Vorlesung

15

Entwurf von Algorithmen

Inhalt

Vorlesung 15

Lernziele: Strukturierung eines Problems und Lösung durch systematische Vorgehensweisen. Systematischer Entwurf und Testen der Lösung, Idee und Einsatz von Greedy-, Divide-and-Conquer- und Backtracking-Algorithmen. Aspekte der Parallelisierung von Code. Grenzen der Parallelisierung.

1 Entwurfsmethoden

Für die Lösung eines Problems mit einem Programm gibt es keine Patentrezepte, aber einige bewährte Vorgehensweisen, die wir in diesem Kapitel besprechen möchten. Allen gemeinsam ist die Erkenntnis, dass man zuerst das Problem gedanklich durchdringen sollte und sich dann erst hinsetzt, um den Code in einer geeigneten Programmiersprache zu schreiben.
Bei den dazu verwendeten Beispielen gibt es immer nur eine endliche Anzahl von Lösungen, die durch eine endliche Anzahl von Parametern $\mathbf{c} = (P_1, \ldots, P_N)$ gekennzeichnet sind. Jeder der Parameter kann mehrere Werte annehmen, so dass er als „Dimension" des Problems aufgefasst werden kann. Die Gesamtmenge aller Lösungen wird deshalb auch als „**Lösungsraum**" bezeichnet. Die gesuchte Lösung **c*** ist dann nur ein ganz bestimmter Punkt in dem N-dimensionalen Lösungsraum.

Der einfachste Fall liegt vor, wenn ich nur eine Sequenz von Befehlen hinschreiben muss, um das Problem zu lösen. Auch eine wiederholte Sequenz ist keine Schwierigkeit. Diese beginnt erst, wenn der Lösungsweg durch viele Bedingungen und Details unklar ist. Für diesen Fall möchte ich drei verschiedene Lösungsstrategien anbieten:

- Greedy-Methode
- Devide-and-Conquer („Salami-Taktik")
- Backtracking

Diese drei Methoden wollen wir nun näher betrachten.

1.1 Die „Greedy“-Methode

Die Greedy-Methode setzt darauf, dass die gesuchte Lösung Schritt für Schritt erreichbar ist und es dafür ausreicht, in jedem Schritt die jeweils beste Entscheidung zu treffen. Dazu benötigt das Verfahren sowohl eine Zustandsbeschreibung **c** (z.B. mehrere Zahlen oder Parameter) sowie eine Gütefunktion R(**c**) für die Entscheidung, welchen neuen Zustand man wählen sollte.

Ein typisches Beispiel für diese Problemklasse ist ein Optimierungsproblem: Gegeben sei ein Parametersatz **c** (eine mögliche Lösung des Problems) sowie eine Gütefunktion R(**c**), die Güte der Lösung berechnet. Gesucht ist nun eine Verfahren, das das optimale **c*** liefert mit der maximalen Güte R(**c***). Ein Beispiel für ein solches Greedy-Verfahren ist die „Evolutionäre Optimierung“: Wir bilden mehrere zufällige Abweichungen **c**’ von unserem initialen Zustand **c** und wählen davon diejenige aus, die das größte R(**c**’) liefert. Den neuen Zustand **c**’ nehmen wir nun als Ausgangszustand und suchen wieder Abweichungen von **c**’, um einen noch besseren Zustand zu finden. Am Ende hoffen wir, uns mit unserem Zustand dem besten Zustand **c*** genügend genähert zu haben und den maximal möglichen Wert von R(**c**) zu erreichen.

Ein anderes Beispiel ist ein selektiver Sortieralgorithmus, den wir näher betrachten wollen. Unser Parametersatz **c** ist in diesem Fall der Satz der Indices (1,…, N) der Zahlen der initialen Folge. Die Gütefunktion ist in diesem Beispiel nicht direkt definiert, sondern wir bemerken nur jede Verbesserung.

Beispiel *Selektives Sortieren*

Sei eine Liste **v** von N=7 natürlichen Zahlen gegeben, die in aufsteigender Reihenfolge sortiert werden sollen, beispielsweise

v = [87, 65, 1, 78, 39, 114, 5], **c** = (1,2,3,4,5,6,7)

Wie können wir diese sortieren, so dass wir die Liste

v* = [1, 5, 39, 65, 78, 87, 114] **c*** = (3,7,5,2,4,1,6)

erhalten?

Dazu gehen wir wie folgt vor:

- Suche eine Liste v’, die besser ist: Finde zuerst das kleinste Element in der Folge der N Elemente und vertausche es mit dem Element in der ersten Position. Dies erhöht sicher die Güte der Lösung **c**.
- Suche eine Liste v”, die noch besser ist: Finde das zweit-kleinste Element (also das kleinste Element unter den verbliebenen N-1 Elementen) und vertausche es mit dem Element in der zweiten Position. Dies erhöht sicher die Güte der Lösung.
- Fahre in dieser Weise fort, bis die gesamte Folge sortiert ist.
- Der entsprechende Python-Code lautet dafür wie folgt:

```
def selection_sort(v):
for p in range(len(v)-1):     # Für alle Werte im Bereich 0..N-2
   # Sei v bereits sortiert im Bereich [0,…,p-1]
   vmin = p
   # vmin ist der Index des aktuell als minimal betrachteten Wertes
   for q in range(p, len(v)):
       if v[q] < v[vmin]:
                  vmin = q

   temp = v[vmin]       # Austausch
   v[vmin] = v[p]       # von v[vmin]
   v[p] = temp          # und v[p]
return v
```

Der Code besteht aus zwei ineinander geschachtelten Schleifen. Er geht die gesamte Liste durch und sucht für jedes Element v[p] aus den restlichen Elementen mittels der zweiten Schleife den Index *q* des kleinsten Elements. Ist das Element gefunden, so wird das Element v[q] mit dem aktuellen Element v[p] vertauscht. Da das nächste Element v[p+1] nur größer sein kann, wird auch durch Vertauschen die Folge der v[p] nur größer.

Die Aufgabe, vorgegebene Daten nach irgendwelchen Kriterien zu sortieren, ist ein klassisches Problem der Informatik und hat viele Lösungen. Gerade bei großen Datenbanken ist die Wahl eines schnellen Algorithmus, also eines Algorithmus mit geringem Berechnungsaufwand und damit geringer Laufzeit, ziemlich wichtig.
In unserem Fall können wir ihn wie folgt ermitteln:
Um das kleinste Element zu finden, müssen wir beim ersten Schritt *N* Elemente betrachten, beim zweiten Schritt eins weniger, also maximal *N-1*, beim dritten *N-2* und so fort. Dies sind also maximal

$$N + (N-1) + (N-2) + \ldots + 2 + 1 = \sum_{i=1}^{N} i = \frac{(N+1)N}{2} = \frac{N^2 + N}{2}$$

Schritte. Ist die Zahl *N* der Elemente groß, so ist $N^2 \gg N$ der dominierende Term und wir sagen: „Die Laufzeit des *Selection_Sort*- Algorithmus ist in der Größenordnung von N^2“.
Die allgemeine Laufzeit eines Sortieralgorithmus ist aber nicht das einzige Gütekriterium. In der Praxis gibt es viele weitere Punkte, die für die Wahl des optimalen Sortieralgorithmus wichtig sind:

- Welche Daten sollen sortiert werden (Integer, Strings, …) ?
- Wie groß ist der Datensatz? Manche Algorithmen sind nur auf großen Datensätzen gut.
- Nach welcher Vergleichsfunktion soll sortiert werden?
- Wie weit sind die Daten vorsortiert?
- Sind doppelte Einträge vorhanden?
- Wie viel Speicher steht zur Verfügung?
- Wieviele Prozessoren stehen zur Verfügung?

Das N-Damen-Problem

Ein weiteres Beispiel für den Einsatz der *Greedy*-Methode ist das sog. N-Damen-Problem.

Beispiel

Gegeben sei ein Schachbrett mit N Spalten und N Zeilen. Zur Veranschaulichung sei eines mit N=4 gezeigt. Aufgabe ist, N = 4 Damen darin so anzuordnen, dass sie sich nicht in einem Zug schlagen können. Zur Erinnerung: Eine Damen kann horizontal, vertikal sowie diagonal rechts und links ziehen. Eine gültige Lösung ist in Abb. 15.1 gezeigt und besteht aus vier Zahlen $\mathbf{c} = (P_1,P_2,P_3,P_4)$ für die vier Spalten des Schachbretts. Jede der Spalten P_i kann nur eine von vier möglichen Zuständen enthalten, bei dem eine Dame auf eines der vier Felder der Spalte platziert wird.

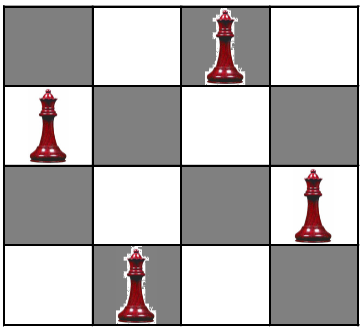

Abb. 15.1 *Eine Lösung $c = (2,4,1,3)$ für das N-Damen-Problem bei $N = 4$*

Wie könnte man das Problem mit dem Greedy-Ansatz lösen? Wir können zwar N Damen zufällig so auf N Reihen positionieren, dass nur jeweils eine Damen pro Reihe existiert, aber dies garantiert uns nicht, dass auch nur pro Spalte eine Dame vorhanden ist und die Damen sich nicht gegenseitig schlagen können (wir machen dabei keine Unterschiede zwischen schwarzen und weißen Damen; jede kann jede schlagen). Bei jeder Dame gibt es also eine Reihe von Konflikten; wenn nicht, hätten wir schon die Lösung. Um die Anzahl der Konflikte zu verringern, suchen wir uns nach unserem zufallsbedingten Anfang die Dame, die die meisten Konflikte hat, und verschieben sie solange in ihrer Spalte, bis die Konfliktzahl minimal ist. Dann suchen wir wieder unter allen Damen die Dame mit der höchsten Konfliktzahl, verschieben sie auf der Spalte und reduzieren die Gesamtzahl der Konflikte so wieder ein Stück, und so fort, bis es keine Konflikte mehr gibt.

Dieser Algorithmus ist zwar intuitiv, aber er garantiert uns nur ein lokales Minimum: Was machen wir, wenn am Ende noch Konflikte existieren und keine Verschiebung irgendeiner Dame die Konfliktzahl mehr mindern kann? In diesem Fall sind wir mit unserem Latein am Ende: Wir setzten die Damen noch einmal neu in Zufallspositionen und beginnen von vorn.

Es gibt noch viele weitere Beispiele für Greedy-Algorithmen, etwa die Suche nach „minimalen Spannbäumen“ in Graphen.

Minimale Spannbäume

Was ist ein Spannbaum? Lässt man von einem Graphen gezielt Kanten (aber keine Knoten) weg, so dass der Graph aus den verbleibenden Knoten und Kanten einen Baum bildet, so „spannt“ der Baum den Graphen „auf“.

DEF *Ein **Spannbaum** ist ein Teilgraph eines ungerichteten Graphen, der ein Baum ist und alle Knoten des Graphen enthält.*

Ein solcher Graph kann beispielsweise die Router in einem Kommunikationsnetz beschreiben. Der Spannbaum enthält dann alle Leitungen, um alle Stationen von einem zentralen Router aus zu erreichen. Notiert man noch die Übertragungsgeschwindigkeit der Leitungen bei den Kanten des Graphen, so wird der Graph als „markiert“ oder kantengewichtet“ bezeichnet.

DEF *Ein Spannbaum heißt* ***minimal****, wenn in demselben Graphen kein anderer Spannbaum mit geringeren Kosten existiert.*

Ein Beispiel für einen solchen minimalen Spannbaum (*minimal spanning tree*, MST) ist in Abb. 15.2 zu sehen.

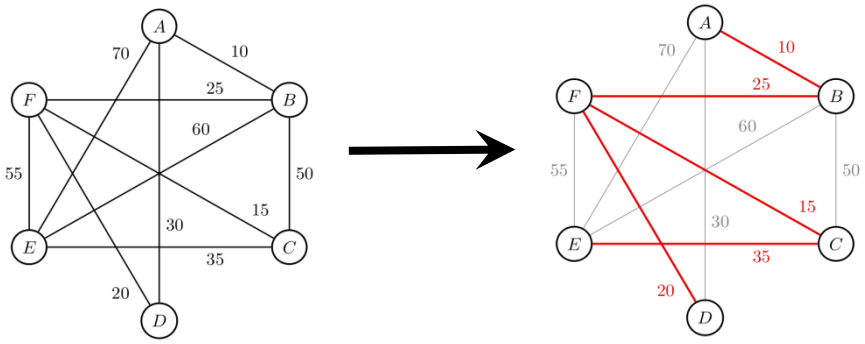

Abb. 15.2 (a) Ein kantengewichteter Graph und (b) ein möglicher minimaler Spannbaum

Allgemein kann es mehrere unterschiedliche minimale Spannbäume mit gleichen Kosten geben.
Verwendung finden minimale Spannbäume bei allen Problemen, bei denen kostengünstige, zusammenhängende Netze, etwa für Kommunikation oder zur Vermeidung von Mehrfachübertragung bei Sprachpaketen, genutzt werden. Aber auch zur Lösung von Problemen bei anderen gewichteten Netzwerken, etwa von Signalwegen in Gen-Netzwerken, können sie beitragen.
Wie kann man solche minimalen Spannbäume automatisch erzeugen? Ein bekanntes Verfahren dafür ist der Greedy-Algorithmus von Kruskal. Seine Hauptidee ist folgende: *Füge fortgesetzt die kürzeste (billigste) Kante aus G hinzu, die zu keinem Zyklus im MST führt*. Dies können wir in folgenden Algorithmus gießen:

KRUSKAL-ALGORITHMUS:

1. *Sortiere die Kanten eines Graphen in aufsteigender Reihenfolge nach den Kosten.*
2. *Setze* $V_{MST} = \{\ \}$ *als Menge aller zusammenhängenden Knoten und beginne mit der ersten Kante.*
3. *Füge die Kante* $e = (v_i,v_j)$ *und die Knoten* v_i, v_j *zum MST hinzu, falls dadurch kein Zyklus erzeugt wird.*
4. *Ist* $V_G = V_{MST}$ *oder die letzte Kante der Liste behandelt worden, STOP.*
5. *Betrachte die nächste Kante und gehe zu 3.*

Bei der Implementierung müssen wir den Algorithmus mit Leben füllen. Dazu müssen wir einzelne Formulierungen wie „*falls dadurch kein Zyklus erzeugt wird*" konkretisieren. Ein Ansatz dazu besteht darin, Teilmengen C_i zu definieren, welche die Knoten des bereits erstellten MST enthalten.

> Sei C_i jeweils eine Menge aus Knoten, die miteinander verbunden sind. Bilde initial für n = *N* Knoten *N* Mengen $C_1,..,C_n$ aus jeweils einem Knoten. Betrachte bei jedem Schritt die Mengen $C_1,..,C_n$ der verbundenen Knoten.
>
> - *Zyklustest:* Eine Kante $e = (v_1,v_2)$ wird nur dann zum MST hinzugefügt, wenn sie nicht schon in einer Menge enthalten ist, also $v_1 \in C_i$, $v_2 \in C_j$ und $C_i \neq C_j$ gilt.
> - In diesem Fall wird ein $C_k = C_i \cup C_j$ erzeugt, das C_i und C_j ersetzt: die beiden Mengen werden verschmolzen.

Eine Implementierung des Algorithmus wird also zuerst ein Graph-Modul definieren mit Operationen für das Erzeugen eines Graphen `__init__`, Hinzufügen von Knoten `addvertex` und Kanten `addedge` sowie Auslesen von Knoten `getvertices`, Kanten `getedges` und Kosten `cost` einer Kante. Die Mengen C_i kann man über die Mengenfunktionen in Python realisieren. Definiert man eine Kante (v_1,v_2) über ein Element e[0] und die Kosten c_{12} als Element e[1], so ist die bewertete Kante $e = [(v_1,v_2), c_{12}]$.
Der Code des Algorithmus kann man dann folgendermaßen formulieren:

```
def kruskal(g):  # Eingabe: g Objekt der Klasse Graph
     sets = []                    # Liste (Menge) der Ci
     for v in g.getvertices():    # Einfügen von allen v's
          sets.append([v])        # in verschiedene Ci's
     mst = Graph()                # Erzeuge leeren MST
     for e in g.getedges():       # Durchlaufen der Kanten
                                    nach ihren Kosten sortiert
          v1 = e[0][0]            # Erster Knoten der Kante
          v2 = e[0][1]            # Zweiter Knoten der Kante
          s1 = findset(sets, v1) # Ci des ersten Knoten
          s2 = findset(sets, v2) # Cj des zweiten Knoten
          if s1 != s2:            # Wenn Ci ungleich Cj:
               mst.addedge(v1,v2,e[1])# Kante(+Kosten) hinzufügen
               joinset(sets,s1,s2)    # Ci und Cj vereinigen
          if len(sets) == 1:
               break  # Wenn nur noch eine verb.Knotenmenge: Ende
     return mst
```

Der Algorithmus stoppt also genau dann, wenn nur noch eine zusammenhängende Menge von Knoten vorhanden ist, oder aber wenn das Ende der Kantenliste erreicht wird.
Dabei lassen sich die Mengenoperationen `findsets` und `joinsets` wie folgt implementieren:

```
# Eingabe: Menge der C's und Knoten v
def findset(sets, v):
      for s in sets:       # Bei allen Listenelementen (Unterlisten)
         if v in s:        # wenn der Knoten in einer Unterliste ist
            return s       # gib die Unterliste zurück
# Ausgabe: Das Ci, das v enthält

# Eingabe: Menge der C's, s1=Ci und s2=Cj
def joinsets(sets, s1, s2):
      sets[sets.index(s1)] = s1 + s2
      sets.remove(s2)
# Ausgabe: die Vereinigung von s1 und s2 wird an Stelle von s1 abge-
speichert
```

1.2 Die Methode „Divide-and-Conquer“

Angenommen, das gegebene Problem ist für uns in der Gesamtheit nicht lösbar. Dann ist es eine gute Idee, das Problem in kleine, lösbare Unterprobleme zu zerteilen, diese Probleme dann zu lösen und die Teillösungen anschließend zur Gesamtlösung zusammenzufügen. Ist auch das Unterproblem nicht lösbar, so kann man auch hier wieder die gleiche Strategie anwenden und das Unterproblem in weitere Unter-Unterprobleme zerteilen. Dies führt zu einer Rekursion bei der Problemlösung.
Eine Veranschaulichung davon stammt von Knuth: Angenommen, man möchte einen großen Haufen Briefe im ganzen Land zustellen. Dann ist es sinnvoll, die Briefe zunächst in einzelne Säcke zu ordnen, für jedes Bundesland einen. Jeder einzelne Sack wiederum lässt sich in dem Bundesland wieder in kleinere Säcke aufteilen, für jeden Kreis (Postleitzahlenbereich) einen. Diese kleineren Säcke lassen sich im Kreis wieder in Orte, und pro

Ort in Straßenzustellbezirke aufteilen und dann vor Ort tatsächlich zustellen. Damit führt die Problemaufteilung nach mehreren Schritten tatsächlich zu einer Lösung.
Das Gesamtverfahren *divide and conquer* lässt sich so in folgende Schritte unterteilen:

1. Teile (**Divide**)

   Teile das Problem der Größe N in (wenigstens) zwei annähernd gleichgroße Teilprobleme, wenn N > 1 ist.
   Ist das Teilproblem nicht lösbar, so teile es erneut und führe so eine Rekursion durch.

2. Herrsche (**Conquer**)

   Ist ein Teilproblem hinreichend klein (z.B. N = 1), so löse es direkt und breche so die Rekursion im Lösungsschema ab.

3. Vereinige (**Merge**)

   Füge die Lösungen für die Teilprobleme zur Gesamtlösung zusammen.

Als Beispiel für diese Methode betrachten wir das Problem, eine gegebene Liste mit *N* Elementen zu sortieren, etwa nach der Größe der Elemente. Als Beispiel stellen einen Vertreter der Sortiermethoden vor, der auf Vergleichen zwischen den Elementen basiert.

Quicksort

Das Verfahren besteht aus folgenden Schritten:

- **Divide**: Teile die zu sortierende Liste in zwei Teillisten so, dass alle Elemente der einen Liste kleiner oder gleich allen Elementen der anderen Liste sind. Dazu wähle ein Element der Liste aus (das Pivot-Element), etwa das Letzte in der Liste. Dann ziehe man alle kleineren Elemente an den Anfang der Liste. Am Schluss wird das Pivot-Element mit einem garantiert größeren Element vertauscht, mit dem Element, das nach dem zuletzt gefundenen kleineren Element steht. Dadurch sind alle Elemente der Liste vor dem Pivot-Element kleiner (oder gleich), alle anderen danach größer. Dann teile man die Liste am Pivot-Element.

  Sortiere beide Teillisten sodann wieder rekursiv, d.h. wähle jeweils ein neues *Pivot*-Element in jeder Teilliste und sortiere die Teilliste, und führe dies sooft durch, bis es nur noch ein Element in einer Teilliste gibt.

- **Conquer**: Sortiere die Teillisten. Ist das Sortieren trivial, so ist die Rekursion abgebrochen.
- **Merge**: Da die beiden sortierten Teillisten bereits Teile der Gesamtliste sind, besteht die Zusammenfassung aus dem Setzen der beiden Indizes für Anfang und Ende der Liste.

Der Code dafür ist

```
"""
  A Python implementation of the quicksort algorithm:
  T. Cormen et al.: Introduction to Algorithms, 2nd ed., pp. 145
"""

def quicksort(A, p, r):
    if p < r:
        q = partition(A, p, r)      # devide it
        quicksort(A, p, q-1)        # and conquer
        quicksort(A, q+1, r)

def partition(A, p, r):
    pivot = A[r]                    # set new pivot element
    i = p - 1
    for j in range(p,r):
        if A[j] <= pivot:
            i = i + 1               # index of last smaller found
            exchange (A, i,j)       # shift it to the beginning
    exchange (A, i+1,r)       # shift pivot to last bigger element
    return i+1

def exchange(A,i,j):          # exchange two elements in the list
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

Es gibt auch Varianten des Quicksort. So kann man beispielsweise auch als *Pivot*-Element nicht das letzte, sondern das erste oder das mittlere Element der Liste wählen. Der Rest bleibt aber gleich bis auf den Austausch am Schluss der Vergleichsoperationen: Am Schluss schiebe das *Pivot*-Element an den Platz des zuletzt gefundenen kleineren Elements, so dass nun auch wieder alle Elemente vor dem *Pivot*-Element kleiner und die danach stehen größer sind als das *Pivot*-Element.

Ein anderes Beispiel für den Divide & Coquer-Ansatz ist die Sortiermethode „Merge Sort“, die auf dem elementweisen Vergleich zweier Listen basiert.

Merge sort

Als Beispiel wenden wir die Methode wieder auf das Sortierproblem an und gehen dabei folgendermaßen vor:

- **Divide**: Teile die zu sortierende Liste in zwei gleichgroße Teillisten und sortiere beide Teillisten rekursiv, d.h. teile sie solange, bis das Sortieren trivial wird, da es nur noch ein Element in der Liste gibt.
- **Conquer**: Sortiere die Teillisten. Ist das Sortieren trivial, so ist die Rekursion abgebrochen.
- **Merge**: Vereinige die beiden sortierten Teillisten zu einer sortierten Gesamtliste.

In Abb. 15.3 ist dies grafisch an unserem Beispiel dargestellt.

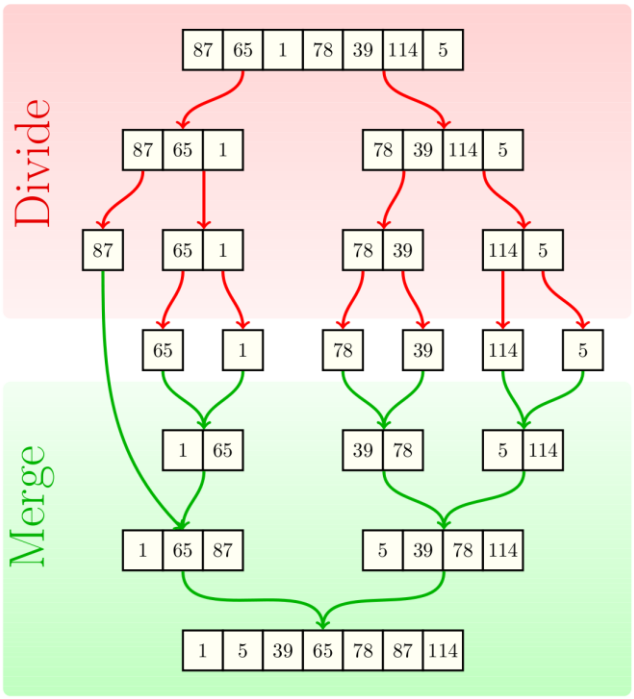

Abb. 15.3 *Die Teilung und Vereinigung der Teillisten bei* merge_sort.

Der rekursive Code ist in Python wie folgt:

```
def merge_sort(list):
if len(list) <= 1:              # Rekursionsverankerung
   return list
else:
   length = len(list)
   # Liste in zwei Teillisten teilen (divide)
   list1 = list[:length/2]
   list2 = list[length/2:]
   # Rekursiver Funktionsaufruf (conquer) auf die
   # Teillisten und Rückgaben vereinigen (merge)
   return merge(merge_sort(list1),merge_sort(list2))
```

Dabei ist Vereinigung von zwei Sublisten so definiert, dass durch beide Teillisten parallel durchgegangen wird und jeweils das kleinere Element beider Listen in die Hauptliste einsortiert wird, siehe Abb. 15.4.

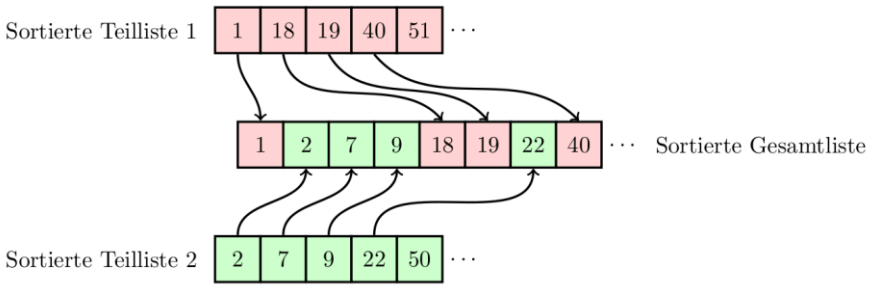

Abb. 15.4 *Die Vereinigung zweier vorsortierter Listen bei* merge_sort.

Der Code dafür sieht folgendermaßen aus:

```
def merge(list1, list2):
merged_list = []              #initialisiere leere Gesamtliste
while len(list1) != 0 and len(list2) != 0:
   # Solange beide Listen noch Elemente besitzen, wird das
   # kleinste Element aus den Teillisten entfernt und in
   # die Gesamtliste eingefügt
if list1[0] > list2[0]:
   merged_list.append(list2.pop(0))
else:
   merged_list.append(list1.pop(0))
   # Falls eine Teilliste noch Elemente enthält,
   # werden sie angehängt
if len(list1) != 0:
   merged_list.extend(list1)
if len(list2) != 0:
   merged_list.extend(list2)

return merged_list
```

Dabei gibt die Methode pop(0) das erste Element der Liste zurück.

Wie groß ist der Aufwand dieses Sortieralgorythmus? Betrachten wir unser Beispiel in Abb. 15.3. Für eine Liste der Länge *N* haben nach maximal $\log_2(N) + 1$ *Divide*-Ebenen alle Teillisten die Länge 1. Nach genau so vielen *Merge*-Ebenen entsteht die sortierte Gesamtliste. In jeder Ebene werden nun höchstens *N* Elemente bewegt. Die Laufzeit ist also in einer Größenordnung von $N \cdot \log_2(N)$.

1.3 Die Methode „Backtracking“

Beim *Greedy*-Verfahren wurden alle Lösungsschritte untersucht, die möglich sind, und dann der beste Schritt ausgewählt. Dies muss nicht zwangsläufig zu dem besten Lösungsweg führen, sondern kann auch zu einem suboptimalen, lokalen Optimum führen. Manchmal erreicht man die beste Lösung nicht auf dem geraden Weg, sondern auch durch Umwege, die anfangs nicht immer gut erscheinen.

Diesen Beobachtungen trägt das *Backtracking*-Verfahren Rechnung. Dabei wird ein gewählter Lösungsschritt weitergegangen und unter allen Alternativen ein weiterer Schritt gewählt solange, bis der Weg zu Ende ist, etwa weil die gewünschte Lösung nicht mehr möglich ist. In diesem Fall geht das Verfahren einen Schritt zurück („back-track“) und versucht erneut, mit einer der Alternativen zur Lösung zu kommen. Sind alle Alternativen erschöpft, so geht das Verfahren einen weiteren Entscheidungsknoten zurück und so fort, bis es im schlimmsten Fall alle denkbaren Alternativen abgesucht hat, also den gesamten Entscheidungsbaum untersucht hat. Ist beim *Backtracking* eine gute Entscheidungsmethode implementiert, um ertraglose Äste des Entscheidungsbaums nicht weiter zu verfolgen, so muss das Verfahren nicht alles ausprobieren („nicht den gesamten Lösungsraum untersuchen“), sondern kann schneller zum Ziel kommen.

Ein möglicher Python-Code ist rekursiv, etwa die *in-order*-Traversierung eines Baumes:

```
def track (Baum):                    # Lösungssuche mit back-tracking

    if Baum.Links != None :          # Existiert ein linker Unterbaum?
      if LsgOK(Baum.Links):          # und die Lösung ist ausreichend?
        track(Baum.Links)            # dann: in-order linker Unterbaum

    if Test(Baum.Wurzel) :           # Lösung gefunden?
       print(„Lösung gefunden!", Baum.Wurzel)

    if Baum.Rechts != None :         # Existiert ein rechter Unterbaum?
      if LsgOK(Baum.Rechts):         # und die Lösung ist ausreichend?
        track(Baum.Rechts)           # dann: in-order rechter Unterbaum
```

wobei die Funktionen `Test()` die Lösung vollständig testet und `LsgOK()` nur bestimmt, ob der Zweig abgebrochen werden soll oder nicht.

Im Unterschied zum *Greedy*-Verfahren untersucht also das Backtracking-Verfahren potentiell alle denkbaren Lösungen und ist damit nicht unbedingt schneller, aber genauer.

Beispiel *Sudoku*

Betrachten wir als Beispiel für diesen Algorithmus die Sudoku-Rätsel, siehe Abb. 15.5.

4	3				5	9	2	8
	2	9					7	5
			1					
9	4	6	5		7	8	3	
1								
7			9	8	2		6	1
					4		9	6
		5	7	3				
	6							

Abb. 15.5 *Ein 9×9 Sudoku-Rätsel*

Hier besteht die Aufgabe darin, die leeren Felder des Quadrats mit den Zahlen von 1 bis 9 zu füllen. Dabei gibt es aber folgende Nebenbedingungen:

- Jedes quadratische Unterfeld der Größe 3×3 enthält alle Zahlen von 1 bis 9. Hier dürfen also keine zwei Zahlen gleich sein.
- In jeder Spalte dürfen keine zwei gleichen Zahlen vorkommen.
- In jeder Zeile dürfen keine zwei gleichen Zahlen vorkommen.

Die Backtracking-Lösung ist einfach: beginnend mit dem ersten freien Feld wird einfach angenommen, dass hier eine 1 eingesetzt werden muss. Führt dies zum Widerspruch, so wird eine 2 genommen und so fort. Ist keine der drei Nebenbedingungen verletzt, so wird beim nächsten freien Feld ebenfalls eine 1 angenommen. Ist dadurch eine Nebenbedingung verletzt, so wird stattdessen eine 2 angesetzt und so fort. Führt dies irgendwann zu einem Widerspruch, so dass in einem freien Feld überhaupt keine Zahl eingesetzt werden kann ohne eine der Nebenbedingungen zu verletzten, so muss eine Annahme zurückgegangen werden und eine Alternative gewählt werden. Man beachte, dass bei der Änderung eines übergeordneten Knotens

alle Entscheidungen darunter neu überprüft werden müssen, d.h. mit der Wahl eine Alternative begibt man sich auf einen alternativen Zweig des Entscheidungsbaums. In ist der Entscheidungsbaum für zwei freie Felder gezeigt und die Abfolge, zur Kombination `9 , 1` für die freien Felder zu kommen.

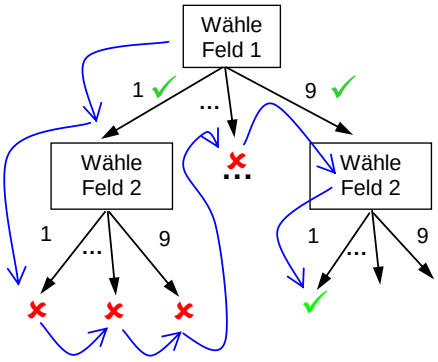

Abb. 15.6 *Ein Entscheidungsbaum für zwei freie Felder*

Der rekursive Code für Sudoku ist relativ einfach und entspricht dem Traversieren eines Baumes mit Abkürzungen, wenn eine Nebenbedingung nicht erfüllt ist; die Laufzeit ist allerdings ziemlich lang. Im schlimmsten Fall muss der gesamte Lösungsraum durchprobiert werden. Beim obigen Sudoku mit N = 49 unbekannten Zahlen, jede im Wert von 1 bis 9, sind somit potentiell 9^{49} Kombinationen abzuprüfen. Trotzdem kann im Normalfall mit einem PC ein 9×9 Sudoku innerhalb einer Sekunde gelöst werden. Erhöhen wir allerdings die Feldgröße und damit die Anzahl der freien Stellen, so geraten wir durch den exponentiellen Zuwachs des Lösungsraums schnell in Laufzeitprobleme. Hier zeigt sich, dass das Backpropagation-Verfahren am besten dann benutzt wird, wenn geeignete Bewertungen (Nebenbedingungen, Qualitätsmerkmale) vorliegen, um viele Äste des Entscheidungsbaums nicht evaluieren zu müssen.

Ein anderes Beispiel für die Anwendung von Backtracking ist das N-Damen-Problem aus Abschnitt 1.1. Wir lösen es, indem wir zunächst den ganzen Lösungsraum in Form eines Baumes konstruieren. Da wir bei *N* Spalten *N* Parameter haben, können wir den Lösungsraum schrittweise konstruieren. Bei jedem Schritt wird immer nur ein Parameter variiert, wobei jeder Parameter einen von *N* Werten annehmen kann. In Abb. 15.7 ist der erste Schritt zu sehen: Parameter 1 (Spalte 1) kann einen von *N* Werten annehmen.

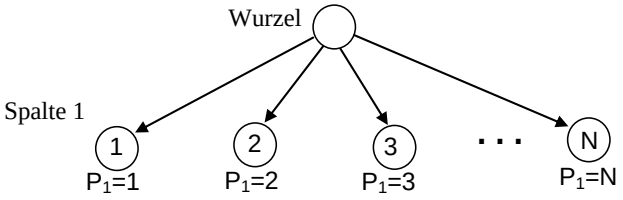

Abb. 15.7 Der Lösungsraum als Entscheidungsbaum für das N-Damen-Problem, 1.Schritt

Variieren wir den zweiten Parameter P_2, so wirkt jeder der *N* Unterknoten wieder als Wurzel mit jeweils weiteren *N* Unterknoten, so dass sich auf der Ebene P_2 insgesamt N·N Knoten ergeben. Führen wir das für *N* Parameter durch, so entsteht insgesamt aus einem Ausgangsknoten (Wurzel) nach dem N-ten Schritt ein Baum der Tiefe *N* mit N^N Blättern und damit N^N Lösungen.

Beispiel *Das N-Damen-Problem*

Bei der Lösung des N-Damen-Problems mittels *Backtracking* setzten wir zuerst eine Lösung an für eine Reihe mit *N* Feldern, etwa die erste Dame auf Feld 1. Dann konstruieren wir die zweite Reihe und probieren alle Spalten aus. Hier bemerken wir, dass der gesamte denkbare Lösungsraum nicht möglich ist: auf Spalte 1 gibt es Konflikt und auch auf Spalte 2. Erst in Spalte 3 gibt es keinen mehr und wir setzen dort die zweite Dame hin, siehe Abb. 15.8.

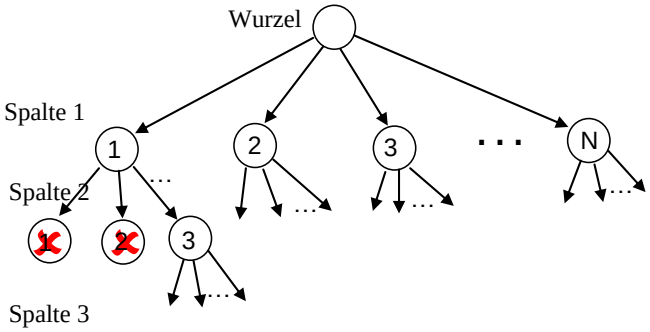

Abb. 15.8 Ein Entscheidungsbaum für das N-Damen-Problem mit ungültigen Lösungen

Nun setzen wir die 3-te Dame in Reihe 3 und gehen mit ihr alle unbesetzte Spalten durch, bis wir entweder eine Lösung finden, oder bei der letzten Spalte ankommen und keine gefunden haben. In letzterem Fall müssen wir nun ein *Backtracking* machen: wir gehen zur letzten Lösung *i*–1 zurück und versuchen an dieser Stelle des Entscheidungsbaums, eine andere, ebenfalls gültige Lösung zu finden. Haben wir sie, so geht es wieder weiter zur *i*-ten Dame und so fort. Haben wir sie nicht, so müssen wir weiter zurück zur *i–2*-ten Lösung und dort eine bessere suchen, darauf dann die Lösung für die *i–1*-te Dame, die *i*-te Dame, und sofort, suchen.

Das *Backtracking* ermöglicht uns, den gesamten Lösungsraum von N Damen auf N×N Feldern durch die Beachtung geeigneter Nebenbedingungen einzuschränken. Betrachten wir dies quantitativ am Entscheidungsbaum in Abb. 15.8, so haben wir in der ersten Zeile für die Dame gerade *N* Möglichkeiten. In der zweiten Reihe haben wir N-1 Möglichkeiten, da ja die Spalte mit der ersten Dame nicht besetzt werden darf. Zusätzlich verbieten sich alle Plätze, die unterhalb schräg rechts oder links von der Dame gelegen sind. In der dritten Reihe sind es nun N-2 Plätze abzüglich der Plätze, die auf einer Diagonalen mit einer der vorigen Damen liegen. Insgesamt haben wir also im Entscheidungsbaum statt N über N^2 weniger als $N(N-1)(N-2)\cdot\ldots\cdot 1 = N!$ Möglichkeiten, was deutlich geringer ist.

Zusammenfassend können wir sagen, dass die *Backtracking*-Methode einen guten Ansatz darstellt, alle Möglichkeiten („den ganzen Suchraum“) durchzuprobieren. Allerdings ist dies nicht unproblematisch:

- Die gesamte Abarbeitung von *N* Parametern mit jeweils *m* möglichen Werten bedeutet, insgesamt m^N Zustände zu überprüfen. Mit wachsendem *N* ist dies ein exponentielles Anwachsen der Zahl möglicher Zustände, also ein exponentielles Ausdehnen des Suchraums („kombinatorische Explosion“). Bei vielen Parametern bedeutet dies lange Suchzeiten.
- Besitzen wir noch zusätzliche Informationen, etwas Einschränkungen oder andere Nebenbedingungen für die Lösung, so sollten wir sie nutzen und einen anderen Algorithmus für die Suche wählen, um die Suchzeiten zu verringern.

2 Entwurf paralleler Programme

Die bisher vorgestellten Methoden, ein Problem mit Hilfe eines Algorithmus systematisch zu formulieren gingen immer von einem zentralen Prozessor aus, den man ein einzelnes Programm abarbeiten lässt. Diese Annahme ist auch in den meisten Fällen sinnvoll, aber durch die Möglichkeiten, die moderne Computersysteme bieten, kommen immer mehr auch andere Rechenmodelle ins Blickfeld des Programmierers. Betrachten wir nur den Leistungszuwachs, den graphische Prozessoren (GPU) im Unterschied zu „normalen" Zentralprozessoren (CPU) in den letzten Jahren erfahren haben. In Abb. 15.9 ist eine solche Übersicht gezeigt.

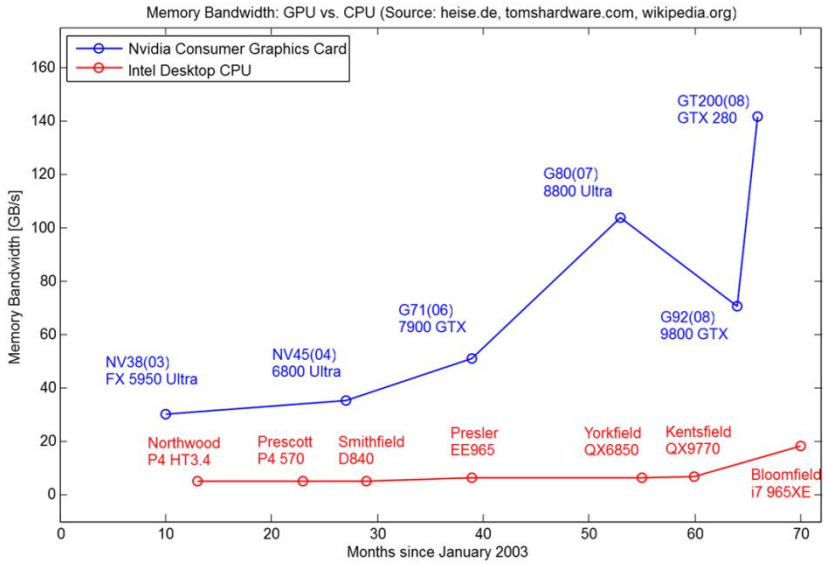

Abb. 15.9 *Leistungszuwachs von CPUs und GPUs*

Wir sehen, dass die CPU als rechnende Einheit selbst mit dem eingebauten Fließkommaprozessor gegen eine hochparallele, schnelle Grafikeinheit sehr schlecht aussieht. Berücksichtigen wir nun noch die Tatsache, dass die Grafikprozessoren inzwischen universell programmierbar sind, so stellt sich die Frage: Wie können wir die Rechenkraft einer solchen Spezialmaschine für schnelle Rechenoperationen gut ausnutzen? Wie beschleunigen wir unser Programm mit einer handelsüblichen Grafikkarte; wie kann man überhaupt ein Programm mit Hilfe eines Parallelrechners mit sehr vielen Prozessoren effizient beschleunigen?

Diese Frage kann man so in der Allgemeinheit nicht beantworten: jede Beschleunigung hängt sowohl von dem Algorithmus als auch von der verwendeten Architektur ab. In den folgenden Abschnitten sollen deshalb beide Aspekte anhand von Beispielen dargestellt werden. Beginnen wir mit dem Schwersten: der Parallelisierung von Algorithmen. Dies ist nicht immer automatisch möglich; die besten und effizientesten parallelen Formulierungen werden durch Menschen verfasst.

2.1 Beispiel Quicksort

Als Beispiel für die Parallelisierung eines sequentiellen Programms betrachten wir den Quicksort-Algorithmus wie wir ihn in Abschnitt 1.2 kennen gelernt haben. Dieser Algorithmus kann relativ leicht parallelisiert werden: Nachdem aus der ursprünglichen Liste zwei Unterlisten erstellt wurden, können diese parallel abgearbeitet werden, da sie unabhängig voneinander sind. Für die Unterlisten dieser Unterlisten gilt das ebenfalls, so dass im zweiten Schritt aus zwei parallelen Arbeiten vier parallele Arbeiten möglich sind und

so weiter. Für den Quicksort-Algorithmus können wir uns den Entscheidungsprozess mit Hilfe eines Binärbaumes visualisieren: Jede Verzweigung bedeutet eine Teilung der Liste (*devide*-Operation), ausgehend von einem *Pivot*-Element als Knoten. Bezeichnen wir jeden Knoten mit dem *Pivot*-Element der Teillistenbearbeitung, die es repräsentiert, so verzweigt sich der Baum solange, bis die jede Teilliste vollständig abgearbeitet (geordnet) ist und die Rekursion wieder zurückkehrt. Traversieren wir den Binärbaum *inorder* von links nach rechts, so erhalten wir die geordnete Liste. Wir erhalten also für die Arbeitsverteilung eine Baumstruktur, die sich immer mehr verzweigt. Im Idealfall der Lastverteilung erhält der Algorithmus bei N Daten und N Prozessoren anstelle von $N \cdot log N$ eine Laufzeit von nur $log\ N$ Zeiteinheiten. Dies soll nun in einer parallelen Formulierung nachvollzogen werden.
Die konkrete Implementierung ist je nach zugrunde liegender Hardware sehr unterschiedlich. Als Beispiel wählen wir uns als Maschine, auf der der Algorithmus ausgeführt werden soll (Zielarchitektur), einen Rechner, der parallel sowohl lesen als auch schreiben kann: die CRCW PRAM (Concurrent-Read, Concurrent-Write Parallel Random-Access Machine). Eine solche Maschine ist nur eine Gedankenkonstruktion; echte Maschinen können dies nicht vollständig implementieren. Trotzdem ist dieses Konzept sehr hilfreich, um die parallelen Möglichkeiten der Algorithmen auszuloten.
Angenommen, wir haben N Listenelemente und auch N Prozessoren. Jedem Prozessor mit dem Prozessor-ID pID = i sei ein Wert A[i] zugewiesen sowie zwei Bereiche, `leftChild`[i] und `rightChild`[i].
Für die Sortierung wählen wir uns zuallererst ein *Pivot*-Element. Dies wird mit folgendem symbolischen Code getan:

```
global root
def choose_Pivot(A,1,N):
   for pID in range(1,N):     # for each processor
          root = pID
          parent[pID] = root
          leftChild[pID] = rightChild[pID] = N+1
```

Dabei ist die for-Schleife **parallel** zu verstehen: alle Prozessoren tun das Gleiche und versuchen gleichzeitig, der Variablen `root` ihren Index zuzuweisen. Da die CRCW PRAM keine Bevorzugung kennt, nimmt die Speicherzelle den Wert des Prozessors an, der in der Warteschlange als Letzter auf die Speicherzelle zugreift.
Nach dieser zufälligen Initialisierung des *Pivot*-Elements beginnt die parallele Sortierung. Dazu vergleichen alle parallel ihren Wert mit dem *Pivot*-Element. Haben sie einen Wert kleiner als A[`parent`[pID]], so schreiben sie ihn in den `leftChild`-Bereich des `parent`[pID], andernfalls in den `rightChild`-Bereich.

```
def build_tree(A):
   while pID != root :
          if A[pID]< A[parent[pID]] or
                 (A[pID] == A[parent[pID]] and pID < parent[pID]):
                 leftChild[parent[pID]] = pID
                 if pID == leftChild[parent[pID]]: break
                 else: parent[pID] = leftChild[parent[pID]]
          else
                 rightChild[parent[pID]] = pID
                 if pID == rightChild[parent[pID]]: break
                 else: parent[pID] = rightChild[parent[pID]]
```

Da auch hier wieder nur ein Prozessor erfolgreich ist, werden beide Kind-Knoten jeweils nur einem Prozessor und damit jeweils einem *Pivot*-Element für den weiteren Teilbaum zugewiesen. Dies gilt zunächst nur für den Wert `root` von `parent`[i]. Ist dies erreicht und als Kindsknoten zwei neue Elternknoten bestimmt, so versuchen alle übrigen Prozessoren nun, sich auch hier bei den neuen Eltern als Kinder einzutragen. Auch hier sind wie-

der pro Elternknoten nur zwei Prozessoren erfolgreich. Dies wird solange weitergeführt, bis alle Prozessoren den `exit`-Befehl durchlaufen haben und damit als Knoten im Binärbaum vorhanden sind. Zwar sind die neuen Pivot-Elemente der Teilbäume wieder nur zufällig bestimmt, aber die Bewerbungen der Prozessoren als neue Knoten sind auf der richtigen Seite rechts oder links vom Elternknoten, so dass insgesamt der korrekte Binärbaum entsteht. Die Laufzeit zum Baumaufbau ist dabei diejenige, um ein Blatt zu erreichen: $O(\log N)$. Am Schluss muss man nur einmal den Baum traversieren, um die korrekte Reihenfolge aufzulisten.

Beispiel

A = (87, 65, 1, 78, 39, 114, 5)
pID = (1, 2, 3, 4, 5, 6, 7)

Betrachten wir als Beispiel die zu ordnende Liste aus Abschnitt 1.1. Jedem der N=7 Prozessoren ist die darüber stehende Zahl der Liste zugeordnet. Alle Prozessoren führen den gleichen Code aus, wobei als Ergebnis einer Zeile der Wert einer Variablen vom letzten Prozessor bestimmt wird, der darauf schreibt. Nehmen wir nun an, dass bei der Initialisierung Prozessor 5 gewonnen hat, also root = 5 gilt und alle Prozessoren als `parent` zunächst `root` notiert haben. Nun bestimmen alle Prozessoren außer 5, ob ihr Wert A[pID] größer oder kleiner als der Wert A[root] = 39 ist. Für Prozessoren 3 und 7 ist er kleiner; für Prozessoren 1,2,4 und 6 ist er größer. Den if-Teil der Abfrage und damit die Zuordnung des linken Kindes absolvieren also Prozessoren 3 und 7, von denen wir annehmen, dass 3 gewinnt und mit `break` die Endlosschleife abbricht. Prozessore 7 gewinnt nicht und notiert nur, dass sein Elternteil Prozessor 3 ist.

Im `else`-Teil versuchen Prozessoren 1,2,4,6 das rechte Kind zu werden. Nehmen wir an, dass 4 gewinnt und mit `break` abbricht, so notieren sich die anderen drei, dass Prozessor 4 der Elternteil ist. In Abb. 15.10 ist der Zustand des Baumes nach der ersten Runde zu sehen. Wurzel und 1. Ebene liegen schon fest; für die zweite Ebene ist noch nicht klar, wer linkes und rechtes Kind ist.

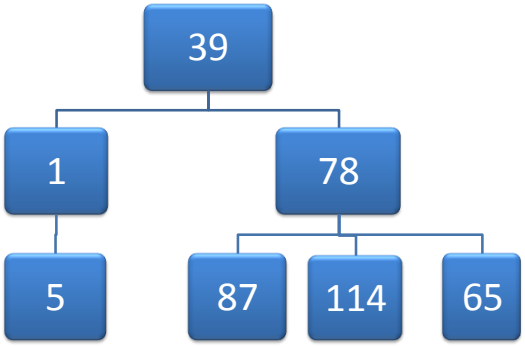

Abb. 15.10 *Zwischenzustand des Baumes*

Nun geht es in die zweite Runde. Alle Prozessoren vergleichen ihr Listenelement mit dem des Elternteils. Prozessor 2 bemerkt, dass seine Zahl 65 kleiner als die Zahl 78 seines Elternteils 4 ist, also bewirbt er sich als linkes Kind von 4. Da sonst kein anderer Prozessor dabei ist, gewinnt er und wird linkes Kind. Die anderen Prozessoren haben nur größere Zahlen als ihre Eltern, so dass sie Kandidaten für rechte Kinder werden. Prozessor 7 hat mit seiner Zahl 5 keine Konkurrenz bei seinem Elternteil, wird sofort rechtes Kind und bricht mit `break` ab. Die Prozessoren 1 und 6 dagegen bewerben sich beide bei dem Elternteil 4 mit der Zahl 78. Nehmen wir an, dass Prozessor 6 gewinnt, so notiert Prozessor 1 den Prozessor 6 als Elternteil und muss in die dritte Runde, in der er sich schließlich als linkes Kind anheftet. Nun sind alle Prozes-

soren zum Stillstand gekommen; der resultierende Baum der Tiefe 3 ist in Abb. 15.11 zu sehen.

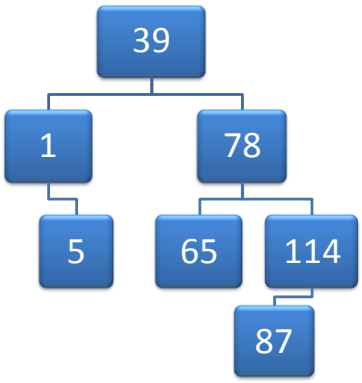

Abb. 15.11 *Endzustand des Baumes*

Traversiert man den fertigen Baum, so ist die Liste tatsächlich geordnet, wie verlangt.

2.2 Instruktionsfluss und Datenfluss

Im vorigen Abschnitt haben wir eine parallele Formulierung des Quicksort-Algorithmus kennen gelernt ohne zu sagen, wie man auf diese Formulierung gekommen ist. Wie kann man nun grundsätzlich ein paralleles Programm konstruieren? Welche alternativen Formulierungen gibt es? Dazu muss man grundsätzlich alle Operationen, die parallel zueinander möglich sind, auch parallel zur Ausführung vorsehen. Die Schwierigkeit besteht nun darin, zum einen den Algorithmus möglichst parallel zu ersinnen, zum anderen, die vorhandenen Parallelismen zu erkennen und explizit für den Compiler zu formulieren. Betrachten wir die zweite Möglichkeit etwas näher.

Der Ablauf von Programmen wird in einer Programmiersprache durch die Ablaufprinzipien *Kontrollfluss* und *Datenfluss* bestimmt. Beide Prinzipien bestimmen für jede Operation in einem Algorithmus den Zeitpunkt, zu dem die Operation ausgeführt werden darf. Während das Kontrollflussprinzip die Bearbeitungsreihenfolge der einzelnen Operationen streng (imperativisch) vorgibt, verlangt das Datenflussprinzip lediglich, dass bei der Abarbeitung der Operationen die Datenabhängigkeiten zwischen den einzelnen Operationen nicht verletzt werden, also ein Ergebnis erst errechnet und dann weiter verwendet wird und nicht umgekehrt. Das Datenflussprinzip eröffnet dadurch Möglichkeiten, die Effizienz von Programmen z.B. durch Permutation von unabhängigen Datenoperationen oder durch Parallelarbeit zu erhöhen. Betrachten wir diesen Unterschied genauer.

Das Kontrollflussprinzip

Das **Kontrollflussprinzip** basiert auf dem Grundsatz, dass eine Operation genau dann zur Ausführung kommt, wenn die vorhergehende beendet wurde. Der Kontrollfluss eines Programms bestimmt die Bearbeitungsreihenfolge der Instruktionen. Dieses Prinzip wird lediglich durch die sogenannten *Kontrollkonstrukte* unterbrochen. Statt die Ausführung bei der nächsten Operation im Speicher fortzusetzen, wird durch eine Kontrolloperation (Maschinenbefehl „Jump“) eine beliebige andere Operation als Fortsetzung des Programms ermittelt. Das Kontrollflussprinzip wird in imperativen Programmiersprachen durch die Kontrollkonstrukte implementiert. Moderne imperative Programmiersprachen stellen die folgenden Kontrollkonstrukte zur Verfügung:

- Schleife mit Vorbedingung : `while` (Bedingung) `do` {Anweisungen}
- Schleife mit Nachbedingung : `do` {Anweisungen} `while` (Bedingung)

- Unbedingter Sprung : `goto` (Label)
- Verzweigung : `if`(Bedingung) {Anweisungen} `else` {Anweisungen}
- Mehrfachverzweigung : `switch` bzw. `case`
- Sequentialisierungsoperator : **;**

Die Semantik des Sequentialisierungsoperators „**;**“ (Semikolon) wird bei der Beschreibung von Programmiersprachen in der Literatur häufig vernachlässigt. Von Programmierern wird er häufig als eine Art Zeilenende interpretiert. Seine Bedeutung als Trennungssymbol streng sequentiell auszuführender Operationen wird üblicherweise nicht vermittelt. Der Sequentialisierungsoperator hat die in frühen Programmiersprachen üblichen Zeilennummern ersetzt.

Das Kontrollflussprinzip wird auf der Hardwareebene durch die Architekturen implementiert, die einen Befehlszähler (*program counter* PC) haben.

Das Datenflussprinzip

Im Unterschied dazu kann man auch ein Programm ganz anders, nämlich als Abarbeitungsreihenfolge von Daten und nicht von Befehlen, ansehen. Der Datenfluss in Algorithmen wird durch die **Datenabhängigkeiten** zwischen den Instruktionen bestimmt. Das Datenflussprinzip basiert auf den Datenabhängigkeiten der Operationen in Algorithmen. Was ist eine „Datenabhängigkeit“? Man sagt, dass zwischen zwei beliebigen Operationen genau dann eine Datenabhängigkeit besteht, wenn eine Operation das Ergebnis der anderen weiterverarbeitet. Die Operationen sind dann datenabhängig. Besteht diese Art der Datenabhängigkeit nicht, dann sind die Operationen datenunabhängig

Beispiel Reihenfolge von Operationen

Seien drei Operationen gegeben:

```
1: C=B+A;
2: D=E*F;
3: G=C/D;
```

Die ersten beiden Operationen im Beispiel sind datenunabhängig, denn keine der beiden Operationen benötigt das Ergebnis der anderen. Die Operation 3 hängt von 1 und 2 ab, denn sie verwendet deren Ergebnisse. Instruktion 3 muss also in jedem Fall nach 1 und 2 ausgeführt werden. Die Operationen 1 und 2 können beliebig vertauscht werden oder sogar parallel berechnet werden, aber Operation 3 muss immer danach erfolgen. Die Möglichkeit der automatischen Ermittlung parallel ausführbarer Operationen motiviert die wissenschaftliche Untersuchung des Datenflussprinzips.

Aus den Datenabhängigkeiten der Operationen lässt sich das nun folgendes Abarbeitungsprinzip ableiten, das als **Datenflussprinzip** bezeichnet wird:

> *Eine Operation gilt als ausführbar, sobald alle ihre Operanden verfügbar sind* (nicht erst dann, wenn der Befehlszähler auf sie zeigt).

Ein Berechnungszyklus nach dem Datenflussprinzip besteht dann aus:

- der Berechnung aller ausführbarer Operationen
- der Ermittlung aller Operationen, die durch die neuen Ergebnisse ausführbar werden.

Datenflussprogramme laufen üblicherweise so ab, dass eine Teilmenge aller Befehle als *ausführbar* markiert ist. Diese Befehle werden nach der Produktion der Ergebnisse als *ausgeführt* gekennzeichnet und in die weitere Bearbeitung nicht mit einbezogen. Durch die neuen Ergebnisse können neue Operationen als „ausführbar“ ermittelt werden. Wichtig ist dabei, dass die Reihenfolge der Abarbeitung bei den ausführbaren Operationen keine

Rolle mehr spielt. Sie dürfen in einer beliebigen Reihenfolge oder sogar parallel berechnet werden.

Das Datenflussprinzip findet seine Anwendung in den Datenflusssprachen und in den Datenflussrechnern, die im folgenden Abschnitt behandelt werden. Eine weitere wichtige Anwendung dieses Prinzip stellt die Optimierung der Codegeneratoren von Compilern dar. Gerade bei der RISC-Architektur moderner Mikroprozessoren hat sich gezeigt, das unterschiedliche Permutationen einer Befehlssequenz unterschiedlich effizient abgearbeitet werden. Dabei wird Permutierbarkeit datenunabhängiger Operationen ausgenutzt, um die günstigste Anordnung zu finden.

Datenflusssprachen ermöglichen idealerweise die Codierung von Algorithmen unter Berücksichtigung des Datenflussprinzips. Diese akademische Anforderung an Datenflusssprachen wurde in den tatsächlich implementierten Sprachen jedoch nur teilweise verwirklicht. Sie sind eher durch die architekturellen Merkmale der Rechner geprägt, für die sie geschrieben wurden. Das heißt, sie enthalten Konstrukte, die spezielle Eigenschaften des Rechners unterstützen. Diese Tatsache hat zusammen mit der eher prozedural strukturierten Denkweise des Menschen dazu geführt, dass das Datenflussprinzip und die Datenflusssprachen als allgemeines Programmierparadigma kaum Anwendung gefunden haben.

2.3 Parallele Konstrukte

Die Formulierung des Sachverhalts „dieses Stück Programm kann von mehreren Prozessoren parallel abgearbeitet werden“ benötigt zum einen eine syntaktische Form, zum anderen eine Laufzeitumgebung, die diesen Wunsch mittels Kommunikation mit den beteiligten Prozessoren in die Tat umsetzt. Betrachten wir zuerst die syntaktische Seite.

Kontrollflussprogrammierung

Die einfachste Art und Weise, einen parallelen Sachverhalt zu formulieren, besteht in einer Aufzählung aller parallel abzuarbeitenden Codestücke sowie die Kennzeichnung der Aufzählung durch Schlüsselworte. Beispielsweise bedeutet in parallelem Pascal das Konstrukt

```
PARBEGIN
    procedure1(a,b);
    d := procedure2();
    procedure3(a,b);
PAREND
```

dass alle drei Prozeduren unabhängig voneinander parallel ausgeführt werden können, aber nicht müssen. Die konkrete Zuordnung von parallelem Code zu einzelnen Prozessoren wird vom Compiler initiiert und vom Linker oder von der Laufzeitumgebung ausgeführt. Die Zuordnung kann auch erst lastabhängig zur Laufzeit erfolgen, etwa durch spezielle Instanzen. Die Ausführung muss nur folgende Spielregeln befolgen:

- Alle als selbständig gekennzeichneten Codestücke (hier: die Prozeduren, durch Semikolon getrennt) können parallel oder auch sequentiell in beliebiger Reihenfolge ausgeführt werden. Um die Reihenfolge besser strukturieren zu können, gibt es auch Varianten, bei denen innerhalb des PARBEGIN/PAREND-Blocks auch zwischen parallel ausführbaren Instruktionen nochmals das Schlüsselwort PARALLEL steht. Steht es nicht da, ist die sequentiell aufgeführte Reihenfolge auch sequentiell gemeint.
- Nach der Ausführung werden die Ergebnisse zurückgeführt und im Hauptprogramm gesammelt. Erst wenn das letzte parallele Codestück zu Ende gegangen ist, wird das

Hauptprogramm fortgesetzt. Das PAREND-Statement bedeutet also eine Barriere, die erst freigegeben wird, wenn aller Code davor ausgeführt wurde.

- Der Kontrollfluss kann sich also bei der Parallelausführung verzweigen und muss sich dann wieder vereinen. Die Verzweigung und Wiedervereinigung wird durch entsprechende Schlüsselworte angegeben.
- Ein anderes Beispiel für parallel ausführbare Konstrukte bietet die Programmiersprache Occam, die 1985 für eine Familie von kommunizierenden Prozessoren (Transputern) geschaffen wurde und auf dem Konzept der *communicating sequential processes* CSP des Informatikers Hoare beruht. Hier bedeutet

  Par

  Befehl 1

  Befehl 2

  Befehl 3

  eine mögliche parallele Ausführung der drei Befehle. Der Par-Block wird durch das Schlüsselwort „Seq" wieder aufgehoben. Par- und Seq-Blöcke können geschachtelt werden.

Datenflussprogrammierung

Die Basis der Datenflusssprachen bilden die Datenflussgraphen, die zur Darstellung von Algorithmen zusammen mit ihren Datenabhängigkeiten verwendet werden. Compiler für Datenflusssprachen übersetzen die Programme in Datenflussgraphen, die dann direkt auf Datenflussrechnern abgebildet werden können. Die Datenflussgraphen stellen in diesem Sinne eine Art *Assembler* der Datenflussrechner dar. In Abb. 15.12 ist ein Datenflussgraph zur Berechnung des Polynoms ax^3+bx^2+cx+d dargestellt. Aus dem Datenflussgraphen sind die Datenabhängigkeiten direkt abzulesen. Operationen, die nach diesem Schema in keiner Datenabhängigkeit zueinander stehen, sind parallel berechenbar.

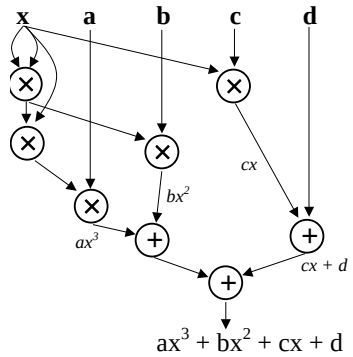

Abb. 15.12 *Datenflussgraph zur Berechnung eines Polynoms*

Die wesentlichen Merkmale der Datenflusssprachen sind:

- D*atengetriebene Auswertungsreihenfolge*

  Wenige Kontrollstrukturen: Durch das datengetriebene Ausführungsprinzip erübrigen sich in vielen Fällen explizite Kontrollkonstrukte. Im Besonderen kann auf einen Sequentialisierungsoperator (;) und auf Parallelisierungsoperatoren verzichtet werden.
- *Single Assignment Principle*

  Dieses Prinzip besagt, dass einer Variablen nur zu Beginn ihrer Gültigkeit ein einziges Mal ein Wert zugewiesen werden darf. Der Grund für dieses Prinzip wird in den Beispielen zur Datenflusssprache *Id* erläutert.

- *Kein Zugriff auf externe Variablen*

  Durch den Ausschluss externer Variablenzugriffe wird die Seiteneffektfreiheit der Datenflusssprachen erreicht.

Die folgenden Sprachen stellen die wichtigsten und bekanntesten Datenflusssprachen dar:

Id (Irvine dataflow)

Val (Value oriented algorithmic language)

Sisal (Streams and Iterations in a Single Assignment Language)

Anhand der Sprache *Id* werden im folgenden einige Merkmale der Datenflusssprachen erörtert. Die wichtigste Struktur von *Id* ist der Block. Ein Block ähnelt den Blöcken der prozeduralen Programmiersprachen mit dem wesentlichen Unterschied, dass die Reihenfolge, in der die Operationen berechnet werden, nicht durch die Auflistung im Programmtext gegeben ist, sondern erst bei der Ausführung des Programms durch den Datenflussrechner festgelegt wird.
Ein Block in *Id* wird wie folgt notiert:

```
{
        Anweisung;
        Anweisung;
        Anweisung;
        finally x;
}
```

Das Semikolon „**;**“ ist zwar zulässig, hat aber in *Id* keine Bedeutung als Sequentialisierungsoperator. Die Ausführungsreihenfolge der Anweisungen wird durch den Datenflussrechner anhand der Datenabhängigkeiten bestimmt. Der Block stellt einen Teilgraphen des gesamten Datenflussgraphen dar. Die Anweisung *finally* dient der Bestimmung der Variablen, die an nachfolgende (abhängige) Blöcke weitergegeben wird. Durch die automatische Ermittlung der Ausführungsreihenfolge nach dem Datenflussprinzip wird an dieser Stelle auch die Notwendigkeit des *Single Assignment Principle* innerhalb eines Blocks deutlich.

Beispiel Ein Block enthalte die Anweisungen

$$a = b + c \text{ und } a = x^2$$

Beide Operationen seien datenunabhängig. Das Ergebnis beider Operationen, das jeweils in die Variable *a* eingetragen wird, ist dann ohne weitere Annahmen nicht eindeutig bestimmt, da keine Ausführungsreihenfolge vorgegeben ist. Die parallele Ausführung in einem geeigneten Rechner würde sogar zu einem Konflikt führen, da einer Variablen gleichzeitig unterschiedliche Werte zugewiesen werden können.

Neben dem unbedingt auszuführenden Block enthält *Id* Schleifenblöcke und bedingt auszuführende Blöcke. Sie werden durch die Schlüsselwörter *if*, *for* oder *while* gekennzeichnet:

```
{
    x=1
    y=1
    In
    {  for j<-1 to n do
                next x = y
                next y = x + y
                finally x
    }
}
```

Neben den Datenflusssprachen besteht die Möglichkeit der Anwendung der weitverbreiteten prozeduralen Sprachen wie C/C++ oder Pascal zur Programmierung von Daten-

flussrechnern. Dabei wurde prinzipiell davon ausgegangen, dass die Datenabhängigkeiten, die zur Erzeugung eines Datenflussgraphen notwendig sind, auch aus einer prozeduralen Beschreibung extrahiert werden können. Probleme bereitet dabei, dass weder von der Einhaltung des *Single Assignment* Prinzips, noch von der Seiteneffektfreiheit solcher Programme ausgegangen werden darf. Die Konflikte, die dadurch entstehen, können aber durch den Compiler erkannt und aufgelöst werden.

Dieser Ansatz hat trotz des Mehraufwandes in der Konfliktauflösung verschiedene Vorteile:

- Akzeptanz

  Durch die weite Verbreitung der Sprache im Bereich der Anwendungs- und Systemprogrammierung ist die Akzeptanz von Programmentwicklern gegeben.

- Vorhandene getestete Anwendungen

  Es existieren bereits zahllose Applikationen, die ohne die Notwendigkeit der Neuprogrammierung auf dem Zielrechner (in diesem Fall ein Datenflussrechner) zur Ausführung gebracht werden können.

Beispiel

Das folgende C-Programm

```
{
  input  double x;
  output double cosx;

   /* cosinus approximation */
   cosx = 1
          - x*x / 2
          + (x*x)*(x*x) / 24
          - (x*x)*(x*x)*(x*x) / 720
}
```

wird in den folgenden Datenflussgraphen übersetzt

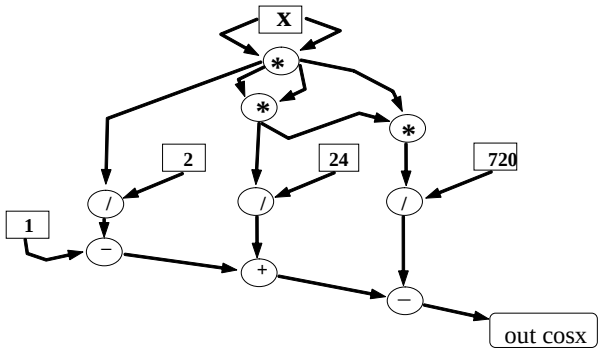

Abb. 2.13 *Datenflussgraph eines C-Programms*

Dieses Beispiel zeigt Möglichkeiten, durch geschickte Verwendung bereits durchgeführter Operationen (der Quadrierung von x) die Zahl der Rechenschritte zu reduzieren.

Allerdings gibt es dabei ein wichtiges Problem: Der mit Datenflussarchitekturen nutzbare Parallelitätsgrad eines Programms hängt entscheidend von der implizit oder explizit parallel formulierten Algorithmus ab. Es ist deshalb wichtig, dass bereits bei der Spezifikation des Anwenderproblems auf unnötige Abhängigkeiten verzichtet wird und der Algorithmus dazu möglichst parallel formuliert wird. Die automatische Umformulierung von

der sequentiellen zu einer parallelen Version des Algorithmus kann dagegen heutzutage auch der beste Compiler noch nicht leisten.

Die Verwendung einer graphischen Datenflusssprache kann diese Aufgabe entscheidend erleichtern.

2.4 Programm- und Daten-Granularität vs. Kommunikation in Rechnerarchitekturen

Im vorigen Abschnitt wurden die Grundstrukturen für eine parallele Programmierung auf der Kontrollfluss- oder Datenflussebene gezeigt. Die sprachlichen Möglichkeiten leiden darunter, dass dabei nicht verbindlich festgelegt werden kann, ob ein Code zur Laufzeit tatsächlich parallel oder nur sequentiell abgearbeitet werden kann. Wonach richtet sich das? Wie sollte die Laufzeitumgebung entscheiden? Obwohl ein Programm wunderschön parallel formuliert ist, kann es trotzdem sein, dass es mehr Zeit zur Ausführung benötigt als seine sequentielle Form. Warum? Der Grund liegt in der allgemeinen, rechnerunabhängigen Formulierung der Parallelität, wie sie im vorigen Abschnitt eingeführt wurde. Die parallele Formulierung allein beschreibt nur die logische Aufteilung der Arbeit, sie sagt aber Nichts darüber aus, wie lange es dauert, den Code oder die dafür benötigten Daten über das Kommunikationsnetz zu dem Prozessor zu senden, wo der Code ausgeführt und die Daten benötigt werden, und die Ergebnisse wieder zurück zu senden. Diese Zeiten können stark variieren: Sie hängen nicht nur von der Ausführungsdauer des Codes ab, sondern auch von der Programmlänge, Datenlänge, Schnelligkeit der Kommunikationsleitung und allgemein vom Kontext, also von der verwendeten Rechnerarchitektur. Es gibt sehr spezielle Multiprozessor-Rechnerarchitekturen, etwa welche bei denen die Kommunikationsstruktur einen Hyperwürfel bildet, oder eine Ringstruktur, oder ein Gitterstruktur. Üblich sind allerdings eher konventionelle Strukturen weniger Rechner im Verbund, etwa Cluster oder anderer stark oder schwach gekoppelter Rechner und Prozessoren.

Unabhängig von dem benutzten Prozessortyp, Bustyp oder Speicherplattenfabrikat muss man einige grundsätzliche Konfigurationen unterscheiden. Im einfachsten, klassischen Fall gibt es nur einen Prozessor, der Haupt- und Massenspeicher benutzt, um das Betriebssystem (BS) und die Programme der Benutzer auszuführen, siehe Abb. 15.14. Die Ein- und Ausgabe (Bildschirm, Tastatur, Maus) ist dabei nicht gezeigt.

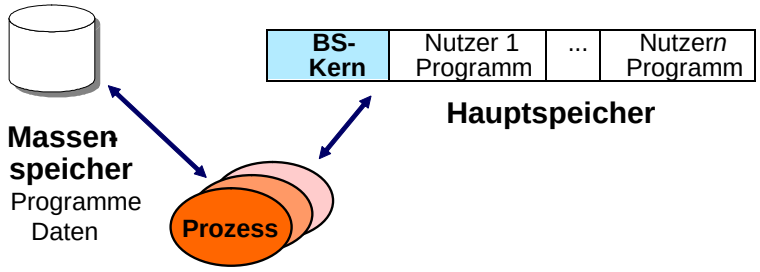

Abb. 15.14 *Ein Einprozessor- oder Mehrkernsystem*

Ein solches **Einprozessorsystem** (*single processor*) kann man mit mehreren Prozessoren (meist bis zu 16 Stück) aufrüsten. Je nachdem, wie man die Prozessoren miteinander koppelt, ergeben sich unterschiedliche Architekturen.

Im einfachsten Fall kann man nur multiple Prozessoren an die Stelle des einzelnen Prozessor setzen, etwa durch Anreicherung mit mehreren Kernen wie sie bereits in einfachen PCs anzutreffen sind. Hierbei ändert sich aber nichts weiter an der Hauptarchitektur, so dass die multiplen Kerne sich die Datenwege zum Hauptspeicher und Ein-/Ausgabe zum Massenspeicher teilen müssen. Dies begrenzt die Leistung des Gesamtsystems stark.

Die nächste Stufe sieht Replikationen der CPU vor, die parallel an einem besonderen Verbindungsnetzwerk (z. B. an einem Multi-Master-Systembus) hängen. In Abb. 15.15 ist ein solches **Multiprozessorsystem** abgebildet.

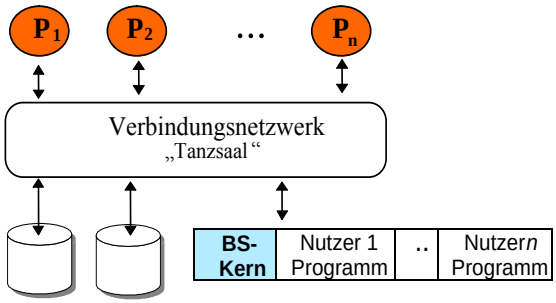

Abb. 15.15 *Multiprozessorsystem*

Die Prozessormodule sind auf der einen Seite des Netzwerks und die Speichermodule auf der anderen lokalisiert. Für jeden Daten- und Programmcodezugriff wird zwischen ihnen eine Verbindung hergestellt, die während der Anforderungszeit bestehen bleibt. Diese Architektur wird deshalb auch „Tanzsaal“-Konzept genannt.

Eine derartige Architektur ermöglicht zwar einen parallelen Zugriff der Prozessoren auf den Hauptspeicher, führt aber auch leicht zu Leistungs- (*Performance*)-Einbußen, da einzelne Netzwerkknoten für Speicherzugriffe bestimmter, oft benutzter Teile des Betriebssystems stark belastet werden („hot spots“) und sich dort Warteschlangen ausbilden. Abhilfe kommt in diesem Fall aus der Beobachtung, dass die Prozessoren meist nur einen eng begrenzten Programmteil referenzieren. Der Speicher kann deshalb aufgeteilt und der relevante Teil „dichter“ an den jeweiligen Prozessor herangebracht werden (Abb. 15.16). Die feste Aufteilung muss natürlich auch durch den Compiler unterstützt werden, der das Anwenderprogramm auf die Rechner entsprechend aufteilt. Ein solches **Mehrrechnersystem** ist auch als „Vorzimmer“-Architektur bekannt.

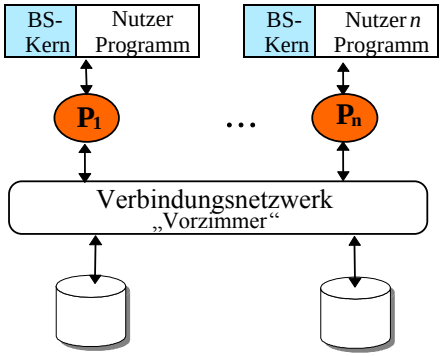

Abb. 15.16 *Mehrrechnersystem*

Als dritte Möglichkeit der Anordnung der Verbindungen existiert das **Rechnernetz**. Hier sind vollkommen unabhängige Rechner mit jeweils eigenem (nicht notwendig gleichem!) Betriebssystem lose miteinander über ein Netzwerk gekoppelt, s. Abb. 15.17. Ist das Netzwerk sehr schnell und sind die Rechner räumlich dicht beieinander, so spricht man auch von einem *Cluster*.

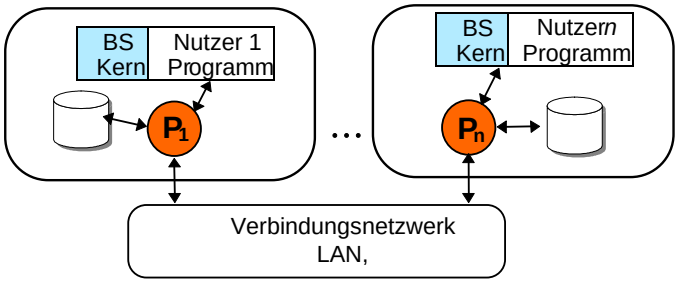

Abb. 15.17 *Rechnernetz*

Ist eine Software auf einem Rechner installiert, so dass der Rechner als Dienstleister (*Server*) Aufträge für einen anderen Rechner (Kunden, *Client*) ausführen kann, so spricht man von einer **Client-Server**-Architektur. Beispiele für Dienstleistungen sind numerische Rechnungen mit Supercomputern (*number cruncher*), Ausdruck von Dateien (*print server*) oder das Bereithalten von Dateien (*file server*).

Jede der vorgestellten Rechnerarchitekturen besitzt ihre Vor- und Nachteile. Für uns ist dabei wichtig: Jede benötigt spezielle Mechanismen, um bei einer Parallelprogrammierung eine Interprozessorkommunikation und Zugriffssynchronisation bei den Betriebsmitteln zu erreichen. Bei jeder Verteilung parallel ausführbaren Codes muss die Frage beantwortet werden: Ist die sequentielle Ausführungszeit zweier Programmteile größer als die Zeit, den Code sowie die zugehörigen Daten zu verschicken und die Ergebnisse zurück zu holen? Es ist klar, dass sich dies bei großen Programmstücken (etwa Module, Methoden ganze oder Programme) lohnt, bei kleinen Programmstücken (etwa Einzelinstruktionen) dies aber zu lange dauert. Entscheidend ist also die Codestückgröße (Programmgranularität) relativ zum Aufwand der Transmission. Betrachten wir dazu als Beispiel zwei Modelle von Parallelarbeit: das Pipeline-Modell mit Aufteilung des Programms und das Parallel-Modell mit Aufteilung der Daten. In Abb. 15.18 ist dies visualisiert.

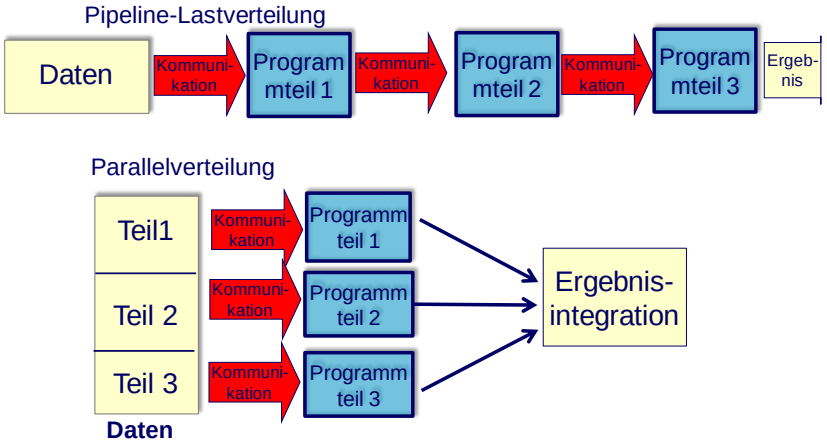

Abb. 15.18 *Parallelverarbeitung durch Programmaufteilung und Datenaufteilung*

Hierbei müssen wir aber auch nach der Aufteilung selbst unterscheiden: verschicken wir Codestücke, Datenstücke (bei einmal initialisiertem Code auf anderem Rechner) oder Beides? Verwendet das Zielsystem eigene Kommunikationsprozessoren, oder muss der Hauptprozessor alles durchführen? Ist der Code und die Daten im selben, globalen Hauptspeicher (s. Abb. 15.14 und Abb. 15.15), oder ist er in einem lokalen Speicher, der von außen zugängig ist (Abb. 15.16), oder muss zum Datenaustausch ein spezieller Kommunikationsprozessor involviert werden (Abb. 15.17)? Die Konsequenzen aus dem

Spannungsfeld zwischen Kommunikationsleistung und Rechnerleistung ist in Abb. 15.19 gezeigt.

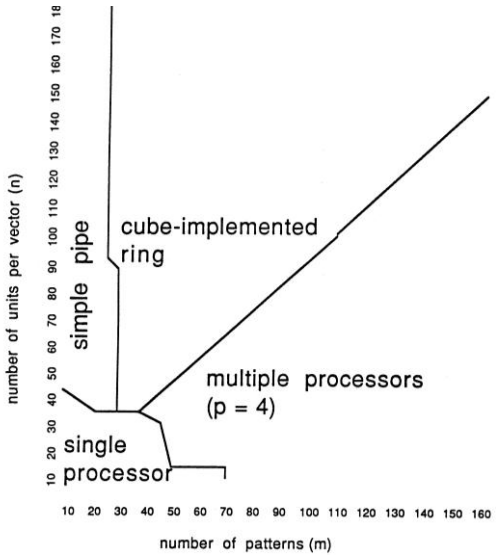

Abb. 15.19 *Optimale Aufteilung von Daten und Programmteilen*

Hier ist für eine rekonfigurierbare Intel Hypercube-Architektur je nach Kontext aus der Anzahl *m* der Datentupel (Mustervektoren) und der Dimension *n* jedes Tupels für einen bestimmten Algorithmus (Backpropagation) eine unterschiedliche Architektur am besten geeignet und liefert am schnellsten die Resultate. Typischerweise schafft bei geringer parallel bearbeitbarer Datenlänge *n* und wenigen Daten *m* ein einzelner Prozessor schneller die Arbeit, als wenn man die wenigen Daten auf andere Prozessoren verteilt. Gibt es wenige, aber lange Daten, so ist eine sequentielle Verarbeitung in Form einer Pipeline am besten, bei der das Programm in seine sequentiellen Teile aufgebrochen und auf die Prozessoren verteilt wird. Hat man aber sehr viele Daten, so ist es besser, stattdessen die Datenmenge aufzuteilen und jedem Prozessor sowohl das Programm als auch seinen zu bearbeitenden Teil der Daten zu schicken und am Ende die Resultate wieder einzusammeln und abzugleichen.

Ein optimaler, paralleler Code, der auf allen Rechnerarchitekturen eingesetzt werden kann, gibt es also nicht. Stattdessen beschränkt man sich in der Praxis darauf, für ein Programm eine Parallelisierung des Kontrollflusses auf eigenständige, parallele Aktivitäten grober Granularität vorzunehmen und sie zur Laufzeit an verschiedene Prozessoren zu verteilen. Aktive Einheiten sind dabei sog. „Prozesse“ und „Threads“, die auf globalem Speicher arbeiten. Wir werden sie in den nächsten Vorlesungen genauer untersuchen.

Die Verteilung und Koordinierung der Prozesse und Threads kann man in Hilfsbibliotheken zusammenfassen. Beispiel für ein solches Softwarepaket ist die Parallel Virtual Machine PVM[ii], die es ermöglicht, mehrere vernetzte Rechner als einen großen, virtuellen Rechner zu verwenden.

2.5 Grenzen der Parallelisierung

Angenommen, wir haben einen Super-Parallelcomputer mit fast unendlich vielen Prozessoren. Kann man damit jedes Programm beliebig schnell ausführen? Was sind die Grenzen dafür?

Die maximal erreichbare Beschleunigung (*speedup* = das Verhältnis „AlteZeit“ zu „NeueZeit“) der Programmausführung durch zusätzliche parallele Prozessoren, I/O-Kanäle usw. ist sehr begrenzt. Maximal können wir die Last auf *m* Betriebsmittel (z. B. Prozessoren) verteilen, also NeueZeit = AlteZeit/*m*, was einen linearen *speedup* von *m* bewirkt. In der Praxis ist aber eine andere Angabe noch viel entscheidender: der Anteil von sequentiellem, nicht-parallelisierbarem Code am Gesamtcodeumfang. Seien die sequentiell ausgeführten Zeiten von sequentiellem Code mit T_{seq} und von parallelisierbarem Code mit T_{par} notiert, so ist mit m→∞ Prozessoren mit $T_{par} \to 0$ der *speedup* bei sehr starker paralleler Ausführung

$$speedup = \frac{\text{AlteZeit}}{\text{NeueZeit}} \to \frac{T_{seq} + T_{par}}{T_{seq} + 0} = 1 + \frac{T_{par}}{T_{seq}}$$

fast ausschließlich vom Verhältnis des parallelisierbaren zum sequentiellen Code bestimmt.

Beispiel

Angenommen, wir haben ein Programm, das zu 90% der Laufzeit parallelisierbar ist und nur für 10% Laufzeit sequentiellen Code enthält. Auch mit der schnellsten Parallelhardware können wir nur den Code für 90% der Laufzeit beschleunigen; die restlichen 10% müssen hintereinander ausgeführt werden und erlauben damit nur einen maximalen *speedup* von (90%+10%)/10% =10.

Da die meisten Programme weniger rechenintensiv sind und einen hohen I/O-Anteil (20-80%) besitzen, bedeutet dies, dass auch die Software des Betriebssystems in das parallele Scheduling einbezogen werden sollte, um die tatsächlichen Ausführungszeiten deutlich zu verringern. In Abb.12.19 sind die zwei Möglichkeiten dafür gezeigt.

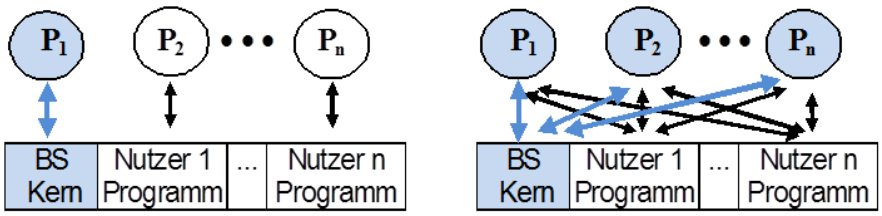

Abb. 15.20 *Asymmetrisches und symmetrisches Multiprocessing*

Links ist die Situation aus Abb. 12.13 vereinfacht dargestellt: Jeder Prozessor bearbeitet streng getrennt einen der parallel existierenden Prozesse, wobei das Betriebssystem als sequentieller Code nur einem Prozessor zugeordnet wird. Diese Konfiguration wird als **asymmetrisches Multiprocessing** bezeichnet.

Im Gegensatz dazu kann man das Betriebssystem – wie auch die Anwenderprogramme – in parallel ausführbare Codestücke aufteilen, die von jedem Prozessor ausgeführt werden können. So können auch die Betriebssystemteile parallel ausgeführt werden, was den Durchsatz deutlich erhöht. Diese Konfiguration bezeichnet man als **symmetrisches Multiprocessing** SMP.

Zusammenfassend kann man also feststellen, dass die Ausführungszeit von parallel geschriebenen Code nicht nur an der Granularität der Arbeitseinheiten (Ausführungsdauer der Programmstücke) und der Verteilung auf die Prozessoren liegt, sondern auch an

dem noch verbleibenden sequentiellen Anteil der Anwendung sowie an dem Parallelisierungsgrad des benutzten Betriebssystems.

---

[i] V.Kumar et al.: Introduction to Parallel Computing, Benjamin/Cummings Publ., Redwood City, CA , USA 1994

[ii] http://www.csm.ornl.gov/pvm/pvm_home.html