Soll es aber parallel gehen, so muss man sich etwas Neues ausdenken. Mit etwas Informatiksachverstand sieht man schnell, dass die Multiplikation mit einem Divide-and-Conquer-Algorithmus recht gut parallelisiert werden kann, was recht einfach einen NC^2-Algorithmus liefert. Für einen NC^1-Algorithmus, unser erstes Hauptziel in diesem Kapitel, ist ein netter kleiner Zusatztrick nötig.

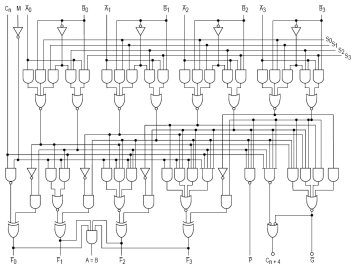
Bei der Division muss man schon mit größeren Kalibern hantieren. In diesem Kapitel wird eine Newton-Iteration für die Division vorgestellt, welche einen NC^2-Algorithmus liefert. Dies ist noch ein sehr kleines Geschütz verglichen mit den mathematischen Schwergewichten, die im Beweis für die TC^0-Vollständigkeit der Division notwendig sind. Da sie für diese Veranstaltung eine Nummer zu groß sind, lassen wir deshalb lieber die Finger davon.

13.1 Schaltkreise

13-4

Zwei Schaltkreise.

Copyright by Qurren, GNU Free Documentation License



Copyright by Stephane Tsacas, GNU Free Documentation License

13.1.1 Gatter und Verbindungen

13-5

Wie unsere Schaltkreise prinzipiell aufgebaut sind.

- Wir werden nur *Boole'sche Schaltkreise* betrachten.
  Das bedeutet, dass nur die Werte 0 und 1 vorkommen, entsprechend Strom fließt und kein Strom fließt oder hohe Spannung und niedrige Spannung.
- Schaltkreise zeichnen sich dadurch aus, dass sie *keine Kreise* bilden.
  (Gibt es Rückkopplungen, also Kreise, so spricht man von Schaltwerken.)
- Der Schaltkreis besteht also aus zwei Dingen:
  1. *Gattern*, an denen (mehrere) 0 und 1 Werte verarbeitet werden, und
  2. *Verbindungen*, also Drähten, die die Gatter miteinander verbinden.

13-6

Welche Arten von Gattern gibt es?

Ein *Gatter* leistet folgendes: Es hat (eventuell mehrere) Eingänge, an denen von vorherigen Gattern in jedem Takt 0en und 1en anliegen. Das Gatter errechnet aus diesen Werten einen neuen Wert (wieder eine 0 oder eine 1) und liefert diesen an seinen Ausgang. Vom Ausgang aus wandern die Werte dann zu den nachgeschalteten Gattern.
Wir werden *Und*-, *Oder*- und *Nicht*-Gatter zulassen. (Bekanntermaßen würden auch beispielsweise Nand-Gatter genügen.)

13.1.2 Definition von Schaltkreisen

Die mathematische Definition eines Schaltkreises. 13-7

▶ Definition

Ein *Schaltkreis* ist ein gerichteter azyklischer Graph mit folgenden Eigenschaften:

- Die *Knoten* des Graphen sind die Gatter.
- Die *Kanten* des Graphen sind die Verbindungen.
- Jedes Gatter hat einen der folgenden *Gattertypen*:
  - *Eingabegatter* – solche Knoten haben Eingrad 0.
  - *Ausgabegatter* – solche Knoten haben Ausgrad 0 und Eingrad 1.
  - *Konstante Gatter* – solche Knoten haben Eingrad 0.
  - *Negationsgatter* – solche Knoten haben Eingrad 1.
  - *Und-Gatter*
  - *Oder-Gatter*
- Die Eingabegatter benennen wir x_1 bis x_n.
- Die Ausgabegatter benennen wir y_1 bis y_m.

Es gibt keine feste Notation für Schaltkreise, sie ist Geschmackssache.

Wie eine Berechnung mittels eines Schaltkreises funktioniert. 13-8

Wir wollen Schaltkreise natürlich benutzen, um Eingaben in Ausgaben umzuformen. Dazu *legt man die Eingabe an den Eingängen an* und schaut, wie die Berechnung vorangeht. Jedes Gatter kann erst dann beginnen zu arbeiten, wenn alle seine (lokalen) Eingänge einen Wert vorweisen.

Formale Definition der Berechnung. 13-9

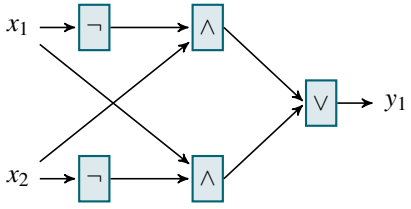
▶ Definition: Berechnete Funktion

Sei C ein Schaltkreis mit n Eingängen und m Ausgängen. Dann berechnet C eine Funktion $f_C\colon \{0,1\}^n \to \{0,1\}^m$ wie folgt:

- Seien b_1 bis b_n Bits.
- Wir definieren rekursiv den *Wert* eines Gatter als
  - b_i für Eingabegatter x_i.
  - den Wert des Vorgängers für Ausgabegatter.
  - i für konstante i-Gatter.
  - $1-v$ für Negationsgatter, wenn v der Wert des Vorgängers ist.
  - das logische Oder aller Vorgängergatter für Oder-Gatter und
  - das logische Und aller Vorgängergatter für Und-Gatter.
- Sind v_1 bis v_m dann die Werte der Ausgabegatter, so ist $f_c(b_1 \cdots b_n) = v_1 \cdots v_m$.

Beispiel eines Schaltkreises 13-10

Beispiel: Ein Schaltkreis, der die XOR-Funktion berechnet.



13.1.3 Die AC- und NC-Klassen

13-11

Welche Ressourcen sind bei Schaltkreisen wichtig?

Sicherlich sind folgende Ressourcen bei Schaltkreisen wichtig:

- *Die Größe* des Schaltkreise (Anzahl Gatter), da dies der Chip-Fläche entspricht.
- *Die Tiefe* des Schaltkreises (maximale Weglänge von Eingabe- zu Ausgabegattern), da dies der Rechenzeit entspricht.
- *Der Fan-In* des Schaltkreises (maximaler Eingrad von Gattern), da Gatter mit hohem Fan-In schwer oder gar nicht zu bauen sind.

13-12

Vom Schaltkreis zur Schaltkreisfamilie.

Problem

Ein einzelner Schaltkreis kann nur genutzt werden, Eingaben *einer festen Länge zu bearbeiten*. Im Allgemeinen wollen wir aber Eingaben *beliebiger Länge* bearbeiten.

Lösung

Der Trick ist, für jede mögliche Eingabelänge n einen eigenen Schaltkreis C_n zu benutzen. Eine solche Liste $C = (C_0, C_1, C_2, \dots)$ nennt man eine *Schaltkreisfamilie*.
Wir schreiben einfach $C(x)$ für $f_{C_{|x|}}(x)$.

Für eine Schaltkreisfamilie C bezeichnet

- $\text{depth}_C(n)$ die *Tiefe von* C_n und
- $\text{size}_C(n)$ die *Größe von* C_n.

Man verlangt in der Regel noch, dass die Schaltkreisfamilien »uniform«, also nicht zu »wild« sind, aber das ist für diese Vorlesung unwichtig.

13-13

Flache Schaltkreisklassen: AC-Klassen und NC-Klassen.

▶ **Definition:** AC-Klassen

Sei i eine natürliche Zahl. Die *Klasse* AC^i enthält alle Sprachen L, deren charakteristische Funktion von einer Schaltkreisfamilie C

- polynomieller Größe (also $\text{depth}_C \in O(n^k)$ für eine Konstante k) und
- der Tiefe $O(\log^i n)$ berechnet werden kann.

▶ **Definition:** NC-Klassen

Sei i eine natürliche Zahl. Die *Klasse* NC^i enthält alle Sprachen L, deren charakteristische Funktion von einer Schaltkreisfamilie C

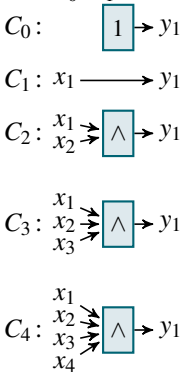
- mit polynomieller Größe,
- mit Tiefe $O(\log^i n)$ und
- mit Fan-In maximal 2 berechnet werden kann.

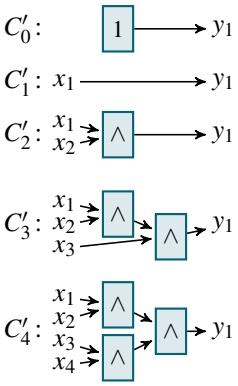
13-14

Ein ganz einfaches Beispiel.

Beispiel

Die Sprache $\{1\}^*$ liegt sowohl in AC^0 via der Schaltkreisfamilie $(C_0, C_1, \dots)$ als auch in NC^1 via $(C'_0, C'_1, \dots)$:





13.2 Addition

Die Problemstellung zum Additionsproblem.

13-15

Problemstellung

Für ein gegebenes n wollen wir die Funktion $p_n\colon \{0,1\}^{2n} \to \{0,1\}^{n+1}$ berechnen.
Sie addiert die ersten n Bits der Eingabe auf die zweite n Bits und liefert das Ergebnis zurück.

Beispiel

- $p_4(01001101) = 10001$.
- $p_4(00010001) = 00010$.

Ziel

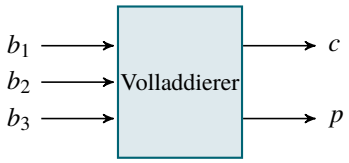
Ein »NC^1-Schaltkreis« für die Funktion.
Genauer ist eine Schaltkreisfamilie polynomieller Größe und logarithmischer Tiefe gesucht, die alle p_n berechnet.

13.2.1 Der naive Addierer

Der naive Addierer.

13-16

▶ Definition: Volladdierer
Ein *Volladdierer* ist ein Schaltkreis mit drei Eingängen und zwei Ausgängen. Er liefert die Binärdarstellung der Summe der drei Eingänge.



Hierbei sind c der Übertrag (Carry) und p die Parität.

▶ Definition: Naiver Addierer
Ein *naiver Addierer* ist eine Folge von Volladdierern, bei denen jeweils zwei Input-Bits und das Carry-Bit des vorherigen Volladdierers anliegen.

Zentraler Nachteil: Die Tiefe des Schaltkreises ist $\Omega(n)$.

13.2.2 Der Carry-Look-Ahead-Addierer

13-17

Die drei Effekte, die zwei Bits haben können.

Der naive Addierer braucht lange, da die Carry-Bits hinten einen Einfluss auf das Ergebnis vorne haben können. Bei einem *Carry-Look-Ahead-Addierer* versucht man deshalb, den Einfluss der Carry-Bits schneller zu berechnen. Sei $a_n \dots a_1 b_n \dots b_1$ die Eingabe und betrachten wir zwei Bits a_i und b_i. Ihr Einfluss auf das Carry-Bit lässt sich in drei Klassen unterteilen:

generate Falls beide Bits 1 sind, erzeugen sie *auf jeden Fall ein Carry-Bit*, unabhängig davon, was an den niederwertigeren Stellen passiert.

kill Falls beide Bits 0 sind, erzeugen sie *auf keinen Fall ein Carry-Bit*.

propagate Falls genau ein Bit 0 ist, so *reichen sie das Carry-Bit der niederwertigen Bits einfach durch*.

13-18

Wie die Effekte zusammenwirken.

Nehmen wir nun an, der Effekt zweier Bits a_i und b_i ist g, k oder p und der Effekt von a_{i-1} und b_{i-1} ist ebenfalls g, k oder p. Dann ist der Gesamteffekt:

Effekt von a_i, b_i	Effekt von a_{i-1}, b_{i-1}	Gesamteffekt
g	g	g
g	k	g
g	p	g
p	g	g
p	p	p
p	k	k
k	g	k
k	k	k
k	p	k

13-19

Der Carry-Look-Ahead-Addierer.

Ein Carry-Look-Ahead-Addierer ist wie folgt aufgebaut:

- Zunächst wird parallel in Tiefe $O(1)$ für jedes Bitpaar (a_i, b_i) der Effekt ausgerechnet.
- Dann wird für jede Stelle der Effekt ausgerechnet, den alle Bits haben, die niederwertiger sind.
- Dazu kann man eine *Postfixsumme* berechnen. Dies geht mit einem Schaltkreis der Tiefe $O(\log n)$ und Größe $O(n)$.
- Ist für jede Stelle der Übertrag der vorherigen Stellen bekannt, kann die Ausgabe in Tiefe $O(1)$ und Platz $O(n)$ berechnet werden.

Beispiel

Bitposition i	4	3	2	1	0
a_i		0	1	1	1
b_i		0	0	0	1
Effekt von a_i und b_i		k	p	p	g
Gesamteffekt bis i …		k	g	g	g
… und folglicher Übertrag c_i		0	1	1	1
Summe $a_i + b_i + c_{i-1}$	0	1	0	0	0

13.3 Multiplikation

Die Problemstellung zum Multiplikationsproblem.

13-20

Problemstellung

Für ein gegebenes n wollen wir die Funktion $m_n\colon \{0,1\}^{2n} \to \{0,1\}^{2n}$ berechnen.
Sie multipliziert die ersten n Bits der Eingabe mit den zweiten n Bits und liefert das Ergebnis zurück.

Beispiel

- $m_4(01001101) = 00110100$.
- $m_4(00010001) = 00000001$.

Ziel

Eine NC^1-Schaltkreisfamilie für die Funktion.

Wie ein NC^2-Schaltkreis für die Multiplikation funktioniert.

13-21

1. Mit einem Schaltkreis *konstanter Tiefe* werden n Zahlen ermittelt, deren Summe das gesuchte Produkt ist. (Schulmethode.)
2. Wir wissen schon, dass sich zwei Zahlen in Tiefe $O(\log n)$ addieren lassen.
3. Addieren wir n Zahlen baumartig, so erhalten wir einen Additionsbaum mit der Tiefe $O(\log n)$.
4. Dies liefert eine Gesamttiefe von $O(\log^2 n)$.

Beispiel: Ein Produkt

$1001 \cdot 1101 =$

$00001001 +$
$00000000 +$
$00100100 +$
$01001000 =$

$00001001 +$
$01101100 =$

01110101

13.3.1 Ein Trick

Ein Trick, mit dem man den Schaltkreis flach bekommt.

13-22

Wir können die Addition von zwei Zahlen *nicht beschleunigen*. Wir können aber *schnell aus drei Zahlen zwei Zahlen machen*.

Neue Aufgabenstellung

Eingabe Drei Zahlen a, b, c als Bitfolge.

Ausgabe Bitfolge zweier Zahlen d und e mit $a+b+c = d+e$.

✎ Zur Übung

Überlegen Sie, wie ein »Aus-3-mach-2-Schaltkreis« *konstanter Tiefe* aussieht.

13.3.2 Der Schaltkreis

13-23

Wie ein NC^1-Schaltkreis für die Multiplikation funktioniert.

Aufbau einer NC^1-Multiplikationsschaltkreisfamilie

1. In einer ersten Schicht *konstanter Tiefe* werden aus der Eingabe n Zahlen erzeugt, deren Summe das Ergebnis liefert. (Schulmethode)
2. In den n Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
3. In den verbleibenden $\frac{2}{3}n$ Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
4. In den verbleibenden $\frac{4}{9}n$ Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
5. Und so weiter, bis nur noch zwei Zahlen übrig sind.
6. Diese werden in Tiefe $O(\log n)$ addiert.

13-24

Verdeutlichung der Schritte des Schaltkreises an einem Beispiel.

1. Eingabe $10010111 \cdot 11001111 = ?$
2. Bildung von zu summierenden Zahlen
3. »3 auf 2«-Reduktion (wiederholt)
4. Endsumme = Endprodukt

```
0000000010010111
0000000100101110
0000001001011100
0000010010111000
0000000000000000
0000000000000000
0010010111000000
0100101110000000

0000001111100101
0000000000111100
0000010010111000
0000000000000000
0010010111000000
0100101110000000

0000011101100001
0000000101111000
0110111001000000
0000001100000000

0000011101100001
0000000101111000

0110100001011001
0000111011000000
0000001100000000

0110010110011001
0001010010000000
0111101000011001
```

13.4 Division

13-25

Die Problemstellung zum Divisionsproblem.

Problemstellung

Für ein gegebenes n wollen wir die Funktion $d_n \colon \{0,1\}^{2n} \to \{0,1\}^n$ berechnen.
Sie dividiert die ersten n Bits der Eingabe durch die zweiten n Bits und liefert den ganzzahligen Anteil zurück.

Beispiel

- $d_4(11010010) = 0110$.
- $d_4(00010001) = 0001$.

Ziel

Eine möglichst flache Schaltkreisfamilie für die Division.

13.4.1 Division in modernen Prozessoren

Division beim Itanium Prozessor. 13-26

Literatur

[1] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual* Volume 1, Revision 2.2, 2002.

```
     frcpa.s0 f8,p6 = f6,f7;;
(p6) fnma.s1 f9 = f7,f8,f1 ;;
(p6) fma.s1 f8 = f9,f8,f8
(p6) fma.s1 f9 = f9,f9,f0 ;;
(p6) fma.s1 f8 = f9 ,f8,f8
(p6) fma.s1 f9 = f9,f9,f0 ;;
(p6) fma.s1 f8 = f9,f8,f8 ;;
(p6) fma.d.s1 f9 = f6,f8,f0 ;;
(p6) fnma.d.s1 f6 = f7,f9,f6 ;;
(p6) fma.d.s0 f8 = f6,f8,f9
```

- `frcpa` liefert eine Approximation das Kehrwerts.
- `fma` steht für »floating point multiply and add«.
- `fnma` steht für »floating point negate multiply and add«.

13.4.2 Vorbereitungen

Vorbereitungen: Ein paar Vereinfachungen. 13-27

Es ist im Prinzip *egal*, ob wir mit *Nachkommastellen* rechnen oder nicht – ein einfaches Verschieben des Kommas genügt. Wenn wir mit Kommazahlen rechnen, *dann reicht es* offenbar, einen *Kehrwert zu berechnen*. Das Ergebnis a/b errechnet sich dann als $(1/b) \cdot a$ – und wir wissen schon, wie man schnell multipliziert. Da wir wieder das Komma beliebig verschieben können, wollen wir nun *Kehrwerte von Zahlen zwischen* $1/2$ *und* 1 berechnen. Solche Zahlen haben in Binärschreibweise die Form $0{,}1\ldots$.

Vorbereitungen: Approximative Lösungen und unser neues Ziel. 13-28

▶ Definition: n-Approximation

Sei $c \in \mathbb{R}$ eine Zahl. Eine *n-Approximation* von c ist eine Zahl $\tilde{c}$ mit

$$|c - \tilde{c}| \leq 2^{-n}.$$

Problemstellung

Eingabe n-Bit Zahl d mit $1/2 \leq d \leq 1$.

Ausgabe $2n$-Approximation von $1/d$.

Die *Motivation* für die neue Problemstellung ist, dass für zwei n-Bit Zahlen c und d und für eine $2n$-Approximation d' von $1/d$ gilt: $c \cdot d'$ ist eine n-Approximation von c/d.

13.4.3 Das Newton-Verfahren

13-29 **Eigenschaften des Newton-Verfahrens.**
Das *Newton-Verfahren* geht auf Isaac Newton zurück. Es ist ein *numerisches Verfahren*, um Nullstellen von Funktionen zu bestimmen. Es arbeitet *iterativ*: In jeder Iteration wird die bestehende Approximation (im Allgemeinen sehr stark) verbessert. Wie alle numerischen Verfahren funktioniert es nur unter bestimmten Bedingungen.

13-30 **Ein Satz, der die Konvergenz des Newton-Verfahrens allgemein beschreibt.**

▶ **Satz**
Sei f eine im Intervall $[a,b]$ zweifach stetig differenzierbare Funktion mit $f'(x) \neq 0$ für alle $x \in [a,b]$. Sei

$$\max_{x\in[a,b]} \left| \frac{f(x)\cdot f''(x)}{f'(x)^2} \right| \leq 1.$$

Dann existiert exakt eine Nullstelle x von f im Intervall $[a,b]$ und die Folge

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

konvergiert gegen die Nullstelle für alle $x_0 \in [a,b]$.

13-31 ✎ **Zur Übung**
Wir wollen den Kehrwert einer Zahl d berechnen. Wie lautet eine (sinnvolle) Funktion f, deren Nullstelle gerade dieser Kehrwert ist? Wie lautet die Iterationsvorschrift

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

für Ihre Funktion f?

13-32 **Die Fixpunktiteration für die Division.**
Die Funktion $f(x) = d - 1/x$ hat offenbar die Nullstelle $1/d$. Die Iterationsvorschrift für diese Funktion lautet wegen $f'(x) = 1/x^2$ folglich:

$$x_{n+1} = x_n - \frac{d - 1/x_n}{x_n^2} = x_n - dx_n^2 + x_n = x_n(2 - dx_n).$$

Leider ist die Bedingungen für die Konvergenz der Newtoniteration nur in einer Umgebung des Kehrwertes erfüllt (also für gute anfängliche Näherungen).

13-33 **Satz über die Konvergenz der Iterationsvorschrift für die Division.**

▶ **Satz**
Sei d mit $1/2 \leq d \leq 1$ gegeben. Sei $x_0 = 1$ der feste Startwert und sei

$$x_{k+1} = x_k(2 - dx_k).$$

Dann ist x_k eine $(2^k - 2)$-Approximation von $1/d$.

Beweis. Wir zeigen zwei Behauptungen:

1. Es gilt $x_k = \sum_{i=0}^{2^k-1} (1-d)^i$.
2. Es gilt, dass $\sum_{i=0}^{2k+1} (1-d)^i$ eine $2k$-Approximation von $1/d$ ist.

□

Zur Behauptung $x_k = \sum_{i=0}^{2^k-1}(1-d)^i$. 13-34

Wir zeigen die Behauptung durch Induktion. Der Induktionsanfang für $k = 0$ ist korrekt. Für den Induktionschritt rechnen wir wie folgt:

$$\begin{aligned}
x_{k+1} &= x_k(2-dx_k) = 2x_k - dx_k^2 = 2x_k + \big(-1+(1-d)\big)x_k^2 \\
&= 2x_k - x_k x_k + x_k(1-d)x_k \\
&= 2x_k - x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + x_k(1-d)x_k \\
&= x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + \sum_{i=0}^{2^k-1}(1-d)^{i+1}x_k \\
&= x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + \sum_{i=1}^{2^k}(1-d)^i x_k \\
&= x_k + (1-d)^{2^k}x_k \\
&= \textstyle\sum_{i=0}^{2^k-1}(1-d)^i + (1-d)^{2^k}\sum_{i=0}^{2^k-1}(1-d)^i \\
&= \sum_{i=0}^{2^{k+1}-1}(1-d)^i
\end{aligned}$$

Zur Behauptung, dass $\sum_{i=0}^{2k+1}(1-d)^i$ eine $2k$-Approximation von $1/d$ ist. 13-35

Es gilt:

$$\begin{aligned}
\left|1/d - \sum_{i=0}^{2k+1}(1-d)^i\right| &= \left|\sum_{i=0}^{\infty}(1-d)^i - \sum_{i=0}^{2k+1}(1-d)^i\right| \\
&= \sum_{i=2k+2}^{\infty}(1-d)^i \le \sum_{i=2k+2}^{\infty}\frac{1}{2^i} \\
&= 2^{-2k-1} < 2^{-2k}.
\end{aligned}$$

13.4.4 Der Schaltkreis

Von der Iteration zum Schaltkreis. 13-36

▶ Satz

Es gibt eine NC^2*-Schaltkreisfamilie für die Division.*

Beweis.

1. Für gegebene Eingaben a und b wird erstmal das »Komma bei b zurechtgerückt«.
2. Dann wenden wir $\log n$ mal die Iterationsvorschrift $x_{k+1} = x_k(2-dx_k)$ an. Jede dieser $\log n$ Iteration benötigt Tiefe $O(\log n)$.
3. Dann multiplizieren wir das Ergebnis mit a, was in Tiefe $O(\log n)$ geht.
4. Die Gesamttiefe ist also $O(\log^2 n)$. □

Bemerkung: Es gibt einen (wahnsinnig komplizierten) NC^1-Schaltkreis für die Division.

Zusammenfassung dieses Kapitels

13-37

▶ Schaltkreisfamilien
Eine *Schaltkreisfamilie* enthält für jedes n genau einen Schaltkreis mit genau n Eingabegattern.

▶ AC^i-Klassen und NC^i-Klassen
Eine Sprache liegt in AC^i, wenn ihre charakteristische Funktion von einer Schaltkreisfamilie berechnet wird

1. polynomieller Größe und
2. Tiefe $O(\log^i n)$.

Die Sprache liegt in NC^i, wenn *zusätzlich*

3. der Fan-In maximal Zwei ist.

▶ Addition
(Das Entscheidungsproblem zur) Addition liegt in NC^1.

▶ Multiplikation
(Das Entscheidungsproblem zur) Multiplikation liegt in NC^1.

▶ Division
(Das Entscheidungsproblem zur) Division liegt in NC^2.

▶ Ausblick

- Es ist leicht zu zeigen, dass die Addition sogar in AC^0 liegt.
- Es schwer zu zeigen, dass die Division in NC^1 liegt.
- Es schwer zu zeigen, dass Multiplikation und Division *nicht* in AC^0 liegen.

Übungen zu diesem Kapitel

Übung 13.1 Konzept der approximativen Lösung, leicht

Gegeben sind m Zahlen $\tilde{a}_1, \ldots, \tilde{a}_m$, die n-Approximationen der Zahlen $a_1, \ldots, a_m$ sind.

1. Bestimmen Sie ein möglichst großes k, so dass $\sum_{i=1}^{m} \tilde{a}_i$ eine k-Approximation für $\sum_{i=1}^{m} a_i$ ist.
2. Beweisen Sie Ihre Behauptung aus dem ersten Teil.

Übung 13.2 Schaltkreisfamilie für Palindromsprache, mittel, mit Lösung

Zeigen Sie, dass die Sprache der Palindrome über dem Alphabet $\{0,1\}$ in der Schaltkreisklasse AC^0 liegt.

In den folgenden beiden Aufgaben geht es darum, den 8-Bit Carry-Look-Ahead-Addierer im Detail zu modellieren. Ein Addierer weist prizipiell drei Arten von großen (zusammengesetzten) Gattern auf:

- Die ersten Gatter nehmen zwei Eingangsbits und produzieren daraus (kodiert) eine der drei Ausgaben g, p oder k (für *kill, generate, propagate*).
- Die zweiten Gatter nehmen nehmen zwei Codes von g, p oder k und liefern als Ausgaben den Gesamteffekt g, p oder k nach der Tabelle 13-18.
- Die dritten Gatter sind Volladdierer.

Übung 13.3 Konkretes Aussehen der Gatter, mittel, mit Lösung

Geben Sie für jede der drei oben aufgeführten großen Gatterarten die konkrete Schaltung mittels $\wedge$-, $\vee$- und $\neg$-Gattern an.

Übung 13.4 Postfix-Addition, mittel

Überlegen Sie, wie die Verdrahtung im Falle einer Post-Fix-Addition aussieht, und geben Sie die Verschaltung des gesamten Schaltkreises mittels der großen Gatter an.

Übung 13.5 Die Newton-Iteration, leicht

Führen Sie für die Werte $a = \frac{2}{3}$, $b = \frac{3}{4}$ und $c = 0{,}9$ die Newton-Iteration zur Bestimmung des Kehrwerts durch. Geben Sie jeweils die ersten vier Werte der Iteration exakt an und bestimmen Sie, auf wie viele Dezimalstellen dies genau ist.

Übung 13.6 Konzept der approximativen Lösung, leicht

Gegeben sind m Zahlen $\tilde{a}_1, \ldots, \tilde{a}_m$, die n-Approximationen der Zahlen $a_1, \ldots, a_m$ sind.

1. Bestimmen Sie ein möglichst großes k, so dass $\sum_{i=1}^{m} \tilde{a}_i$ eine k-Approximation für $\sum_{i=1}^{m} a_i$ ist.
2. Beweisen Sie Ihre Behauptung aus dem ersten Teil.

Übung 13.7 Schaltkreisfamilie für Paritysprache, mittel

Die Sprache PARITY besteht genau aus den Wörtern über dem Alphabet $\{0, 1\}$, die eine ungerade Anzahl von 1en enthalten. Zeigen Sie, dass PARITY $\in \mathrm{NC}^1$.

Übung 13.8 Schnelle Addition, schwer

Zeigen Sie, dass es eine AC^0-Schaltkreisfamilie für die Addition gibt.

Tipps: Gehen Sie zunächst wie bei der NC^1-Schaltkreisfamilie vor, vermeiden Sie dann aber den Baum zur Berechnung des Effektes.

Übung 13.9 Iterationsfunktion zur Wurzelbestimmung, leicht

Betrachten Sie die Funktion $f(x) = x^2 - d$, die für die Wurzelberechnung einer Zahl d verwendet werden kann. Konstruieren Sie ausgehend von f die Iterationsvorschrift der Folge x_n, und berechnen Sie konkret die ersten vier Glieder der Folge für die Wurzel von 2. Nehmen sie hierbei den Startwert $x_0 = 2$ an. Auf wie viele Dezimalstellen sind die berechneten Werte genau?

Übung 13.10 Konvergenz der Iterationsfunktion, mittel

Beweisen Sie, dass die Folge x_n quadratisch gegen den genauen Wert der Wurzel konvergiert. Sie sollen also beweisen, dass für jede Zahl d, eine Konstante c und alle $n \geq 2$ Folgendes gilt:

$$|x_{n+1} - \sqrt{d}| \leq c|x_n - \sqrt{d}|^2$$

Hierbei dürfen Sie davon ausgehen, dass $x_n \geq \sqrt{d}$ für alle $n \geq 2$ ist. Folgern Sie, dass ein NC^3-Schaltkreis für die Berechnung der Wurzelfunktion existiert.

Übung 13.11 Verbesserung des Wurzel-Algorithmus, mittel

In dieser Aufgabe wollen wir einen NC^2-Schaltkreis für die Wurzelbestimmung angeben. Betrachten Sie hierfür die Funktion $g(x) = \frac{1}{x^2} - d$. Entwickeln Sie aus g eine Iterationvorschrift für eine Folge y_n, die gegen den inversen Wert von $\sqrt{d}$ konvergiert. Sie dürfen hierbei davon ausgehen, dass die Folge y_n quadratische Konvergenz aufweist. Wie können Sie hieraus einen NC^2-Schaltkreis für die Wurzelberechnung von d konstruieren?

Teil IV
Untere Schranken

Das Motto des vorherigen Teils könnte sein – frei nach Huxley – *Oh brave new parallel world, that has such algorithms in it!* In diesem Teil wird die allgemeine Euphorie etwas gedämpft werden, denn es soll um die Frage gehen, was sich *nicht* parallelisieren lässt, ob es also untere Schranken für die Parallelisierbarkeit von Problemen gibt.

Versucht man, untere Schranken für die Rechenzeit von parallelen Programmen zu beweisen, hat man schnell mit den gleichen Problemen zu kämpfen wie bei der P-NP-Frage: Man schafft es nicht. Anstatt das aber einfach zuzugeben und sich eine ehrliche Arbeit zu suchen, forschen Theoretiker trotzdem munter weiter daran herum. In Bezug auf die P-NP-Frage sind dabei Resultate der folgenden Art herausgekommen: »Ehrlich gesagt wissen nicht, ob das Erfüllbarkeitsproblem schwierig ist, aber wir wissen ganz sicher, dass es in NP nicht *noch schwierigere* Problem gibt.« Ganz ähnliche Sachen lassen sich in Bezug auf Parallelisierbarkeit aussagen: »Niemand weiß, ob Schaltkreisauswertung parallelisierbar ist, aber wir wissen ganz sicher, dass es in P nicht *noch schlechter parallelisierbare* Probleme gibt.«

Ein Hoffnungsschimmer in Bezug auf echte untere Schranken bleibt jedoch: Wir werden untere Schranken für einige Probleme bei PRAMs zeigen können. Jedoch erweist sich auch dieser beim genaueren Hinschauen als eher dürftig: Wir müssen »etwas schummeln« und das Maschinenmodell verändern und einschränken.

14-1

Kapitel 14
Untere Schranken – Adversaries
Was definitiv nicht parallel geht

14-2

Lernziele dieses Kapitels

1. Konzepte des Vergleichsbaums und des Adversary-Arguments kennen
2. Untere-Schranken-Beweise für Comparison-PRAMs kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn Sie einen kommerziell erfolgreichen Film drehen wollen, brauchen sie (a) eine Love-Story (heteronormativ und -stereotyp mit heroischem maskulinen Part und erotischem femininen Part), (b) einen möglichst fiesen Bösewicht und natürlich (c) ein Special-Effects-Studio samt Renderfarm. Uns soll in diesem Kapitel lediglich Teil b interessieren: Der böse Widersacher, der dem Helden gerne die Schau stiehlt (der Bösewicht ist fast immer männlich, es sei denn er ist weiblich oder – wie »das Böse« im Fünften Element – einfach so böse, dass kein Geschlecht mehr zugeordnet werden kann). Das Techtelmechtel von Neo und Trinity wäre ohne Mr. Smith nicht der Rede wert; Luke würde ohne seinen Vater wahrscheinlich Saufkumpane von Herrn Hutt sein; James würde ohne Goldfinger die Hotelbar nicht mehr verlassen; Shrek würde ohne Prinz Charming friedlich in seinem Sumpf leben, anstatt sich durch eine dritte Fortsetzung quälen zu müssen.

In der Theoretischen Informatik heißt der böse Widersacher schlicht »der Adversary«. Wie es sich für einen Bösewicht gehört, sollte man ihn eigentlich nicht mögen, ohne ihn geht es aber nicht. Zur Erinnerung: In einem Unteren-Schranken-Beweis geht es darum zu zeigen, dass kein Algorithmus ein bestimmtes Problem in einer bestimmten Zeit lösen kann. Der Adversary sieht seine Aufgabe gerade darin, es »jedem Algorithmus« möglichst schwer zu machen. Er wird immer gerade die Eingabe auswählen, die besonders viele Probleme bereitet. Er lässt den Algorithmus sogar anfänglich im Unklaren darüber, wie die Eingabe überhaupt genau lautet; er liefert immer »in letzter Minute« Informationen, die dann auch noch für den Algorithmus möglichst ungünstig ausfallen. Deshalb ist es der Adversary, der uns hilft, untere Schranke zu zeigen: Aufgrund seiner unermüdlichen Suche nach »fiesen Eingaben« werden wir beispielsweise zeigen können, dass sich das Maximum von n Zahlen (unter einigen Zusatzvoraussetzungen) nicht schneller als in Zeit $\Omega(\log\log n)$ berechnen lässt – was übrigens auch unsere obere Schranke für dieses Problem war.

14.1 Starke Laufzeitschranken

14-4

Laufzeitschranken für PRAMs.

Für bestimmte Arten von PRAMs und bestimmte Problem lassen sich *untere Schranken für die Laufzeit* beweisen. Wir werden zeigen, dass *Comparison*-PRAMs mit p Einheiten zum Finden eines Elementes in einer *sortierten Liste der Länge* n mindestens $\Omega(\log n/\log p)$ Schritte benötigt. Wir werden zeigen, dass *Comparison*-PRAMs Maxima bestenfalls in Zeit $\Omega(\log\log n)$ berechnen können.

14-5

»Starke« untere Schranken für die Laufzeit von PRAMs.

Die Resultate für Comparison-PRAMs sind *starke* Resultate – sie gelten absolut und ohne Annahmen. Wir beweisen sie durch *Adversary-Argumente*. Wir »erkaufen« die starken Resultate dadurch, dass unser Modell (die Comparison-PRAM) nicht sonderlich mächtig ist.

14.2 Maschinenmodell

14-6

Das Modell: Die Comparison-PRAM.

Es hat *noch niemand geschafft*, (nicht triviale) untere Schranke für PRAMs zu beweisen. Wir betrachten deshalb ein *schwächeres* Modell von PRAMs: Die *Comparison*-PRAM.

▶ **Definition:** Comparison-PRAM

Eine *Comparison*-PRAM greift auf die Eingabespeicherzellen nur auf folgende Weise zu:

- Sie kann zwei Eingabezellen vergleichen.
- Sie kann eine Eingabezelle mit einem gegebenen Wert vergleichen.
- Sie erfährt, wie der Vergleich ausgegangen ist.
- Anderweitig darf sie *nicht* in die Zellen »hineinschauen«.

Skript

Für an Formalisierungen Interessierte sei hier im Skript noch eine genauere formale Definition der Comparison-PRAM angegeben:

▶ **Definition:** Syntax der Comparison-PRAM

Syntaktisch ist eine Comparison-PRAM eine normale PRAM mit einem zusätzlichen Befehl:

18. ***if*** GRRi $<$ GRRj ***goto*** k

▶ **Definition:** Semantik der Comparison-PRAM

Die Semantik einer Comparison-PRAM ist genauso definiert wie die Semantik einer PRAM mit folgenden Erweiterungen:

1. Die Semantik des Befehls ***if*** GRRi $<$ GRRj ***goto*** k lautet wie erwartet: $\langle \text{PC}\rangle_{t+1}^p = k$, falls $\langle \text{GR}\langle \text{R}i\rangle_t^p\rangle_t < \langle \text{GR}\langle \text{R}j\rangle_t^p\rangle_t$, sonst $\langle \text{PC}\rangle_{t+1}^p = \langle \text{PC}\rangle_t^p + 1$.
2. Die Register GR1 bis GRn, wobei n die in GR0 gespeicherte Eingabelänge ist, können nur durch obigen Befehl adressiert werden. Eine Berechnung, bei der durch einen anderen Befehl auf eines dieser Register zugegriffen wird (lesend oder schreibend), endet sofort. Dies wäre zum Beispiel der Fall bei $n = 2$ und dem Befehl GR1 $\leftarrow$ R1 oder bei $n = 6$ und dem Befehl R2 $\leftarrow$ GRR8 mit $\langle \text{R8}\rangle_t^p = 5$.

14.3 Methoden

14.3.1 Decision-Trees

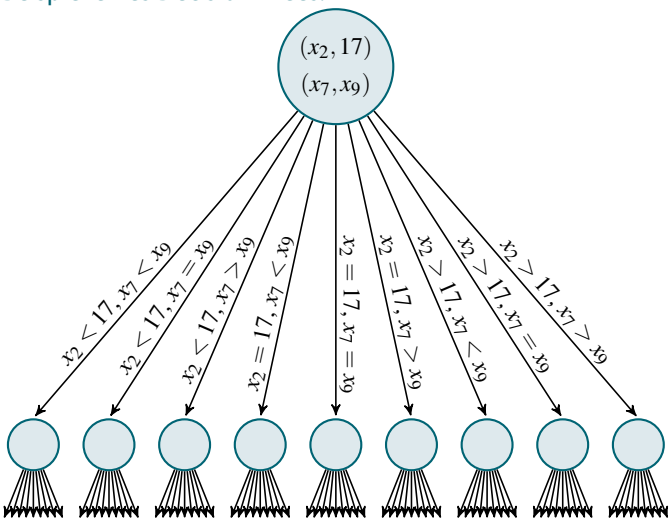
Unser zentrales Mittel zum Beweis unterer Schranken für Comparison-PRAMs. 14-7

Bei *vergleichsbasierten* Modellen sind *Vergleichsbäume* oft nützlich. Vergleichsbäume werden beispielsweise bei dem Beweis verwendet, dass Sortieren Zeit $\Omega(n \log n)$ benötigt.

▶ Definition: Paralleler Vergleichsbaum vom Grad d

- Ein *paralleler Vergleichsbaum vom Grad d* ist ein Baum, in dem jeder innere Knoten (Nicht-Blatt-Knoten) genau 3^d Kinder hat.
  Dabei gibt es je ein Kind für jedes mögliches Ergebnis von d Vergleichsoperationen, bei denen als Ergebnis $<$, $=$ oder $>$ herauskommen kann.
- Die Blätter sind *mit Ausgaben gelabelt*.
- Das *Ergebnis* eines Baumes für eine Eingabe ist definiert als das Label des Blattes, das man beim Durchschreiten des Baumes »entsprechend der Eingabe« erreicht.

Beispiel eines Decision-Trees. 14-8



Das Verhältnis von Comparison-PRAMs und parallelen Vergleichsbäumen. 14-9

Beobachtung

Für jede Comparison-PRAM *mit p Einheiten*, die *s Schritte rechnet*, gibt es einen parallelen Vergleichsbaum *vom Grad p* und der *Tiefe s*, der auf allen Eingaben das gleiche Ergebnis liefert.

▶ Folgerung

1. *Kann ein Problem von einer Comparison-PRAM mit p Einheiten in Zeit s gelöst werden, so gibt es auch einen parallelen Vergleichsbaum vom Grad p und Tiefe s für das Problem.*
2. *Hat jeder parallele Vergleichsbaum vom Grad p für ein Problem Tiefe mindestens s, so braucht eine Comparison-PRAM bei p Einheiten mindestens Zeit s.*

14.3.2 Adversary-Argumente

Allheilmittel bei unteren Schranken: Adversary-Argumente. 14-10

Es gibt ein *generelles Verfahren*, das bei unteren Schranken oft zum Einsatz kommt: Adversary-Argumente. Ein *Adversary* ist ein Gegenspieler, der versucht, einer Maschine das Leben möglichst schwer zu machen. Für Untere-Schranken-Beweise bedeutet das, dass der Adversary immer *gerade die Lösungen auswählt, die besonders lange brauchen*.

14.4 Untere Schranken

14.4.1 Suchen

14-11

Untere Schranke für das Suchen.

▶ Satz

*Suchen in sortierten Listen benötigt auf einer Comparison-*PRAM *mit p Einheiten eine Zeit von $\Theta(\log n/\log p)$.*

Beweis. Für die *obere Schranke* müssen wir einfach einen Algorithmus angeben. Grobe Idee: Teile die Liste in Blöcke der Größe n/p auf; jede Einheit überprüft für einen Block, ob der gesuchte Wert in seinem Block ist; genau ein Block gewinnt, dort wiederholt man die Suche rekursiv. Für die *untere Schranke* zeigen wir, dass jeder Vergleichsbaum vom Grad p mindestens Tiefe $\Omega(\log n/\log p)$ hat. □

14-12

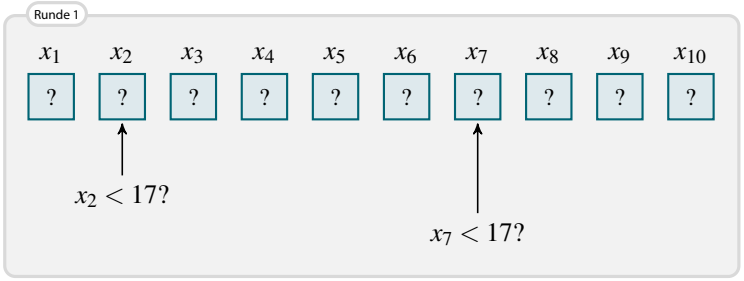
Vorgehen des Adversary.

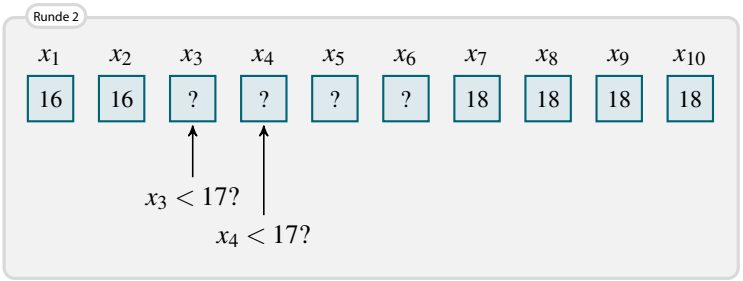
Der Adversary baut eine Eingabe der Länge n, so dass ein korrekter Vergleichsbaum wenigstens Tiefe $\log n/\log p$ haben muss:

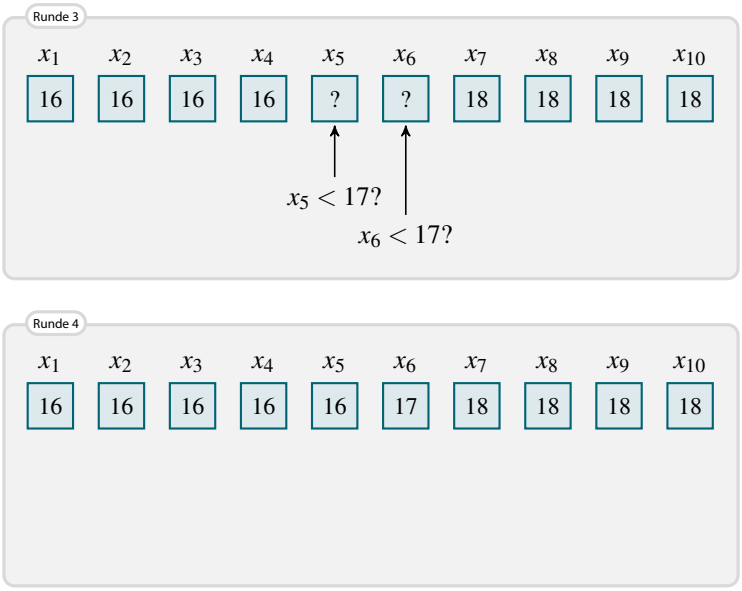
- Er betrachtet die Wurzel und die p Eingaben x_{i_1} bis x_{i_p}, die mit einer Zahl verglichen werden.
- Diese teilen die n Eingaben in Intervalle. Das größte von ihnen muss dann eine Größe von grob n/p haben (genauer $(n-p)/(p-1)$).
- Er legt die Eingabe so fest, dass der gesuchte Wert gerade in diesem Intervall landet. *Genauer legt er sich noch nicht fest.*
- Er betrachtet nun das Kind der Wurzel, das den Vergleichsergebnissen für das Intervall entspricht.
- Wieder wird dieses in p Intervalle geteilt, von denen eines mindestens Größe n/p^2 haben muss.
- Nun legt sich der Adversary weitere Werte fest, so dass der gesuchte Wert in diesem Intervall landet.
- Und so weiter.

14-13

Wo ist die 17?

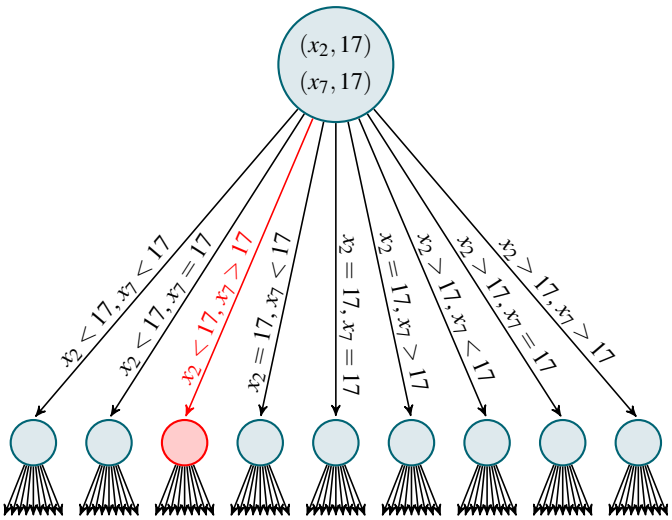






Beispielvorgehen des Adversarys. 14-14

Für $n = 10$ ist das größte Intervall zwischen den Indizes 2 und 7.



Zur Tiefe eines Vergleichsbaumes für die Suche. 14-15

Beobachtungen

1. So lange das größte Intervall, das der Adversary findet, noch Größe mindestens 2 hat, kann in dem Baum noch kein Blatt sein.
2. Für die Tiefe s des Baumes gilt also (grob) $n/p^s \leq 1$, beziehungsweise $\log n / \log p \leq s$.

14.4.2 Maxima finden

Untere Schranke für Maxima. 14-16

▶ Satz

*Das Finden von Maxima in unsortierten Listen benötigt auf einer Comparison-*PRAM *mit n Einheiten Zeit* $\Theta(\log \log n)$.

Beweis. Für die *obere Schranke* sei an den Algorithmus mit den doppelt-logarithmischen Bäumen erinnert. Für die *untere Schranke* benutzen wir wieder ein Adversary-Argument. □

14-17

Vorgehen des Adversarys.

Hauptziel des Adversarys

Der Adversary versucht zu *verheimlichen, welches Element das Maximum ist*. Dazu versucht er, in jeder Runde r eine möglichst große Menge M_r von Knoten zu finden, die alle folgende Eigenschaft haben: *Keiner der bisherigen Vergleiche kann ausschließen, dass dieser Knoten das Maximum ist.* Solange M_r noch mindestens zwei Elemente enthält, kann der Algorithmus noch nicht fertig sein.

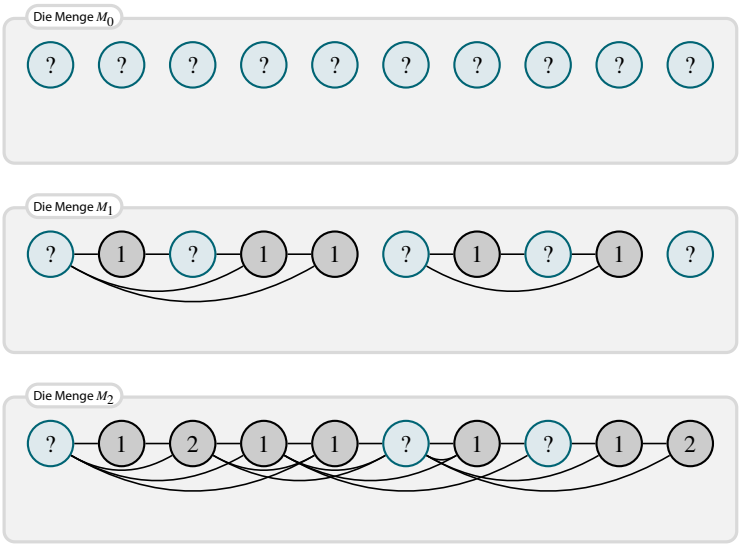
Aktionen des Adversarys in Runde r

1. Identifikation der Menge M_r.
2. Für alle Listenelemente *außerhalb* M_r, für er sich *noch nicht festgelegt* hat, legt sich der Adversary auf den Wert r fest.

14-18

Identifikation der Menge M_r.

- Der Adversary baut einen Graphen, dessen Knotenmenge gerade die Listenelemente in M_{r-1} sind (wobei M_0 alle Listenelemente sind).
- Er zieht eine Kante in dem Graphen, wenn zwei Elemente verglichen wurden.
- Nach dem Satz von Turan (Graphen mit a Knoten und b Kanten enthalten $a^2/(2b+a)$ unabhängige Knoten) gibt es eine unabhängige Menge der Größe $|M_{r-1}|^2/(2n+|M_{r-1}|)$.
- Als M_r wählt er diese Menge.



14-19

Analyse der Größe der unabhängigen Menge.

Die Rekursionsformel lautet $|M_r| \geq |M_{r-1}|^2/(2n+|M_{r-1}|) \geq |M_{r-1}|^2/(3n)$. Also:

- $|M_0| = n$
- $|M_1| \geq n^2 \frac{1}{3n} = n/3$
- $|M_2| \geq \frac{n^2}{3^2} \frac{1}{3n} = n/3^3$
- $|M_3| \geq \frac{n^2}{3^6} \frac{1}{3n} = n/3^7$
- $|M_4| \geq \frac{n^2}{3^{14}} \frac{1}{3n} = n/3^{15}$
- $|M_r| \geq n/3^{2^r-1}$

Wir erreichen also $|M_r| = 1$ erst für $r \in \Omega(\log\log n)$.

Zusammenfassung dieses Kapitels

14-20

1. *Suchen in sortierten Listen* mittels Comparison-PRAMs dauert $\Theta(\log n/\log p)$.
2. *Maximum-Finden in unsortierten Listen* mittels n-Einheiten-Comparison-PRAMs dauert $\Theta(\log\log n)$.
3. Dies beweist man mittels *Adversary-Argumenten*.

Übungen zu diesem Kapitel

Übung 14.1 Untere Schranke für das Finden des Maximums, leicht

1. Beweisen Sie mit Hilfe eines Adversaries, dass es keinen sequentiellen Algorithmus zum Bestimmen des Maximums einer Liste von n beliebigen Zahlen mit einer Laufzeit von $o(n)$ geben kann.
2. Lösen Sie die vorherige Aufgabe mit der zusätzlichen Annahme, dass in der Liste nur die Zahlen 1, 2 und 3 vorkommen.

Übung 14.2 Untere Schranke für MAJORITY, schwer

Die Funktion MAJORITY sei so definiert, dass sie als Eingabe eine Liste von natürlichen Zahlen erhält und die Ausgabe eine Zahl aus der Liste ist, die am häufigsten vorkommt.

1. Bestimmen Sie mit Hilfe eines Adversaries die untere Schranke für die Laufzeit eines Algorithmus, der die Funktion MAJORITY berechnet.
2. Lösen Sie die vorherige Aufgabe mit der zusätzlichen Annahme, dass in der Liste der n Elemente maximal $k < n$ verschiedene Zahlen vorkommen.
3. Nehmen Sie an, Sie können auf die Elemente der Liste nur mittels eines Operators zugreifen, der Ihnen für ein Paar von Positionen eine der drei Informationen $<, =, >$ zurückgibt. Mit wievielen Vergleichen kommen Sie aus, um MAJORITY zu lösen?

Übung 14.3 Vergleichsbasiertes Sortieren, mittel

Zeigen Sie, dass es keinen Sortieralgorithmus gibt, der eine Liste von n beliebigen Zahlen in Zeit $o(n \log n)$ sortieren kann.

Übung 14.4 Untere Schranke für Medianbestimmung, schwer

1. Bestimmen Sie mittels eines Adversaries die untere Schranke für die Laufzeit eines Algorithmus, der den Median einer Liste von n natürlichen Zahlen berechnet.
2. Lösen Sie die vorherige Aufgabe mit der zusätzlichen Annahme, dass es maximal drei verschiedene Zahlen in der Liste gibt.
3. Wie kann der Median in Linearzeit bestimmt werden (Stichwort Selektionsalgorithmen)?

Übung 14.5 Sonderfälle des Sortierens, mittel

1. Wie kann eine Liste von n Binärstrings der Länge k in $O(n \cdot k)$ sortiert werden? Warum gilt hier nicht die untere Schrank von $O(n \log n)$, die für das Sortieren im Allgemeinen gilt?
2. Sie bekommen eine Liste von Nullen und Einsen der Länge n, die sortiert werden soll. Auf die Elemente der Liste können Sie nur mittels eines Operators zugreifen, der Ihnen für ein Paar von Positionen eine der drei Informationen $<, =, >$ zurückgibt. Wieviele dieser Vergleiche brauchen Sie mindestens, um die Liste zu sortieren?

Anhang

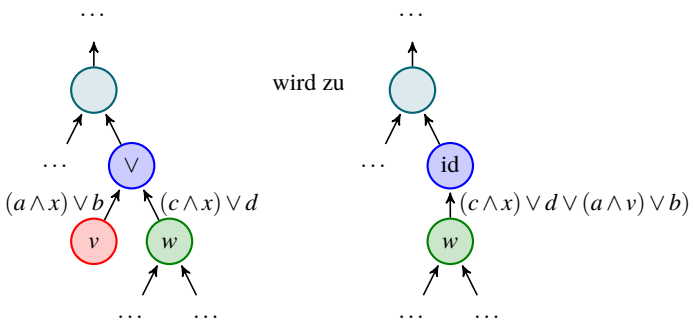
Lösungen

Beispiellösungen zu ausgesuchten Übungen

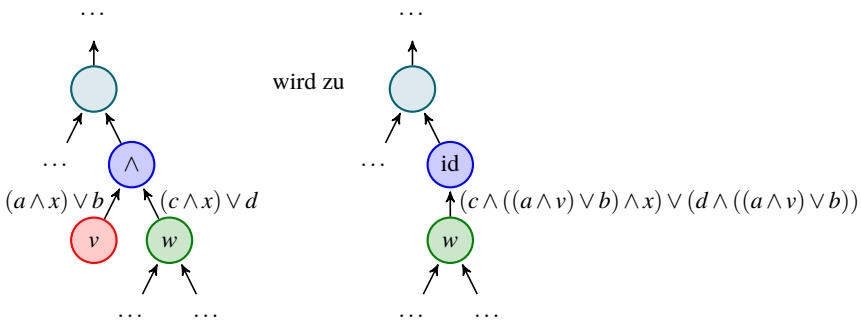
Lösung zu 12.1

Wir benutzen Algorithmus *Evaluate* von Seite 12-27 mit abgeänderten Linearformen und abgeänderter *Rake*-Prozedur. Die neuen Linearformen haben die Form $(a \wedge x) \vee b$ für $a, b \in \{0, 1\}$. Die neue *Rake*-Prozedur benutzt statt Multiplikation das boolesche *and* und statt Addition das boolesche *or*. Das sieht wie folgt aus:

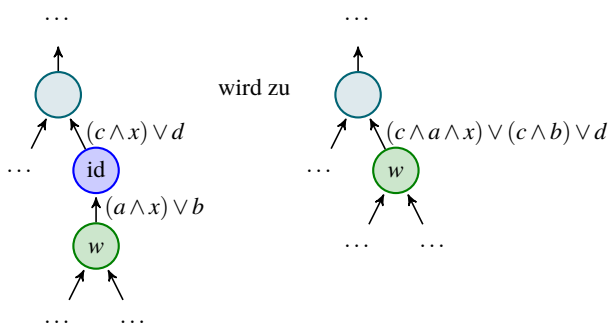
- Löschen von Blatt v wenn Elternknoten u ist ein *Or*-Knoten ist:



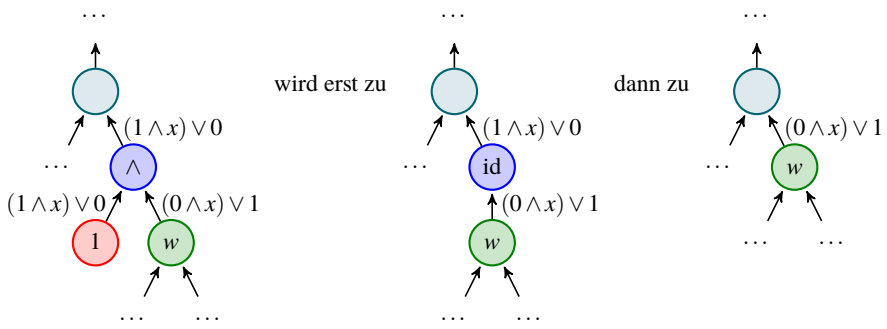
- Löschen von Blatt v wenn Elternknoten u ist ein *Or*-Knoten ist:



- Anschließendes Löschen des entstehenden *Id*-Knotens:



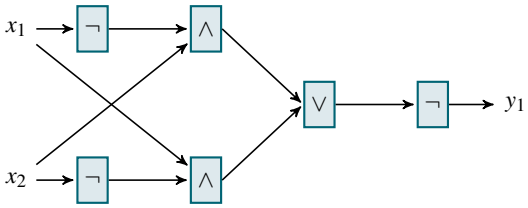
Ein Beispiel:



Lösung zu 13.2

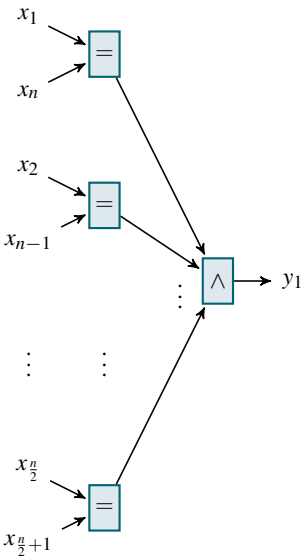
Die Klasse AC^0 enthält aller Sprachen, die von Schaltkreisfamilien der Tiefe $O(\log^0 n) = O(1)$ (also konstant) und polynomieller Größe erkannt werden. Der Fan-In kann dabei beliebig groß sein.

Um so eine Schaltkreisfamilie zu entwerfen, implementieren wir erst ein Schaltkreis der die $=$-Funktion berechnet:

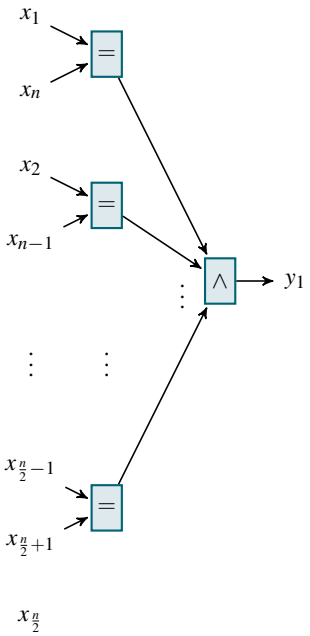


Diesen Schaltkreis konstanter Tiefe und Größe nutzen wir jetzt als $=$-Gatter. Für jedes n entwerfen wir einen Schaltkreis:

- Für gerade n:



- Für ungerade n:

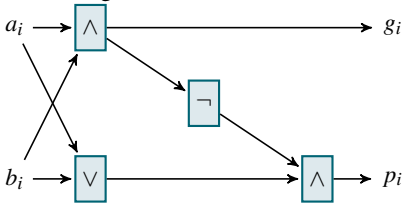


Die Schaltkreise haben konstante Tiefe und Größe $O(n)$ und erkennen genau die Palindrome, also ist die Sprache der Palindrome in AC^0.

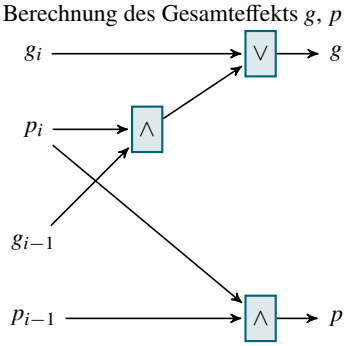
Lösung zu 13.3

Kodiere g, p und k durch zwei Bits g und p. Dabei bedeutet $g = p = 0$, dass $k = 1$, und $g = p = 1$ kommt nicht vor. Wir benutzen nur Gatter mit Fan-In kleiner oder gleich 2, damit wir Schaltkreise der NC-Familie bekommen (Addition liegt ja in NC^1).

- Berechnung des Effekts zweier Bits:



- Berechnung des Gesamteffekts g, p aus den Effekten g_i, p_i, g_{i-1}, p_{i-1}:



- Jetzt der Volladdierer. Dabei ist c_i der Übertrag und s_i die Parität, also dass i-te Bit der gesuchten Summe: