Fachhochschule Südwestfalen
University of Applied Sciences
Standort Hagen
Fachbereich Technische Betriebswirtschaft

Funktionale Programmierung und Streams in Java

Unterlagen für das 3. Semester Wirtschaftsinformatik

Andreas de Vries

Version: 30. Januar 2017

Dieses Skript unterliegt der *Creative Commons License* 3.0
(http://creativecommons.org/licenses/by-nc/3.0/deed.de)

Inhaltsverzeichnis

1 Funktionale Programmierung **3**
1.1 Lambda-Ausdrücke 4
1.2 Methodenreferenzen 9
1.3 Ist Java echt funktional? 9

2 Streams: Nebenläufigkeit mit Collections **11**
2.1 Sequenzielle Programme und Mehrkernarchitekturen 11
2.2 Datenströme und Pipelines 12
2.2.1 Auswertungen von Datenströmen: lazy und eager 13
2.3 Streams in Java 13
2.3.1 Arten von Streams 13
2.3.2 Stream-Operationen 14
2.3.3 Map und Reduce 16
2.3.4 Reduce auf parallelisierten Streams 17
2.3.5 Zustandslose Operationen 21

3 Das Fork-join-Modell und parallele Algorithmen **22**
3.1 Fork-join 22
3.2 Präfixsummen 23
3.3 Reduce-Joins 25

4 Übungsaufgaben **26**

A Hintergrund und Vertiefung **28**
A.1 Amdahl’sches und Gustafson’sches Gesetz 28
A.2 Rückruffunktionen 30
A.3 Anmerkungen zur Parallelisierung in Java 8 31

Literaturverzeichnis **33**

Kapitel 1

Funktionale Programmierung

> *Wenn wir in zehn Jahren unsere Hardware als Entwickler noch auslasten wollen, wenn wir 100 Cores am Desktop-Rechner haben wollen, dann müssen wir an unseren Programmiermodellen grundsätzlich was ändern.*
>
> Klaus Alfert auf der W-JAX 09

In einer funktionalen Programmiersprache können Funktionen definiert werden und es gibt keinen Unterschied zwischen Daten und Funktionen. Funktionen können also auch in Variablen gespeichert und als Parameter übergeben werden [10, S. 6]. Die funktionale Programmierung steht damit im Gegensatz zur *imperativen Programmierung*, denn die Trennung von Daten und Programmlogik wird aufgehoben und jede Variable kann nur einmal einen Wert bekommen (`final` in Java). Rein funktionale Sprachen kennen sogar überhaupt keine Variablen oder Schleifen, denn ein Algorithmus wird hier durch eine Abfolge von Funktionsaufrufen ausgeführt; für Wiederholungen werden also keine Schleifen verwandt, sondern ausschließlich Rekursionen. So können mathematisch unsinnige Ausdrücke der imperativen Programmierung wie `x=x+1` gar nicht erst auftreten.

Warum aber überhaupt funktionale Programmierung? In objektorientierten Sprachen ist eine Methoden immer eindeutig einer Klasse zugewiesen. Um Funktionalität aufzurufen, muss eine Methode entweder statisch sein (was an sich bereits der reinen Objektorientierung widerspricht) oder eine Objektinstanz muss zuvor erzeugt werden. Die Methode wird dann mit *Daten* als Parameter aufgerufen. Will man jedoch die *Funktionalität* als Parameter übergeben, so kommt man an funktionaler Programmierung nicht vorbei. Auch führt die daten- und objektorientierte Programmierung zu erheblichen Problemen bei parallelen Programmabläufen auf Mehrkernprozessoren oder Rechnerclustern. Da in der rein funktionalen Programmierung Variablen als veränderbare Datenspeicher nicht existieren, kann es keine Nebeneffekte *(side effect)* geben, und somit auch keine Synchronisationsprobleme bei zentralen Speichern *(shared memory)*.

Die funktionale Programmierung widerspricht grundsätzlich dem objektorientierten Paradigma, gemäß dem Objekte im Laufe ihres Lebenszyklus verschiedene Zustände annehmen können. Da ein Zustand in der Regel durch die jeweils aktuelle Belegung der Attribute des Objekts bestimmt ist, müssen sie in diesem Fall veränderlich sein. Java als objektorientierte Sprache wird damit nie eine rein funktionale Sprache sein können. Allerdings ermöglicht Java seit der Version 8 zumindest funktionale Syntax, nämlich durch Lambda-Ausdrücke und Methodenreferenzen. Als eine gute Einführung dazu sei [11] empfohlen.

1.1 Lambda-Ausdrücke

Ein *Lambda-Ausdruck*

```
(<T1> x, <T2> y)  ->  x*x + y*y;
```

Parameterliste — Funktionsrumpf

besteht aus einer Parameterliste mit Datentypen in runden Klammern, dem Pfeil-Symbol `->` und dem Funktionsrumpf. Kann der Compiler den Datentyp der Parameter erkennen so spricht man von *impliziter Typbestimmung*. In diesem FallG können die Datentypen weggelassen werden. Hat man zusätzlich nur einen Parameter, so können auch die Klammern weggelassen werden:

```
x -> x*x + 1;
```

Man kann ebenso das reservierte Wort `return` verwenden, allerdings muss der Funktionsrumpf dann in geschweifte Klammern gesetzt werden:

```
(x) -> {return x*x + 1;};
```

Die Syntax in Java für Lambda-Ausdrücke ist insgesamt recht vielfältig, eine Übersicht ist in Tabelle 1.1 aufgelistet.

Syntax des Lambda-Ausdrucks	**Regel**
Parameterlisten	
`(int x)            -> x + 1`	Parameterliste mit einem Parameter und expliziter Typangabe
`int x              -> x + 1`	**Falsch:** Parameterliste mit Typangabe stets in Klammern!
`(x)                -> x + 1`	Parameterliste mit einem Parameter ohne explizite Typangabe
`x                  -> x + 1`	Parameterliste mit einem Parameter ohne explizite Typangabe: Klammern dürfen weggelassen werden
`(int x, short y) -> x + y`	Parameterliste mit zwei Parametern und expliziten Typangaben
`int x, short y   -> x + y`	**Falsch:** Parameterliste mit Typangaben stets in Klammern!
`(x, y)             -> x + y`	Parameterliste mit zwei Parametern ohne explizite Typangaben
`(x, short y)       -> x + y`	**Falsch:** keine Mischung von Parametern mit und ohne explizite Typangabe!
`()                 -> 42`	Parameterliste darf leer sein
Funktionsrümpfe	
`(x,y) -> x*x - y*y`	Rumpf mit nur einem Ausdruck

`x -> {`
...*Anweisungen*; ...
`return` *wert*;
`}`
	Rumpf als Block mit mehreren Anweisungen und abschließender `return`-Anweisung

`(int x) -> return x + 1`	**Falsch:** die `return`-Anweisung nur innerhalb eines Blocks mit geschweiften Klammern `{...}`!

Tabelle 1.1. Syntax von Lambda-Ausdrücken in Java. Die rötlich hinterlegten Ausdrücke sind falsch. Modifiziert nach [4]

Ein Lambda-Ausdruck erzeugt auf diese Weise eine *anonyme* Funktion in Java, die eines der Interfaces aus dem Paket `java.util.function` implementiert. Das Paket besteht aus den sogenannten *funktionalen Interfaces*. Die wichtigsten Interfaces daraus sind in der folgenden

Tabelle aufgelistet:

Interface	**Lambda-Ausdruck**	**Auswertungs-methode**	**Bemerkung**
`Consumer<T>`	`(T x) -> <void>;`	`accept(T x)`	ein Parameter, keine Rückgabe (z.B. durch Aufruf einer `void`-Methode)
`Supplier<T>`	`() -> const;`	`get()`	kein Parameter, Rückgabe vom Typ `T`
`Predicate<T>`	`x -> x < 5;`	`test(T x)`	ein Parameter vom Typ `T`, Rückgabe vom Typ `boolean`
`Function<T,R>`	`x -> x*x - x + 1;`	`apply(T x)`	ein Parameter vom Typ `T`, Rückgabe vom Typ `R`
`BiFunction<T,U,R>`	`(x,y) -> x*x - y*y;`	`apply(T x, U y)`	zwei Parameter vom Typ `T` und `U`, Rückgabe vom Typ `R`
`UnaryOperator<T>`	`x -> x*x - x + 1;`	`apply(T x)`	ein Parameter vom Typ `T`, Rückgabe vom Typ `T`
`BinaryOperator<T>`	`(x,y) -> x - y;`	`apply(T x, T y)`	zwei Parameter vom Typ `T` und Rückgabe vom Typ `T`

(1.1)

(Die beiden Datentypen `UnaryOperator` und `BinaryOperator` sind spezielle Varianten der Datentypen `Function` und `BiFunction`, in denen alle Datentypen ihrer Parameter und ihrer Rückgaben stets gleich sind.) Eine anonyme Funktion kann wie eine Variable eine Referenz erhalten, um dann später aufgerufen und verwendet zu werden, zum Beispiel

```
BiFunction <Double, Double, Double> f = (x,y) -> x*x + y*y;
```

für eine von zwei Variablen abhängende Funktion

$$f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \qquad f(x, y) = x^2 + y^2$$

(mit der Näherung `Double` $\approx \mathbb{R}$). Solche Funktionsreferenzen werden oft *Rückruffunktionen* oder *Callback*-Funktionen genannt, da sie nicht dort ausgeführt werden, wo sie übergeben werden, sondern in der Umgebung der aufrufenden Funktion, oft in der API (siehe Abschnitt A.2).

Eine Funktion kann mit der Methode `apply` ausgewertet werden. Das folgende Programm wertet zwei statisch definierte Funktionen f und g aus:

```
1  import java.util.function.Function;
2  import java.util.function.BiFunction;
3  
4  public class Funktionen {
5     public static Function<Integer, Integer> f = (x) -> x*x - 1;
6     public static BiFunction<Integer, Integer, Integer> g = (x,y) -> x*x*x + 2*x - 1;
7  
8     public static void main(String... args) {
9        int x = 3, y=4;
10       String ausgabe = "f("+x+") = " + f.apply(x);
11       ausgabe += "\ng("+x+","+y+") = " + g.apply(x,y);
12       javax.swing.JOptionPane.showMessageDialog(null, ausgabe, "Funktionen", -1);
13    }
14 }
```

Zu beachten ist, dass in einer Klasse die Funktionreferenzen verschiedene Namen haben müssen, auch wenn sie verschiedene Parameterlisten haben. (Anders als bei Methoden wird für Lambda-Ausdrücke die Signatur der Referenz in Java erst bei ihrem Aufruf ausgewertet.)

Lambda-Ausdrücke innerhalb von Methoden: Closures. Werden Lambda-Ausdrücke innerhalb von Methoden definiert, so dürfen die verwendeten Parameter nicht bereits vorher deklariert sein. In diesem Fall ergibt sich ein Kompilierfehler:

```
public static void main(String... args) {
   int x = 3;
   Function<Integer, Integer> f = x -> x*x - 1; // !! Fehler !!
}
```

Ein Lambda-Ausdruck in einer Methode kann als eine „innere" Funktion betrachtet werden. Eine solche Funktion heißt *Closure*. Eine Closure kann auf Variablen zugreifen, die außerhalb von ihr deklariert sind, aber zum Zeitpunkt der Closuredeklaration einen bestimmten Wert haben. Diese müssen aber `final` sein oder „effektiv final", was soviel heißt, dass bei einer nachträglichen Änderung des Variablenwertes sofort ein Kompilierfehler ausgelöst wird.

```
public static void main(String... args) {
   int c = 5;
   Function<Integer, Integer> f = x -> x*x - c; // Konstante c hier festgesetzt
   int x = 3;
   String ausgabe = "f("+x+") = " + f.apply(x);
   c = -5; // Konstante wird geändert: Kompilierfehler!
   x = 5;  // erlaubt, da x in Lambda-Ausdruck ja die freie Variable ist
}
```

Mit Closures kann man zum Beispiel Funktionenscharen oder „Funktionenfabriken" programmieren, also Funktionen, die parameterabhängig Funktionen zurückgeben:

```
1 // Funktionenschar, mit dem Scharparameter "name":
2 Function<String, Function<String,String>> fabrik =
3    name -> (instrument -> name + " spielt " + instrument);
4
5 Function<String, String> a = fabrik.apply("Anna");
6 Function<String, String> b = fabrik.apply("Bert");
7
8 System.out.println(a.apply("Geige") + ", " + b.apply("Bass"));
9 // => Anna spielt Geige, Bert spielt Bass";
```

Die äußere Funktion definiert hier eine Variable `name`, auf die die innere anonyme Funktion (in Klammern) auch Zugriff hat. Bei jedem Aufruf der äußeren Funktion wird der jeweilige Wert der Variablen in der inneren Funktion „eingefroren" und mit ihr zurückgegeben. Daher gibt die Funktion `anna()` einen anderen Wert zurück als die Funktion `bert()`. Die runden Klammern um den inneren Lambda-Ausdruck können auch weggelassen werden, der zweite Pfeiloperator hat höhere Präferenz als der erste. Zur besseren Lesbarkeit des Ausdrucks allerdings werden sie hier aufgeführt.

Zudem kommen Closures zum Einsatz bei dem Funktionalkalkül, also der Algebra mit Funktionen. Man kann beispielsweise die Summe $f + g$ zweier Funktionen f, $g : \mathbb{R} \to \mathbb{R}$ definieren, indem man $(f + g)(x) = f(x) + g(x)$ definiert. In Java könnte der Additionsoperator für zwei Funktionen also so aussehen:

```
1    public static Function<Integer, Integer> f = x -> x*x - 1;
2    public static Function<Integer, Integer> g = x -> x*x + 1;
3
4    public static BinaryOperator<Function<Integer, Integer>> add =
5       (f,g) -> (x -> f.apply(x) + g.apply(x));
```

```
6
7    public static void main(String... args) {
8       int x = 3, y = 4;
9       String ausgabe = "f("+x+") = " + f.apply(x);
10      ausgabe += "\ng("+x+") = " + g.apply(x);
11      ausgabe += "\n(f+g)("+x+") = " + add.apply(f,g).apply(x);
12      javax.swing.JOptionPane.showMessageDialog(null, ausgabe, "Funktionen", -1);
```

Funktionen höherer Ordnung. Eine *Funktion höherer Ordnung* ist eine Funktion, die als Argument eine Funktion erwartet oder deren Ergebnis eine Funktion ist. Beispielsweise kann die Kepler'sche Fassregel (oder auch Simpson-Regel)

$$I_f(a,b) \approx \frac{b-a}{6}\left(f(a) + 4f\left(\tfrac{a+b}{2}\right) + f(b)\right) \tag{1.2}$$

als Näherungsformel für das Integral $I_f(a,b) = \int_a^b f(x)\,\mathrm{d}x$ einer integrierbaren Funktion $f: [a,b] \to \mathbb{R}$ mit a, $b \in \mathbb{R}$ in Java wie folgt implementiert werden:

```
1  import java.util.function.Function;
2  import java.util.function.BiFunction;
3  import static java.lang.Math.*;
4
5  public class Integral {
6     /** Quadraturformel Kepler'sche Fassregel I(f)(a,b) für eine Funktion f.*/
7     public static Function<Function<Double,Double>, BiFunction<Double, Double, Double>>
8        I = f -> ((a,b) -> (b-a)/6 * (f.apply(a) + 4*f.apply((a+b)/2) + f.apply(b)));
9
10    public static Function<Double,Double> f = x -> 3*x*x;
11    public static Function<Double,Double> g = x -> 1/x;
12
13    public static void main(String... args) {
14       double a = 0, b = 1;
15       System.out.println("I_f("+a+", "+b+") = " + I.apply(f).apply(a,b)); // 1.0
16       a = 1; b = E;
17       System.out.println("I_g("+a+", "+b+") = " + I.apply(g).apply(a,b)); // 1.00788994
18       a = 0; b = 1;
19       System.out.println("I_h("+a+", "+b+") = " + I.apply(x -> sqrt(1-x*x)).apply(a,b)); //.744
20    }
21 }
```

Die tatsächlichen Werte sind

$$\int_0^1 3x^2\,\mathrm{d}x = x^3\Big|_0^1 = 1, \qquad \int_1^{\mathrm{e}} \frac{\mathrm{d}x}{x} = \ln x\Big|_1^{\mathrm{e}} = 1 \tag{1.3}$$

und [12, S. 163]

$$\int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\arcsin x + x\sqrt{1-x^2}}{2}\Bigg|_0^1 = \frac{\pi}{4} \approx 0{,}78539816339745. \tag{1.4}$$

(Für die Herleitung dieser Gleichung mit Integration durch Substitution siehe z.B. [1, §19.15]; der Graph der Funktion $\sqrt{1-x^2}$ beschreibt in dem Intervall $[0,1]$ einen Viertelkreis mit Radius 1.)

Ereignisbehandlung. Bei der Programmierung graphischer Oberflächen, sogenannter GUI's *(graphical user interface)*, werden durch die API vordefinierte Bedienelemente wie Schaltflächen *(Buttons)*, Auswahllisten *(Select Boxes)* oder Textfelder *(Text Fields)* in einem Dialogfenster programmiert. Damit diese Elemente jedoch nicht nur statisch in dem Fenster angezeigt werden, sondern auf Eingaben des Anwenders reagieren, müssen sie mit einem oder mehreren *Ereignisbehandlern (Event Handler* oder auch *Event Listenern)* verknüpft werden. Die Idee dahinter ist, dass jede Eingabe über ein Bedienelement ein bestimmtes Ereignis *(Event)* erzeugt, also beispielsweise einen Mausklick oder eine Tastatureingabe. Ist das Bedienelement mit einem Ereignisbehandler verknüpft, der genau auf diese Eingabe „horcht", so führt er eine vom Programmierer implementierte Methode aus, die *Ereignisbehandlungsroutine (event handling method)*.

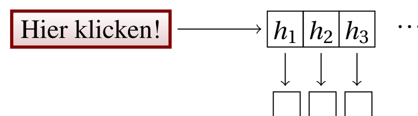


Abbildung 1.1. Ein GUI-Element und mit ihm verknüpfte Ereignisbehandler und deren Behandlungsroutinen.

Diese Programmiermodell nennt man *Ereignisbehandlung* oder *Event Handling*. In Java ist es schon seit jeher realisiert und hat mit funktionaler Programmierung zunächst nicht direkt etwas zu tun. Allerdings ist es mit funktionaler Programmierung möglich, die Ereignisbehandlungsroutine ganz kurz als Lambda-Ausdruck zu implementieren:

```java
1  import javax.swing.JButton;
2  import static javax.swing.JOptionPane.*;
3
4  public class EventListener {
5     public static void main(String... args) {
6        JButton button = new JButton("Klick mal hier!");
7        // Event Handler durch Lambda-Ausdruck definiert:
8        button.addActionListener(e -> showMessageDialog(null, "Geklickt!"));
9
10       showMessageDialog(null, button);
11    }
12 }
```

Alternativ müsste man dazu viel aufwändiger explizit eine anonyme innere oder eine eigene äußere Klasse für den Ereignisbehandler programmieren und darin die Behandlungsroutine implementieren, z.B.:

```java
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import static javax.swing.JOptionPane.*;
5
6  public class EventListenerClassic {
7     public static void main(String... args) {
8        JButton button = new JButton("Klick mal hier!");
9        // Event Handler durch anonyme Klasse definiert:
10       button.addActionListener(new ActionListener() {
11          public void actionPerformed(ActionEvent e) {
12             showMessageDialog(null, "Geklickt!");
```

```
13            }
14        });
15
16        showMessageDialog(null, button);
17    }
18 }
```

Man erkennt an diesem einfachen Beispiel auch sofort, dass man einen Lambda-Ausdruck zur Implementierung eines Event Handlers nur verwenden kann, wenn es nur eine einzige zu implementierende Methode gibt. Was aber ist bei einem Event Handler mit mehreren zu implementierenden Methoden zu tun? Wie sollte man z.B. einen `MouseListener` mit einem Lambda-Ausdruck für vier Methoden programmieren? Nun, es geht nicht. Es bleibt einem in einem solchen Fall nach wie vor nur die klassische Variante, also eine innere anonyme oder eine extra Klasse, siehe `http://stackoverflow.com/questions/21833537`.

1.2 Methodenreferenzen

Mit Lambda-Ausdrücken programmiert man anonyme Funktionen. Mit Hilfe einer *Methodenreferenz* kann man auf bereits existierende Methoden verweisen. Dazu wird der doppelte Doppelpunktoperator `::` verwendet, hier am Beispiel der `forEach`-Methode einer Liste, die eine Funktion `x -> void` erwartet, d.h. eine Instanz des Interfaces `Consumer`:

```
public static void main(String[] args) {
   java.util.List<Integer> prim = java.util.Arrays.asList(2,3,5,7,11,13,17,19,23);
   prim.forEach(System.out::println);
}
```

Hierbei gilt für die Methodenreferenz die Äquivalenz der Anweisungen

$$\texttt{System.out::println} \qquad \Longleftrightarrow \qquad \texttt{x -> System.out.println(x)} \tag{1.5}$$

Insgesamt ersetzt damit der funktionale Ausdruck

```
prim.forEach(System.out::println);
```

also die klassische for-each Schleife

```
for (int x : prim) {
   System.out.println(x);
}
```

1.3 Ist Java echt funktional?

Trotz der Lambda-Ausdrücke ist Java allerdings noch keine funktionale Sprache. In rein funktionalen Sprachen können Wiederholungen nicht durch Schleifen, sondern nur durch Rekursionen durchgeführt werden, also durch Selbstaufrufe von Funktionen. In Java ist das (in Version 8) allerdings nicht möglich [LK]. Zudem zeigt ein etwas genauerer Blick in das Paket `java.util.function`, dass Java nur Funktionen mit höchstens zwei Parametern ermöglicht. Deren Haupttypen wurden bereits in Tabelle (1.1) aufgeführt. Echt funktional ist Java mit Version 8 also sicherlich nicht. Dennoch ermöglicht Java funktionale Programmierung auf „alltagsüblichem“ Niveau, d.h. man kann Lambda-Ausdrücke verwenden und Referenzen auf Funktionen

übergeben. Und wenn man für eine Funktion mehr als zwei Parameter braucht, nun ja, dann nimmt man halt ein Array als Parameter ...

Ein großer Vorteil der Java-Lösung mit funktionalen Interfaces ist jedenfalls, dass sie die parallele Programmierung mit Streams ermöglichen. Dieser Aspekt wird im folgenden Kapitel näher betrachtet.

Kapitel 2

Streams: Nebenläufigkeit mit Collections

2.1 Sequenzielle Programme und Mehrkernarchitekturen

Ein wichtiger Aspekt bei der effizienten Verarbeitung großer Datenstrukturen auf nebenläufigen Architekturen wie Mehrkernprozessoren oder Rechnerclustern ist die Möglichkeit, einen geeigneten Programmabschnitt echt parallel ablaufen zu lassen. Nicht jede Operation eines Algorithmus lässt sich parallelisieren, manche Anweisungsabschnitte können nur sequenziell erfolgen. Drei einfache Beispiele mögen dies illustrieren, nämlich die Initialisierung eines Arrays mit einem Standardwert:

```
for (int i = 0; i <= 256; i++) {
   x[i] = 1;
}
```

die Vektoraddition:

```
for (int i = 0; i <= 256; i++) {
   z[i] = x[i] + y[i];
}
```

und die Berechnung und Speicherung einer Folge (hier $x_i = x_{i-1} + i$) in einem Array:

```
x[0] = 1;
for (int i = 1; i <= 256; i++) {
   x[i] = x[i-1] + i;
}
```

Alle drei Beispiele sind hier sequenziell programmiert. Da aber für die ersten beiden Beispiele keine Abhängigkeiten zwischen Array-Einträgen mit verschiedenen Indizes existieren, wären diese durchaus auf einem Mehrkernprozessorsystem echt parallelisierbar, indem verschiedene Indizes auf verschiedenen Prozessoren ablaufen. Auf einem solchen Rechnersystem haben wir

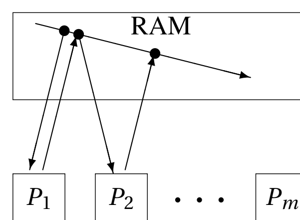


Abbildung 2.1. Sequenzielle Programmierung auf paralleler Prozessorarchitektur?

also eine Situation programmiert, wie sie in Abbildung 2.1 dargestellt ist: Wir haben unseren

Rechner „sequenziell ausgebremst“. Die Berechnung der Folge dagegen ermöglicht keine Parallelisierung, der Eintrag mit Index i hängt von dessem Vorgänger ab.

Grundsätzlich hängt der potenzielle Geschwindigkeitsgewinn eines auf mehrere Prozessorkerne parallelisierbaren Programms vom Gustafson’schen Gesetz ab, siehe Gleichung (A.9) im Anhang. Der Anteil p_G der parallelisierbaren Programmanteile sollte hierbei maximiert werden. Der Compiler kann prinzipiell jedoch nicht automatisch entscheiden, wann ein Programmabschnitt oder eine Schleife parallel verarbeitet werden kann, diese Entscheidung kann nur der Programmierer treffen. Er muss die Möglichkeit haben, durch unterschiedliche Syntaxanweisungen parallele oder sequenzielle Ausführung festzulegen. In Java wird das durch das Stream Paket `java.util.stream` für die Collections ermöglicht.

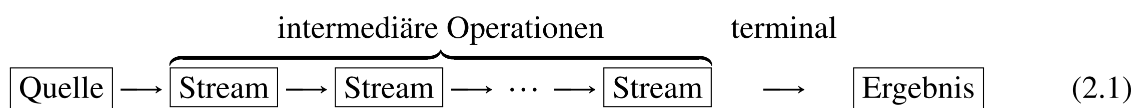
2.2 Datenströme und Pipelines

Ein *Datenstrom (Stream)* ist eine Folge gleichstrukturierter Elemente, deren Ende nicht im Voraus festgelegt ist.

$$\text{Stream} = \begin{array}{|c|c|c|c|c|}\hline x_1 & x_2 & x_3 & x_4 & x_5 \\ \hline\end{array}\ \cdots$$

Auf Datenströme sind sequenzielle und parallele Auswertungen anwendbar, nicht aber direkte und gezielte Zugriffe auf einzelne Elemente. Datenströme sind daher keine Datenstrukturen wie Collections oder Arrays. Das Konzept der Datenströme geht auf die Pipes zurück, die 1973 von Douglas McIlroy für das Betriebssystem UNIX erfunden wurden. Eine *Pipe* (englisch für Rohr, Röhre) bezeichnet einen gepufferten Datenstrom zwischen zwei Prozessen oder Threads nach dem FIFO-Prinzip. Das heißt vereinfacht, dass die Ausgabe eines Threads als Eingabe für einen weiteren verwendet wird.[1]

Eine *Pipeline* ist eine Abfolge von Operationen auf einen Datenstrom. Sie besteht aus einer Quelle *(source)*, mehreren intermediären Operationen *(intermediate operations)* und einer terminalen Operation *(terminal operation)*. Die Quelle kann eine Datenstruktur, eine Daten erzeugende Funktion („Generator“) oder ein I/O-Kanal sein. Aus der Quelle wird der Stream dann erzeugt, technisch ist er oft lediglich ein Verweis auf die Quelle.



(2.1)

Eine intermediäre Operation bearbeitet die einzelnen Datenelemente und erzeugt einen neuen Datenstrom; auf diese Weise können beliebig viele intermediäre Operationen hintereinander geschaltet werden. Eine terminale Operation schließlich erzeugt Daten als ein aggregiertes Endergebnis des Datenstroms oder beendet ihn einfach nur [6].[2] Intermediäre und terminale Operationen werden zusammengefasst auch *Aggregatoperationen* oder *Bulk Operationen* genannt.[3]

---

[1] Datenströme stellen ein datenstromorientiertes Kommunikationsmodell dar, d.h. der Empfänger nimmt einen linearen Datenstrom entgegen, dem er nicht ansieht, in welchen Portionen die Daten ursprünglich gesendet worden sind [9, §3.3]. Daraus entwickelte sich die „datenstromorientierte Programmierung“ oder „Datenstromprogrammierung“ *(data flow programming)*, in der ein Programm als gerichteter Graphen modelliert wird. Ältere Vetreter solcher Sprachen sind SimuLink, Pure Data oder LabVIEW, neuere Sprachen, die dieses Paradigma verwenden, sind Clojure, Lucid oder Oz.

[2] Java API Documentation `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps`

[3] Java API Documentation `http://docs.oracle.com/javase/tutorial/collections/streams/` [2016-03-15]

2.2.1 Auswertungen von Datenströmen: lazy und eager

In intermediären Operationen werden Ausdrücke üblicherweise *lazy (faul, bequem)* ausgewertet. Eine *lazy Evaluation* bezeichnet eine Art der Auswertung, bei der das Ergebnis des auszuwertenden Ausdrucks nur insoweit berechnet wird, wie es benötigt wird.[4] Im Gegensatz dazu wird bei einer *eager Evaluation* jeder Ausdruck sofort ausgewertet oder verarbeitet. Intermediäre Operationen sind stets *lazy*, da der Inhalt des Streams erst dann verarbeitet wird, wenn die terminale Operation beginnt.

2.3 Streams in Java

`Stream` ist das zentrale Interface in Java, das parallelisierbare Operationen für Collections und Arrays ermöglicht, aber auch unendliche Folgen von Elementen.

2.3.1 Arten von Streams

`Stream` ist das zentrale Interface des Pakets `java.util.stream`. Ein Stream ist kein Datenspeicher wie eine Collection oder ein Array, sondern lediglich ein Verweis auf die Elemente seiner Quelle, auf der eine Pipeline von Operationen ausgeführt werden kann [5]. Streams werden anhand ihrer Quelle unterschieden, es gibt Collection-basierte Streams, Array-basierte Streams, durch eine Generatorfunktion gebildete unendliche Streams und schließlich I/O-Kanal-basierte Streams. Die Syntax für einen Collection-basierten Stream lautet einfach:

```
List<String> text = ...;
Stream<String> seq = text.stream();
Stream<String> par = text.parallelStream();
```

Mit der Collection-Methode `stream` wird auf diese Weise ein sequenzieller Stream erzeugt, mit `parallelStream` ein parallelisierter Stream. Die Syntax für Array-basierte Streams ist etwas vielfältiger, die einfachsten Varianten lauten:

```
String[] text = {"Da", "steh", "ich", "nun"};
Stream<String> seq = Arrays.stream(array);
Stream<String> par = Arrays.stream(array).parallel();
```

Hier wird die statische Methode `stream` aus der Klasse `java.util.Arrays` zur Erzeugung eines sequenziellen Streams verwendet, will man einen parallelisierten Stream benutzen, muss man ihn mit der Stream-Operation `parallel` parallelisieren. Eine bequeme Variante ist daneben die statische Methode `of` aus dem Stream-Interface, die eine Liste von Objekten erwartet:

```
Stream<String> seq = Stream.of("Da", "steh", "ich", "nun");
```

Sequenzielle unendliche Streams werden mit den statischen Stream-Methoden `generate` und `iterate` erzeugt, die beide eine Generatorfunktion erwarten, `iterate` zusätzlich einen Startwert *(seed)* für das erste Element:

```
Stream<Double> random = Stream.generate(Math::random); // Zufallszahlen
Stream<Integer> seq = Stream.generate(2, n -> 2*n); // Zweierpotenzen 2, 4, 8, ...
```

Für die primitiven Datentypen `int`, `long` und `double` gibt es die primitiven Streamtypen `IntStream`, `LongStream` und `DoubleStream`, die ihre eigenen speziellen Methoden haben und effizienter sind als die entsprechenden Streams `Stream<Integer>`, `Stream<Long>` und `Stream<Double>`.

[4] `http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html#laziness` [2016-03-15]

Erwähnt sei außerdem, das es streamerzeugende Methoden in dem Interface `CharSequence` sowie in den Klassen `Pattern` und `Files` gibt.

```java
import java.util.stream.*;
import java.math.BigInteger;
import static java.math.BigInteger.ONE;
import static java.math.BigInteger.ZERO;
import static java.math.BigInteger.valueOf;

public class Streams {
   public static void main(String... args) {
      Stream<Integer> squares;
      int sum;

      // -- Stream aus Array: --
      squares = java.util.Arrays.stream(new Integer[] {1, 4, 9, 16, 25, 36, 49});

      sum = squares
         .peek(System.out::println)   // intermediär
         .reduce(0, (n,a) -> n + a);  // terminal

      System.out.println("sum = " + sum);

      // -- unendliche Streams: --
      squares = Stream.iterate(1, n -> n + 1); // unendlicher Stream

      sum = squares
         .map(n -> n*n)
         .limit(7)
         .peek(System.out::println)   // intermediär
         .reduce(0, (n,a) -> n + a);  // terminal

      System.out.println("sum = " + sum);

      Stream<BigInteger> odds = Stream.iterate(ONE, n -> n.add(valueOf(2)));
      BigInteger summe = odds
         .skip(100000000)   // zustandsbehaftet, benötigt Rechenzeit
         .limit(7)          // zustandsbehaftet
         .peek(System.out::println)  // Nebeneffekt, aber zustandslos
         .reduce(ZERO, (n,a) -> n.add(a));  // terminal

      System.out.println("sum = " + summe);
   }
}
```

2.3.2 Stream-Operationen

Ein `Stream` ermöglicht parallelisierbare Operationen für Collections und Arrays, sogenannte *aggregierte* oder *Bulk Operationen*. Die wichtigsten dieser Operationen sind `filter`, `map`, `parallel`, `sequential`, `peek`, `reduce` und `forEach`. Alle Operationen erwarten als Parameter einen

Lambda-Ausdruck oder eine Methodenreferenz, `filter` und `map` sind intermediär, `reduce` und `forEach` terminal, siehe Tabelle 2.1. Die Methode `filter` entfernt alle Elemente des Streams,

Signatur	**Beispiel**
intermediäre Operationen	
`Stream<T> filter(Predicate<T> p)`	`stream.filter(x -> x < 5);`
`Stream<R> map(Function<T,R> f)`	`stream.map(x -> x*x);`
`Stream<T> parallel()`	`stream.parallel()`
`Stream<T> sequential()`	`stream.sequential()`
`Stream<T> peek(Consumer<T> action)`	`stream.peek(System.out::println)`
terminale Operationen	
`T reduce(T start, BinaryOperator<T> f)`	`stream.reduce(0, (a,x) -> 2*x -1 + a);`
`void forEach(Consumer<T> action))`	`stream.forEach(System.out::println);`

Tabelle 2.1. Gängige Operationen von Streams.

die für die übergebene Prädikatfunktion `false` zurück gibt, `map(f)` erzeugt einen neuen Stream, der statt des Elements x_i das Element $f(x_i)$ enthält, während `peek` einen Einblick in den jeweils aktuellen Stream ermöglicht. Die Methode `reduce` wendet die übergebene Funktion $f = f(a,x)$ rekursiv mit dem Startwert (hier: $a_0 = 0$) an. Die Operationen

Ein parallelisierter Stream wird erzeugt, indem man für eine Collection die Methode `parallelStream` aufruft. Mit der Methode `stream` dagegen wird ein sequenzieller Stream erzeugt. Das folgende Beispiel soll dies demonstrieren:

```java
1  import java.util.Arrays;
2  import java.util.List;
3  import java.util.LinkedList;
4  import java.util.function.Function;
5  import static java.util.Collections.addAll;
6
7  public class ParallelLinkedList {
8     public static void main(String... args) {
9        LinkedList<String> text = new LinkedList<>();
10       addAll(text, "Da steh ich nun, ich armer Tor".split("[\\s;,.!?]"));
11       int summe = text.stream()
12          .filter(s -> s.length() > 0)
13          .peek(x -> System.out.print(x + ", "))
14          .map(s -> s.length())
15          .peek(x -> System.out.print(x + ", "))
16          .reduce(0, (a,x) -> x + a);
17       System.out.println("\n" + text + ": " + summe + " Buchstaben");
18       summe = text.parallelStream()
19          .filter(s -> s.length() > 0)
20          .peek(x -> System.out.print(x + ", "))
21          .map(s -> s.length())
22          .peek(x -> System.out.print(x + ", "))
23          .reduce(0, (a,x) -> x + a);
24       System.out.println("\n" + text + ": " + summe + " Buchstaben");
25
26       List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
27       Function<Integer, Integer> f = x -> x*x;
```

```
28        summe = liste.parallelStream()
29           .map(f)
30           .peek(x -> System.out.print(x + ", "))
31           .reduce(0, (a,x) -> a+x);
32        System.out.println("\nsumme = "+summe);
33     }
34  }
```

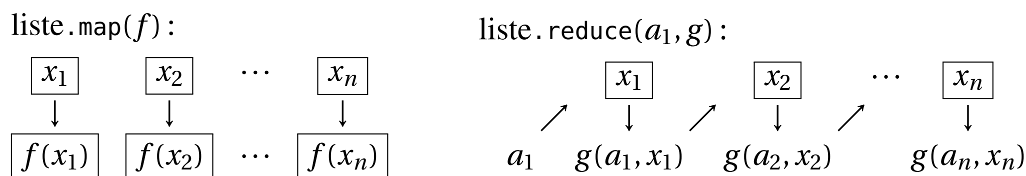
Auf einem Dual-Core Ultrabook ergibt dieses Programm zum Beispiel die folgende Ausgabe:

```
Da, 2, steh, 4, ich, 3, nun, 3, ich, 3, armer, 5, Tor, 3,
[Da, steh, ich, nun, , ich, armer, Tor]: 23 Buchstaben
ich, Tor, 3, ich, armer, 5, 3, steh, nun, 3, 3, 4, Da, 2,
[Da, steh, ich, nun, , ich, armer, Tor]: 23 Buchstaben
9, 4, 16, 25, 1,
summe = 55
```

Während für den sequenziellen Stream für jeden Eintrag jeweils die Pipeline `filter`–`map` durchgeführt wird, ist die Abarbeitung im parallelisierten Stream zwar für jeden Eintrag für sich sequenziell, jedoch die Reihenfolge der Operationen ist zufällig.

2.3.3 Map und Reduce

In der funktionalen Programmierung wenden die beiden Routinen `map` und `reduce` (letztere machmal auch `fold` genannt) eine übergebene Funktion `f` auf die Elemente einer Liste an und aggregieren deren Auswertungen zu einem Endergebnis. Während `map` die Funktion auf jedes Element anwendet und somit eine Liste mit derselben Größe zurückgibt, ergibt `reduce` einen einzelnen Rückgabewert, in der Regel eine Zahl. Die Funktionsweise von `map` und `reduce` bei ihrer Anwendung auf die Liste `liste` $= [x_1, x_2, \ldots, x_n]$ und die Funktion $f(x)$ bzw. die Funktion $g(a, x)$ und den Startwert a_1 ist durch folgendes Schema illustriert:



Hierbei erwartet `reduce` eine zweiparametrige Funktion $g(a, x)$, deren erster Parameter a den bis jetzt kumuliert errechneten Wert darstellt und x den aktuell einzusetzenden Listenwert. In der Illustration ist der Startwert a_1, und für die folgenden Werte gilt

$$a_{n+1} = g(a_n, x_n) \tag{2.2}$$

Das folgende Beispiel zeigt `map` mit der Funktion $f(x) = x^2$ angewendet auf eine Liste von Zahlen:

```
List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
Function<Integer, Integer> f =  x -> x*x;
liste.parallelStream()
   .map(f)
   .peek(x -> System.out.print(x + ", "))  // 9, 4, 16, 25, 1,
   .reduce(0, (a,x) -> a+x); // terminale Operation, sonst passiert nichts ...
```

Ein typisches Beispiel für `reduce` ist die Berechnung einer akkumulierenden Funktion aller Listenelemente:

```
List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
BinaryOperator<Integer> g =  (a, x) ->  2*x - 1 + a;
System.out.println(liste.stream().reduce(0,g));  // 25
```

Die Berechnung einer Summe wie $\sum_{i=1}^{4} i^3$ lässt sich mit `map` und `reduce` wie folgt programmieren:

```
List<Integer> liste = Arrays.asList(1, 2, 3, 4);
int aggregat = liste.stream().map(i -> i*i*i).reduce(0, (a,x) -> x + a);
System.out.println(aggregat);  // 100
```

2.3.4 Reduce auf parallelisierten Streams

Wendet man reduce unbedacht auf einen parallelisierten Stream an, so kann man falsche Ergebnisse erlangen.

Beispiel 2.1. *(Fehlerhaftes Reduce)* Betrachten wir eine Liste $[1,2,\ldots,5]$ von Zahlen und die Funktion $g(a,x) = 2x - 1 + a$, also

```
List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
BinaryOperator<Integer> g =  (a,x) ->
   {System.out.println("+++ g("+a+","+x+") = "+(2*x - 1 + a)); return 2*x - 1 + a;};
int summe = liste.stream().reduce(0,g);
```

Mit diesem sequenziellen Stream erhalten wir damit das (korrekte) reduce-Ergebnis `summe` = 25:

```
+++ g(0,1) = 1
+++ g(1,2) = 4
+++ g(4,3) = 9
+++ g(9,4) = 16
+++ g(16,5) = 25
```

Dieselben Anweisungen für einen parallelisierten Stream allerdings,

```
List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
BinaryOperator<Integer> g =  (a,x) ->
   {System.out.println("+++ g("+a+","+x+") = "+(2*x - 1 + a)); return 2*x - 1 + a;};
int summe = liste.parallelStream().reduce(0,g);
```

ergeben mit der Ausgabe

```
+++ g(0,3) = 5
+++ g(0,1) = 1
+++ g(0,4) = 7
+++ g(0,2) = 3
+++ g(0,5) = 9
+++ g(1,3) = 6
+++ g(7,9) = 24
+++ g(5,24) = 52
+++ g(6,52) = 109
```

fälschlicherweise das Ergebnis `summe = 109`! Analysiert man die Ausgaben, die die Funktion g jeweils zu Beginn eines Aufrufs erstellt, etwas genauer, so erkennt man für die beiden Ergebnisse durch reduce die in Abbildung 2.2 dargestellten Berechnungsbäume. Anhand des Be-

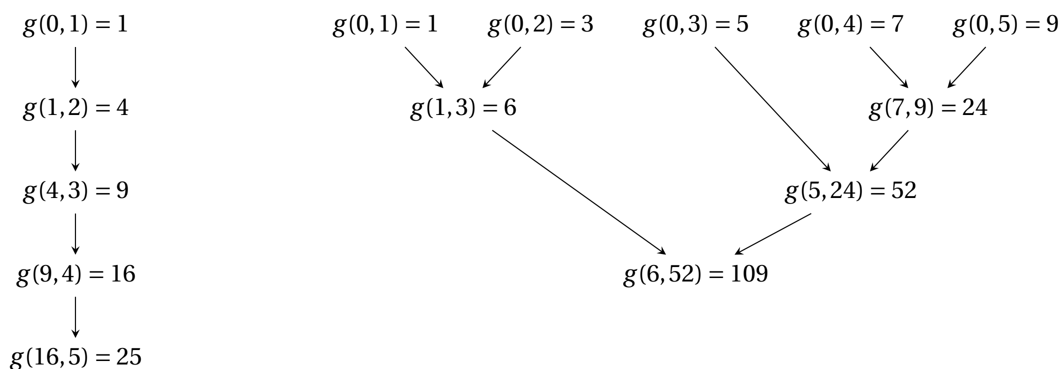


Abbildung 2.2. Reduce-Berechnung eines Streams mit der nicht-assoziativen Funktion $g(a, x) = 2x - 1 + a$: sequenziell (links) und parallel (rechts).

rechnungsbaums für den parallelisierten Stream ist zu erkennen, dass zunächst alle Startwerte $g(0, n)$ berechnet und die Teilergebnisse in drei Stufen jeweils paarweise kombiniert werden. Bei fünf verfügbaren Prozessorkernen würden also nur vier Berechnungsschritte benötigt, im Gegensatz zu den fünf Schritten im sequenziellen Fall. (Auch mit vier Prozessorkernen wäre dies möglich, nur müsste dann $g(0, 3)$ erst in der zweiten Stufe auf einem der freigewordenen Prozessorkerne ablaufen, z.B. nach $g(0, 2)$ oder $g(0, 5)$.) □

Was ist für den parallelisierten Stream in Beispiel 2.1 schiefgelaufen? Die Ursache liegt in der Rechenweise, mit der reduce auf einem parallelisierten Stream vorgeht und die in Abbildung 2.2 zu erkennen ist. Zunächst werden nämlich alle Streamelemente x_n mit dem Startwert a in die Funktion g eingesetzt und dann die Ergebnisse sukzessive jeweils paarweise wieder in g eingesetzt, also nach dem Schema:

$$\begin{array}{ccc} g(a,x) & & g(b,y) \\ & \searrow \quad \swarrow & \\ & g\big(g(a,x),\, g(b,y)\big) & \end{array} \tag{2.3}$$

Diese paarweise Zusammenführung von berechneten Zwischenwerten wird in parallelen Rechnerarchitekturen *Join* genannt. Wollen wir reduce auf einem parallelisierten Stream anwenden, so können wir vorher den Stream mit der intermediären Operation `sequential()` zu einem sequenziellen Stream umformen; diese Lösung klappt zwar immer, hat aber natürlich die Nachteile, dass einerseits Rechenzeit für eine rein technische Operation vergeudet und andererseits die parallele Prozessorarchitektur via Join nicht ausgenutzt wird. Um auf einem parallelisierten Stream korrekt zu arbeiten, muss die Funktion $g(a, x)$ eine Eigenschaft besitzen, die bei einem Join stets das richtige Ergebnis liefert: Die reduce-Funktion $g : \mathbb{R}^2 \to \mathbb{R}$ muss *assoziativ* sein, d.h.[5]

$$\boxed{g\big(g(a,x),\, y\big) = g\big(a,\, g(x,y)\big).} \tag{2.4}$$

Betrachten wir dazu die Funktion $g : \mathbb{R}^2 \to \mathbb{R}$,

$$g(a,x) = 2x - 1 + a \tag{2.5}$$

[5] `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Associativity`, oder [11, S. 88]

aus Beispiel 2.1. Diese Funktion ist nicht assoziativ im Sinne von (2.4), denn es gilt allgemein:

$$g\big(g(a,x),y\big) = g(2x-1+a,y) = 2\,(2x-1+a)-1+y = 4x+y+2a-3,$$

aber

$$g\big(a,g(x,y)\big) = g(a,2x-1+y) = 2\,(2x-1+y)-1+a = 4x+2y+a-3.$$

Für ganz spezielle Wertekombinationen können zwar beide Funktionswerte gleich sein (nämlich genau dann wenn $y = a$ gilt), aber im Allgemeinen sind sie verschieden. Speziell für $a = 0$, $x = 1$ und $y = 2$ gilt beispielsweise

$$g\big(g(a,x),y\big) = g(g(0,1),2) = g(1,2) = 4,$$

aber

$$g\big(a,g(x,y)\big) = g(0,g(1,2)) = g(0,4) = 7.$$

Ersetzt man in den Quelltextausschnitten dagegen die Funktion durch $g(a,x) = x+a$, so ergeben sich die Ausgaben

```
+++ g(0,1) = 1
+++ g(1,2) = 3
+++ g(3,3) = 6
+++ g(6,4) = 10
+++ g(10,5) = 15
```

für den sequenziellen Stream, und

```
+++ g(0,3) = 3
+++ g(0,5) = 5
+++ g(0,4) = 4
+++ g(0,2) = 2
+++ g(4,5) = 9
+++ g(0,1) = 1
+++ g(3,9) = 12
+++ g(1,2) = 3
+++ g(3,12) = 15
```

für den parallelisierten Stream. Das Ergebnis ist im parallelen Fall also korrekt, da die Funktion g assoziativ ist.

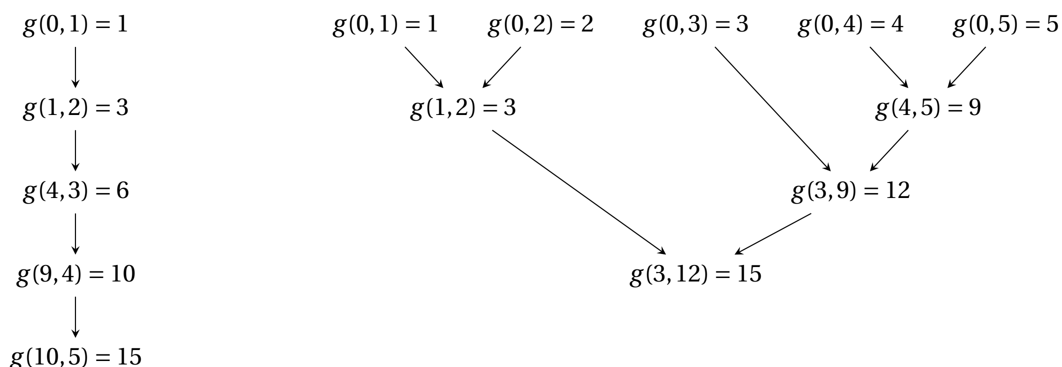


Abbildung 2.3. Reduce-Berechnung eines Streams mit der assoziativen Funktion $g(a,x) = a+x$: sequenziell (links) und parallel (rechts).

Für die Liste $[1,2,\ldots,6]$ ergibt die Berechnung eines sequenziellen Streams entsprechend die Ausgabe

```
+++ g(0,1) = 1
+++ g(1,2) = 3
+++ g(3,3) = 6
+++ g(6,4) = 10
+++ g(10,5) = 15
+++ g(15,6) = 21
```

und eines parallelisierten Streams die Ausgabe

```
+++ g(0,4) = 4
+++ g(0,2) = 2
+++ g(0,6) = 6
+++ g(0,3) = 3
+++ g(0,1) = 1
+++ g(0,5) = 5
+++ g(2,3) = 5
+++ g(1,5) = 6
+++ g(5,6) = 11
+++ g(4,11) = 15
+++ g(6,15) = 21
```

Vgl. Abbildung 2.4. Die effektive Laufzeit allerdings wird bei der Verteilung der Rechenleis-

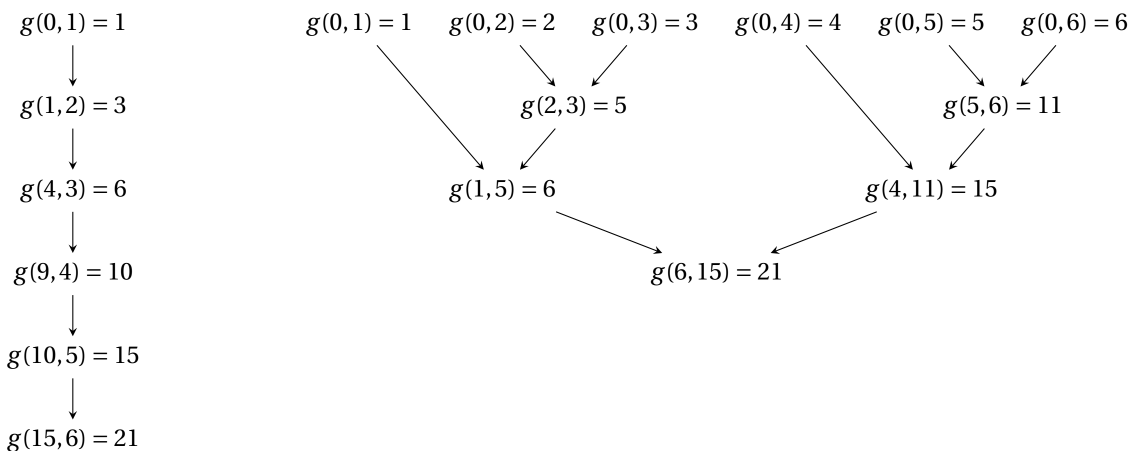


Abbildung 2.4. Reduce-Berechnung eines Streams mit der assoziativen Funktion $g(a, x) = a + x$ für die Quelle $\{1,2\ldots,6\}$: sequenziell (links) und parallel (rechts).

tung auf mehrere Prozessorkerne insgesamt kürzer. Für große Datenmengen wird daher ein paralleler Stream effizienter als ein sequenzieller bearbeitet werden können. Näheres siehe in §3 auf Seite 23.

Wann sollte man einen Stream parallelisieren? Aufgrund der gerade genannten Schwierigkeiten, die eine Parallelisierung hervorrufen kann, sind die folgenden Bemerkungen angebracht. Grundsätzlich sollte für eine Parallelisierung eines Streams seine Datenmenge genügend groß sein, denn die Parallelisierung erfordert vermehrte Speicherverwaltung. Aber auch der Datentyp der Datenquelle hat Einfluss auf die Effizienz einer Parallelisierung, primitive Datentypen und Strings sind schneller parallelisierbar als komplexe Objekte. Und nicht zuletzt die Collection-Art der Datenquelle spielt eine Rolle: Indizierte Datenstrukturen wie eine ArrayList oder ein Array, aber auch ein `IntStream.range`-Konstruktor, sind leicht zu splitten und daher effizient

parallelisierbar; eine `HashSet` oder eine `TreeSet` sind nicht ganz so effizient parallelisierbar, während eine `LinkedList` oder die `Stream.iterate` und `BufferedReader.line` nur schlecht parallelisierbar sind (nämlich mit Aufwand $O(n)$ bei n Elementen). Siehe dazu [11, §6].

2.3.5 Zustandslose Operationen

Eine wichtige Unterscheidung von intermediären Streamoperationen ist die zwischen *zustandslos (stateless)* und *zustandsbehaftet (stateful)*. Zustandslose Operationen arbeiten isoliert auf einem Element und benötigen keine Informationen über andere Elemente. Typische zustandslose Operationen sind `filter` und `map`: Die `filter`-Operation benötigt jeweils ein Element und das Prädikat, mit dem es bewertet wird, `map` braucht ein Element und die Abbildungsvorschrift darauf.

Zustandsbehaftete Operationen dagegen benötigen zusätzlich zu dem Element und der spezifizierten Funktion weitere Kontextinformation. Die Standardoperationen `limit` und `skip` beispielsweise müssen mitzählen, wieviel Elemente sie bereits besucht haben. Auch die Operation `sorted` benötigt Kontextinformationen über alle Elemente, da zur Ausführung jedes Element betrachtet werden muss.

Intermediäre zustandsbehaftete Operationen können nicht immer parallisiert auf einem Stream ausgeführt werden. Da sie stets Kontextinformationen mehrerer oder aller Elemente benötigen, müssen vorherige parallele Operationen synchronisiert werden, wie in Abbildung 2.5 dargestellt. Grundsätzlich sind also nur zustandslose Operationen ohne Weiteres parallelisierbar. In-

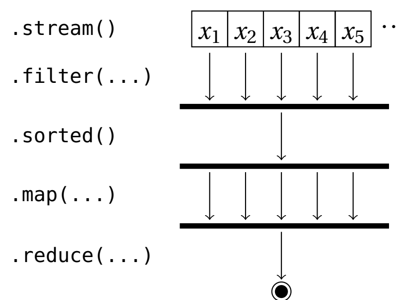


Abbildung 2.5. Zustandsbehaftete Operationen (hier: `sorted`) können nicht parallelisiert auf einem Stream arbeiten.

termediäre zustandsbehaftete Operationen bilden einen Engpass in der parallelen Verarbeitung von Streams und sollten soweit wie möglich vermieden werden.

Allerdings können bestimmte zustandsbehaftete Operationen zumindest zu wesentlichen Teilen parallelesiert werden: Beispielsweise können rekursive Algorithmen nach dem *Divide-and-conquer*-Prinzip in ihren geteilten Rekursionsschritten (natürlich!) parallel ablaufen, erst die *conquer*-Schritte bilden dann die Engpässe (gewissermaßen so etwas wie kleine *reduce*-Operationen). Prominente Vertreter sind die parallele Präfixsummierung *(parallel prefix sum)* und der parallelisierte Quicksort [2, §4.3, §9.4], [KM, 2012_5].

Im Prinzip kann man jede imperativ programmierte Schleifenkonstruktion durch funktionale datenstrombasierte *(„fluent“)* Programmierung umformen. Für ein illustratives Beispiel siehe [11, §3 S. 31ff].

Kapitel 3

Das Fork-join-Modell und parallele Algorithmen

Wie im vorigen Abschnitt gesehen, können nur zustandslose Operationen vollkommen parallelisiert werden. Allerdings können zumindest große Teile der rekursiven *Divide-and-Conquer*-Algorithmen parallelisiert werden, indem in den Rekursionsschritten die Rechenlast auf mehrere Prozessorkerne verlegt wird. Das Prinzip ist in Abbildung 3.1 skizziert. Es basiert auf einer

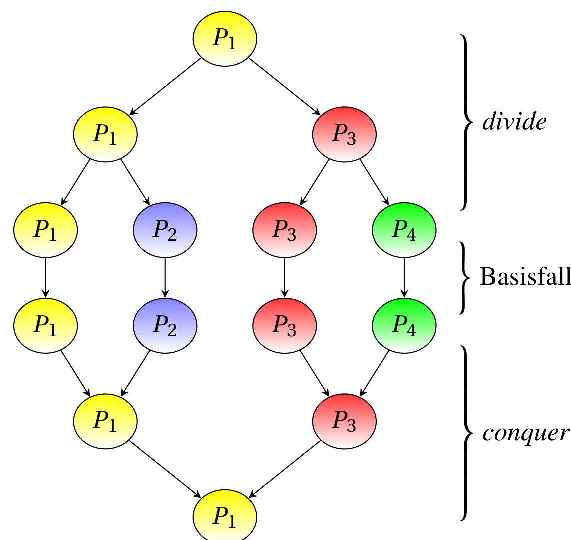


Abbildung 3.1. Parallele Verarbeitung eines rekursiven *Divide-and-Conquer*-Algorithmus mit vier Prozessoren $P_1, \ldots, P_4$. Nach dem Fork-join-Modell.

Rechenlastverteilung nach dem Fork-join-Modell.[1]

3.1 Fork-join

Die Lastverteilung eines parallel ausgeführten Programms, das nach dem *Fork-join-Modell* abläuft, kann als ein gerichteter azyklischer Graph (DAG) dargestellt werden (Abbildung 3.2), der sogenannte *Kostengraph (cost graph)* des Programms. Hierbei stellt ein Knoten einen auf einem einzelnen Prozessor ablaufenden Prozessschritt oder Thread dar, eine Kante den Aufruf eines neuen Prozessschritts mit Datenübergabe. Bei einer Verzweigung *(fork)* werden mehrere

[1] http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, http://en.wikipedia.org/wiki/Fork%E2%80%93join_model

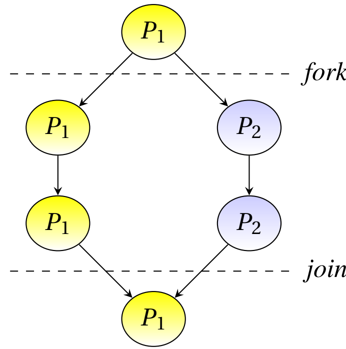


Abbildung 3.2. Lastverteilung eines Programms nach dem Fork-join-Modell mit zwei Prozessorkernen P_1 und P_2, dargestellt als dessen Kostengraph.

unabhängige Prozessschritte aufgerufen, bei einer Vereinigung *(join)* dagegen die Teilergebnisse mehrerer Prozessschritte synchronisiert und zusammengefasst und von einem einzelnen Prozessorkern weiterverarbeitet.

Die *Arbeit (work)* $w_p(n)$ eines parallel ausgeführten Programms ist definiert als seine effektive Laufzeit und hängt ab von der Problemgröße n, der Knotenzahl $k(n)$ seines Kostengraphen und der sequenziellen Laufzeit $t_s(n)$, d.h. der Laufzeitsumme aller einzelnen auf den Knoten ablaufenden Teilprozesse:

$$T_1(n) = w_p(n) = k(n)\, t_s(n). \tag{3.1}$$

Die Arbeit ist also die Laufzeit $T_1(n)$, die ein einzelner Prozessor für den Algorithmus benötigen würde. Demgegenüber bestimmt die Höhe $h(n)$ des azyklischen Graphen die Laufzeit $T_\infty(n)$, die im schlimmsten Fall bei unendlich viel verfügbaren Prozessoren benötigt werden. Für die Laufzeit $T_p(n)$ im Falle von p verfügbaren Prozessoren gilt allgemein [KM, §2012_4]

$$\frac{T_1(n)}{p} \leqq T_p(n), \qquad T_\infty(n) \leqq T_p(n) \leqq T_1(n) \qquad (p \in \mathbb{N}), \tag{3.2}$$

und für eine asymptotisch optimale Laufzeit T_p^* gilt per Definition

$$T_p^*(n) = \frac{T_1(n)}{p} + T_\infty(n). \tag{3.3}$$

(Hier dominiert der erste Summand für kleine p, der zweite für große p.) Für Map-Reduce-Operationen gilt beispielsweise

$$T_\infty(n) = O(\log n), \qquad T_1(n) = O(n), \qquad T_p^*(n) = O\left(\frac{n}{p}\right) + O(\log n). \tag{3.4}$$

In Java wurde das Fork-join-Modell mit den Klassen `ForkJoinTask` und `ForkJoinPool` im Paket `java.util.concurrent` implementiert.[2] Es garantiert eine Laufzeit

$$T_p(n) \leqq \text{const} \cdot \left(\frac{T_1(n)}{p} + T_\infty(n)\right). \tag{3.5}$$

3.2 Präfixsummen

Die n-te Partialsumme s_n einer Folge $(a_0, a_1, \ldots)$ ist definiert als die Summe ihrer ersten n Glieder,

$$s_n = \sum_{k=1}^{n} a_k. \tag{3.6}$$

[2] `http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html`

In der Informatik wird mit *Präfixsumme (prefix sum)* der Algorithmus bezeichnet, der jeden Eintrag a_k eines Arrays durch die Partialsumme s_k ersetzt [ES, ParallelAlgorithms.pdf]. Eine

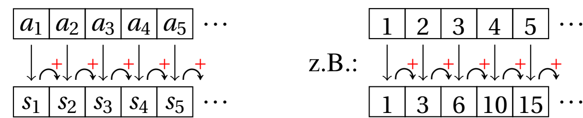


Abbildung 3.3. Präfixsumme eines Arrays.

sequenziell programmierte Routine für die Präfixsumme eines `int`-Arrays sieht in Java wie folgt aus:

```
1  public static void prefixSum(int[] a) {
2     for(int k = 1; k < a.length; k++) {
3        a[k] = a[k-1] + a[k];
4     }
5  }
```

Die Zeitkomplexität dieses Algorithmus ist $T_s(n) = O(n)$, die Speicherkomplexität $S_s(n) = O(1)$. Ein parallelisierbarer Algorithmus dagegen ist wie folgt gegeben:[3]

```
0. s[1] = a[1];
1. for k = 1, ..., n/2 :
      y[k] = a[2k-1] + a[2k]; // Hilfsarray y[1..n/2]
2. for k = 1, ..., n/2:
      berechne rekursiv z[k] = y[1] + y[2] + ... + y[k]; // Hilfsarray z[1..n/2]
3. for k = 2, ..., n:
      if (k gerade) {
         s[k] = z[k/2];
      } else {
         s[k] = z[(k-1)/2] + a[k];
      }
return;
```

Die Schritte 1 und 2 können jeweils parallelisiert ausgeführt werden, müssen nach ihrer Ausführung jedoch gejoint werden. Bei optimaler Prozessorzahl $p_* = n/\log n$ ergibt sich eine Laufzeit

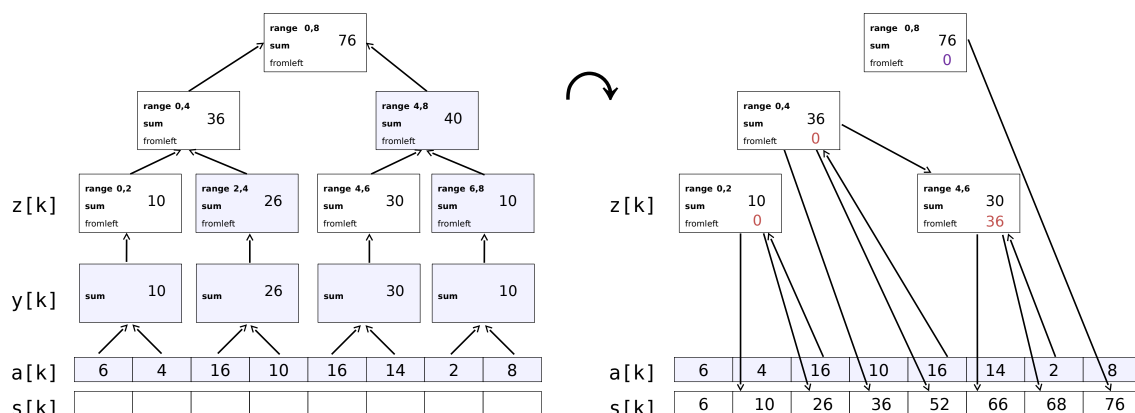


Abbildung 3.4. Parellele Berechnung der Präfixsumme.

3 http://en.wikipedia.org/wiki/Prefix_sum

pro Prozessor von $t_p = O(\log n)$, d.h. die Arbeit beträgt $w_p(n) = O(n)$ [KM, 2012_5]. Die Präfixsummierung kann auf vielfältige Weise eingesetzt werden und hat sich zu einem Basisalgorithmus paralleler Rechnerarchitekturen entwickelt. Beispielsweise kann er verwendet werden, um Additionen parallel zu ermöglichen oder dünn besetzte Arrays zu komprimieren [ES, ParallelAlgorithms.pdf].

In Java stellt die Klasse `Arrays` im Paket `java.util` Präfixsummierngen in mehreren Varianten für die primitiven Datentypen `int`, `long` und `double` bereit. Die statische Methode `parallelPrefix` erwartet zwei Parameter, ein Array und eine assoziative (!) Funktion $f(x, y)$, die die Kumulierung festlegt.

3.3 Reduce-Joins

Experimentell wurden die in Tabelle 3.1 dargestellten Join-Strukturen für ein Reduce auf einer Java VM der Version OpenJDK 1.8.0_121 auf Ubuntu Linux mit 4 Prozessorkernen festgestellt.

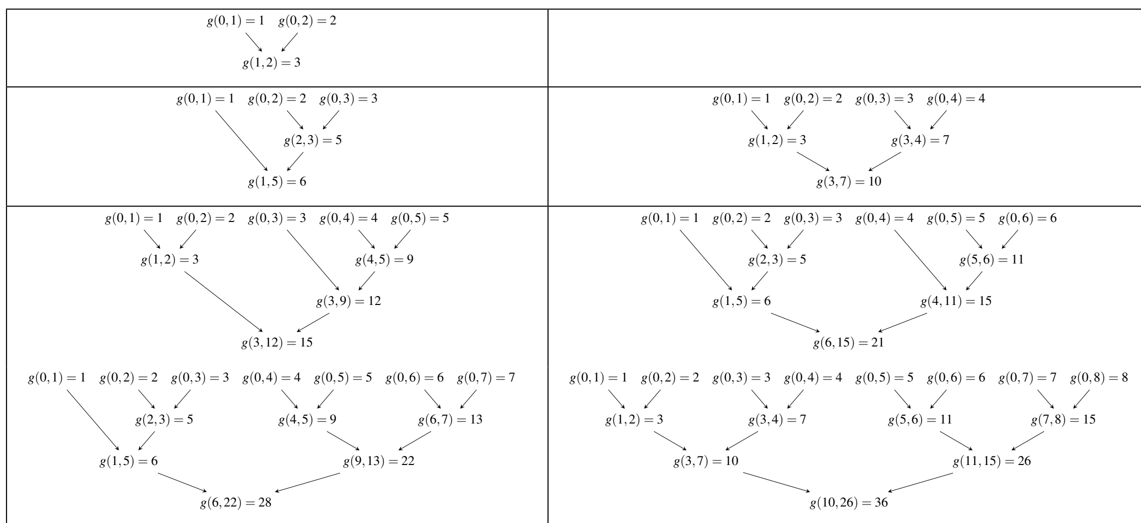


Tabelle 3.1. Joinstrukturen der Java VM (OpenJDK version 1.8.0_121 auf Ubuntu Linux mit 4 Prozessorkernen) für die assoziative reduce-Funktion $g(a, x) = a + x$.

Kapitel 4

Übungsaufgaben

Aufgabe 4.1. Welche der folgenden Lambda-Ausdrücke sind gültige Implementierungen des Interfaces `Function<Long,Long>`?

```
x -> x+1;
(x,y) -> x + y;
x -> x == 1;
```

Welche Interfaces können die anderen beiden Lambda-Ausdrücke implementieren?

Aufgabe 4.2. (a) Erläutern Sie kurz den Zusammenhang zwischen Streams und Collections in Java.

(b) Was sind intermediäre Streamoperationen und wann können sie auf parallelisierten Streams ausgeführt werden?

(c) Gegeben sei die Pipeline in folgendem Quelltextausschnitt:

```
List<Integer> liste = Arrays.asList(1,2,3,4,5);
int summe = liste
  .stream()
  .reduce(5, (a,x) -> a*x + 2);
```

Wie wird die Liste in einen parallelen Stream umgeformt? Liefert die Pipeline für einen parallelisierten Stream stets das korrekte Ergebnis?

Aufgabe 4.3. Es sollen alle Zahlen in einer Liste miteinander multipliziert und das Ergebnis mit 5 multipliziert werden. Die Pipeline im folgenden Quelltext führt diese Aktion auch erfolgreich durch.

```
List<Long> liste = Arrays.asList(1L,2L,3L,4L,5L,6L,7L,8L,9L,10L,11L,12L,13L,14L,15L);
long summe = liste
   .stream()
   .reduce(5L, (a,x) -> a*x);
```

Nach einer Umformung in einen parallelen Stream allerdings liefert diese Pipeline ein falsches Ergebnis. Finden Sie die Ursache und ändern die Pipeline, so dass sie auch parallelisiert korrekt läuft. (Hinweis: Beachten Sie die ursprüngliche Zielsetzung des Programms.)

Aufgabe 4.4. Mit den Anweisungen

```
int[] liste = new int[n];
Arrays.parallelSetAll(liste, i -> i+1);
```

kann man ein `int`-Array `liste` erzeugen, das die Einträge `[1, 2, ..., n]` hat. Testen sie damit experimentell, ab welcher Arraygröße Ihr Rechner die Präfixsumme echt parallel berechnet. (Bei meinem Ubuntu-Rechner mit 4 CPU-Kernen ist dies für $n \geqq 33$ der Fall.)

Anhang A

Hintergrund und Vertiefung

A.1 Amdahl'sches und Gustafson'sches Gesetz

Sei t_s die Laufzeit des seriellen Teils eines Programms, $t_p(N)$ die Laufzeit des parallelisierbaren Teils des Programms auf einer Rechnerarchitektur mit N Prozessoren. Dann gilt für die Gesamtlaufzeit $T(N)$ des Programms

$$T(N) = t_s + t_p(1)/N, \tag{A.1}$$

wenn wir annehmen, dass die parallelisierbaren Teile ideal auf die N Prozessoren aufgeteilt werden können, so dass sie alle gleich lang arbeiten, d.h.

$$t_p(1) = N\,t_p(N), \qquad \text{oder} \qquad t_p(N) = t_p(1)/N. \tag{A.2}$$

Für den Beschleunigungsfaktor $S(N) = T(1)/T(N)$ gegenüber einer Berechnung mit einem Einprozessorrechner gilt

$$S(N) = \frac{T(1)}{T(N)} = \frac{t_s + t_p(1)}{t_s + t_p(1)/N}. \tag{A.3}$$

Definiert man den *nichtskalierten parallelisierbaren Anteil* p_A der Laufzeit des Programms auf *einem* Prozessor durch

$$p_A = \frac{t_p(1)}{t_s + t_p(1)} \tag{A.4}$$

so erhält man nach Umformung[1]

$$\boxed{S(N) = \frac{1}{(1-p_A) + p_A/N} \leqq \frac{1}{1-p_A}.} \tag{A.5}$$

Das ist das *Amdahl'sche Gesetz* (1967). Es besagt also insbesondere, dass es für den Beschleunigungsfaktor einen Sättigungswert $S_{\max}(p_A) = \frac{1}{1-p_A}$ mit $S(N) \leqq S_{\max}(p_A)$ gibt (egal wie groß N auch sein mag!), der für ein gegebenes Programm allein durch seinen nichtskalierten parallelisierbaren Anteil p_A gegeben ist.

Beispiel A.1. Gegeben sei ein für einen Einzelprozessor geschriebenes Programm, das einen parallelisierbaren Anteil von 50% habe. Mit (A.5) ergibt sich damit auf einem Dual-Core-Prozessor ($N = 2$) bzw. einem 2^n-Prozessorsystem jeweils eine Laufzeitbeschleunigung von

$$S(2) = \frac{1}{\frac{1}{2} + \frac{1}{4}} = \frac{4}{3}, \qquad S(2^n) = \frac{1}{\frac{1}{2} + \frac{1}{2^{n+1}}} = \frac{2^{n+1}}{2^n + 1}. \tag{A.6}$$

[1] $S(N) = \dfrac{t_s + t_p(1)}{t_s + t_p(1)/N} = \dfrac{1}{\frac{t_s}{t_s+t_p(1)} + \frac{1}{N}\frac{t_p(1)}{t_s+t_p(1)}} = \dfrac{1}{(1-p_A) + p_A/N}.$

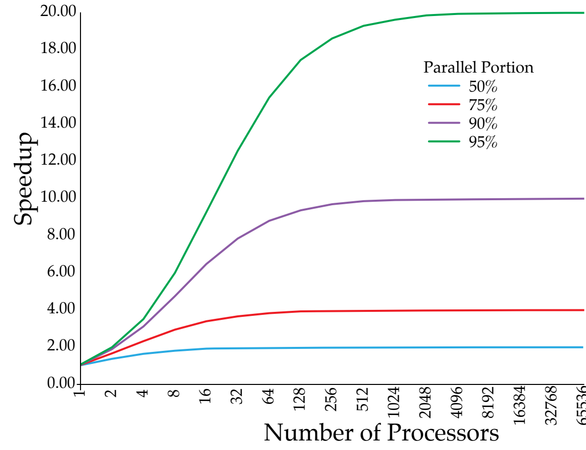


Abbildung A.1. Das Amdahl'sche Gesetz (A.5): Speedup $S(N)$ abhängig von der Anzahl N der Prozessoren

Entsprechend gilt $S(4) = \frac{8}{5}$ für ein Quad-Core-System ($N = 4$ oder $n = 2$). Der maximale Beschleunigungsfaktor für das Programm lautet also $S_{\max}(\frac{1}{2}) = 2$. □

Verwendet man allerdings statt des Anteils (A.4) den *skalierten parallelisierbaren Anteil* $p_G = p_G(N)$ der Laufzeit des Programms auf N Prozessoren,

$$p_G = \frac{t_p(1)}{t_s + t_p(N)} \overset{(A.2)}{=} \frac{t_p(1)}{t_s + t_p(1)/N} \quad \overset{N\to\infty}{\longrightarrow} \quad \frac{t_p(1)}{t_s}, \tag{A.7}$$

so ist[2]

$$\boxed{S(N) = 1 + (N-1)\,p_G = O(N).} \tag{A.8}$$

Das ist das *Gustafson'sche Gesetz* (1988). Der Unterschied zum Amdahl'schen nichskalierten Anteil p_A ist also, dass beim Gustafson'schen Anteil der Laufzeitgewinn des parallelisierbaren Anteils in Gleichung (A.2) durch die N Prozessoren berücksichtigt ist (und als ideal angesehen wird). Die beiden Anteilsparameter p_A und p_G sind voneinander abhängig, es gilt

$$p_A = \frac{N p_G}{1 + (N-1) p_G} \qquad \text{oder} \qquad p_G = \frac{p_A}{(1-p_A)\,N + p_A}. \tag{A.9}$$

Gustafson's law argues that even using massively parallel computer systems does not influence the serial part and regards this part as a constant one. In comparison to that, the hypothesis of Amdahl's law results from the idea that the influence of the serial part grows with the number of processes.

Beispiel A.2. Gegeben sei ein Programm, das auf einem 2^n-Kernprozessor einen skalierten parallelisierbaren Anteil von $p_G(2^n) = 50\%$ habe. Mit (A.8) ergibt sich damit für dieses Programm eine Laufzeitbeschleunigung gegenüber einem Einzelprozessor von

$$S(2^n) = \frac{1}{2} + \frac{2^n}{2} = \frac{2^n + 1}{2} = 2^{n-1} + \frac{1}{2}. \tag{A.10}$$

Mit (A.9) ist der nichtskalierte parallelisierbare Anteil $p_A = \frac{2^{n-1}}{1+(2^n-1)/2} = \frac{2^n}{2^n+1}$. Für einen Dual-Core-Prozessor ($n = 1$) ist also $S(2) = \frac{3}{2}$ und $p_A = \frac{2}{3}$, ganz im Einklang mit dem Amdahl'schen Gesetz (A.5). □

[2] Mit (A.1) folgt $t_s = (1-p_G)T(N)$ und $t_p(N) = p_G T(N)$, also mit (A.2) sofort $T(1) = t_s + t_p(1) = t_s + N t_p(N) = (1 - p_G + p_G N)\,T(N)$,

Historisch wurde das Amdahl'sche Gesetz vorwiegend pessimistisch im Hinblick auf die Effizienz massiv-paralleler Prozessorarchitekturen interpretiert. In der Tat konnte erst Gustafson über 20 Jahre später die vorherrschende Skepsis beenden. Dennoch ist das Amdahl'sche Gesetz sehr aussagekräftig für Programme, die vorwiegend sequenziell programmiert nun einfach auf Mehrkernprozessoren ablaufen: Der Laufzeitgewinn ist in diesen Fällen sehr gering [3].

A.2 Rückruffunktionen

Ein wichtiges Programmierkonzept der funktionalen Programmierung sind Rückruffunktionen. Eine *Rückruffunktion (callback function)* bezeichnet in der Informatik eine Funktion, die einer anderen Funktion als Parameter übergeben und von dieser unter gewissen Bedingungen aufgerufen wird. Dieses Vorgehen folgt dem Entwurfsmuster der Kontrollflussumkehr *(Inversion of Control, IoC)*. Als ein einfaches Beispiel soll das folgende kurze Programm diese Eigenschaften illustrieren:

```
1  import java.util.Arrays;
2  import java.util.function.Function;
3  import java.util.stream.Stream;
4
5  public class Callback {
6     public static Function<Integer, Integer> callback = x -> x*x;
7
8     public static void main(String... args) {
9        Stream<Integer> stream = Arrays.stream(new Integer[]{1, 2 , 3, 4, 5});
10       Stream<Integer> squares = stream.map(callback);
11       int sum = squares.reduce(0, (a,n) -> n + a); // => 55
12    }
13 }
```

Hier wird die Funktion `callback`$(x) = x^2$ in Zeile 10 von der API-Funktion `map` eines Streams aufgerufen und dort auch ausgeführt. Ihr „Rückruf" ergibt einen Stream, der in jedem Eintrag

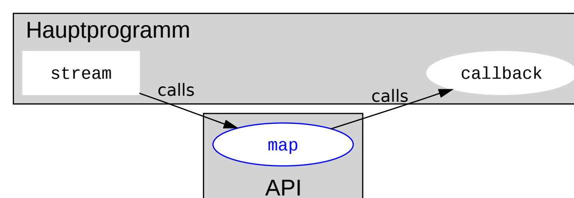


Abbildung A.2. Aufrufstruktur einer Rückruffunktion. (Modifiziert nach `https://en.wikipedia.org/wiki/File:Callback-notitle.svg`)

die Quadratwerte des ursprünglichen Streams enthält. Die Aufrufstruktur ist in Abbildung A.2 skizziert.

Es gibt zwei Arten von Rückruffunktionen. Eine Rückruffunktion heißt *blockierend*, oder auch *synchron*, wenn sie mit ihrer Referenz aufgerufen wird und ihr Ausführungsergebnis abgewartet wird, um weiterverarbeitet zu werden. Das obige Programmierbeispiel ist eine solche synchrone Rückruffunktion.

Bei einer *verzögerten (deferred)* oder *asynchronen Rückruffunktion* wird die Rückruffunktion in einem anderen Thread ausgeführt, das Hauptprogramm wartet aber nicht auf das Ergebnis der Rückruffunktion. Stattdessen wird das Ergebnis der Rückruffunktion „möglichst schnell"

verarbeitet, wenn es eingetroffen ist. Typische Anwendungsfälle solcher verzögerten Rückruffunktionen sind Ereignisbehandler, die nach einem gefeuerten Ereignis die Rückfruffunktion ausführen, oder Aufrufe serverseitiger Skripte im Web.

In der nebenläufigen Programmierung spricht man bei asynchronen Rückruffunktionen auch oft von *Promises*.[3]

A.3 Anmerkungen zur Parallelisierung in Java 8

Im Wesentlichen gibt es zwei Arten von Funktionalität, für die eine Parallelisierung auf mehrere Threads oder Prozessoren lohnt. Zum Einen ist dies *rechenintensive Funktionalität*, also Algorithmen mit hoher Laufzeit und hohem CPU-Bedarf. Zum Anderen ist da die *blockierende Funktionalität*, die wenig CPU-Zeit benötigt, aber auf Aufrufergebnisse warten muss. Typische Beispiele dafür sind synchrone Netzwerkanfragen über HTTP oder synchrone Datenbankzugriffe [8].

Das Parallelisierungskonzept in Java 8 nun ist jedoch dafür ausgelegt, optimiert rechenintensive Funktionalität zu parallelisieren, für blockierende Funktionalität ist es nur bedingt geeignet [7].

[3] `https://docs.angularjs.org/api/ng/service/$q#the-promise-api`

Literaturverzeichnis

[1] FORSTER, O. : *Analysis 1*. 9. Wiesbaden : Vieweg, 2008

[2] GRAMA, A. ; GUPTA, A. ; KARYPIS, G. : *Introduction to Parallel Computing*. Harlow : Pearson Education, 2003. – http://parallelcomp.uw.hu/index.html

[3] KREFT, K. ; LANGER, A. : '3. Die Kosten der Synchronisation'. In: *Java Magazin* 10 (2008), S. 109–112. – http://www.AngelikaLanger.com/Articles/EffectiveJava/39.JMM-CostOfSynchronization/39.JMM-CostOfSynchronization.html

[4] KREFT, K. ; LANGER, A. : 'Effective Java: Let's Lambda!'. In: *Java Magazin* 12 (2013), S. 21–27

[5] KREFT, K. ; LANGER, A. : 'I have a Stream ...'. In: *Java Magazin* 8 (2014), S. 16–23

[6] KREFT, K. ; LANGER, A. : 'Was sind Streams?'. In: *Java Magazin* 6 (2014), S. 18–22

[7] KREFT, K. ; LANGER, A. : 'Java 8: Stream Performance'. In: *Java Magazin* 10 (2015), S. 28–36

[8] KREFT, K. ; LANGER, A. : 'Einmal um den Block'. In: *Java Magazin* 4 (2016), S. 11–19

[9] OECHSLE, R. : *Parallele Programmierung mit Java Threads*. München Wien : Carl Hanser Verlag, 2001

[10] PIEPMEYER, L. : *Grundkurs Funktionale Programmierung mit Scala*. München Wien : Carl Hanser Verlag, 2010. – http://www.grundkurs-funktionale-programmierung.de/

[11] WARBURTON, R. : *Java 8 Lambdas. Functional Programming for the Masses*. Sebastopol : O'Reilly, 2014. – http://oreilly.com/catalog/errata.csp?isbn=9781449370770

[12] ZEIDLER, E. (Hrsg.): *Teubner Taschenbuch der Mathematik. Teil 1*. Leipzig : B. G. Teubner, 1996

Internetquellen

[ES] Stefan Edelkamp (2010): *Algorithm Engineering.* Lecture notes. http://www.tzi.de/~edelkamp/lectures/ae/ [2017-01-27]

[KM] Michelle Kuttler (2012): *Multithreading in Java.* Lecture notes. https://vula.uct.ac.za/access/content/group/7a800d75-b55a-4e05-91da-7341b8692710/ParallelJava/Slides/ [2017-01-27]

[LK] Angelika Langer und Klaus Kreft (2013): *Lambda Expressions in Java. Reference*, S. 12 & S. 49ff, `http://www.angelikalanger.com/Lambdas/LambdaReference.pre-release.pdf` [2014-06-04]

[SP] Java Streams und Pipelines. `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps` [2017-01-27]

Index

`->`, 4
`::`, 9
`DoubleStream`, 13
`ForkJoinPool`, 23
`ForkJoinTask`, 23
`IntStream`, 13
`LongStream`, 13
`Stream`, 13
`filter`, 14
`forEach`, 14
`limit`, 21
`map`, 14
`parallelPrefix`, 25
`parallelStream`, 15
`parallel`, 14
`peek`, 14
`reduce`, 14
`sequential`, 14
`skip`, 21
`sorted`, 21

Aggregatoperation, 12
aggregierte Operation, 14
Amdahl’sches Gesetz, 28
anonyme Funktion, 4
apply, 5
Arbeit (paralleler Algorithmus), 23
assoziativ, 18
asynchrone Rückruffunktion, 30

blockierende Funktionalität, 31
blockierende Rückruffunktion, 30
Bulk Operation, 14

Callback-Funktion, 5
Closure, 6

Datenstrom, 12
datenstromorientiertes Kommunikationsmodell, 12
deferred callback, 30
divide-and-conquer, 21
DoubleStream, 13

eager Evaluation, 13
effektiv final, 6
Ereignisbehandler, 8, 31
Event Handler, 8
Event Listener, 8

filter, 14
fold, 16
forEach, 14
fork and join, 22
Funktion, 4
Funktion, innere –, 6
funktionale Interfaces, 4
funktionale Programmiersprache, 3
Funktionalkalkül, 6
Funktionenschar, 6
Funktionsauswertung, 5

Gustafson’sches Gesetz, 29

höhere Ordnung, Funktion, 7

imperative Programmierung, 3
impliziter Typbestimmung, 4
innere Funktion, 6
IntStream, 13
Inversion of Control, 30
IoC, 30

Join, 18

Kontrollflussumkehr, 30
Kostengraph, 22

Lambda-Ausdruck, 4
lazy Evaluation, 13
limit, 21
LongStream, 13

map, 14
Methodenreferenz, 9

Nebeneffekt, 3

parallel, 14
parallelPrefix, 25
parallelStream, 15
peek, 14
Pfeil-Symbol, 4
Pipe, 12
Pipeline, 12
Präfixsumme, 24
Promise, 31

Quicksort, 21

Rückruffunktion, 5, 30
rechenintensive Funktionalität, 31
reduce, 14, 16
Rekursion, 9

Schar, Funktionen-, 6

Seiteneffekt, 3
sequential, 14
skip, 21
sorted, 21
Stream, 13
synchrone Rückruffunktion, 30

verzögerte Rückruffunktion, 30

zustandslose Streamoperation, 21