Wolfgang Reisig

Verteiltes Rechnen: Im wesentlichen das Herkömmliche oder etwas grundlegend Neues?

Antrittsvorlesung

16. Dezember 1993

Humboldt-Universität zu Berlin
Fachbereich Informatik

Herausgeberin:
Die Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr. Marlis Dürkop

Redaktion:
Christine Gorek
Forschungsabteilung der Humboldt-Universität
Unter den Linden 6
10099 Berlin

Herstellung:
Linie DREI, Agentur für Satz und Grafik
Wühlischstraße 33
10245 Berlin

Heft 27

Redaktionsschluß:
03. 03. 1994

Einleitung

In der Tradition der Antrittsvorlesung wählt man üblicherweise ein Thema der aktuellen Forschung und stellt es für den interessierten Laien verständlich dar.

So wähle ich ein Thema über verteilte Systeme: Das ist aktuell, weil derzeit zahlreiche Aufgaben von zentralen Rechenzentren wegverlagert werden, hin zu verteilten Rechnernetzen. Und um verständlich zu bleiben, stelle ich Beispiele vor.

Wenn man eine Aufgabe nicht mit einem herkömmlichen Rechner sondern auf einem Rechnernetz lösen will, werden ganz neue Aspekte von Algorithmen sichtbar und es stellen sich grundlegende Fragen nach Prinzipien des Problemlösens in verteilten Systemen, eben nach dem im Titel angekündigten „Verteilten Rechnen“.

Im ersten Kapitel entwickeln wir am Beispiel des Sortierens aus einem klassischen Verfahren heraus einen verteilten Sortieralgorithmus. So wird insbesondere deutlich, was an der Annahme von Verteiltheit spezifisch ist.

Im zweiten Kapitel wird gezeigt, daß zahlreiche bekannte Algorithmen einen „verteilten Kern“ haben. Eine für verteilte Algorithmen spezifische Fragestellung behandelt das dritte Kapitel: Die Feststellung der Terminierung konkreter Abläufe. Schließlich stellen wir im vierten Kapitel die Frage aus dem Titel dieser Vorlesung: Hat verteiltes Rechnen prinzipiell neue Eigenschaften, oder kann man es im wesentlichen im herkömmlichen Rahmen behandeln?

1. Ein Beispiel: Verteiltes Sortieren

Ich beginne mit einem „klassischen“ Beispiel, dem Sortieren. Gegeben seien Karteikarten, jede mit der Anschrift einer Firma. Sie sollen alphabetisch sortiert werden. Dazu wird der Kartenstapel durchgeblättert, und falsch geordnete Nachbarn werden vertauscht.

Ein Beispiel sei der Kartenstapel

Gans Elch Bär Fuchs Cobra Dachs Aal. (1)

Der erste Vergleich, *Gans - Elch*, führt zu

Elch Gans Bär Fuchs Cobra Dachs Aal. (2)

Dann wird das aktuelle zweite Kartenpaar, *Gans - Bär*, verglichen (und vertauscht), dann das dritte usw. bis der erste Durchlauf mit

Elch Bär Fuchs Cobra Dachs Aal Gans. (3)

endet. Ein zweiter Durchlauf blättert wieder von links nach rechts durch und vertauscht ggf. die Karten, bis nach sechs Durchläufen der Stapel sortiert ist.

Um dies allgemein zu formulieren, bezeichne a_{12} den Vergleich der ersten beiden Karten zusammen mit ihrer eventuellen Vertauschung. a_{23}, a_{34} etc. bezeichne die entsprechende Aktion mit der zweiten und dritten, der dritten und vierten Karte, etc. Bezeichnen wir aktuelle Anordnungen („Zustände“) des Kartenstapels mit z_0, z_1, z_2 etc., so erzeugt das Verfahren ein Verhalten der Form

$$z_0 \xrightarrow{a_{12}} z_1 \xrightarrow{a_{23}} z_2 \longrightarrow \cdots \xrightarrow{a_{67}} z_6 \xrightarrow{a_{12}} \cdots \longrightarrow z_{35} \xrightarrow{a_{67}} z_{36} \qquad (4)$$

Versuchen wir nun einmal eine andere, „liberalere“ Variante: Warum eigentlich den Kartenstapel von vorn nach hinten durchblättern? Warum überhaupt eine feste Reihenfolge vorgeben?

Warum nicht nach Belieben Paare benachbarter Karten herausnehmen, prüfen und ggf. vertauschen?
Beispielsweise, erst a_{67} und dann a_{23} ausführen mit dem Resultat

Gans Bär Elch Fuchs Cobra Aal Dachs. (5)

Machen wir es geschickt, so reichen 16 Schritte zum Sortieren aus.

Dieses Verfahren („man nehme irgendein Paar benachbarter Karten aus dem Stapel, prüfe und vertausche ggf. die Karten und lege sie zurück. Dann wähle man ein weiteres Paar und verfahre entsprechend“) ist nicht deterministisch: Es gibt dabei nicht (wie oben) *einen*, sondern *viele* Abläufe der Form

$$z_0 \xrightarrow{a_{i\,i+1}} z_1 \xrightarrow{a_{j\,j+1}} z_2 \xrightarrow{a_{k\,k+1}} z_3 \cdots \qquad (6)$$

wobei i, j, k etc. Zahlen zwischen 1 und 6 sind.

Eine weitere, „noch liberalere“ Variante bietet sich sofort an: Warum immer Nachbarn vergleichen? Nehmen wir doch beliebige Paare, beispielsweise *Gans* und *Aal* in (1) mit dem Resultat

Aal Elch Bär Fuchs Cobra Dachs Gans. (7)

Dann reichen sogar vier Schritte zum sortieren. Jeder einzelne Ablauf (von den vielen möglichen) hat dann die Form:

$$z_0 \xrightarrow{a_{i\,i'}} z_1 \xrightarrow{a_{j\,j'}} z_2 \xrightarrow{a_{k\,k'}} z_3 \cdots \qquad (8)$$

wobei i, i′, j, j′, k, k′ etc. Zahlen zwischen 1 und 7 sind.

Bislang haben wir uns auf Abläufe der Form

$$z_0 \xrightarrow{\alpha_1} z_1 \xrightarrow{\alpha_2} z_2 \longrightarrow \cdots \qquad (9)$$

beschränkt, wobei z_0, z_1, ... Zustände und α_1, α_2, ... elementare Aktionen bezeichnen. Sortieren können wir auch in einer allge-

meinen Weise, indem mehrere elementare Aktionen, beispielsweise a_{12} und a_{45}, zugleich, in einem Schritt, eintreten. In diesem Fall notieren wir

$$s \xrightarrow[a_{45}]{a_{12}} s' \tag{10}$$

Tatsächlich reichen nun sogar zwei derartige Schritte, um (1) zu sortieren:

$$s_0 \xrightarrow[a_{23}]{a_{17}} s_1 \xrightarrow[a_{46}]{a_{35}} s_2 \tag{11}$$

Der allgemeine Fall besteht also aus einer Sequenz

$$s_0 \xrightarrow{A_1} s_1 \xrightarrow{A_2} s_2 \longrightarrow \cdots \tag{12}$$

von Schritten, wobei jeder Schritt $s_{n-1} \xrightarrow{A_n} s_n$ die Gestalt

$$s_{n-1} \xrightarrow[\substack{a_{i\,i'} \\ a_{j\,j'} \\ \vdots}]{} s_n \tag{13}$$

hat, mit paarweise verschiedenen Indizes i ,i′,j , j′, ...

Die obigen Konzepte, insbesondere (9) bilden also einen Spezialfall von (12), wobei jedes A_n nur *einen* Elementarschritt $a_{ii'}$ enthält.

Haben wir mit (12) also die denkbar allgemeinste Variante? Betrachten wir nochmals, was *Sortieren* bedeutet. Der Kartenstapel hat „Positionen“: In (1) liegt auf der ersten Position *Gans*, auf der zweiten *Elch* etc.

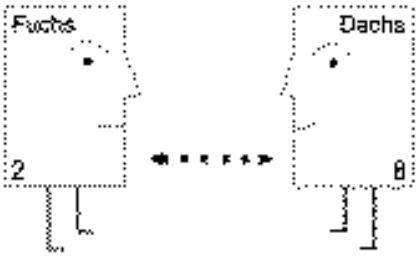
Der Stapel legt diese Positionsnummern implizit fest. Sie gehen beispielsweise verloren, wenn der Stapel umfällt. Deshalb ist es besser, auf jeder Karteikarte – mit Bleistift – die aktuelle Posi-

tionsnummer zu notieren und bei Vertauschen durch die neue Position zu ersetzen. Für (1) erhalten wir so

(14)

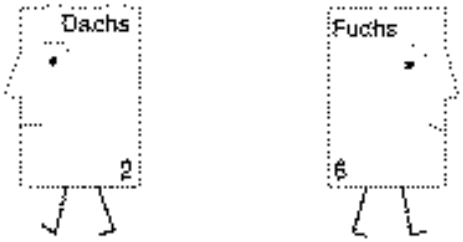
Jetzt ist der Stapel überflüssig!
Die Karteikarten können räumlich verteilt vorliegen, beispielsweise jede auf einem anderen Rechner eines Rechnernetzes. Dennoch kann man sie sortieren: Man muß die alphabetische Ordnung der Namenseinträge und die Ordnung der Positionsnummern in Übereinstimmung bringen.

Wie gelingt das nun im „allgemeinsten“ Fall? Wie sieht überhaupt der allgemeinste Fall aus? Vielleicht so, daß die Karteikarten „herumwandern“ und sich gelegentlich „begegnen“:



(15)

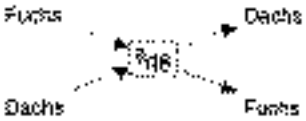
Dann vergleichen sie Ihre Positionsnummern, tauschen sie ggf. aus und wandern weiter:



(16)

Nach hinreichend vielen solchen Begegnungen sind die Karteikarten geordnet.

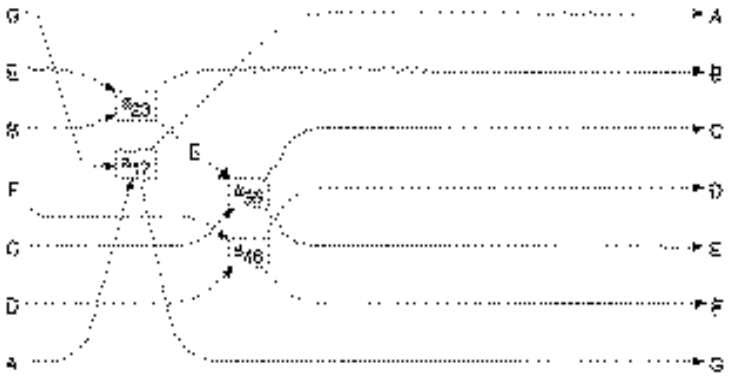
Wie kann nun ein solches Verhalten dargestellt werden, in Analogie zu (9) und (12)? Zwei Karteikarten begegnen sich unabhängig von allen anderen Karteikarten. Im Zustand (1) begegnet beispielsweise *Elch* zunächst *Bär* und dann *Cobra*. Völlig unabhängig davon begegnen sich einerseits *Gans* und *Aal*, andererseits aber auch *Fuchs* und *Dachs*. Wir notieren solche Begegnungen graphisch in der Form



(17)

mit der Konvention, daß die Karte mit der niedrigen Positionsnummer oben steht und die Karte mit der höheren Nummer unten. Jedes mögliche Gesamtverhalten der Karteikarten setzt sich nun aus solchen „Begegnungen“ zusammen.

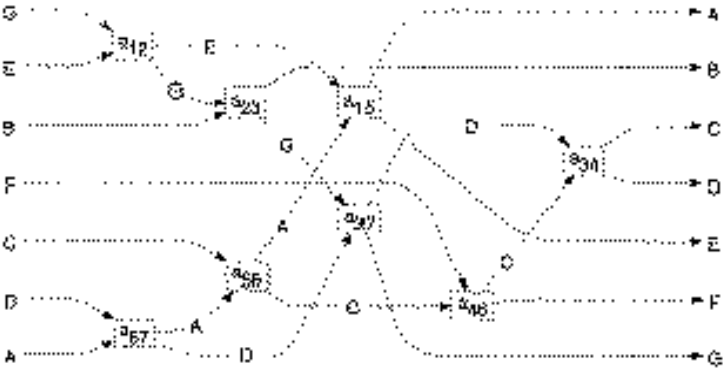
Ein Beispiel ist



(18)

Hier ist dargestellt, daß a_{23} vor a_{35} eintritt und daß unabhängig von dieser Sequenz sowohl a_{17} als auch a_{46} eintritt.

Ein anderes Beispiel ist



(19)

Hier kommen nicht nur vier, sondern acht Aktionen vor. Sie bilden teilweise *Ketten*, z.B.

$$a_{67} \longrightarrow a_{56} \longrightarrow a_{46} \longrightarrow a_{34} \qquad (20)$$

von denen sich andere abspalten (z.B. kann man in (20) nach a_{67} und a_{56} mit a_{15} fortfahren) oder erst abspalten und wieder zusammenkommen (z.B. trifft sich $a_{67} \longrightarrow a_{37} \longrightarrow a_{34}$ mit (20) am Anfang und am Ende).

Kann man (18) und (19) als Ausprägung von (12) auffassen? Dies ist nicht möglich, obwohl beispielsweise in (18) und in (11) die selben Elementaroperationen (a_{17}, a_{23}, a_{35} und a_{46}) vorkommen. Sie sind jedoch verschieden geordnet: Die Ordnung von (18) ist schwach: a_{23} liegt vor a_{35}; alle anderen Paare sind ungeordnet. Stellen wir die direkte Nachfolge in einer Ordnung durch Pfeile dar, so führt (18) zur Ordnung

$$\begin{array}{ccc} a_{23} & \longrightarrow & a_{35} \\ a_{46} & & a_{17} \end{array} \qquad (21)$$

Diese Ordnung beschreibt die *kausalen* Abhängigkeiten der Elementaroperationen: a_{35} baut auf dem Ergebnis von a_{23} auf; alle anderen Elemtaroperationen arbeiten unabhängig voneinander, mit der anfangs gegebenen Konfiguration (1). Die Sequenz der zwei Schritte (11) beschreibt a_{17} und a_{23} als gleichzeitig; entspre-

chend treten a_{35} und a_{46} gleichzeitig ein. Die beiden Schritte sind *zeitlich* geordnet: a_{17} liegt vor a_{35} und vor a_{46} und entsprechend liegt a_{23} vor a_{35} und vor a_{46}:

(22)

Diese Ordnung ist nicht nur *stärker* als die von (18), sie ist auch in einer Weise „regelmäßig“, die zeitliche Ordnung im allgemeinen von kausaler Ordnung unterscheidet: In zeitlicher Ordnung sind zwei Aktionen genau dann ungeordnet, wenn sie *gleichzei - tig* eintreten.
Gleichzeitigkeit ist *transitiv* (wenn a und b gleichzeitig sind und wenn b und c gleichzeitig sind, dann auch a und c). Und in der Tat ist die Nicht-Ordnung in (22) transitiv, während sie in (21) nicht transitiv ist. Dort sind beispielsweise a_{23} und a_{17}, aber auch a_{17} und a_{35} ungeordnet, nicht jedoch a_{23} und a_{35}.

Zusammengefaßt formuliert, kann kausal geordnetes Verhalten reicher strukturiert sein als zeitlich geordnetes.

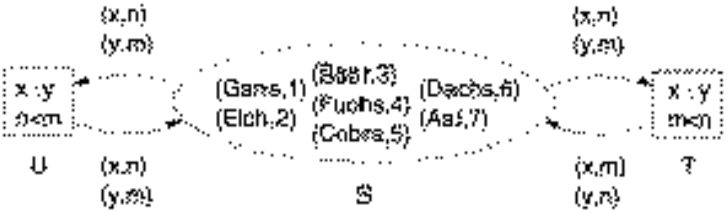
Eine Kette einer kausalen Ordnung (z.B. 20) repräsentiert kausales Aufeinanderfolgen von Aktionen, während eine Kette einer zeitlichen Ordnung (z.B. a_{17}, a_{46} in (22)) Aktionen in ein Zeitraster einordnet. So sind alle Ketten einer zeitlichen Ordnung gleich lang, während Ketten einer kausalen Ordnung verschieden lang sein können.

Bislang haben wir mögliches Verhalten von Systemen und Programmen untersucht und zum Teil formal dargestellt. Systeme und Programme selbst haben wir nur informell, umgangssprachlich beschrieben.
Verhalten der in (9) beschriebenen Form, also Sequenzen aus Zuständen und elementaren Aktionen, ist die „klassische“ Form der (operationellen) Semantik von Programmen. Wie solche Programme für das Sortieren aussehen können, ist bekannt und braucht hier nicht diskutiert zu werden. Verhalten der in (12) be-

schriebenen Form, also Sequenzen aus Zuständen und Mengen elementarer Aktionen, wird als Semantik für eine Reihe von System- und Programmkonzepten verwendet, darunter „parallelen Programmen“, „systolischen Algorithmen“ und „parallelen random access machines“.

Wie sehen aber Systeme und Programme mit „verteilten“ Abläufen wie z.B. (18) oder (19) aus, die also aus halbgeordneten Aktionen und lokal vorliegenden Teilzuständen bestehen?

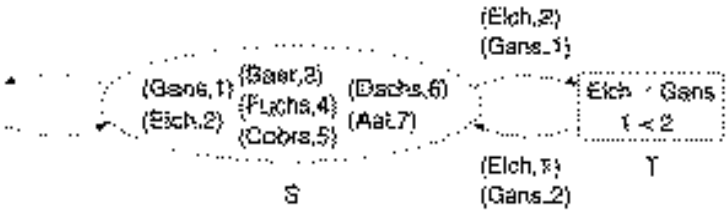
Zur Darstellung derartiger „verteilter“ Algorithmen gibt es eine überraschend einfache Technik. Verteiltes Sortieren wird beispielsweise so dargestellt:



(23)

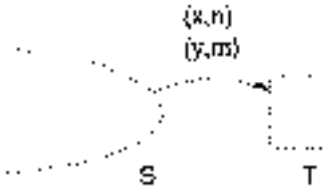
Dabei enthält der Kreis S die Karteikarten mit ihren jeweiligen Positionsnummern (in (23) mit den durch (1) gegebenen Anfangspositionen). Jeder Kasten beschreibt ein *Aktionsschema*. Eine konkrete Aktion entsteht durch Belegung der Variablen x, y, n und m in den Pfeilinschriften (x,n) und (y,m) durch Karteikarten mit ihren aktuellen Positionsnummern.

Die Belegung von (x,n) mit (*Elch*,2) und von (y,m) mit (Gans,1) führt beispielsweise für T zur Aktion



(24)

Diese Aktion ist in der Tat aktiviert, weil zwei Voraussetzungen dafür erfüllt sind. Zum einen erfüllt diese Belegung die beiden Bedingungen x ‹ y und m < n in T (‹ bezeichne die alphabetische, < die numerische Ordnung), zum anderen ergibt die Belegung der Anschriften am „herausnehmenden“ Pfeil



(25)

Karteikarten (nämlich (*Elch*,2) und (*Gans*,1)), die aktuell in S verfügbar sind.

Tritt die Aktion nun ein, so wird gemäß der interpretierten Pfeilanschriften (*Elch*,2) und (*Gans*,1) aus S entnommen, und (*Elch*,1) und (*Gans*,2) zu S hinzugefügt. Jede T-Aktion *vertauscht* also die Positionsnummern der beteiligten Karteikarten; entsprechend zeigen die Inschriften in U, daß jede U-Aktion Karteikarten vergleicht und feststellt, daß sie bereits korrekt geordnet sind.

Wichtig ist nun der *lokale* Charakter der Aktion: Eine konkrete Aktion bezieht sich nicht auf einen Gesamtzustand, sondern auf zwei einzelne Karteikarten. Betreffen Aktionen verschiedene Karteikarten, so treten sie unabhängig voneinander ein. Beziehen sich zwei Aktionen auf die selbe Karteikarte, so entsteht ein *Konflikt*, der in einem konkreten Fall nichtdeterministisch zu Gunsten einer der Aktionen entschieden wird.

Das System (23) beschreibt somit unendlich viele verschiedene Verhaltensmöglichkeiten. Zwei davon sind (18) und (19).
Mit (23) ist nun die allgemeinste Form des „Sortierens durch Vertauschen“ gegeben. Jede der zu Beginn beschriebenen Varianten ist ein Spezialfall davon, in dem nur Sequenzen von Aktionsmengen oder gar nur eine Aktion zugleich erlaubt sind.
Besonders interessant am verteilten Sortieren nach (23) ist das Erreichen einer *globalen* Eigenschaft („wo immer die Karten sich

befinden, sie sind als Ganzes gesehen geordnet“) aufgrund *loka - ler* Aktionen (zwei Karten begegnen sich und tauschen ggf. ihre Positionsnummern aus). Diese Art des Sortierens findet man in realen Systemen wieder, beispielsweise in einem geschüttelten Gemisch aus Wasser und Öl: Läßt man es ruhig stehen, sortieren sich die Ölteile nach oben und die Wasserteile nach unten, durch lokales Vertauschen benachbarter Wasser- und Ölpartikel.

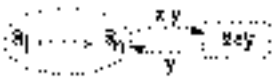
2. Weitere verteilte Algorithmen

(23) zeigt einen ganz typischen verteilten Algorithmus. Verschiedene Aktionen operieren jeweils auf einem Teil der verfügbaren Daten. Aktionen treten unabhängig voneinander ein, liegen logisch geordnet hintereinander oder stehen miteinander im Konflikt. Im Konfliktfall gibt es mehrere Verhaltensweisen.

Derartige verteilte Algorithmen gibt es für zahlreiche wohlbekannte Probleme; einige davon seien hier beispielhaft aufgeführt.

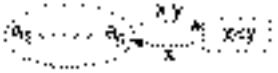
2.1. Verteiltes Finden der größten Zahl in einer endlichen Menge von Zahlen

Die algorithmische Idee besteht darin, aus der jeweils vorliegenden Zahlenmenge $\{a_1,...,a_n\}$ zwei beliebige Zahlen herauszunehmen, miteinander zu vergleichen, die kleinere von beiden zu vernichten und die größere zurückzulegen („die Großen fressen die Kleinen“):



(26)

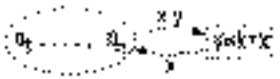
Symmetrisch dazu („die Kleinen fressen die Großen“) findet man die kleinste Zahl:



(27)

2.2. Verteiltes Sieb des Eratosthenes

Das bekannte „Sieb des Eratosthenes“ zum Herausfiltern der Primzahlen aus den ersten n natürlichen Zahlen hat einen ähnlichen „verteilten Kern“ („jede Zahl frißt ihre Vielfachen“):

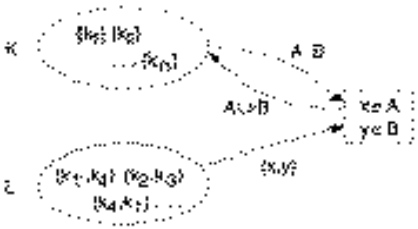


(28)

Dabei ist k eine („freie“) Variable, die durch irgendeine natürliche Zahl zu belegen ist. Dieses Beispiel zeigt nun besonders deutlich die verschiedenen Beziehungen unter den Aktionen. Bezeichnen wir mit a_{nm} jene Aktion die durch Belegen von x mit n und von y mit m entsteht, (wobei es ein k mit $y = k \cdot x$ geben muß), so ergibt sich beispielsweise: a_{39} tritt völlig unabhängig von allen anderen Aktionen ein. Das selbe gilt für $a_{5\ 25}$. Hingegen steht a_{26} in Konkurrenz zu a_{36}: Nach dem Eintritt einer der beiden Aktionen kann die andere nicht mehr eintreten. Interessant ist auch a_{48}: Nicht nur steht a_{48} in Konkurrenz zu a_{28} (über die „Vernichtung“ von „8“), sondern auch in Konkurrenz zu a_{24} (über die „Vernichtung“ von „4“).

2.3. Verteiltes Erkennen der Konnektivität eines ungerichteten Graphen

Hier unterscheiden wir zweckmäßigerweise zwei Datenmengen: Die Knoten und die Kanten des Graphen. Die Knoten seien zunächst als einelementige Mengen notiert. Dies ist zweckmäßig, weil der verteilte Algorithmus diese Mengen zu größeren, zusammenhängenden Mengen vereinigt:

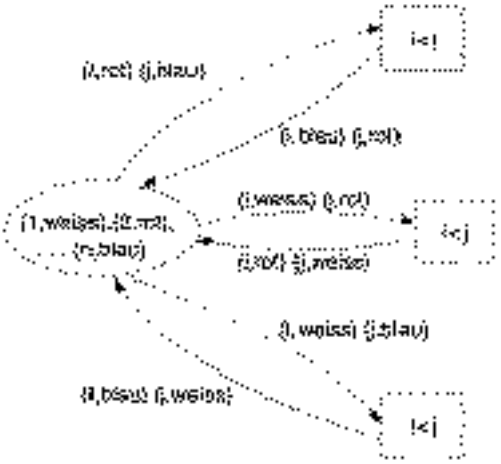


(29)

A und B sind hier Variablen über Mengen von Knoten; (x,y) beschreibt eine Kante von x nach y. Der zugrundeliegende Graph hängt offenbar genau dann zusammen, wenn zum Schluß alle Knoten in einer einzigen Menge in K versammelt sind.

2.4. Verteilte Lösung des „Dutch National Flag“-Problems

Bei diesem speziellen Sortierproblem (nach Dijkstra) sind n Positionen mit Farben belegt: Jede Position ist Anfangs „blau“ oder „rot“ oder „weiß“. Die Farbgebung der Positionen sind so umzutauschen, daß die Positionen mit niederen Indizes alle blau, die mit mittleren alle rot und die mit hohen Indizes alle weiß sind:



(30)

2.5. Weitere Beispiele

Zahlreiche Algorithmen zur Konstruktion spezieller Teilgraphen oder zur Erkennung spezieller Eigenschaften von Graphen haben einen verteilten Kern, so die Konstruktion minimaler spannender Bäume, kürzeste Wege oder konvexer Hüllen. Auch Algorithmen auf Zeichenketten können verteilt werden, beispielsweise das Erkennen einer längsten aufsteigenden Teilsequenz in einer Zahlenfolge.

3. Lokales Erkennen globaler Eigenschaften

Das *lokale* Erkennen einer globalen Eigenschaft ist ein zentraler Aspekt verteilten Rechnens. Beim verteilten Sortieren (23) können beispielsweise die Karten sortiert sein, ohne daß irgendeine Instanz dies erkennt. Spezielle — verteilte — Algorithmen sind erforderlich, damit eine lokale Instanz eine globale Eigenschaft kennenlernt. Solche Algorithmen basieren oft auf einem Verfahren, das bestätigte Informationen in einem autonomen, also von keiner Instanz überblickbaren Netzwerk erzeugt. Dieses Verfahren wird im folgenden in seiner allgemeinen Form vorgestellt.

Gegeben sei:
- eine Menge P von „Agenten“ (Prozessoren, Rechnerknoten, ...),
- zu jedem Agenten p eine Menge N(p) anderer Agenten, die „Nachbarn“ von p, wobei Nachbarschaft symmetrisch ist,
- ein ausgezeichneter „Initiator“-Agent p_0.

Ein solches System läßt sich als ungerichteter Graph mit einem „Anfangsknoten“ darstellen. Ein Beispiel ist:

(31)

Gesucht wird ein Verfahren, mit dem
- p_0 alle anderen Agenten informiert,
- nach einiger Zeit p_0 weiß, daß alle anderen Agenten informiert sind.

Dabei sollen folgende Nebenbedingungen erfüllt sein:

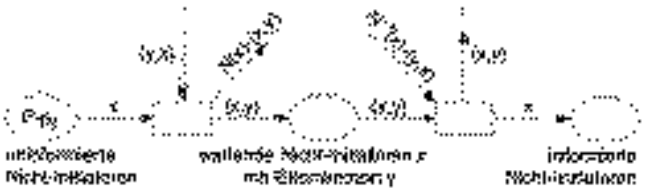
- Das Agentensystem hängt zusammen (als Graph aufgefaßt, sind alle Knoten über Kantenzüge miteinander verbunden).

– Mit Ausnahme des Initiators arbeitet jeder Agent nach den selben Regeln.
– Jeder Agent „kennt“ seine Nachbarn und kann nur mit ihnen kommunizieren. Andere Agenten bleiben ihm unbekannt.

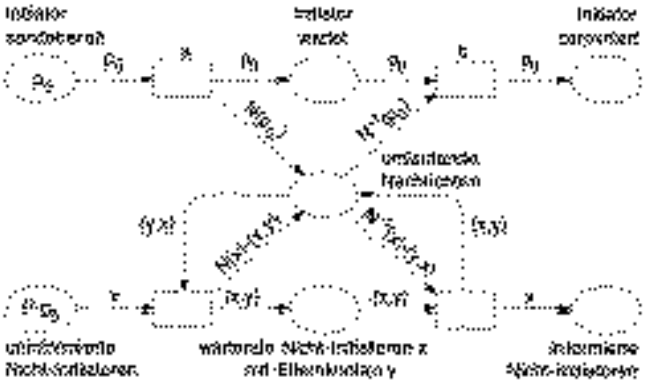
Wir beschreiben zunächst das Verfahren für den Initiator p_0. Vom Zustand „sendebereit“ sendet er an alle seine Nachbarn $N(p_0)$ eine Nachricht, wartet dann auf $N^{-1}(p_0)$, also auf Nachrichten *von* allen seinen Nachbarn und weiß anschließend, daß alle Agenten informiert sind:

(32)

Alle anderen Agenten verhalten sich nach dem selben Schema: Ein Agent x wählt unter den an ihn gerichteten Nachrichten eine aus, deren Absender y eine besondere Rolle spielt. Aus Gründen die erst später ersichtlich werden, nennen wir y dann den *Eltern - knoten* von x. Dann schickt x die Nachricht an alle Nachbarn, mit Ausnahme des Elternknotens, wartet auf Nachrichten von allen diesen Nachbarn und verschickt zuletzt die Nachricht auch an seine Elternknoten:


(33)

Der gesamte Algorithmus hat also folgende Form:



(34)

Der Nachweis der Korrektheit dieses Verfahrens sprengt den Umfang dieser Abhandlung. Schon die *Formulierung* seiner Korrektheit erfordert sorgfältig konstruierte Ausdrucksmittel (aus dem Gebiet der *temporalen Logik*), die hier nicht eingeführt werden sollen. Wir können jedoch *intuitiv* argumentieren und erkennen leicht, daß die nichtdeterministische Auswahl von Elternknoten einen *spannenden Baum* im System erzeugt. Er enthält p_0 als Wurzelknoten und schränkt die Kantenmenge so ein, daß von jedem Knoten zum Wurzelknoten genau ein Weg existiert. Beispiele für spannende Bäume für das System (31) sind:

(35)

und

(36)

Der Korrektheitsbeweis für dieses Verfahren argumentiert über solche spannenden Bäume. Insbesondere stellt sich heraus, daß die halbgeordneten Abläufe (vgl. (18) und (19)) eineindeutig den spannenden Bäumen entsprechen.

Mit diesem Algorithmus kann man nun konkrete Probleme lösen, indem man den Nachrichten gewisse Informationen aufprägt. Ein Beispiel sei die Sortiertheit von Karten, wenn jede Karte auf einem anderen Agenten liegt. Will der Initiator-Agent erkennen ob die Karten sortiert sind, so prägt er den Nachrichten an seine Nachbarn seinen Karteneintrag mit Positionsnummer auf. Hat ihm nach einiger Zeit jeder Nachbar einen Karteneintrag geschickt und sind alle mit seiner Karte korrekt geordnet, so steht für ihn die globale Ordnung fest. Sendet ein Nachbar ihm einen „Alarm“, so steht die Sortiertheit noch nicht fest.

Jeder einzelne Agent prägt den von ihm verschickten Nachrichten seinen Karteneintrag mit Positionsnummern auf, bis er einen „Alarm“ empfängt oder er beim Vergleich eines empfangenen Karteneintrags mit seiner eigenen Karte Nichtordnung erkennt. Dann prägt er allen abgehenden Nachrichten „Alarm“ auf.

4. ... etwas grundlegend Neues?

Hier gehen wir der Frage nach, in welchen grundlegenden Aspekten herkömmliche Algorithmen sich von verteilten Algorithmen unterscheiden.
Drei Aspekte heben wir hervor: das Erkennen von Endzuständen, die Rolle unbeschränkter Abläufe und die Funktionalität von Programmen.

Dazu betrachten wir zunächst typische Anwendungen verteilter Algorithmen und gehen dann allgemein und vertieft auf das (im Zusammenhang mit dem verteilten Sortieren schon kurz behandelte) Problem des Erkennens von Endzuständen ein. Danach fragen wir, warum die Semantik verteilter Algorithmen allgemein so schwer faßbar ist und diskutieren Versuche, die Ausdrucks-

kraft verteilter Algorithmen abstrakt zu charakterisieren. Abschließend fassen wir die Unterschiede zwischen herkömmlichen und verteilten Algorithmen noch einmal zusammen.

4.1. Anwendungen verteilter Algorithmen

Verteilte Algorithmen werden typischerweise für Systeme verwendet, die ihrer physikalischen oder logischen Struktur nach selbst verteilt sind, wo also ein zentrales Management technisch-organisatorisch ausgeschlossen ist. Beispiele sind verteilte Betriebs- und Datenbanksysteme, Rechnernetze, rechnerintegrierte Steuerungen technischer Prozesse und rechnergestützte Kommunikationssysteme aller Art, beispielsweise das weltweite digitale Telefonsystem. Zur Beschreibung vieler Aufgaben in solchen Systemen können herkömmliche Algorithmen nicht verwendet werden. Die Gründe dafür werden im weiteren deutlich.

Die im ersten und zweiten Kapitel vorgestellten verteilten Algorithmen sind so gesehen eher untypisch.

4.2. Das Erkennen von Endzuständen

Ein Ablauf eines herkömmlichen Algorithmus setzt sich schrittweise fort bis ein Zustand erreicht ist, in dem kein weiterer Schritt mehr anliegt. Bei dem anfangs in (1) - (4) beschriebenen Sortierverfahren ist dies bei n Karten nach spätestens n^2 Schritten der Fall. Ob ein konkret erreichter Zustand ein solcher „Endzustand“ ist oder nicht, wird als triviale Fragestellung angesehen.

Bei einem verteilten Algorithmus ist es hingegen überhaupt nicht einfach festzustellen, ob die gesamte Rechnung beendet ist. Rechnungen des verteilten Sortierens nach (23) enden ohnehin nicht: Es sind immer wieder U-Aktionen möglich, also Vergleiche korrekt geordneter Karteikarten. Als Variante von (23) kann man das gesamte Aktionschema U streichen und nur T-Aktionen zulassen. Dann endet jede Rechnung. Aber dennoch kann – anschaulich

formuliert – keine Karteikarte jemals sicher sein, daß ihre aktuelle Positionsnummer auch die endgültige Nummer ist, daß sie also beim weiteren „Herumwandern“ nur Karteikarten mit richtig geordneten Positionsnummern begegnen wird. Terminierung ist für verteilte Algorithmen ein Beispiel für eine globale Eigenschaft, die lokal erkannt werden muß. Im dritten Kapitel wurde gezeigt, daß dies unter sehr schwachen Annahmen immer möglich ist. Allerdings wurde auch klar, daß dafür ein gewisser Aufwand erforderlich ist.

4.3. Die Semantik herkömmlicher und verteilter Algorithmen

Ein herkömmlicher Algorithmus soll für jede erlaubte Eingabe eine endlich lange Rechnung erzeugen, also nach endlicher Zeit einen Endzustand erreichen. Im Endzustand liegen dann die Ausgabedaten vor. Wenn kein Endzustand erreicht wird, die Rechnung also unbeschränkt voranschreitet, wird ein Fehler angenommen: Der Algorithmus ist auf die Eingabe nicht anwendbar.

Mit diesen Grundannahmen kann die Bedeutung eines Programms P abstrakt als eine Funktion f_P beschrieben werden, die jedem Satz von Eingabedaten die zugehörigen Ausgabedaten zuordnet. Diese Funktion ist *partiell* (zu einem Satz a von Eingabewerten gibt es nicht notwendigerweise einen Wert $f_P(a)$), weil ja ggf. kein Endzustand erreicht wird.

Nun gibt es eine weit ausgebaute Theorie über derartige Funktionen. Sie unterscheidet insbesondere „berechenbare“ von „nicht berechenbaren“ Funktionen.

Das Konzept der berechenbaren Funktionen ist robust gegenüber einer Reihe technischer Verallgemeinerungen. Man kann beispielsweise die Möglichkeit vorsehen, daß ein Zustand nicht genau *einen* Folgezustand hat, sondern daß jeweils einer aus einer Reihe von Kandidaten beliebig ausgewählt wird. Im ersten Kapitel führte diese Idee beispielsweise von (1) nach (5). Auch kann man, wie in (10) - (13), in einem Schritt zugleich mehrere Ak-

tionen zulassen. Alle diese zusätzlichen Möglichkeiten führen nicht aus der Klasse der berechenbaren Funktionen heraus.

Beschreibt man die Bedeutung eines Programms als berechenbare Funktion, so ergibt sich sofort eine *Äquivalenz*: zwei Programme sind äquivalent wenn sie die gleiche Bedeutung haben (also die gleiche Funktion realisieren).

Die Semantik eines verteilten Algorithmus ist im allgemeinen nicht als partielle Funktion beschreibbar. Zwei Gründe dafür seien hervorgehoben: Zum einen sind nicht endende Rechnungen oftmals nicht fehlerhaft, sondern beabsichtigt, zum anderen verwischt die Beschränkung auf Ein-/Ausgabebeziehungen oft entscheidende Aspekte der Semantik eines verteilten Algorithmus. Die Verteiltheit und der relative Entstehungszeitpunkt von Ausgabedaten sind oft entscheidend für verteilte Algorithmen.

Im Algorithmus zur bestätigten Information aus Kapitel 3 sendet letztendlich jeder Agent an jeden Nachbarn genau einmal eine - inhaltsleere - Nachricht. Ausschließlich die relative Anordnung der Sende- und Empfangszeitpunkte (z.B. „ ... nach Empfang der Nachricht vom Elternknoten ...“) garantiert die Korrektheit des Verfahrens. *Synchronisation* ist also eine zentrale elementare Operation verteilter Algorithmen.

4.4. Abstrakte Charakterisierung der Ausdruckskraft herkömmlicher und verteilter Algorithmen

Hier soll ein besonderes spekulatives Argument erörtert werden: Die in 4.3. bereits erwähnten berechenbaren Funktionen werden gemeinhin als Rahmen für realisierbare Algorithmen angesehen. Hier soll nun diskutiert werden, ob verteilte Algorithmen keine Funktionen berechnen: Unbeschränkte Rechnungen bedeuten nichts Fehlerhaftes, und eine Ein/Ausgabebeziehung trifft nicht den Kern dessen, was ein verteilter Algorithmus leistet. Dennoch können diese Konzepte zum Teil mit herkömmlichen Mitteln konstruiert werden. Eine unbeschränkte Rechnung kann bei-

spielsweise als Limes ihrer endlichen Anfangsstücke charakterisiert werden, gegebenenfalls mit unbeschränkten Ein- und Ausgabeströmen, die ebenfalls Limiten ihrer endlichen Anfangsstücke sind.

Um alle berechenbaren Funktionen zu realisieren, reicht ein erstaunlich kleiner Prozessor (eine „universelle Turingmaschine“); die Realisierbarkeit eines konkreten Algorithmus hängt dann nur noch von der Größe des verfügbaren Speichers ab. Um jeden beliebigen Algorithmus zu realisieren, reicht also *ein* Prozessor und – als Limes endlicher Speicher – ein unendlich großer Speicher.

Verteilte Algorithmen arbeiten mit beliebig vielen Prozessoren, jeder mit lokalem Speicher. Eine „universelle“ Konstruktion (analog einer universellen Turingmaschine) benötigt als Limes unendlich viele Prozessor/ Speicher-Einheiten.

Es stellt sich nun die Frage, ob die beiden Limiten in jeder relevanten Hinsicht gleich sind. Eine Antwort darauf setzt allerdings eine schärfere Charakterisierung verteilter Algorithmen voraus, die derzeit nicht erkennbar ist.

4.5. Unterschiede zwischen herkömmlichen und verteilten Algorithmen

Die Titelfrage dieses Beitrags (in welcher Hinsicht sind verteilte Algorithmen etwas grundlegend Neues?) ist derzeit nicht abschließend beantwortbar. Dazu gibt es noch zu wenig Erfahrungen. Welche Konzepte wirklich zentral sind (z.B. Synchronisation als elementare Operation, kausale Ordnung statt zeitliche Ordnung), welche Abstraktions- und damit welche Äquivalenzbegriffe angemessen sind und mit welchen Techniken die Korrektheit eines Algorithmus besonders gut beweisbar ist, muß der Umgang mit verteilten Algorithmen erst noch zeigen.

Vielleicht ist die derzeitige Konstruktion und Verwendung verteilter Algorithmen immer noch zu sehr vom herkömmlichen, se-

quentiellen Denken durchzogen, und vielleicht ergeben sich zukünftig bisher unbeachtete Anwendungsfelder.

Die folgende Tabelle stellt noch einmal die in diesem Beitrag diskutierten Unterschiede zwischen herkömmlichen und verteilten Algorithmen zusammen:

	herkömmlich	verteilt
Elementaroperation	Lesen/Schreiben des Speichers	Synchronisation
Rolle interner Operationen	abstrahierbar	entscheidend
Erkennen eines Endzustandes	trivial	aufwendig, eigener Algorithmus nötig
Semantik eines Algorithmus	partielle Funktion f:IN ——-> OUT	?
Divergenz	Fehler	beabsichtigt
Äquivalenz von Algorithmen	kanonisch: gleiche Funktion	?
Ausdruckskraft beschreibbar durch	Turingmaschine: ein Prozessor mit unbeschränktem Speicher	unbeschränkt viele Prozessor/Speicher-Einheiten

Literatur

Banâtre J.-P., Coutant A., Le Metayer D.: A Parallel Machine for Multiset Transformation and its Programming Style. Future Generations Computer Systems 4, (1988)

Berry G., Bourdol G.: The Chemical Abstract Machine. TCS 96, (1982)

Chandy K. M., Misra J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading MA (1988)

Lamport L., Lynch N.: Distributed Computing: Models and Methods. Handbook of Theoretical Computer Science, Chapter 18 Elsevier Publ. Comp. (1990)

Manna Z., Pnueli A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag (1992)

Mattern F.: Verteilte Basisalgorithmen. Informatik-Fachberichte 226, Springer-Verlag (1989)

Raynal M.: Distributed Algorithms and Protocols. Wiley (1988)

Reisig W.: Petri Nets and Algebraic Specifications TCS, 80 (1991)

Tel G.: Topics in Distributed Algorithms. Cambridge International Series on Parallel Computation 1, Cambridge University Press (1991)

Wolfgang Reisig

Studium der Physik und Informatik in Karlsruhe und Bonn, dort Diplom 1974.
Wiss. Mitarbeiter an der Univ. Bonn 1974 - 1976.
Wiss. Mitarbeiter und Assistent an der RWTH Aachen 1976-1983, dort Promotion 1979.
Vertretungsprofessor an der Univ. Hamburg 1983/84.
Projektleiter bei der Gesellschaft für Mathematik und Datenverarbeitung 1984 - 1988, Habilitation an der Univ. Bonn 1987.
Professor an der TU München 1988 - 1993.
Professor an der Humboldt - Universität zu Berlin seit 1993.

Wichtigste Veröffentlichungen

Petrinetze- eine Einführung. Springer-Verlag, 1992
(italienisch 1984, englisch 1985, japanisch, polnisch 1988, chinesisch 1989)
Systementwurf mit Netzen. Springer-Verlag, 1985
(englisch 1992)
Das Verhalten verteilter Systeme. Oldenbourg-Verlag, 1987
Petri Nets and Abstract Data Types. Theoretical Computer Science 80, 1991

In der Reihe **Öffentliche Vorlesungen** sind erschienen:

1 *Volker Gerhardt*
Zur philosophischen Tradition der Humboldt-Universität

2 *Hasso Hofmann*
Die versprochene Menschenwürde

3 *Heinrich August Winkler*
Von Hitler zu Weimar
Die Arbeiterbewegung und das Scheitern der ersten deutschen Demokratie

4 *Michael Borgolte*
„Totale Geschichte“ des Mittelalters?
Das Beispiel der Stiftungen

5 *Wilfried Nippel*
Max Weber und die Althistorie seiner Zeit

6 *Heinz Schilling*
Am Anfang waren Luther, Loyola und Calvin - ein religionssoziologisch-entwicklungs-geschichtlicher Vergleich

7 *Hartmut Harnisch*
Adel und Großgrundbesitz im ostelbischen Preußen 1800 - 1914

15 *Gert-Joachim Glaeßner*
Demokratie nach dem Ende des Kommunismus

16 *Arndt Sorge*
Arbeit, Organisation und Arbeitsbeziehungen in Ostdeutschland

17 *Achim Leube*
Semnonen, Burgunden, Alamannen
Archäologische Beiträge zur germanischen Frühgeschichte

18 *Klaus-Peter Johne*
Von der Kolonenwirtschaft zum Kolonat
Ein römisches Abhängigkeitsverhältnis im Spiegel der Forschung

19 *Volker Gerhard*
Die Politik und das Leben

20 *Clemens Wurm*
Großbritannien, Frankreich und die westeuropäische Integration

21 *Jürgen Kunze*
Verbfeldstrukturen

22 *Winfried Schich*
Die Havel als Wasserstraße im Mittelalter:
Brücken, Dämme, Mühlen, Flutrinnen

23 *Herfried Münkler*
Zivilgesellschaft und Bürgertugend
Bedürfen demokratisch verfaßte Gemeinwesen einer sozio-moralischen Fundierung?

24 *Hildegard Maria Nickel*
Geschlechterverhältnis in der Wende
Individualisierung versus Solidarisierung

25 *Christine Windbichler*
Arbeitsrechtler und andere Laien
in der Baugrube des Gesellschaftsrechts
Rechtsanwendung und Rechtsfortbildung

26 *Ludmila Thomas*
Rußland im Jahre 1900
Die Gesellschaft vor der Revolution

Es erscheinen demnächst:

28 *Ernst Osterkamp*
Die Seele des historischen Subjekts

29 *Rüdiger Steinlein*
Märchen als poetische Erziehungsform
Zum kinderliterarischen Status der Grimmschen „Kinder- und Hausmärchen“