Parallelisierung aller APL-Operationen

Wolfgang J. Paul und Jürgen Sauermann

Die Sprache APL basiert auf einem äußerst eleganten Kalkül für Vektoren, Matrizen und Tensoren. APL-Operationen kommen inzwischen in vielen modernen Programmiersprachen vor. Ihre Parallelisierung ist daher nicht nur für Benutzer von APL von Interesse. Wir zeigen, daß sich alle APL-Operationen effizient parallelisieren lassen. Die wesentlichen Werkzeuge beim Beweis sind

1. *Zahlensysteme mit gemischter Basis für die Berechnung von Speicheradressen von Tensorelementen und*
2. *ein auf* Smoothing *basierender Routing-Algorithmus.*

11.1 Ziele

Die Sprache APL [7, 9, 22] basiert auf einem äußerst eleganten Kalkül für Vektoren, Matrizen und Tensoren. APL-Operationen kommen inzwischen in vielen modernen Programmiersprachen wie FORTRAN 90 vor [17]. Die Parallelisierung dieser Operationen ist daher für die Praxis von Interesse und zwar nicht nur für Benutzer von APL.

Wir zeigen in dieser Arbeit, daß sich *alle* APL-Operationen parallelisieren lassen. Dabei werden wir uns natürlich auf die umfangreiche theoretische Literatur über parallele Algorithmen stützen [6, 10, 14]. Um Algorithmen aus der theoretischen Literatur praktikabel zu machen, muß man typischerweise an drei Stellen noch Arbeit investieren:

1. Die Anzahl p der Prozessoren ist fest und wächst nicht mit der Größe n der Eingabe. Wir behandeln nur den Fall $p < n$.
2. Die Größe n der Eingabe ist keine Zweierpotenz, und man bläst die Eingabe möglichst auch nicht künstlich auf.
3. Wenn man mehrere Algorithmen in ein System (in unserem Fall einen parallelen APL-Interpreter [21, 24]) integrieren will, braucht man ein einheitliches Format für die Ein- und Ausgabe.

In Abschnitt 11.2 spezifizieren wir die Anforderungen an die Zielmaschine, auf der die parallelisierten APL-Operationen ausgeführt werden sollen. In den Abschnitten 11.3 und 11.4 geben wir einen kurzen Überblick über APL-Werte, und wir beschreiben, wie Daten auf der Zielmaschine abgelegt und wiedergefunden werden. Als zentrales Hilfsmittel benutzen wir hier eine Zahlendarstellung mit gemischter Basis [24]. Wir setzen nicht voraus, daß der Leser mit APL-Jargon vertraut ist.

In den Abschnitten 11.5 bis 11.8 beschreiben und analysieren wir parallele Algorithmen für *alle* APL-Operationen außer der Inversion von Matrizen

und der Lösung linearer Gleichungssysteme[1]. Die Abschnitte enthalten Algorithmen für Routing, Reshape, Parallel Prefix und Sortieren, gefolgt von Anwendungen für die Realisierung von APL-Operationen. Solange nicht sortiert wird, erreichen wir in der Regel Laufzeit $O(n/p)$.

Lokale Laufzeiten schätzen wir asymptotisch ab. Die Kommunikationszeiten berechnen wir bis auf Terme niederer Ordnung exakt. Dies gestattet in Abschnitt 11.9 die Klassifikation der entwickelten Algorithmen nach ihrer Kommunikationszeit. Überdies kann man die exakten Analysen für die Abschätzung und Vorhersage von Laufzeiten auf realen Maschinen benutzen. Darauf werden wir aus Platzgründen hier nicht näher eingehen.

Große Teile der Abschnitte 11.3, 11.5.4 bis 11.5.6 und 11.6 stammen aus [24]. In den Abschnitten 11.7.1, 11.7.2 und 11.8 brauchen wir nur bekannte Standardalgorithmen in naheliegender Weise anzupassen.

11.2 Zielmaschine

Wir definieren zunächst eine abstrakte Zielmaschine. Damit legen wir einfach diejenigen Mechanismen fest, die zur Ausführung unserer Algorithmen notwendig sind. Jede reale Maschine, die die abstrakte Maschine effizient simuliert, kann auch unsere Algorithmen effizient ausführen.

11.2.1 Abstrakte Zielmaschine. Die abstrakte Zielmaschine hat einen *Master-Prozessor* M und p *Slave-Prozessoren* P_i, $i \in \{0, \ldots, p-1\}$. Wir setzen voraus, daß p eine Zweierpotenz ist. Alle Berechnungen verlaufen in Runden. Es gibt zwei Sorten von Runden. In *compute*-Runden rechnen alle Prozessoren lokal. In *communicate*-Runden tauschen sie über insgesamt drei Netzwerke Daten aus.

Die Slaves sind untereinander durch ein sogenanntes *Shifting-Network* [26] verbunden. In solchen Netzen kann man für jede Kommunikationsrunde eine feste *Shift-Distanz* d einstellen. Ist die Distanz d eingestellt, so kann für alle i gleichzeitig Prozessor P_i an Prozessor $P_{i+d \bmod p}$ Daten senden. Dabei kann er n Daten in $\sigma + \rho n$ Schritten übertragen[2]. Hierbei ist σ die *Startup-Zeit* für die Kommunikation, und ρ ist die *inverse Bandbreite*, die jedem Prozessor zur Verfügung steht. Beispiele für Shifting-Networks sind Crossbars, Permutationsnetze [28] und Barrel-Shifter (siehe z.B. [18]).

Der Master ist mit den Slaves über zwei Kontrollbusse verbunden. Über Kontrollbus 1 kann der Master an die Slaves broadcasten. Auf Kontrollbus 2 liest der Master das bitweise logische ODER aller Signale, die von den Slaves auf diesen Bus gelegt werden. Bus 2 kann offensichtlich zur *Barrier-Synchronisation* [5] benutzt werden. Diese Verbindung zwischen Master und

[1] Parallele Algorithmen für diese Probleme sind ausführlich untersucht worden [19].

[2] Wir gehen davon aus, daß Prozessoren gleichzeitig senden und empfangen können; andernfalls muß man in zwei Teilrunden kommunizieren. Für die Analyse verdoppeln sich dann σ und ρ.

Slaves wurde in [2] vorgeschlagen. Über eine Realisierung wurde in [21] berichtet. Spezielle Hardware zum Synchronisieren ist überraschenderweise erst 1992 in kommerziellen Maschinen eingesetzt worden, und zwar im Kontrollbus der CM-5 [15]. In unseren Algorithmen werden die Kontrollbusse fast nur zur Synchronisation zwischen den Runden benutzt. Die Zeit zum Synchronisieren kann man im Parameter σ berücksichtigen. Die Zeit für die übrige Kommunikation über die Kontrollbusse vernachlässigen wir.

Für die Analyse der lokalen Laufzeiten behandeln wir die Prozessoren als Random Access Maschinen (RAMs) mit uniformem Kostenmaß [1], d.h. Elementaroperationen auf elementaren Daten haben Kosten 1. Das RAM-Modell aus [1] erlaubt als elementare Daten die ganzen Zahlen. Die Datenworte realer Maschinen haben jedoch eine beschränkte Breite. Deshalb passen wir unsere Analyse in zweierlei Hinsicht an:

- Wir nehmen an, daß n elementare Daten, die auf einem Prozessor gespeichert sind, in Zeit $O(n)$ sortiert werden können. Aus praktischer Sicht ist dies erfüllt, wenn man Bucket-Sort benutzt [1].
- Wir kodieren nicht unbeschränkt Tupel von natürlichen Zahlen in einzelne natürliche Zahlen, denn andernfalls bräuchte jeder Prozessor in jeder Runde nur ein Datum zu schicken. Bei Adreßrechnungen werden wir sogar massiv Tupel von natürlichen Zahlen durch einzelne natürliche Zahlen kodieren. Dort ist aber die Größe des Ergebnisses in natürlicher Weise (im Wesentlichen durch die Problemgröße n) beschränkt.

Die so definierte abstrakte Maschine ist beinahe ein BSP-Computer [16, 27], der nur eine eingeschränkte Menge von h-Relationen unterstützt.

11.2.2 Reale Zielmaschinen. Will man wissen, wie gut eine reale Maschine die Kommunikationsmechanismen der abstrakten Maschine simuliert, muß man nur die Größen σ und ρ bestimmen. Wir erinnern daran, daß wir in σ die Zeit zum Synchronisieren zwischen den Runden mit untergebracht haben. Zur Bestimmung der (ungünstigsten) inversen Bandbreite ρ braucht man nur $p-1$ Kommunikationsmuster zu betrachten. Man muß sich in diesem Zusammenhang vor der Annahme hüten, daß die pro Prozessor verfügbare Bandbreite bei modernen Maschinen vom globalen Kommunikationsmuster unabhängig ist. Beim Benchmarken einer Intel-Paragon [4] mit $p = 64$ von insgesamt 140 Prozessoren haben wir bei den $p-1$ möglichen Shifts Schwankungen von ρ um einen Faktor knapp unter 1.5 gemessen.

11.3 APL-Werte

Elementare Daten haben in APL den Typ integer, real, boolean oder character. Alle anderen APL-Werte sind Felder A, die aus elementaren Daten bestehen. Ihre Dimension ist höchstens 8 und wird in APL *rank* bzw. *Rang* genannt. Indizes beginnen bei 0 oder 1. Wir werden hier immer voraussetzen, daß sie bei 0 beginnen. Ein solches Feld wird durch ein 4-Tupel $A = (t, d, s, r)$

spezifiziert. Hierbei ist t der Typ der Komponenten des Feldes, d ist die Dimension von A, $s = (s_{d-1}, \ldots, s_0)$ ist der Vektor der oberen Grenzen der Indizes $+1$ und wird *Shape* von A genannt. Schließlich ist r die Folge der Feldelemente in lexikographischer Reihenfolge ihrer Indizes. Die Folge r wird *Ravel* von A genannt. Ist beispielsweise $d = 2$, so ist A eine Matrix, und der Ravel von A besteht aus den Matrixelementen in Zeilenordnung. Elementare Daten werden als APL-Werte mit Dimension 1 und Shape-Vektor (1) angesehen. Wir bezeichnen den Ravel von A mit $(,A)$ und das j-te Element des Ravels mit $(,A)[j]$.

Sei nun $A = (t, d, s, r)$ ein APL-Wert und es sei $n = \prod_{j=0}^{d-1} s_j$ die Anzahl seiner Elemente. Für (d-dimensionale) Indizes i von Feldelementen $A[i]$ bezeichne $s \perp i$ die Anzahl der Indizes, die lexikographisch kleiner als i sind. Nach Definition des Ravels gilt

$$A[i] = (,A)[s \perp i] \tag{11.1}$$

für alle Feldindizes i. Ein leichter Induktionsbeweis zeigt

$$s \perp i = \sum_{k=0}^{d-1} \left(i_k \prod_{x=0}^{k-1} s_x \right) .$$

Das ist aber nichts anderes als die natürliche Zahl, die im Zahlensystem mit der gemischten Basis s die Darstellung i hat. Mit einem solchen Zahlensystem messen wir alle unsere Zeit. Wenn wir wissen wollen, wieviele Sekunden des Tages um 3:35:16 Uhr mittags vorbei sind, so ist das $(2, 12, 60, 60) \perp (1, 3, 35, 16)$.

Für $k \in \{0, \ldots, d-1\}$ sei $I_k = \{0, \ldots, s_k - 1\}$ die Menge der möglichen Werte, welche die Komponente i_k eines Feldindex i annehmen kann. Sei $j \in \{0, \ldots, n-1\}$ und es sei $s \top j \in I_{d-1} \times \cdots \times I_0$ die Darstellung von j im Zahlensystem mit der gemischten Basis s. Dann gilt $s \top (s \perp i) = i$ für alle i und Gleichung (11.1) liefert

$$A[s \top j] = (,A)[j] \text{ für alle } j\,. \tag{11.2}$$

Mit dieser Gleichung kann man aus einer Ravelposition den zugehörigen Feldindex bestimmen.

Wir beenden diesen Abschnitt mit zwei Identitäten, die sich später als sehr nützlich erweisen werden. Ist $d = 6$ und $s_x = 10$ für alle x (d.h. wir rechnen mit gewöhnlichen Dezimalzahlen), dann sind

$$\begin{aligned} 123456 &= (10,10,10,10,10,10) \perp (1,2,3,4,5,6) \\ &= (100, 10, 1000) \perp (12, 3, 456) \text{ und} \\ 12345 &= (10,10,10,10,10) \perp (1,2,3,4,5) = (100, 1000) \perp (12, 345) \end{aligned}$$

Beispiele für diese Identitäten. Allgemein sei $j \in \{0, \ldots, d-1\}$, $i_{up} = (i_{d-1}, \ldots, i_{j+1})$, $i_{low} = (i_{j-1}, \ldots, i_0)$, $s_{up} = (s_{d-1}, \ldots, s_{j+1})$, $s_{low} = (s_{j-1}, \ldots,$

$s_0)$, $h = \prod_{x=j+1}^{d-1} s_x$, $m = s_j$ und $l = \prod_{x=0}^{j-1} s_x$. Dann verifiziert man mit einer leichten Rechnung

$$s \perp i \quad = \quad (h, m, l) \perp (s_{up} \perp i_{up}, i_j, s_{low} \perp i_{low}) \tag{11.3}$$

und

$$(s_{up}, s_{low}) \perp (i_{up}, i_{low}) \quad = \quad (h, l) \perp (s_{up} \perp i_{up}, s_{low} \perp i_{low}) . \tag{11.4}$$

11.4 Speicherung von APL-Werten

Die Arbeit zwischen Master- und Slave-Prozessoren wird ganz einfach aufgeteilt. Die Slaves speichern und manipulieren die Ravel-Elemente. Der Master koordiniert die Arbeit. Wir geben an, wie Ravels auf den Slaves abgelegt werden.

Eine Adresse im globalen Adreßraum aller Slaves ist ein Paar (a, j). Hierbei ist j die Nummer eines Slaves und a ist eine Adresse im lokalen Speicher dieses Prozessors. Wir betrachten also den gesamten Adreßraum als eine Matrix mit p Spalten, d.h. mit einer Spalte für jeden Prozessor. Wir ordnen die globalen Adressen lexikographisch, d.h. in Zeilenordnung.

Für jeden APL-Wert A speichern wir $(, A)[0]$ auf Prozessor 0, d.h. an irgendeiner globalen Adresse der Form $(b, 0)$. Für alle j speichern wir Ravel-Element $(, A)[j]$ am j-ten Nachfolger von Adresse $(b, 0)$ im globalen Adreßraum. Dies ist Adresse $(b + \lfloor j/p \rfloor, j \bmod p)$. Die lokale Adresse b heißt die *Basisadresse* von A, und die Größe $\lfloor j/p \rfloor$ heißt das *lokale Displacement* von Ravel-Element $(, A)[j]$. Für alle j ist also Ravel-Element $(, A)[j]$ auf Slave $P_{j \bmod p}$ mit lokalem Displacement $\lfloor j/p \rfloor$ gespeichert.

Ist n die Anzahl der Elemente von Ravel r, dann speichert Slave P_i genau

$$\nu(i) = \begin{cases} \lceil n/p \rceil & \text{falls} \quad i < n \bmod p \\ \lfloor n/p \rfloor & \text{sonst} \end{cases}$$

Ravel-Elemente.

Die eben definierte Art, Ravels oder Vektoren, die als Zwischenergebnisse von Rechnungen auftreten, abzuspeichern, nennen wir *Zeilenordnung.* Gelegentlich speichern wir Vektoren $x = (x_0, \ldots, x_{n-1})$, die als Zwischenergebnisse auftreten, auch in *Spaltenordnung*: Es sei $\nu(i)$ wie oben definiert und es sei $V(i) = \sum_{j \le i} \nu(j)$. In Spaltenordnung werden für alle i die Elemente $x_{V(i-1)}, \ldots, x_{V(i)-1}$ auf Slave i gespeichert.

Sei nun der Ravel des APL-Werts A mit Shape s in Zeilenordnung gespeichert. Für alle $k \in \{0, \ldots, p-1\}$ und $j \in \{0, \ldots, \nu(k) - 1\}$ sei $i(j, k)$ der Feldindex des Ravel-Elements von A, das auf Prozessor k mit Displacement j gespeichert ist. Dann gilt

Theorem 11.1. *Jeder Prozessor k kann* lokal *die Folge der Feldindizes $i(j, k)$ in Zeit $O(\lfloor n/p \rfloor)$ mit konstant vielen Divisionen berechnen.*

Beweis. Aus der Definition von $i(j,k)$ und Gleichung (11.2) folgt

$$A[i(j,k)] = (,A)[pj+k] = A[s\top(pj+k)]$$

für alle j und k. Dies gilt für alle APL-Werte, insbesondere für Werte, deren Elemente paarweise verschieden sind. Es folgt

$$i(j,k) = s\top(pj+k)$$

für alle j und k. Dies liefert den folgenden sehr einfachen Algorithmus: Berechne $i(0,k) = s\top k$ und $s\top p$ mit konstant vielen Divisionen. Berechne $i(j+1,k)$ durch Addition von $i(j,k)$ und $s\top p$ im Zahlensystem mit der gemischten Basis s. Hier addiert man einfach Stelle für Stelle nach der Schulmethode. □

Weil wir gewöhnlich Ravels in Zeilenordnung speichern, erfordern gewisse APL-Operationen überhaupt keine Kommunikation, insbesondere die komponentenweise Verknüpfung von Werten gleicher Dimension. Hierzu gehören die gewöhnlichen Vektoroperationen. Die lokale Laufzeit ist in diesem Fall $O(\lceil n/p \rceil)$.

11.5 Routing

Wir haben das folgende Routing-Problem zu lösen: Für $i,j \in \{0,\ldots,p-1\}$ will Prozessor i genau $n(i,j)$ Pakete an Prozessor j senden. Jedes Paket enthält die Zieladresse j und eine konstante Zahl L von elementaren Daten. Im folgenden sei $n(i) = \sum_j n(i,j)$ die Gesamtzahl von Paketen, die Prozessor i schickt. Dann ist $N = \max_i n(i)$ die größte Zahl von Paketen, die irgendein Prozessor sendet, $m(j) = \sum_i n(i,j)$ ist die Gesamtzahl von Paketen, die Prozessor j empfängt, $M = \max_j m(j)$ ist die größte Zahl von Paketen, die irgendein Prozessor empfängt und $n = \sum_i n(i)$ ist die Zahl aller Pakete. Offensichtlich ist $\sigma + \rho L \max\{N,M\}$ eine untere Schranke für die Laufzeit von Algorithmen für das obige Routing-Problem.

11.5.1 Triviales Routen. Der triviale Routing Algorithmus (Rt0) arbeitet in $p-1$ Runden. Jedes Paket wird direkt zu seinem Bestimmungsort geschickt. In Runde k wird Shift-Distanz k eingestellt. Die Kommunikationszeit für Runde k ist $\sigma + \rho L \max_i n(i, i+k \bmod p)$. Selbst wenn $N = M = n/p$ gilt, kann dieser Algorithmus Laufzeit $\Theta(nL)$ haben, beispielsweise falls gilt: $n(i,2i) = n(p/2+i, 2i+1) = n/p$ für alle $i < p/2$ und $n(i,j) = 0$ für alle übrigen i und j (die Prozessoren tauschen Daten im Muster eines Perfect Shuffle aus).

Wir benutzen Algorithmus (Rt0) immer dann, wenn wir einen Ravel von Zeilenordnung in Spaltenordnung oder umgekehrt umspeichern. Hat der Ravel Länge n, dann ist $n(i,j) \le \lceil \lceil n/p \rceil / p \rceil$ für alle i und j. Die Kommunikationszeit ist höchstens

$$\begin{aligned}(p-1)(\sigma+\lceil\lceil n/p\rceil/p\rceil\rho) &\leq p\sigma+(p-1)(\lceil n/p\rceil/p+1)\rho\\ &\leq p\sigma+(\lceil n/p\rceil+p-1)\rho\\ &\leq p\sigma+(n/p+p)\rho\,. \qquad (11.5)\end{aligned}$$

11.5.2 Smoothing. Der folgende Algorithmus (Rt1) hat Laufzeit $O(N+M+p^2)$. Er benutzt *Smoothing*, eine Technik, die auch für das Lösen von Routingproblemen auf Gittern benutzt wird [23]. Wir routen in zwei Phasen. In jeder Phase wird der triviale Algorithmus benutzt. In Phase 1 verteilt für alle i und j Prozessor i seine $n(i,j)$ Pakete mit Ziel j so gleichmäßig wie möglich auf alle Prozessoren. Jeder Prozessor erhält höchstens $\lceil n(i,j)/p\rceil$ dieser Pakete. In Phase 2 werden die Pakete an ihr Ziel geschickt.

Für $x\in\{1,2\}$ sei $n_x(i,s)$ die Zahl von Paketen, die Prozessor i in Phase x an Prozessor s schickt. Dann ist

$$n_1(i,s)\leq\sum_j\lceil n(i,j)/p\rceil<n(i)/p+p\leq N/p+p$$

und

$$n_2(i,s)\leq\sum_i\lceil n(i,s)/p\rceil<m(s)/p+p\leq M/p+p-1\,.$$

In Phase 1 enthält jedes Paket $L+1$ elementare Daten, nämlich die ursprünglichen Daten und die Zieladresse. Die Kommunikationszeit in Phase 1 ist also höchstens $\sigma p+(N+p^2)(L+1)\rho$. In Phase 2 muß die Zieladresse nicht mit übertragen werden, und die Kommunikationszeit ist $\sigma p+(M+p^2)L\rho$. Sei $RT(N,M,L)$ die größte Kommunikationszeit, die auftritt, wenn man Algorithmus (Rt1) mit Parametern N, M und p startet. Dann gilt

Theorem 11.2. $RT(N,M,L)\leq 2p\sigma+((L+1)N+LM+(2L+1)p^2)\rho$.

11.5.3 Transposition, Rotation und Spiegelung. Routing-Algorithmen liefern sofort Algorithmen für alle APL-Operationen, welche nur die Ravel-Elemente eines einzelnen APL-Werts permutieren. Insbesondere gilt

Theorem 11.3. *Transposition, Rotation und Spiegelung von APL-Werten mit n Elementen können in Zeit $O(\lceil n/p\rceil+p^2)$ berechnet werden. Die Kommunikationszeit ist höchstens $2p\sigma+(5n/p+5p^2+5)\rho$.*

Beweis. Es sei A der Operand und B das Ergebnis. Weiter sei $s=shape(A)$ und $t=shape(B)$. Jeder Prozessor k bestimmt nach Theorem 11.1 in $O(\lceil n/p\rceil)$ Schritten die Feldindizes i aller Elemente $A[i]$, die auf ihm gespeichert sind. Für jede der oben genannten Operationen gibt es eine einfache bijektive Funktion j, so daß das Element $A[i]$ an $B[j(i)]=(,B)[t\perp j(i)]$ zugewiesen werden muß. Dieser Wert muß auf Prozessor $(t\perp j(i)) \bmod p$ mit Displacement $\lfloor(t\perp j(i))/p\rfloor$ gespeichert werden. Deshalb routet jeder Prozessor für jeden seiner Feldindizes i das Paket $(A[i],\lfloor t\perp j(i)\rfloor)$ an Prozessor $(t\perp j(i)) \bmod p$. Der Rest ist trivial. Die Kommunikationszeit ist begrenzt durch

$$RT(\lceil n/p \rceil, \lceil n/p \rceil, 2) \leq 2p\sigma + (5\lceil n/p \rceil + 5p^2)\rho,$$

und der Satz folgt.

In Zukunft wenden wir die Ergebnisse der Abschnitte 11.3 und 11.4 an, ohne das explizit zu erwähnen.

11.5.4 Indizierung. Die Sprache APL hat sehr allgemeine Mechanismen, um die Elemente eines APL-Wertes B mit den Elementen eines zweiten APL-Wertes I zu indizieren. Im einfachsten Fall sind sowohl B als auch $I = (I[0], \ldots, I[n-1])$ Vektoren. In diesem Fall bezeichnet $B[I]$ den Vektor $(B[I[0]], \ldots, B[I[n-1]])$. Dieser Vektor kann als die rechte Seite einer Zuweisung $A \leftarrow B[I]$ verwendet werden. Dies nennt man eine *gather*-Operation oder *indexed reference*. Diese Zuweisung hat die gleiche Wirkung wie die sequentielle Folge der Zuweisungen $A[0] \leftarrow B[I[0]]; \ldots; A[n-1] \leftarrow B[I[n-1]]$.

Der Vektor $B[I]$ kann aber auch als linke Seite einer Zuweisung $B[I] \leftarrow A$ verwendet werden. Dies nennt man eine *scatter*-Operation oder *indexed assignment*. Diese Zuweisung hat die gleiche Wirkung wie die sequentielle Folge der Zuweisungen $B[I[0]] \leftarrow A[0]; \ldots; B[I[n-1]] \leftarrow A[n-1]$.

Für $j \in \{0, \ldots, p-1\}$ sei $Q(j) = \#\{I[k] : I[k] = j \bmod p\}$. Dies ist die Zahl von verschiedenen Elementen $B[I]$, die auf Prozessor P_j bearbeitet werden müssen. Weiter sei $Q = \max_j\{Q(j)\}$ und $N = \lceil n/p \rceil$. Offensichtlich ist $\rho \cdot \max\{N, Q\}$ eine untere Schranke für die Laufzeit jedes Algorithmus für indexed assignment oder indexed reference. Die Basisalgorithmen für diese beiden Operationen können jetzt leicht formuliert werden.

Für indexed reference kreiert jeder Prozessor für jedes seiner Elemente $I[k]$ ein Paket $(I[k], k)$. Diese Pakete werden zu Prozessor $I[k] \bmod p$ geroutet. Dort wird die erste Komponente durch $B[I[k]]$ ersetzt. Die Pakete $(B[I[k]], k)$ werden an Prozessor $k \bmod p$ zurückgeschickt. Danach können die Prozessoren den Ravel von $B[I]$ lokal erzeugen.

Die Laufzeit dieses Verfahrens wird durch den Routing-Algorithmus bestimmt. Während des Vorwärts-Routens ist $N = \lceil n/p \rceil$, es ist $m(j) = \#\{k : I[k] = j \bmod p\}$, und $M = \max_j\{m(j)\}$. Während des Zurückroutens sind die Rollen von N und M vertauscht. Die Pakete beim Routen haben Länge $L = 2$. Es folgt

Theorem 11.4. *Indexed reference kann in Zeit $O(N + M + p^2)$ ausgeführt werden. Die Kommunikationszeit ist begrenzt durch $4p\sigma + (5N + 5M + 10p^2)\rho$.*

Für indexed assignment kreiert jeder Prozessor j Pakete $(A[k], I[k], k)$. Ein solches Paket wird zu Prozessor $I[k] \bmod p$ geschickt. Jeder Prozessor sortiert die Pakete, die er empfangen hat, nach den dritten Komponenten. Mit Bucket-Sort geht das in Zeit $O(M)$. Die Zuweisungen $B[I[k]] \leftarrow A[k]$ werden in der Reihenfolge der sortierten Liste (also nach aufsteigendem k) lokal durchgeführt. Die Pakete haben beim Routen Länge $L = 3$, und es folgt

Theorem 11.5. *Indexed assignment kann in Zeit $O(N + M + p^2)$ durchgeführt werden. Die Kommunikationszeit ist beschränkt durch $2p\sigma + (4N + 3M + 7p^2)\rho$.*

11.5.5 Reduktion. Sei S eine Menge, $f : S^2 \to S$ eine Funktion und $v = (v_{n-1}, \ldots, v_0)$ ein Vektor aus S^n. Die *f-Reduktion* f/v ist wie folgt definiert: $u_0 = v_0, u_j = f(v_j, u_{j-1})$ für alle $j > 0$ und $f/v = u_{n-1}$.

In APL können Reduktionen auf viele Arten durchgeführt werden: auf Vektoren, auf den Zeilen oder Spalten einer Matrix oder allgemein entlang einer beliebigen Dimension eines APL-Wertes. Formal sei A ein APL-Wert wie in Abschnitt 11.3 definiert. Es sei $j \in \{0, \ldots, d-1\}$ eine Dimension. Schließlich sei $i = (i_{up}, i_{low})$ mit $i_{up} \in I_{d-1} \times \ldots \times I_{j+1}$ und $i_{low} \in I_{j-1} \times \ldots \times I_0$. Die *Achse* von A in Richtung j durch i ist der Vektor

$$A(j|i) = (A[i_{up}, 0, i_{low}], \ldots, A[i_{up}, s_j - 1, i_{low}]).$$

Die f-Reduktion $f/[j]A$ von A in Richtung j hat als Ergebnis den APL-Wert A' mit Shape $s' = (s_{up}, s_{low})$ und $A'[i] = f/A(j|i)$ für alle i.

Die parallele Berechnung von allgemeinen f-Reduktionen scheint auf den ersten Blick ein unübersichtliches und kniffliges Problem zu sein. Mit den vorliegenden Hilfsmitteln läßt es sich jedoch auf erstaunlich leichte Weise lösen. Zunächst führen wir das Problem, ein beliebiges Feld A in Richtung j zu reduzieren, auf das Problem zurück, ein 3-dimensionales Feld B in Richtung 1 zu reduzieren:

Im Ravel $(, A)$ sind die Elemente von Achse $A(j|i)$ an den Positionen $s\bot(i_{up}, y, i_{low})$ mit $y \in \{0, \ldots, s_j - 1\}$ gespeichert. Das Resultat der Reduktion der Achse wird im Ravel $(, A')$ des Ergebnisses an Position $(s_{up}, s_{low})\bot(i_{up}, i_{low})$ gespeichert. Seien h, m und l wie in Abschnitt 11.3 definiert, und es sei B das 3-dimensionale Feld mit Shape (h, m, l) und Ravel $(, A)$. In diesem Ravel sind die Elemente von Achse $B(1|(s_{up}\bot i_{up}, s_{low}\bot i_{low}))$ an den Positionen $(h, m, l)\bot(s_{up}\bot i_{up}, y, s_{low}\bot i_{low})$ mit $y \in \{0, \ldots, s_j - 1\}$ gespeichert.

Das Resultat der Reduktion dieser Achse muß im Ravel $(, B')$ des Ergebnisses an Position $(h, l)\bot(s_{up}\bot i_{up}, s_{low}\bot i_{low})$ gespeichert werden. Gleichungen (11.3) und (11.4) implizieren nun unmittelbar, daß in beiden Fällen auf den Ravels genau die gleichen Operationen durchgeführt werden.

Nun müssen wir nur noch die f-Reduktion eines 3-dimensionalen Felds in Richtung 1 berechnen. Dies gelingt mit dem Routing-Algorithmus (Rt1). Sei $n = hml$. Für jedes Element $B[i, j, k]$ wird das Paket $(B[i, j, k], i, k)$ an Prozessor $(h, l)\bot(i, k) \bmod p$ geschickt. Die Reduktion der Achsen geschieht danach lokal in Zeit $O(m\lceil hl/p \rceil)$. Im Routing-Algorithmus ist $L = 3, N = \lceil n/p \rceil$ und $M = m\lceil hl/p \rceil = m\lceil n/(mp) \rceil$. Es folgt

Theorem 11.6. *Reduktion von APL-Werten mit n Elementen entlang Achsen der Länge m kann in Zeit $O(\lceil n/p \rceil + m)$ durchgeführt werden. Die Kommunikationszeit ist beschränkt durch $2p\sigma + (7n/p + 3m + 7p^2 + 4)\rho$.*

Man kann dieses Verfahren verbessern, wenn man Assoziativität und Kommutativität von f ausnutzt. Diese Situation kann man durch Modifikation der Eingabe oft erreichen. Beispielsweise kann man eine $-$-Reduktion in eine $+$-Reduktion umwandeln, wenn man das Vorzeichen jeder zweiten Komponente des Inputs umdreht. Schwierigkeiten machen schließlich nur noch die Schaltfunktionen NAND und NOR, auf die wir kurz in Abschnitt 11.7.3 zurückkommen.

11.5.6 Join, take und drop. Seien A und B APL-Werte mit Dimension d und Shape $s = (s_{d-1}, \ldots, s_0)$ bzw. $t = (t_{d-1}, \ldots, t_0)$. Es sei $t_j = s_j$ für alle $j \neq i$. Dann ist der *join* $A,[i]B$ von A und B entlang Dimension i das Feld C mit Shape $(s_{d-1}, \ldots, s_{i+1}, s_i + t_i, s_{i-1}, \ldots, s_0)$, wobei für alle j gilt: $C(i|j)$ besteht aus $A(i|j)$ konkateniert mit $B(i|j)$. Zur Berechnung des joins müssen nur die beiden Ravels von A und B umgespeichert werden. Die Kommunikationszeit ist $RT(\lceil n/p \rceil + \lceil m/p \rceil, \lceil (n+m)/p \rceil, 2)$, und man erhält

Theorem 11.7. *Der join zweier APL-Werte mit n und m Elementen kann in Zeit $O((n+m)/p)$ berechnet werden. Die Kommunikationszeit ist beschränkt durch $2p\sigma + (5(n+m)/p + 5p^2 + 8)\rho$.*

Sei $B = (B[0], \ldots, B[n-1])$ ein Vektor, und es sei a eine ganze Zahl. Wir definieren den Vektor $a \uparrow B$ (*take a of B*). Gilt $0 < a \leq n$, so besteht $a \uparrow B$ aus den ersten a Elementen von B. Gilt $n < a$, so besteht $a \uparrow B$ aus B gefolgt von $n - a$ Elementen z; hierbei ist z das Leersymbol, falls B vom Typ char ist, andernfalls ist $z = 0$. Gilt $a < 0$ und $|a| \leq n$, so besteht $a \uparrow B$ aus den letzten a Elementen von B. Gilt $a < 0$ und $n < |a|$, so besteht $a \uparrow B$ aus $n - |a|$ Elementen z gefolgt von B. Für $a > 0$ muß man für die Berechnung von $a \uparrow B$ überhaupt nicht kommunizieren. Für $a < 0$ braucht man eine einzige Kommunikationsrunde mit Distanz $-(n - |a|) = -n - a$.

Die Operation $a \downarrow B$ (*drop a of B*) kann einfach als $(-(n-a)) \uparrow B$ definiert werden.

Die Operationen take und drop sind auch auf Feldern B mit Dimension $k > 1$ und Vektoren $a = (a_{k-1}, \ldots, a_0)$ definiert. Es sei $t = (t_{k-1}, \ldots, t_0) = \mathit{shape}(B)$. Wir behandeln nur die Operation take.

Die Dimension von $a \uparrow B$ ist k. Der Shape von $a \uparrow B$ ist $(|a_{k-1}|, \ldots, |a_0|)$.

$a \uparrow B[j] = z$, falls für irgendein i gilt: $t_i \leq j_i < a_i$ oder $a_i < 0$ und $j_i < |a_i| - t_i$. Andernfalls ist $a \uparrow B[j] = B[q]$, wobei für alle i gilt:

$$q_i = \begin{cases} j_i & \text{falls} \quad a_i > 0\,, \\ t_i - |a_i| + j_i & \text{sonst.} \end{cases}$$

Umgekehrt ist $B[q] = a \uparrow B[j(q)]$ für ein $j(q)$ genau dann, wenn für alle i gilt: $q_i < a_i$ oder $a_i < 0$ und $q_i \geq t_i - |a_i|$. In diesem Fall gilt

$$j(q)_i = \begin{cases} q_i & \text{falls} \quad a_i > 0\,, \\ |a_i| - t_i + q_i & \text{falls} \quad a_i < 0\,, \end{cases}$$

und Element $B[q]$ endet im Ravel des Ergebnisses an Position $j'(q) = (|a_{k-1}|, \ldots, |a_0|)\perp j(q)$. Also kann man $a \uparrow B$ lokal berechnen, nachdem man für alle q das Paar $(B[q], j'(q))$ an Prozessor $j'(q)$ mod p geroutet hat. Damit jeder Prozessor die Zahlen $j'(q)$ berechnen kann, sammelt der Master die k Elemente von a und broadcastet sie. Sei n die Anzahl der Elemente von B und $m = \prod_i |a_i|$. Dann ist die Kommunikationszeit durch $RT(\lceil n/p \rceil, \lceil m/p \rceil, 2)$ beschränkt, und wir erhalten

Theorem 11.8. *Ist n die Anzahl der Elemente von B und ist m die Anzahl der Elemente von $a \uparrow B$ bzw. $a \downarrow B$, dann kann man $a \uparrow B$ bzw. $a \downarrow B$ in Zeit $O((n+m)/p)$ berechnen. Die Kommunikationszeit ist beschränkt durch $2p\sigma + (3n/p + 2m/p + 5p^2 + 5)\rho$.*

11.6 Reshape

Es sei s ein Vektor und A ein APL-Wert. Dann ist $s\rho A$ (s *Reshape* A) ein APL-Wert B mit Shape s. Der Ravel von B besteht aus hinreichend vielen Kopien des Ravels von A. Die letzte Kopie wird an der passenden Stelle abgeschnitten.

11.6.1 Rekursives Verdoppeln. Wir nehmen an, daß ein Ravel r' der Länge m aus einem Ravel r der Länge n erzeugt werden soll. Gilt $n \geq m$, so muß nur eine Kopie von r erzeugt werden. Sei also $c = m/n > 1$. Dann wird r' aus r durch rekursives Verdoppeln von r in Runden $i, i \in \{1, \ldots, \lceil \log c \rceil\}$, erzeugt. Für jedes i haben wir nach Runde i genau 2^i Kopien von r, also ist in der i-ten Runde die Shiftdistanz gleich $n \cdot 2^{i-1}$ mod p und jeder Prozessor sendet höchstens $\lceil n2^{i-1} \rceil$ Daten. Spätestens in Runde $\log p + 1$ ist die Shiftdistanz gleich 0 und die Rechnung kann lokal zu Ende geführt werden. Sei nun $s = \lceil \log(\min\{p, c\}) \rceil$. Die Kommunikationszeit ist dann

$$\begin{aligned} s\sigma + \sum_{i=1}^{s} \lceil 2^{i-1} n/p \rceil \rho &\leq s\sigma + (s + (n/p)2^s)\rho \\ &\leq \sigma \log p + (\log p + (n/p)2\min\{p, c\})\rho \\ &\leq \sigma \log p + (\log p + 2(n/p)(m/n))\rho \\ &\leq \sigma \log p + (\log p + 2m/p)\rho\,. \end{aligned}$$

Zum lokalen Kopieren braucht man Zeit $O(\lceil m/p \rceil)$. Also gilt

Theorem 11.9. *Reshape eines Ravels der Länge n auf Länge $m > n$ kann in Zeit $O(\lceil m/p \rceil + \log p)$ durchgeführt werden. Die Kommunikationszeit ist beschränkt durch $\sigma \log p + (2m/p + \log p)\rho$.*

11.6.2 Äußeres Produkt. Es seien A bzw. A' APL-Werte mit gleichem Typ, Dimension d bzw. d', Shape s_{up} bzw. s_{low}. Weiter sei f eine Funktion mit zwei Argumenten. Das *äußere Produkt* $A \circ .fA'$ ist ein Feld mit Dimension $d + d'$, Shape (s_{up}, s_{low}) und Elementen $(A \circ .fA')[i, i'] = f(A[i], A'[i'])$.

Seien h und l definiert wie in Abschnitt 11.3. Es seien $B = (, A)$ und $B' = (, A')$ die Ravels von A und A'. Dann folgt aus (11.4), daß die Ravels von $A \circ .fA'$ and $B \circ .fB'$ identisch sind.

Sei C eine $h \times l$-Matrix, deren Spalten Kopien von B sind und sei C' eine $h \times l$-Matrix, deren Zeilen Kopien von B' sind. Dann kann $B \circ .fB'$ aus den Ravels von C und C' durch eine Vektoroperation in Zeit $O(\lceil hl/p \rceil)$ berechnet werden.

Nach Theorem 11.9 kann C' aus B' durch eine Reshape-Operation in Zeit $O(\lceil hl/p \rceil + \log p)$ erzeugt werden, wobei die Kommunikationszeit durch $(\log p)\sigma + (2hl/p + \log p)\rho$ beschränkt ist.

Sei C^T die Transponierte von Matrix C. Matrix C^T kann aus B wie eben durch eine Reshape-Operation erzeugt werden. Man erhält C aus C^T durch Transposition. Mit Theorem 11.3 folgt

Theorem 11.10. *Das äußere Produkt zweier APL-Werte mit h bzw. l Elementen kann in Zeit $O(\lceil hl/p \rceil + p^2)$ berechnet werden. Die Kommunikationszeit ist durch $(2 \log p)\sigma + (9hl/p + 5p^2 + 2 \log p + 5)\rho$ beschränkt.*

11.6.3 Mehrdimensionales Indexing. Für $i \in \{0, \ldots, k-1\}$ sei X^i ein APL-Wert mit Dimension d_i und Shape s_i. Es sei B ein APL-Wert mit Dimension k und Shape s. Dann ist $B[X^{k-1}, \ldots, X^0]$ ein APL-Wert mit Dimension $D = \sum d^i$ und Shape $t = (s^{k-1}, ..., s^0)$. Für $i \in \{0, \ldots, k-1\}$ sei das d^i-Tupel j^i ein Index für X^i. Wir definieren

$$B[X^{k-1}, \ldots, X^0][j^{k-1}, \ldots, j^0] = B[X^{k-1}[j^{k-1}], \ldots, X^0[j^0]] .$$

Ist beispielsweise B eine Matrix und X bzw. Y sind Folgen von Zeilenindizes bzw. Spaltenindizes, so ist $B[X, Y]$ der Minor von B, der aus den Elementen mit Zeilenindex in X und Spaltenindex in Y besteht.

Der Ausdruck $B[X^{k-1}, \ldots, X^0]$ kann als rechte oder linke Seite von Zuweisungen auftreten. Die Zuweisung $A \leftarrow B[X^{k-1}, \ldots, X^0]$ hat die gleiche Wirkung wie die sequentielle Folge der Zuweisungen $A[j] \leftarrow B[X^{k-1}, \ldots, X^0][j]$ für alle j. Die Reihenfolge ist nicht wichtig.

Im Falle eines indexed assignment soll eine Teilmenge der Elemente von B seinen Wert verändern. Hierzu setzt man $d^i = 1$ für alle i voraus; dann haben B und $B[X^{k-1}, \ldots, X^0][j]$ die gleiche Dimension. Die Zuweisung $B[X^{k-1}, \ldots, X^0] \leftarrow A$ hat die gleiche Wirkung wie die sequentielle Folge der Zuweisungen $B[X^{k-1}, \ldots, X^0][j] \leftarrow A[j]$ in lexikographischer Ordnung der Indizes j.

Wir realisieren höherdimensionales Indexing mit Hilfe der Algorithmen für den eindimensionalen Fall und für das äußere Produkt.

Für $x \in \{0,\ldots,k-1\}$ sei $S_x = \prod_{j<x} s_j$. Sei nun $(,A)[i]$ ein Element des Ravels von A. Dann ist $i = t\perp j$ für genau ein $j = (j^{k-1},\ldots,j^0)$ wegen der Eindeutigkeit der verwendeten Zahlendarstellungen. Beim indexed assignment muß das Ravel-Element $(,A)[i] = A[j]$ zugewiesen werden an

$$\begin{aligned} B[X^{k-1},\ldots,X^0][j] &= B[X^{k-1}[j^{k-1}],...,X^0[j^0]] \\ &= (,B)[s\perp(X^{k-1}[j^{k-1}],...,X^0[j^0])] \\ &= (,B)[X^{k-1}[j^{k-1}]S_{k-1} + \cdots + X^0[j^0]S_0] \\ &= (,B)[S_{k-1}X^{k-1} \circ . + \cdots \circ . + S_0X^0[j]]\,. \end{aligned}$$

Ist also Y das äußere Produkt $S_{k-1}X^{k-1} \circ . + \cdots \circ . + S_0X^0$, dann wird $(,A)[i]$ zugewiesen an $(,B)[Y[j]] = (,B)[(,Y)[t\perp j]] = (,B)[(,Y)[i]]$. Überdies ist $j \leq_{lex} j'$ genau dann wenn $i = t\perp j \leq t\perp j' = i'$. Man kann also statt des indexed assignment $B[X^{k-1},\ldots,X^0] \leftarrow A$ einfach das indexed assignment $(,B)[(,Y)] \leftarrow (,A)$ mit den eindimensionalen Ravels von B, Y und A ausführen.

Das gleiche Argument zeigt, daß man statt $A \leftarrow B[X^{k-1},\ldots,X^0]$ einfach $(,A) \leftarrow (,B)[(,Y)]$ ausführen kann.

Für die Berechnung von Y muß der Master den Shape-Vektor s broadcasten. Wir verzichten auf eine Laufzeitanalyse.

11.7 Parallel Prefix-Berechnung

Es sei S eine Menge und $\circ : S^2 \to S$ eine assoziative Funktion. Die Funktion $PP^\circ : S^n \to S^n$,

$$PP^\circ_j(s_0,\ldots,s_{n-1}) = \begin{cases} s_0 \circ \cdots \circ s_j & \text{falls} \quad j > 0 \\ s_0 & \text{falls} \quad j = 0 \end{cases}$$

für alle $j \in \{0,\ldots,n-1\}$, heißt *Parallel Prefix* der Länge n von $\circ$. Wir passen die Schaltkreise zur Berechnung von Parallel Prefix aus [13] in naheliegender Weise an.

11.7.1 Simulation von Parallel Prefix-Schaltkreisen. Parallel Prefix-Schaltkreise baut man aus Gattern mit zwei Inputs, welche die Funktion $\circ$ berechnen. Für Zweierpotenzen n liefert die Standardkonstruktion aus [13] Parallel Prefix-Schaltkreise mit Tiefe $2\log n - 1$ und nicht mehr als $2n + 2$ Gattern (siehe etwa [11]). Für $n = p$ übersetzt man diese Schaltkreise direkt in einen Algorithmus mit Laufzeit $O(\log n) = O(\log p)$ und Kommunikationszeit $(2\log n - 1)(\sigma + \rho)$.

Sei nun $n > p$ und der Vektor s sei in *Spaltenordnung* gespeichert. Für alle i seien $\nu(i)$ und $V(i)$ definiert wie in Abschnitt 11.4, und es sei $S_i = (s_{V(i-1)},\ldots,s_{V(i)-1})$ die Folge der Elemente von s, die auf Prozessor i gespeichert ist. Dann wird $PP^\circ(S)$ wie folgt berechnet:

i) Jeder Prozessor i berechnet lokal $PP^\circ(S_i)$.
ii) Für alle i sei $P(i)$ die letzte Komponente von $PP^\circ(S_i)$.
Wir berechnen $PP^\circ(P(0),\ldots,P(n-1))$ mit dem Algorithmus für den Fall $n = p$.
iii) Sei $(R(0),\ldots,R(n-1))$ das Ergebnis dieser Rechnung. Dann ist $R(i) = PP^\circ(s_0,\ldots,s_{V(i)-1})$ für alle i. Prozessor $i-1$ sendet $R(i-1)$ an Prozessor i für $i > 1$.
iv) Für $i > 0$ berechnet Prozessor i

$$R(i-1) \circ PP^\circ(S_i)[j] = PP^\circ(S)[V(i-1)+j]$$

für alle $j \in \{0,\ldots,\nu(i)-1\}$.

Es folgt

Theorem 11.11. *Für Eingabe und Ausgabe in Spaltenordnung kann Parallel Prefix der Länge n in Zeit $O(\lfloor n/p \rfloor + \log p)$ berechnet werden. Die Kommunikationszeit ist beschränkt durch $(2\log p)(\sigma+\rho)$.*

11.7.2 Encode und Decode. Sei $s = (s_{n-1},\ldots,s_0)$ ein Vektor der Länge n, und es sei A ein APL-Wert mit m Elementen und Shape t. Die APL-Operation *encode* angewendet auf s und A erzeugt das Feld $B = s\top A$ mit Shape (n,t) und $s\perp B(0|j) = A[j]$ für alle Indizes j.

Die APL-Operation *decode* erzeugt aus einem Vektor s der Länge n und einem APL-Wert B mit Shape (m,n) einen APL-Wert $A = s\perp B$ mit Shape m und $A[j] = s\perp B(0|j)$ für alle j.

Bei der parallelen Berechnung dieser Funktionen werden die Multiplikationen $S_k = \prod_{x<k} s_x$ durch eine Parallel Prefix-Berechnung durchgeführt. In der Praxis ist n klein, und die Parallelisierung ist nicht von großem Interesse. Wir verzichten auf die Details.

11.7.3 NAND-Reduktion. Die Schaltfunktionen NAND und NOR sind nicht assoziativ. Da ihr Wertebereich endlich ist, kann man Ihre Berechnung aber mit Techniken aus [13] leicht auf eine Parallel Prefix-Berechnung zurückführen.

11.7.4 Replikation und Kompression. Sei A ein APL-Wert mit Dimension d und Shape $s = (s_{d-1},\ldots,s_0)$. Sei $i \in \{0,\ldots,d-1\}$, es sei $m = s_i$ und es sei $c = (c_0,\ldots,c_{m-1})$ ein Vektor mit s_i Elementen vom Typ integer. Für alle $t \in \{0,\ldots,m-1\}$ sei $C_t = \sum_{j\le t} |c_j|$, es sei $C = (C_0,\ldots,C_{m-1})$ und $\zeta = C_{m-1}$.

Wir definieren $B = c/[i]A$, die *c-Replikation* von A in Richtung i. Der APL-Wert B hat Shape $s' = (s_{d-1},\ldots,s_{i+1},\zeta,s_{i-1},\ldots,s_0)$. Für jeden Index j wird die Achse $B(i|j)$ aus $A(i|j)$ in der folgenden Weise konstruiert: Es seien $j_{up} = (j_{d-1},\ldots,j_{i+1})$ und $j_{low} = (j_{i-1},\ldots,j_0)$. Für $t \in \{0,\ldots,m-1\}$ sei $\alpha(t,j)$ die Folge, die aus c_t Kopien des t-ten Elements $A[j_{up},t,j_{low}]$ von Achse $A(i|j)$ besteht, falls $c_t > 0$, die leere Folge ist, falls $c_t = 0$, und die Folge aus $|c_t|$ Elementen z wie in Abschnitt 11.5.5, falls $c_t < 0$. Dann ist

$B(i|j) = \alpha(0,j), \ldots, \alpha(m-1,j)$. Sind alle Komponenten von c aus $\{0,1\}$, so heißt die eben definierte Operation eine *Kompression*.

Ist also $c_t > 0$, dann muß für alle j das Element $A[j_{up}, t, j_{low}]$ von A in die Elemente $B[j_{up}, C_{t-1} + x, j_{low}]$ kopiert werden für $x \in \{0, \ldots, c_t - 1\}$. Ist $c_t < 0$, so muß z in diese Elemente kopiert werden.

Sei nun $h = \prod_{j=i+1}^{d-1} s_j$ und sei $l = \prod_{j=0}^{i-1} s_j$. Nach Gleichung (11.4) können wir dann $(,B)$ berechnen, indem wir $(,A)$ als den Ravel eines Feldes A' mit Shape (h,m,l) ansehen und den Ravel von $c/[1]A'$ berechnen. Wir behandeln zunächst den Fall $m \geq p$.

i) Wir bringen c in Spaltenordnung (mit Algorithmus (Rt0)), berechnen die Elemente C_t durch eine Parallel Prefix-Berechnung und bringen das Resultat zurück in Zeilenordnung. Die Kommunikationszeit ist begrenzt durch $(2p + 2\log p + 1)\sigma + (2m/p + 2p + 2\log p + 1)\rho$.

ii) Als nächstes müssen wir die Elemente $A[x,t,y]$ und c_t bzw. C_{t-1} zusammenbringen. Man ist versucht, die Vektoren c und C zu broadcasten, aber das kostet Zeit $\Theta(m)$. Das ist schlecht, falls $hl < p$.
Deshalb routen wir die Elemente $A[x,t,y]$ an den Prozessor, der c_t und C_{t-1} speichert.Wir kreieren Pakete $(A[x,t,y],(h,m,l)\perp(x,t,y))$ und routen sie zu Prozessor $t \bmod p$. Danach hat jeder Prozessor höchstens $hl\lceil m/p\rceil \leq hlm/p + hl$ Pakete. Die Kommunikationszeit ist beschränkt durch

$$\begin{aligned} RT(\lceil hml/p\rceil, hl\lceil m/p\rceil, 2) &\leq 2p\sigma + (3\lceil hml/p\rceil + 2hl\lceil m/p\rceil + 5p^2)\rho \\ &\leq 2p\sigma + (7hml/p + 5p^2 + 3)\rho, \end{aligned}$$

da $1 \leq m/p$.

iii) Wir ersetzen Elemente $A[x,t,y]$ durch z, falls $c_t < 0$ gilt, und wir ersetzen jedes $c_t < 0$ durch $|c_t|$. Wir entfernen alle Pakete $A[x,t,y]$ mit $c_t = 0$. Für alle verbleibenden Pakete $A[x,t,y]$ ist nun $c_t \geq 1$.

iv) Jeder Prozessor i speichert $\nu(i)$ Elemente c_t, wobei $\nu(i)$ wie in Abschnitt 11.4 definiert ist. Sei

$$\zeta_i = \sum_{j=0}^{\nu(i)-1} c_{i+jp}\,.$$

Das ist die Summe der Elemente c_t, die auf Prozessor i gespeichert sind. Für jedes solche Element c_t gibt es hl Elemente $A[x,t,y]$, die kopiert werden müssen in die Positionen $(h,z,l)\perp(x, C_{t-1}+u, y)$ für alle $u \in \{0,\ldots,c_t-1\}$. Es liegt nahe, ein Paket $(A[x,t,y],(h,\zeta,l)\perp(x,C_{t-1}+u,y))$ zu kreieren für jede Kopie, die von Element $A[x,t,y]$ gemacht werden soll, und diese Pakete dann an ihre Bestimmungsorte zu schicken. Hierbei braucht man aber schon Zeit $hl\zeta_i\rho$, um alle Pakete von Prozessor i abzuschicken. Das ist schlecht, falls $\zeta_i > \max\{m/p, \zeta/p\}$ für irgendein i gilt.
Wir verwenden daher nochmals *Smoothing* und führen *Kommando-Pakete* ein. Ein solches Paket f hat 3 Komponenten (a,b,d). Hierbei

ist a ein elementarer Wert, b ist eine Position im Ravel von B, und $d \geq 1$ ist eine natürliche Zahl. Man *führt* das Kommandopaket (a, b, d) *aus*, indem man Element a in die Positionen $b, b+l, \ldots, b+(d-1)l$ des Ravels von B kopiert. Eine *Kommandofolge* f mit Länge ν ist eine Folge von Kommandopaketen $f(i) = (a(i), b(i), d(i)), i \in \{0, \ldots, d(\nu)-1\}$. Die Summe der Zahlen $d(i)$ heißt das *Gewicht* der Kommandofolge. Für alle i sei $D(i) = \sum_{j \leq i} d(j)$.
Spaltung der Kommandofolge f *bei Länge* μ erzeugt die beiden Kommandofolgen $f(0) \ldots f(\mu-1)$ und $f(\mu) \ldots f(\nu-1)$. Spaltung der Folge *bei Gewicht* k erzeugt ebenfalls zwei Kommandofolgen: sei $k \in \{1, \ldots, D(n-1)\}$ und es sei $D(i-1) < k \leq D(i)$. Ist $k = D(i)$, so sind es die beiden Folgen $f(0) \ldots f(i)$ und $f(i+1) \ldots f(n-1)$. Ist $k < D(i)$, so sind es die beiden Folgen

$$f(0) \ldots f(i-1)(a(i), b(i), (k - D(i-1)))$$

und

$$(a(i), b(i) + l(k - D(i-1)), d(i) - (k - D(i-1)))f(i+1) \ldots f(\nu-1)\,.$$

Wir bemerken, daß die erste der beiden Folgen genau Gewicht k hat. Die zweite Folge hat das verbleibende Gewicht.
Wir kehren zur Beschreibung des Algorithmus zurück. Jeder Prozessor i kreiert eine Kommandofolge S_i, die aus den Kommandopaketen $(A[x,t,y], (h,\zeta,l) \perp (x, C_{t-1}, y), c_t)$ besteht für alle x, y und alle t, so daß $c_t > 1$ gilt und c_t auf Prozessor i gespeichert ist. Diese Kommandofolge hat Gewicht $\zeta_i hl$ und Länge $\nu(i)hl < \lceil m/p \rceil hl < 2hml/p$, da $m \geq p$.
Von S_i spalten wir sukzessive die längste Kommandofolge ab, die Gewicht höchstens $\lceil 2\zeta_i hl/p \rceil$ und Länge höchstens $\lceil 4hml/p \rceil$ hat. Mit jeder solchen Spaltung sinkt das Gewicht der verbleibenden Folge um den entsprechenden Betrag. Da Gewicht und Länge höchstens $p/2$-mal sinken können, haben wir zum Schluß höchstens p Folgen $S_{i,j}, j \in \{0, \ldots, p-1\}$. Für jedes i und j wird die Folge $S_{i,j}$ zu Prozessor j geroutet. Die Kommunikationszeit ist beschränkt durch

$$\begin{aligned} RT(p\lceil 4hml/p^2 \rceil, p\lceil 4hml/p^2 \rceil, 3) &< 2p\sigma + (7p\lceil 4hml/p^2 \rceil + 7p^2)\rho \\ &< 2p\sigma + (28hlm/p + 7p^2 + 7p)\rho\,. \end{aligned}$$

v) Jeder Prozessor hat nun Kommandofolgen mit Gesamtgewicht höchstens

$$y = \sum_i 2\lceil \zeta_i hl/p \rceil \leq 2hl\zeta/p + 2p\,.$$

Die Folge wird in einzelne Kommandos $(a, b, 1)$ gespalten, die letzte Komponente der Kommandopakete wird entfernt, und (a, b) wird an Prozessor $b \bmod p$ geschickt. Dort wird a an Position b von $(, B)$ kopiert. Die Kommunikationszeit ist beschränkt durch

$$\begin{aligned} RT(y, \lceil hl\zeta/p \rceil, 2) &\leq 2p\sigma + (3(2hl\zeta/p + p) + 2(hl\zeta/p + 1) + 5p^2)\rho \\ &\leq 2p\sigma + (8hl\zeta/p + 5p^2 + 3p + 2)\rho. \end{aligned}$$

Für $m \geq p$ folgt damit schon

Theorem 11.12. *Es sei c ein Vektor mit m Elementen, A sei ein APL-Wert mit Shape (h, m, l) und es sei $\zeta = \sum_i |c_i|$. Dann kann $c/[1]A$ in Zeit $O(hml/p + h\zeta l/p + p^2)$ berechnet werden. Die Kommunikationszeit ist beschränkt durch*

$$(8p + 2\log p + 1)\sigma + (35hlm/p + 8hl\zeta/p + 2m/p + 17p^2 + 12p + 2\log p + 6)\rho.$$

Beweis. Wir müssen noch den leichteren Fall $m < p$ behandeln. Statt der obigen Schritte i) und ii) wird Vektor c einfach gebroadcastet. Danach berechnet jeder Prozessor die Vektoren C lokal. Das Broadcasten geschieht in trivialer Weise durch eine Folge von p Shifts mit Kommunikationszeit $p\sigma + pm\rho < p\sigma + p^2\rho$. Schritt iii) ist wie oben.

iv) Für $i \in \{0, \ldots, p-1\}$ und $t \in \{0, \ldots, m-1\}$ sei $\mu(i,t)$ die Zahl von Paaren (x, y), so daß $A[x, t, y]$ auf Prozessor i gespeichert ist, es sei $\nu(i,t) = \mu(i,t)c_t$ und $K_i = \sum_t \nu(i,t)$. Dies ist die Anzahl von Kopien, die von Elementen gemacht werden müssen, die auf Prozessor i gespeichert sind. Jeder Prozessor i kreiert wie oben eine Kommandofolge S_i. Die Folge hat höchstens Länge $\lceil hlm/p \rceil$ und Gewicht K_i. Wir spalten sukzessive die längste Folge mit Gewicht höchstens $\lceil 2K_i/p \rceil$ und Länge höchstens $\lceil 2hml/p^2 \rceil$ ab. Dadurch erhalten wir höchstens p Folgen $S_{i,j}$. Wie oben wird $S_{i,j}$ an Prozessor j geroutet. Die Kommunikationszeit ist beschränkt durch

$$RT(p\lceil 2hml/p^2 \rceil, p\lceil 2hml/p^2 \rceil, 3) \leq 2p\sigma + (14hlm/p + 7p^2 + 7p)\rho.$$

v) Jeder Prozessor hat nun Kommandofolgen mit Gewicht höchstens

$$\begin{aligned} y &= \sum_i \lceil 2K_i/p \rceil \\ &\leq p + 2\sum_i K_i/p \\ &= p + 2h\zeta l/p, \end{aligned}$$

und der Beweis wird wie oben zu Ende geführt.

□

Wenn der Vektor c nur aus Nullen und Einsen besteht, kann man sich Schritt iii) und den Smoothing-Schritt sparen.

11.8 Sortieren

11.8.1 Simulation von Sortiernetzen. Wir passen Batchers Bitonische Sortierer in naheliegender Weise an [3, 6, 12]. Es sollen n Zahlen sortiert werden. Es ist gleichgültig, ob sie in Zeilenordnung oder Spaltenordnung vorliegen. Das Ergebnis wird in Spaltenordnung erzeugt.

Ein *Sortierer* ist ein Netzwerk, das aus Vergleichern aufgebaut ist [6] und das alle Input-Folgen sortiert ausgibt. Ein *Bitonischer Sortierer* ist ein Netzwerk, das aus Vergleichern aufgebaut ist und alle bitonischen Input-Folgen sortiert ausgibt[3].

Für Zweierpotenzen n kann man Bitonische Sortierer B_n für n Inputs aus $n/2$ Vergleichern und zwei Kopien von $B_{n/2}$ konstruieren. Man konstruiert Sortierer S_n für n Inputs aus zwei Kopien von $S_{n/2}$ (eine sortiert aufsteigend, die andere absteigend) und einem Bitonischen Sortierer B_n (siehe etwa [6]). Netzwerk S_n hat $O(n \log^2 n)$ Vergleicher.

Für $n \geq p$ kostet die direkte Simulation von Netzwerk S_n Zeit $O((n/p) \log^2 n)$. Dies kann man verbessern [25]: jeder Prozessor simuliert n/p Zeilen aus Abb. 28.9 in [6]. Die Netze $S_{n/p}$ bzw. $B_{n/p}$ werden lokal auf den Slaves simuliert (und zwar durch Bucket-Sort bzw. eine Merge-Routine in Zeit $O(n/p)$). Erst dann beginnt man mit der Rekursion. Wir beobachten, daß n hier keine Zweierpotenz zu sein braucht. Es genügt, wenn p eine Zweierpotenz und n ein Vielfaches von p ist.

Für $i \in \{0, \ldots, \log p\}$ sei $m(i) = 2^i n/p$ und es sei $B(i)$ bzw. $S(i)$ die Kommunikationszeit bei der Simulation von $p/2^i$ Netzen $B_{m(i)}$ bzw. $S_{m(i)}$. Offensichtlich ist $B(0) = S(0) = 0$. Für $i > 0$ simuliert man Netzwerke $B_{m(i)}$, indem man zunächst die Netzwerke $B_{m(i-1)}$ simuliert und dann Shifts über die Distanz 2^i und -2^i durchführt. In jeder Kommunikationsrunde werden n/p elementare Daten gesendet oder empfangen. Es folgt $B(i) = B(i-1) + 2(\sigma + (n/p)\rho) = 2i(\sigma + (n/p)\rho)$. Für $i > 0$ gilt $S(i) = S(i-1) + B(i) = S(i-1) + 2i(\sigma + (n/p)\rho) = i(i+1)(\sigma + (n/p)\rho)$. Die Kommunikationszeit ist also beschränkt durch $(\log^2 p + \log p)(\sigma + (n/p)\rho)$. Die Gesamtlaufzeit ist beschränkt durch $O(n \log^2 p/p)$.

Ist n kein Vielfaches von p, so muß in den obigen Formeln n/p zu $\lceil n/p \rceil$ aufgerundet werden. Offensichtlich kann man nach dem gleichen Verfahren auch L-Tupel sortieren. Mithin gilt

Theorem 11.13. *Für $n \geq p$ können Folgen von n L-Tupeln in Zeit $O(n \log^2 p/p)$ sortiert werden. Die Kommunikationszeit ist beschränkt durch $(\log^2 p + \log p)\ (\sigma + L\lceil n/p \rceil \rho)$.*

Verfügt die Zielmaschine über ein Permutationsnetz, so können die oben erwähnten Shifts über Distanzen 2^i und -2^i gleichzeitig durchgeführt werden.

[3] Wir benutzen das Konzept bitonischer [illegible]

11.8.2 Grade up und grade down. Sei $v = (v_0, \ldots, v_{n-1})$ ein Vektor. Dann erzeugt $\uparrow(v)$ (*grade up* von v) einen Vektor $\pi = (\pi(0), \ldots, \pi(n-1))$ so daß gilt: π ist eine Permutation und $v_{\pi(0)} \leq \ldots \leq v_{\pi(n-1)}$. Ist $v_i = v_j$ und $i < j$, dann muß überdies $\pi(i) < \pi(j)$ gelten. Das Ergebnis von $\downarrow(v)$ (*grade down* von v) ist ähnlich definiert; es muß $v_{\pi(0)} \geq \ldots \geq v_{\pi(n-1)}$ gelten. Wir erklären nur, wie grade up berechnet wird.

Man erzeugt lokal Paare (v_j, j) und sortiert sie lexikographisch. Man erhält die Folge von Paaren $(v_{\pi(0)}, \pi(0)), \ldots, (v_{\pi(n-1)}, \pi(n-1))$. Mit Algorithmus (Rt0) bringt man die zweiten Komponenten in Zeilenordnung. Somit gilt

Theorem 11.14. *Grade up und grade down eines Vektors der Länge $n \geq p$ kann in Zeit $O(n \log^2 p/p)$ berechnet werden. Die Kommunikationszeit ist beschränkt durch $(p + \log^2 p + \log p)\sigma + (\log^2 p + \log p)(2\lceil n/p \rceil + n/p + p)\rho$.*

11.8.3 Index of und Element. Sei $c = (c_0, \ldots, c_{n-1})$ ein Vektor, und es sei A ein APL-Wert mit m Elementen. Dann ist $c\iota A$ (Index of) ein APL-Wert B mit dem gleichen Shape wie A, und für alle Indizes j gilt

$$B[j] = \begin{cases} \min\{i | A[j] = c_i\} & \text{falls es ein solches } i \text{ gibt,} \\ n & \text{sonst.} \end{cases}$$

Wir setzen $n + m \geq p$ voraus. Der Ravel von B wird auf folgende Weise berechnet:

i) Der Ravel von c wird mit dem Ravel von A konkateniert. Die Shiftdistanz ist $m \bmod p$, und die Kommunikationszeit ist beschränkt durch $\sigma + \lceil n/p \rceil \rho$.

ii) Jeder Prozessor speichert nun höchstens $\lceil (n+m)/p \rceil$ Ravelelemente. Für jedes Ravelelement c_i wird das Paket $(c_i, 0, i)$ erzeugt. Für jedes Ravelelement $(, A)[k]$ wird das Paket $((, A)[k], 1, k)$ erzeugt. Wir sagen, daß ein Paket *von A kommt*, wenn seine zweite Komponente gleich 1 ist. Wir sortieren die Pakete lexikographisch in Zeit $O(((n+m)\log^2 p)/p)$. Die Kommunikationszeit ist beschränkt durch $(\log^2 p + \log p)\sigma + (\log^2 p + \log p)3\lceil (n+m)/p \rceil \rho$.

iii) Das Resultat der Sortierung ist die Folge von Tripeln $S = S_0, \ldots, S_{n+m-1}$, die in Spaltenordnung gespeichert ist. Wir definieren die Operation $\circ$ auf Tripeln durch

$$(a, b, c) \circ (a', b', c') = \begin{cases} (a, b, c) & \text{falls } a = a', \\ (a', b', c') & \text{sonst.} \end{cases}$$

und führen die Parallel Prefix-Berechnung $C = PP^{\circ}_{n+m} S$ durch. Das Ergebnis ist in Spaltenordnung gespeichert. Die Kommunikationszeit ist beschränkt durch $(2 \log p)(\sigma + 3\rho)$.

iv) Wir vergleichen lokal jedes Tripel $S_i = ((,A)[k], 1, k)$, das von A kommt, mit dem entsprechenden Tripel $C_i = (\alpha_i, \beta_i, \gamma_i)$, das in der Parallel Prefix-Berechnung erzeugt wurde. Es folgt $\alpha_i = (,A)[k]$ und

$$(,B)[k] = \begin{cases} \gamma_i & \text{falls } \beta_i = 0\,, \\ n & \text{sonst.} \end{cases}$$

Wir erzeugen für jedes solche i das Paket $(B[k], \lfloor k/p \rfloor)$ und senden es an Prozessor $k \bmod p$. Die Kommunikationszeit ist beschränkt durch $RT(\lceil (n+m)/p \rceil, \lceil m/p \rceil, 2) = 2p\sigma + (3(n+m)/p + 2m/p + 5p^2 + 5)\rho$, und es folgt

Theorem 11.15. *Es sei c ein Vektor mit n Elementen, A sei ein APL-Wert mit m Elementen, und es sei $n + m \geq p$. Dann kann $c\iota A$ in Zeit $O(((n + m)\log^2 p)/p)$ berechnet werden. Die Kommunikationszeit ist beschränkt durch $(2p + \log^2 p + 3\log p + 1)\sigma + ((3(n+m)/p)(\log^2 p + \log p + 1) + 2m/p + n/p + 5p^2 + 6\log p + 5)\rho$.*

Seien A und B APL-Werte. Dann ist $A \in B$ ein APL-Wert C mit dem Shape von A, und für alle Indizes j gilt

$$C[j] = \begin{cases} 1 & \text{falls } A[j] = B[k] \text{ für ein } k\,, \\ 0 & \text{sonst.} \end{cases}$$

Sei $D = (,B)\iota A$. Dann ist $D[j] < n$ genau dann wenn $A[j]$ im Ravel von B vorkommt. Folglich ist $C[j] = 1$ genau dann wenn $D[j] < n$ gilt.

11.8.4 Zufällige Folgen. Sei $c = (c_0, \ldots, c_{n-1})$ ein Vektor von natürlichen Zahlen. Dann erzeugt $?c$ einen zufälligen Vektor $x = (x_0, \ldots, x_{n-1})$, bei dem für alle j die x_j unabhängig und gleichverteilt in $\{1, \ldots, c_j\}$ sind. Unter der Voraussetzung, daß Prozessoren einzelne Elemente x_j in konstanter Zeit erzeugen können, ist diese Operation eine Vektoroperation[4].

Es seien n und m natürliche Zahlen und es gelte $m \leq n$. Dann erzeugt $m?n$ eine zufällige Folge $S = (S_0, \ldots, S_{m-1})$ mit m *verschiedenen* Elementen aus $\{1, \ldots, n\}$. Die zufällige Folge S ist gleichverteilt in der Menge aller dieser Folgen, d.h. jede Folge tritt mit Wahrscheinlichkeit $(n(n-1)\cdots(n-m+1))^{-1}$ auf. Man kann $m?n$ berechnen, indem man einfach die ersten m Elemente einer zufälligen Permutation von $\{1, \ldots, n\}$ nimmt. Für $n \geq p \log n / \log\log n$ führt man die parallele Berechnung zufälliger Permutationen nach dem Muster von Satz 4.1 aus [8] auf das Sortieren von n Paaren zurück.

11.9 Zusammenfassung

Die Kommunikationszeiten der analysierten Algorithmen sind in Tabelle 11.1 zusammengefaßt. Man sieht, daß der Kommunikationsaufwand der Algorithmen vor Abschnitt 11.7.4 sehr moderat bleibt.

[4] In realen Maschinen kann man nur pseudozufällige Elemente erzeugen.

Tabelle 11.1. Führende Koeffizienten von σ and ρ

Abschn.	Operation	Größe	σ	ρ
11.5.1	Routing	N, M, L	$2p$	$(L+1)N + LM$
11.5.2	Transposition Rotation Spiegelung	n	$2p$	$5n/p$
11.5.3	Indx. Assg. Indx. Ref. 1-dim.	N, M	$2p$ $4p$	$4N + 3M$ $5N + 5M$
11.5.4	Reduktion	n, m	$2p$	$7n/p + m$
11.5.5	Join Take, Drop	n, m n, m	$2p$ $2p$	$5(n+m)/p$ $(3n+2m)/p$
11.6.1	Reshape	m	$\log p$	$2m/p$
11.6.2	Äuß. Prod.	h, l	$2\log p$	$9hl/p$
11.7.1	Par. Prefix	n	$2\log p$	$2\log p$
11.7.4	Replikation	h, m, l, ζ	$8p$	$35hlm/p + 8h\zeta l/p$
11.8.1	Sort	n	$\log^2 p$	$(\log^2 p)n/p$
11.8.2	Grade up Grade down	n	$p + \log^2 p$	$(\log^2 p)n/p$
11.8.3	Index of Element	n, m	$2p$	$3(n+m)(\log^2 p)/p$

Schriftenverzeichnis

1. A.V. Aho, J.E. Hopcroft und J.D. Ullman (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley.
2. D.J. Auerbach, W. Paul, A.F. Bakker, C. Lutz, W.E. Rudge und F.F. Abraham (1987). A special purpose parallel computer for molecular dynamics: Motivation, design, implementation and application. *J. Phys. Chem.* 91, 4881–4890.
3. K.E. Batcher (1968). Sorting networks and their applications. *Proc. AFIPS Spring Joint Conference* 32, 405–416. STSC.
4. T. Bönniger, R. Esser und D. Krekel (1995). CM–5E, KSR2, Paragon XP/S: A comparative description of massively parallel computers. *Parallel Computing* 21(2), 199–232.
5. R. Butler und E. Lusk (1992). User's guide to the p4 parallel programming system. Technical Report ANL–92/17, Argonne National Laboratory.
6. T.H. Cormen, C.E. Leiserson und R.L. Rivest (1990). *An Introduction to Algorithms.* MIT Press.
7. Association for Computing Machinery (1983). Draft proposed standard programming language APL. *APL Quote Quad* 14.

8. T. Hagerup (1991). Fast parallel generation of random permutations. *Proc. 18th ICALP Lecture Notes in Computer Science* 416, 405–416. Springer.
9. K.E. Iverson (1962). *A Programming Language.* Wiley.
10. J. JaJa (1992). *An Introduction to Parallel Algorithms.* Addison-Wesley.
11. J. Keller und W.J. Paul (1995). *Hardware Design, Texte zur Informatik* 15. Teubner.
12. D.E. Knuth (1973). *The Art of Computer Programming* 3. Addison-Wesley.
13. R. Ladner und M. Fischer (1980). Parallel prefix computation. *J. ACM* 27(4), 831–838.
14. F.T. Leighton (1992). *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers.
15. C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang und R. Zak (1992). The network architecture of the connection machine CM-5. *Proc. 4th SPAA*, 272–285. ACM.
16. W.F. McColl (1994). An architecture independent programming model for scalable parallel computing. *Portability and Performance for Parallel Processing*, 43–69. Wiley.
17. M. Metcalf und J. Reed (1990). *Fortran 90 Explained.* Oxford University Press.
18. S.M. Müller und W.J. Paul (1995). *The Complexity of Simple Computer Architectures. Lecture Notes in Computer Science* 995. Springer.
19. J.M. Ortega (1988). *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press.
20. W.J. Paul (1994). A note on bitonic sorting. *Inf. Proc. Letters* 49, 223–225.
21. W.J. Paul und D. Scheerer (1991). The DATIS-P fault tolerant machine. *Proc. 24th Hawaii International Conference on System Sciences*, 560–571. IEEE.
22. S. Pommier (1983). *An Introduction to APL.* Cambridge Computer Science Texts 17.
23. S. Rajasekaran und T. Tsantilas (1990). Optimal routing in mesh-connected processor arrays. *Algorithmica* 8, 21–38.
24. J. Sauermann (1989). *Ein paralleler APL-Rechner.* PhD thesis, Universität des Saarlandes.
25. S.R. Seidel und L.R. Ziegler (1987). Sorting on hypercubes. Michael und Heath, (Eds.) *Hypercube Multiprocessors*, 285–291. SIAM.
26. L.G. Valiant (1975). On non-linear lower bounds in computational complexity. *Proc. 7th STOC*, 45–53. ACM.
27. L.G. Valiant (1990). A bridging model for parallel computation. *Comm. of the ACM* 33(8), 103–111.
28. A. Waksman (1968). A permutation network. *J. ACM* 15, 159–163.