Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 79 (1988)

Heft: 7

Artikel: Datenflussrechner : Konzepte und Anwendungen

Autor: Bührer, R.

DOI: https://doi.org/10.5169/seals-904014

Datenflussrechner – Konzepte und Anwendungen

R. Bührer

Zahlreiche Forschungsprojekte befassen sich zurzeit mit der Frage, wieweit sich Datenflussrechnerkonzepte für polyvalente Hochleistungsrechner eignen. Anhand einer Einführung in die grundlegenden Funktionsprinzipien, in die möglichen Architekturvarianten und Implementationen sowie in die Eigenschaften zugehöriger Programmiersprachen werden die vielversprechenden Eigenschaften und die noch offenen Fragen des Konzepts erklärt.

De nombreux projets de recherche essaient actuellement de montrer dans quelle mesure les concepts d'ordinateurs à circulation de données (dataflow) peuvent être appliqués dans la réalisation d'ordinateurs polyvalents et de haute performance. Au moyen d'une introduction aux principes fondamentaux de fonctionnement, aux différentes variantes d'architecture et aux implémentations possibles, ainsi qu'aux propriétés des langages de programmation utilisés, les propriétés prometteuses du concept et les questions encore ouvertes sont discutées.

Adresse des Autors
Dr. *Richard Bührer*, Institut für Elektronik, ETH-Zentrum, 8092 Zürich.

Obwohl das Prinzip der Datenflussrechner noch keine kommerzielle Reife erreicht hat und teilweise noch umstritten ist [1], zeichnet sich anhand verschiedener Forschungsarbeiten wie [2; 3] usw. immer deutlicher ab, dass diese Computerarchitektur gegenüber zahlreichen anderen Parallelrechnervarianten grosse Vorteile aufweist [4]. Ein erstes wichtiges Argument ist die Tatsache, dass die *funktionale Programmiertechnik*, welche den Applikationsprogrammierer weitgehend davon entlastet, sich mit der expliziten Parallelisierung seiner Programme zu beschäftigen, für die Programmierung eines Datenflussrechners eingesetzt werden kann. Dieser Aspekt ist deshalb von grosser Bedeutung, weil nicht nur in vielen technisch-wissenschaftlichen, sondern auch in industriellen Anwendungen mittel- und langfristige Problemstellungen anstehen werden, welche nur noch durch hochparallele Rechnersysteme bearbeitet werden können. Dabei ist zu beachten, dass bei allgemeinen Anwendungen – im Gegensatz zu eher spezifischen Anwendungen (Aerodynamik, Bildverarbeitung usw.), bei denen grosse, regulär strukturierte Datenmengen verarbeitet werden – der Applikationsprogrammierer kaum mehr in der Lage ist, die im Applikationsprogramm vorhandene Parallelität zu überblicken und bei der Programmierung entsprechend umzusetzen. Diese Aufgabe muss von einem leistungsfähigen Compiler übernommen und zur Programmausführungszeit durch ein entsprechendes Laufzeitsystem unterstützt werden.

Eine weitere wichtige Eigenschaft der Datenflussrechner besteht darin, dass sie verhältnismässig einfach erweitert werden können, indem – der geforderten zusätzlichen Leistung entsprechend – weitere Prozessoren in ein System eingebaut werden können, ohne dass am Applikationsprogramm etwas geändert werden muss. Dies ist eine wichtige Grundvoraussetzung zur Realisierung von Rechnersystemen mit hoher und höchster Leistung.

Das Datenflussprinzip

Das grundlegende Prinzip aller Datenflussrechner beruht auf der Vorschrift, der sogenannten *Feuerungsregel* (Firing Rule), dass eine Rechnerinstruktion dann und nur dann zur Ausführung kommt, wenn alle zugehörigen Datenwerte (die Argumente der Instruktion) vorhanden, d. h. unmittelbar verfügbar, sind. Die Ausführung eines Programms wird somit nicht wie bei einem konventionellen Rechner durch einen Programmzähler gesteuert, vielmehr sind es Datenverfügbarkeiten, die die Prozessoraktivitäten beeinflussen bzw. vorschreiben (Data Driven Program Execution).

Das Datenflussprinzip an sich ist nicht neu. Optimierende Compiler verwenden diese Technik seit langer Zeit zur Zuteilung von Speicherzellen und Akkumulatoren mit dem Ziel, die vorhandenen Rechnerressourcen möglichst optimal und somit auch leistungssteigernd auszunützen. Hardwaremässige Verwendung findet dieses Prinzip z. B. schon in den Rechneranlagen IBM 360/91 und CDC 6600 [5] zur Optimierung des Einsatzes von arithmetischen Einheiten. Eine ganze Rechnerarchitektur auf dieses Konzept abzustützen ist hingegen neu. Eine kurze Übersicht über einige Begriffe, welche zum Verständnis wichtig sind, wird in Tabelle I gegeben.

Synchrone und asynchrone Systeme

Die verschiedenen Varianten der Datenflussarchitekturen, welche in jüngster Zeit mit Erfolg untersucht und implementiert worden sind, können in synchrone und asynchrone Sy-

Begriffe

Datenflussprogramme werden in Form von *Datenflussgraphen* (Dataflow Graphs) dargestellt (Fig. 1). Sie bestehen aus *Knoten* (Nodes), welche eine Operationseinheit, d.h. eine Rechneroperation darstellen, und verbindenden Kanten (Arcs), welche die zugehörige *Datenabhängigkeit* (Data Dependency) bzw. den Informationsfluss darstellen. Daten werden paketartig in Form sogenannter (Data) Tokens entlang dieser Kanten verschoben. Datenpakete, welche einzelnen Knoten zugeführt werden, werden als Input Token jene, die von einzelnen Knoten wegführen (die Resultate der Instruktion also), als Output Tokens bezeichnet. Die Bedingung zur Abarbeitung einer Instruktion, die Feuerungsregel, wird im Datenflussgraphen explizit dargestellt.

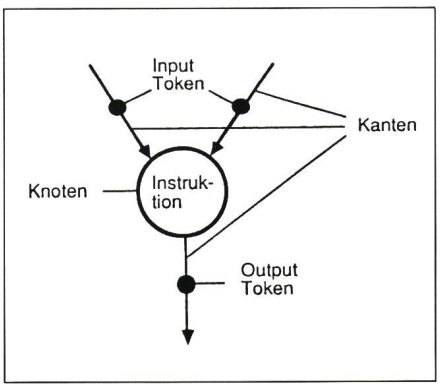
Als *Iteration* wird ein Durchgang (Instanz) einer Schleife (Loop) bezeichnet. Wird eine Programmsequenz mehrfach von verschiedenen Programmstellen aus aufgerufen (z.B. Funktionen oder Prozeduren), so werden verschiedene *Kontexte* (Contexts) des gleichen Unterprogrammes geschaffen.

Monadische Rechnerinstruktionen verarbeiten einen einzigen Operanden, *dyadische* Instruktionen zwei Operanden.

Als *Tag* wird jene Zusatzinformation eines Datentokens bezeichnet, welche darüber Auskunft gibt, wo die zugehörige Instruktion zu finden ist und in welchem Zusammenhang (z. B. Kontekt) diese Instruktion ausgeführt wird.

Tabelle I

steme aufgeteilt werden. Das Konzept der *synchronen* Datenflusssysteme geht von der Überlegung aus, für jeden Knoten des Datenflussgraphen eine der entsprechenden Operation entsprechende Rechnerkomponente bereitzustellen. Diese je nach Applikationsprogramm zum Teil vielfach replizierten, einfachen Rechenelemente sind in der Lage, *eine* bestimmte (Maschinen-)Instruktion auszuführen. Die Instruktion wird, gemäss der Feuerungsregel, beim Eintreffen des entsprechenden Tokens ausgelöst. Die einzelnen Rechenelemente werden durch ein statisches oder dynamisch konfigurierbares Netzwerk verbunden. Der aus Knoten (Rechenelementen) und Kanten (Verbindungen) bestehende Datenflussgraph wird also hardwaremässig implementiert. Die einzelnen, das Netzwerk durchwandernden Tokens bestehen dabei aus den effektiv zu bearbeitenden Datenwerten.



Figur 1 Datenflussgraph
Bezeichnungen

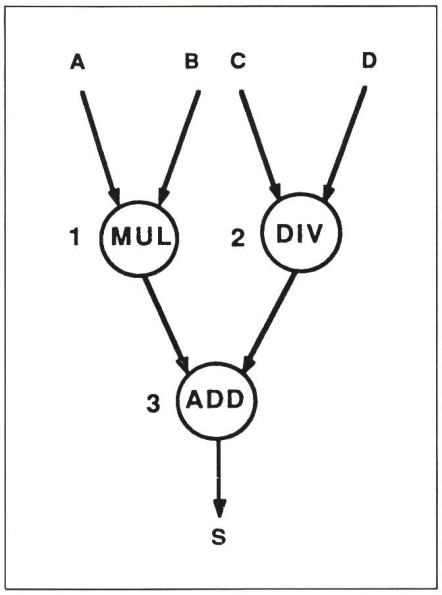
Eine Untervariante dieses Konzepts besteht darin, dass der Instruktionstyp der Rechenelemente vor der Betriebsaufnahme, d.h. vor der Abarbeitung des Applikationsprogramms, programmiert werden kann. Dadurch wird es möglich, eine gewisse Flexibilität bezüglich implementierbarer Programme (d.h. Algorithmen) zu erzielen. Synchrone Datenflussrechner eignen sich insbesondere für jene Anwendungen, welche eine sich vielfach wiederholende Bearbeitung unterschiedlicher Daten umfassen (vgl. z. B. [6]).

Bei den *asynchronen* oder frei programmierbaren Datenfluss-Multiprozessoren verfügt jeder einzelne Prozessor über ein einzelnes Rechenelement, welches jedoch (wie bei konventionellen Von-Neumann-Rechnern [7]) über einen mehr oder weniger umfassenden Instruktionssatz verfügt. Der Datenflussgraph wird hier in tabellarischer Form in einem Programmspeicher abgelegt. Die Tokens müssen zusätzlich zu den Datenwerten eine entsprechende Tabellenadresse enthalten, aus der die zugehörige Instruktion ausgelesen, im Rechenelement dekodiert und anschliessend ausgeführt werden kann. Da die in die erwähnte Tabellenform umgewandelten Applikationsprogramme auf klassische Art in die Programmspeicher eingelesen werden können, stellen asynchrone Datenflussrechner bezüglich Programmierung ein flexibles Rechnerkonzept dar.

Im weiteren wird ausschliesslich auf die asynchronen Datenflusskonzepte eingegangen, welche ihrerseits nochmals in die *statischen* und *dynamischen* Konzepte unterteilt werden. Als Einführung in die beiden Varianten wird die Abarbeitung der Gleichung

$$S = A \cdot B + C/D \qquad (1)$$

als einfaches Beispiel verwendet. Der zugehörige Datenflussgraph ist in Figur 2 dargestellt.



Figur 2 Beispiel eines Datenflussgraphen
Funktion: S = A · B + C/D

Statisches Datenflusskonzept

Das statische Datenflusskonzept beruht darauf, dass die Daten-Tokens vor der Verarbeitung auf vorgegebenen Plätzen im Instruktionsspeicher zwischengespeichert werden (Fig. 3). Sobald in einer monadischen Instruktion *ein* Token verfügbar ist, kann sie nach der Feuerungsregel abgearbeitet, d.h. gefeuert, werden. Bei dyadischen Instruktionen ist der entsprechende Vermerk «#: Anzahl benötigter Tokens» in der codierten Instruktion (Fig. 3) auf zwei gesetzt. Die Instruktion kommt somit erst nach Abspeicherung beider Argumente (Tokens) zur Ausführung. Die Abspeicherung des Datenflussgraphen erfolgt, wie erwähnt, in Form einer Tabelle. Das entsprechende Beispiel geht aus Figur 3 hervor. Der Aufbau eines Tokens wird in Figur 4 dargestellt.

Die Abspeicherung der Token-Werte in die vorgegebenen Plätze im Instruktionsspeicher führt nun aber in jenen Fällen zu Problemen, in denen

	Op-Code	Argument 1	Argument 2	#	Zieladresse 1	Zieladresse 2
1	•	a	b	2	3.l	--
2	/	c	d	2	3.r	--
3	+	a • b	c / d	2	...	...

Figur 3 Tabellarischer Datenflussgraph beim statischen Datenflusskonzept

\# Anzahl benötigter Tokens

l, r Index für linke bzw. rechte Kante des Datenflussgraphen

Programmsequenzen iterativ oder rekursiv abgearbeitet werden.

Beispiel: (2)

```
FOR i  := 0 TO n DO
   s[i]  := a[i] · b[i] + c[i]/d[i]
END;
```

Da die Ankunft der Token $a[i] \ldots d[i]$ zeitlich nicht bestimmbar ist, können - ohne entsprechende Vorkehrungen - die Argumentenplätze im Instruktionsspeicher fälschlicherweise überschrieben werden, da mehrere Tokens der gleichen Instruktion zugeordnet sind.

Zieladresse	l/r	Datenwert

Figur 4 Aufbau eines Tokens beim statischen Datenflusskonzept

l/r Index für linke bzw. rechte Kante des Datenflussgraphen

Die praktische Lösung dieses Problems besteht darin, dass durch die Einführung sogenannter Kontroll-Tokens dieser Stau verhindert wird. Wie Figur 5 entnommen werden kann, wird jede Instruktion dabei erst dann feuerbar, wenn die datenempfangende Instruktion signalisiert hat, dass die vorherige Instanz abgearbeitet ist, die entsprechenden Argumentenplätze also frei geworden sind. Im Instruktionsspeicher muss somit zusätzlich Platz für die Kontroll-Tokens vorgesehen werden. Die Feuerungsregel für Instruktion 1 in Figur 5, beispielsweise, lautet daher

Execute Instr. 1 if Tokens

a[i] & b[i] & s'[i-1] (3)
are available,

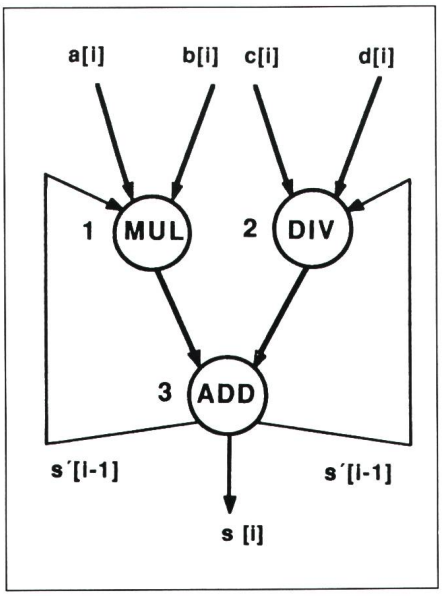
wobei & die UND-Operation bezeichnet. Die erste Iteration benötigt kein Kontroll-Token.

Die Detektion feuerbarer Instruktionen erfolgt anhand einer logischen Vergleichsoperation (2) beim Abspeichern der Daten-Tokens auf die entsprechenden Plätze des Programmspeichers. Ein vollständiger statischer Datenflussrechner nach [18] setzt sich gemäss Figur 6 aus fünf Blöcken zusammen:

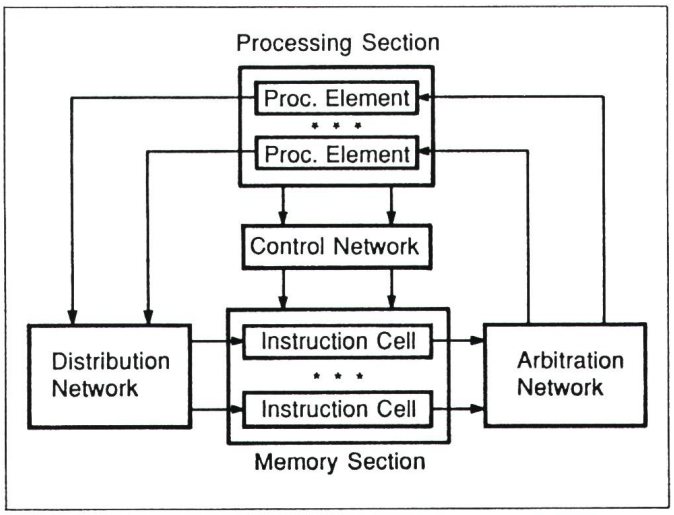
1. aus der Speichereinheit (Memory Section), welche aus einzelnen Instruktionszellen aufgebaut ist, in denen gemäss Figur 3 sowohl die Operationscodes wie auch die Argumente und Zieladressen abgespeichert sind;
2. aus der Ausführungseinheit (Processing Section), welche mittels spezieller Recheneinheiten (Processing Elements) die eigentlichen Datenoperationen durchführt;
3. aus einem Zuweisungsnetzwerk (Arbitration Network), welches sogenannte Instruktionspakete von der Speichereinheit zur Ausführungseinheit übermittelt, die (unter anderem) aus Operationscode, Argumenten und Zieladressen bestehen;
4. aus einem Kontrollnetzwerk (Control Network) zur Übermittlung der obenerwähnten Kontrollinformation zwischen Ausführungseinheit und Speichereinheit und
5. aus einem Verteilnetzwerk (Distribution Network) zur Übertragung von Datenpaketen von der Ausführungseinheit zur Speichereinheit. Eine einzelne Instruktion gelangt dann zur Ausführung, wenn in der Instruktionszelle die nötigen Argumente abgelegt sind und die zugehörige Kontrollinformation eintrifft. Ausführliche Beschreibungen solcher Konzepte findet man z.B. in [8].

Dynamisches Datenflusskonzept

Wie im vergangenen Kapitel erläutert, werden in der statischen Datenflussvariante die Argumente direkt in



Figur 5 Einführung von Kontrolltokens beim statischen Datenflusskonzept



Figur 6 Beispiel eines statischen Datenflussrechners

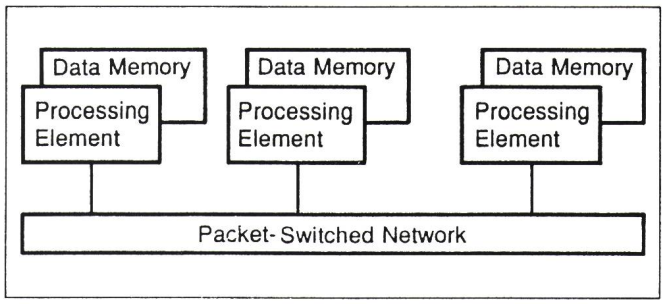
den Instruktionsspeichern zwischengelagert, bevor eine entsprechende Verarbeitung stattfindet. Der grosse Nachteil dieses Prinzips liegt somit darin, dass eine bestimmte Programmsequenz jeweils nur in einem Kontext oder nur für eine einzige Iteration benützt werden kann. Ein parallelisierbarer Loop beispielsweise kann also nur mittels Expansion (für jede Iteration wird eine Kopie des entsprechenden Loop-Körpers im Programmspeicher abgelegt) effektiv parallel abgearbeitet werden. Zudem muss in jeder Instruktion Platz für die Daten-Tokens reserviert werden. Dies führt im allgemeinen Fall zu erheblich grösseren Programmspeicherbelegungen als bei konventionellen Rechnern.

Das dynamische Datenflussprinzip will nun (unter anderem) diesen Nachteil eliminieren, indem die Daten-Tokens nicht mehr im Programmspeicher abgelegt, sondern in einem eigenständigen Token-Speicher ausgelagert werden. Dieser Speicher ist derart ausgelegt, das er einerseits eine selbständige Belegungsverwaltung besitzt und zudem in der Lage ist, anhand der Token-Art festzustellen (Token Matching), ob eine Instruktion gefeuert werden kann oder nicht. Die Daten-Tokens müssen dazu mit Zusatzinformationen versehen werden, welche darüber Auskunft geben, ob nur ein Token für eine Instruktion benötigt oder ob ein Tokenpaar verarbeitet wird. Im zweiten Fall wird eine Tag-Information interpretiert, welche einen eindeutigen Vergleich zur Bestimmung von Token-Paaren erlaubt.

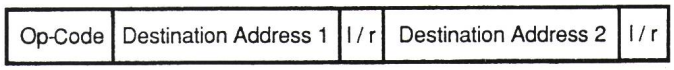
Das Konzept wird im folgenden anhand des Tagged-Token-Datenflussrechners von *Arvind* [3] erläutert.

Wie aus Figur 7 ersichtlich ist, besteht ein derartiger Parallelrechner aus einer Vielzahl von Rechenelementen (Processing Elements) und Datenspeicherblöcken (Data Memories), welche durch ein leistungsfähiges (Packet-Switched) Netzwerk verbunden sind. Die eigentlichen Programmspeicher sind gemäss Figur 10 in den jeweiligen Rechenelementen implementiert.

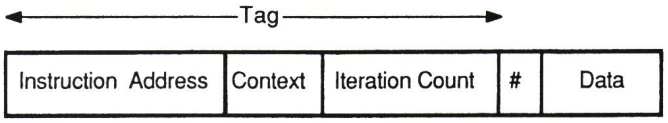
Figur 8 und 9 beschreiben das Instruktionsformat bzw. den Token-Aufbau der Tagged-Token-Maschine. Der Instruktionsaufbau ist gegenüber der Variante von Figur 3 einfacher, da keine Plätze für die Ablage der Datenwerte nötig sind. Die Tokens (Fig. 9) sind hingegen mit der obenerwähnten Zusatzinformation (Tag Information) versehen, welche darüber Auskunft gibt, in welchem Kontext oder in welcher Iteration die mittels Instruktionsadresse referenzierte Instruktion verwendet wird.



Figur 7
Datenfluss-Multiprozessor
Processing-Element: Rechenelement zur Abarbeitung des Datenflussgraphen
Data Memory: Datenspeicherblock zur Ablage von Datenstrukturen
Packet-Switched Network: Verbindungsnetzwerk zur Übertragung von Datentoken



Figur 8
Instruktionsformat des Tagged-Token-Datenflussrechners
nach [3]



Figur 9
Tokenformat
Anzahl benötigter Argumente

Daten-Tokens, welche vom Netzwerk oder über die interne Rückführung zur *Waiting-Matching Section* (W/M Section) gelangen (Fig. 10), werden anhand der internen Information «#: number of tokens to match» (Fig. 9) kontrolliert, ob ein Partner-Token zur Instruktionsabarbeitung benötigt wird (dyadische Operationen: # = 2) oder ob eine monadische Operation mit diesem Token vorgesehen ist (# = 1). Im ersten Fall wird in der W/M Section mittels spezieller Vergleichslogik kontrolliert, ob das Partner-Token, welches eine identische Tag-Information (Fig. 9) aufweist, bereits vorliegt. Trifft dies zu, so werden beide Tokens zur *Instruction-Fetch Unit* weitergeleitet. Liegt das Partner-Token noch nicht vor, wird das angekommene Token in der W/M Section abgelegt. Tokens von monadischen Instruktionen werden in allen Fällen sofort ohne Vergleichsoperation weitergeleitet. Die Instruction-Fetch Unit liest (d.h. kopiert) die vollständige Information aus der zugehörigen Instruktionsspeicherzelle und leitet die Teilinformation gemäss Figur 10 zu den weiteren Funktionsblöcken der Recheneinheit weiter. In der *Output Section* werden vom berechneten Resultat so viele Output Tokens gebildet, wie Destinationsadressen in der Instruktion vorgegeben sind. Die Tokens werden anhand der vorgegebenen Zieladresse und der entsprechenden Belegungstabellen entweder zurück an den Eingang des Rechenelementes übermittelt oder, sofern die Adresse der Zielinstruktion sich in einem anderen Rechenelement befindet, in das Netzwerk geleitet.

Was den physikalischen Aufbau betrifft, liegt der wesentliche Unterschied gegenüber der statischen Datenflussvariante somit darin, dass die Instruktionsspeicher wesentlich kompakter aufgebaut werden können, da die W/M Section die Zwischenspeicherung der Tokens übernimmt. Im weiteren wird der interne Datenverkehr in der Recheneinheit signifikant verringert, da die Übermittlung von Kontroll-Tokens vollständig entfällt. Ein gewisser Mehraufwand entsteht beim dynamischen Rechner hingegen bei der Übermittlung der Daten-Tokens, da diese infolge der zusätzlichen Tag-Informationen etwas grösser ausfal-

len. Der zentrale Vorteil liegt hingegen darin, dass, wie oben erwähnt, mehrfach benützbare (reentrant) Programme einfach realisiert werden können.

Datenstrukturen

Auf die spezielle Handhabung von (verteilten) Datenstrukturen in Datenflussrechnern wird in diesem Rahmen nur kurz anhand der von *Arvind* vorgeschlagenen I-Struktur (I-Structure) eingegangen. Die begrenzte Grösse der W/M Section sowie die zwischen den Funktionsblöcken implementierten Zwischenbuffer lassen nicht zu, dass sämtliche Daten ausschliesslich als Tokens realisiert werden. Es muss daher ein spezifischer Datenspeicher zur Ablage von Arrays, Matrizen usw. vorgesehen werden. Die von *Arvind* vorgeschlagene Lösung basiert auf einem gemäss Figur 7 verteilten Speicher, auf dessen einzelnen Blöcken die Datenstrukturen nach entsprechenden Zuweisungsalgorithmen verteilt werden. Die Semantik dieser I-Struktur beruht darauf, dass ein einzelnes Datenwort nur einmal geschrieben, hingegen beliebig oft gelesen werden darf. Das Abspeichern von Daten erfolgt mittels spezieller *I-Struktur-Write Tokens*, welche statt einer Instruktionsadresse (Fig. 9) eine I-Struktur-Adresse beinhalten. Nach jedem Einschreiben des individuellen Datenwertes wird die entsprechende Speicherzelle mit einem *Valid-Bit* gekennzeichnet. Wird nun in einer Programmsequenz ein I-Struktur-Datum referenziert, wird ein entsprechendes *I-Structure-Read Token* generiert und zum entsprechenden I-Struktur-Block übertragen. Bei verfügbarem Datum wird eine Kopie des fraglichen Datenwertes an jene Instruktion zurückgeschickt, welche im I-Structure-Read Token spezifiziert worden ist. Ist hingegen das Valid-Bit nicht gesetzt, wird der Request so lange sistiert, bis das entsprechende Datum mittels I-Structure-Write Tokens in der I-Struktur abgelegt worden ist. Auf diesem Datenkonzept lassen sich höhere Datenstrukturen aller Art aufbauen.

Vorteile des Datenflussprinzips

1. Ausnützung der Parallelität von Anwenderprogrammen

Die meisten der heute kommerziell verfügbaren Hochleistungsrechner erreichen üblicherweise dann ihre maximale Leistung, wenn die entsprechenden Programme ein hohes Mass an Vektorisierbarkeit aufweisen. Es gilt daher, mittels optimierender (z.B. Fortran-) Compiler die Anwendungen so zu programmieren, dass diese Anforderung möglichst gut erfüllt ist. Abgesehen von speziellen Applikationen (wie Aerodynamik, Seismik, Luftraumüberwachung usw.), wo repliziert Berechnungen vieler gleicher Objekte (mit unterschiedlichen Daten bzw. Randwerten) vorliegen können, weisen jedoch viele allgemeine Programme ein äusserst kleines Mass an Vektorisierbarkeit auf, und die mittlere Leistung der erwähnten Rechnersysteme fällt drastisch zusammen. Die Datenflussrechner sind diesbezüglich nahezu unempfindlich, da, dank der funktionalen Programmierweise, jegliche Art von Parallelität eines Programms automatisch und umfassend durch den Compiler aufgedeckt und durch die Maschine ausgenützt werden kann.

2. Elimination des Memory-Latency-Problems

Ein wichtiges Problem bei konventionellen Multiprozessoren, sowohl bei gemeinsamen wie auch bei verteilten Speichern, wird als *Memory Latency* (Speicher-Zugriffsverzögerung) bezeichnet. Man versteht darunter die Tatsache, dass, wann immer bei der (von-Neumann-artigen) Ausführung einer Instruktion eine Referenz auf einen Datenwert ausgeführt wird (Daten-Fetch), mit Wartezeiten gerechnet werden muss. Im Fall des gemeinsamen Speichers ist dies auf Speicherzugriffskollisionen zurückzuführen, da mehrere Rechner zur gleichen Zeit auf die gleiche Speicherbank zugreifen oder aber der Zugriff über einen gemeinsamen Bus erfolgt. Bei verteilten Speichern wird die Wartezeit in vielen Fällen noch länger, da ein Zugriffspfad vom Prozessor zur entsprechenden nichtlokalen Speicherzelle etabliert werden muss. In beiden Fällen wird der entsprechende Prozessor in einen mehr oder weniger langen Wartezustand versetzt.

In Datenflussrechnern mit verteilten Speichern ist dieses Problem entscheidend entschärft, da die Beschaffung der Daten von der zugehörigen Verarbeitung entkoppelt ist und die entsprechende Latenzzeit durch die Abarbeitung anderer ausführbarer Instruktionen überbrückt werden kann. Dies ermöglicht eine wesentlich bessere Prozessorausnützung und somit eine höhere spezifische Rechenleistung (Anzahl Instruktionen pro Zeiteinheit und Prozessor).

3. Synchronisationsaspekte

Die durch die Datenverfügbarkeit gesteuerte Programmabarbeitung des Datenflussrechners eliminiert das in vielen Parallelrechnerkonzepten nur schwerfällig lösbare Problem der Datensynchronisation zwischen kommunizierenden Unterprogrammen. Komplizierte und verhältnismässig rechenintensive Synchronisationsmechanismen wie Semaphoren, Rendez-vous, Fork-and-Joins usw. entfallen, da die zur Synchronisation nötigen Operationen (z.B. die W/M Section in Figur 10) hardwaremässig implementiert werden können. Die bekannten Lese-Warte-Sequenzen (ein konsumierender Prozessor will ein Datum lesen, bevor es vom entsprechenden produzierenden Prozessor generiert wurde) entfallen vollständig. Bei verteilten Datenspeichern ermöglicht die Einführung expliziter Data-Fetch- bzw. Data-Store-Operationen (im Kapitel «Datenstrukturen»), Prozessoren auch im Fall von Referenzen auf nichtlokale Daten für andere Operationen freizubehalten. Kontextumschaltungen sind vollständig zeitverzugslos, weil konzeptionell jede individuelle Instruktion ihren eigenen Kontext aufweist.

4. Erweiterbarkeit

Wie im folgenden Kapitel gezeigt wird, sollte sich der Programmierer von Parallelrechnersystemen für allgemeine Anwendungen nicht um den Grad der durch die Maschine verarbeitbaren Parallelität kümmern müssen. Diese Aufgaben sollen durch einen entsprechenden Compiler übernommen werden, der den vollständigen parallelisierten Programmgraphen entwickelt. Die zur Laufzeit anfallenden Programm- bzw. Lastzuweisungen werden durch entsprechende Komponenten eines (ebenfalls als Datenflussprogramm implementierten) Betriebssystemkerns übernommen. Erweiterungen des Rechnersystems durch Implementation zusätzlicher Prozessoren bedingen daher keine Änderungen des Anwenderprogramms. Das Laufzeitsystem kann, mit Ausnahme spezieller Systemkonfigurationstabellen, unverändert übernommen werden.

5. Realisierbarkeit

Ein wichtiger Punkt bei der Abschätzung der Qualität eines Rechner-

Einzahlungsschein / Recepissé / Ricevuta

Einzahlung für / Versement pour / Versamento per

Schweizerischer Elektrotechnischer Verein SEV
8034 Zürich

Konto Compte Conto 80-6133-2

Fr. c.

Einbezahlt von / Versé par / Versato da

Die Annahmestelle
L'office de dépôt
L'ufficio d'accettazione

Vor der Einzahlung abzutrennen / A détacher avant le versement / Da staccare prima del versamento

Einzahlung Giro

Einzahlung für / Versement pour / Versamento per

Schweizerischer Elektrotechnischer Verein SEV
8034 Zürich

Konto Compte Conto 80-6133-2

Fr. c.

Mitteilungen / Communications / Comunicazioni

639002 No. 6764

Journée d'information de l'ASE 31.5.1988, Montreux

Giro aus Konto
Virement du compte
Girata dal conto

Einbezahlt von / Versé par / Versato da

Versement Virement

Versamento Girata

SR 3.88

PTT 441.02 (5 9/10 x 4 1/6") OCR 90

8000613322

8000613322

Association Suisse des Electriciens (ASE)

Inscription

Prière de retourner ce bulletin d'inscription avant le **17 mai 1988** à l'Association Suisse des Electriciens, Services administratifs, case postale, 8034 Zurich

Journée d'information de l'ASE

Les nouvelles recommandations de l'ASE pour les installations de protection contre la foudre

Montreux – Mardi, 31 mai 1988, Casino de Montreux

Veuillez compléter à la machine ou en lettres d'imprimerie.

Inscription No. 6764 **639002**

Nom	Prénom	Adresse	Carte de participation:			
			Non membre Fr. 250.—	Membre de l'ASE et de l'ASMFA Fr. 150.—	Etudiants/Apprentis: Non membre Fr. 90.—	Etudiants/Apprentis: Membre de l'ASE et de l'ASMFA Fr. 70.—

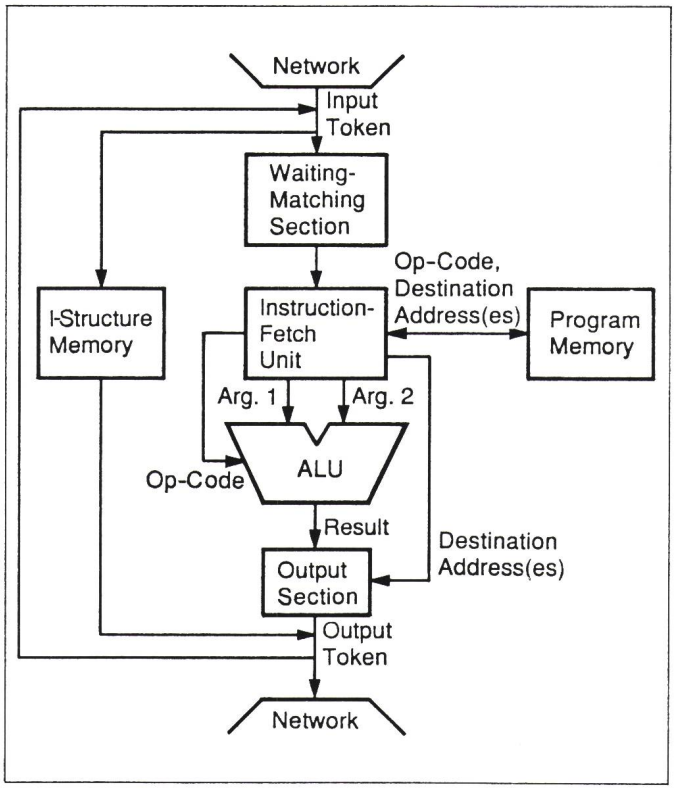
Paiements au moyen du bulletin de versement ci-joint s'il vous plaît.
Délai pour l'inscription: le **17 mai 1988** au plus tard.
Prière d'indiquer sur toute correspondance et bulletin de versement les deux numéros de référence ci-dessus.

Adresse pour l'envoi des documents:

Téléphone-No

Date:

Signature:



Figur 10
Rechenelement (Processing Element) des Tagged-Token-Datenflussrechners
nach [3]

konzepts betrifft die Realisierungsmöglichkeiten einer entsprechenden Rechner*hardware*. Eine diesbezügliche Beurteilung der verschiedenen Komponenten eines dynamischen Datenflussrechners zeigt, dass zu deren Implementation weder exotische Technologien noch unverhältnismässig komplexe Bausteine nötig sind. Die Verfügbarkeit von Halbleiter-Speicherelementen grosser Kapazität sowie die vielfältigen Möglichkeiten, leistungsfähige Netzwerke für Paketvermittlung zu realisieren [9], unterstreichen das Potential dieser Rechnerarchitektur.

Programmierung von Datenflussrechnern

Wie in allen parallelen Rechnersystemen spielt die Frage der Programmierung auch bei Datenflussrechnern eine ausserordentlich wichtige Rolle. Stichworte wie *Mächtigkeit, Modularität, Benutzerakzeptanz usw.* müssen entsprechend berücksichtigt werden. Ein grosses Gewicht muss auch den marktstrategischen Aspekten beigemessen werden, da viele Endbenützer nur dann Rechnersysteme akzeptieren werden, welche in einer neuen Sprache zu programmieren sind, wenn eine signifikante Erhöhung der Computerrechenleistung garantiert wird oder bedeutende Vorteile bei der Erstellung korrekter, den Spezifikationen entsprechender Programme geboten werden.

Da eine umfassende Darstellung der Eigenschaften von Programmiersprachen für Datenflussrechner den vorliegenden Rahmen sprengen würde, wird im folgenden nur eine Auswahl der wichtigsten Punkte gemäss [10] gegeben. Zur Vertiefung sei auf [10] (Sprache VAL), [11] (Sprache Id) oder [12] (Sprache SISAL) verwiesen.

1. Freiheit von Seiteneffekten

Seiteneffekte treten z.B. dann auf, wenn in einem konventionellen Multiprozessor eine gemeinsame Variable von mehreren Prozessoren modifiziert werden kann. Sofern nicht durch strenge Zugriffsrechte eine verbindliche Schreibe- und Lesedisziplin erzwungen wird, können fehlerhafte Resultate produziert werden. In Datenflussrechnern existieren keine Variablen im Sinne von Sprachen wie Fortran, Pascal usw., da Zwischenwerte in Form von Datenpaketen (Tokens) übermittelt werden und keine effektiven (Daten-) Speicherplätze belegen (Call by-Value statt Call by-Reference). Dadurch können bei der Verwendung von skalaren Argumenten auch keine Seiteneffekte auftreten. Da hingegen auch in Datenflussrechnern für Datenstrukturen Speicherbereiche vorgesehen sind, welche von mehreren Prozessoren bearbeitet werden, muss das Seiteneffektproblem trotzdem gelöst werden. Die entsprechende Regel besteht darin, dass Daten enthaltende Speicherzellen nicht überschrieben werden dürfen. Konsequenterweise muss daher bei jeder beabsichtigten Schreiboperation in einen bereits mit gültigen Daten versehenen Array oder Record usw. ein vollständig neuer Datenbereich erstellt werden, in den der neue Wert an der entsprechenden Stelle eingeschrieben und alle unveränderten Daten in die weiteren Plätze kopiert werden. Diese enorm aufwendige Arbeit kann durch die Einführung hierarchischer Datenstrukturen wesentlich verringert werden. Eine vielversprechende, das Problem weiter entschärfende Methode besteht in der Verwendung von sogenannten I-Strukturen (vgl. Kapitel «Datenstrukturen» [3].

2. Lokalitätsprinzip (im Sinne der Verwendung von Variablennamen)

Das Lokalitätsprinzip ist grundsätzlich dann gewährleistet, wenn die Instruktionen eines Programms keine unnötigen, weitreichenden Datenabhängigkeiten aufweisen. Es wird z.B. dann verletzt, wenn in einem Programm gleiche Variablennamen an verschiedenen Stellen benützt werden, obwohl zwischen ihnen kein Zusammenhang existiert. Diese Methode wird oft verwendet, um in einem konventionellen Rechner Speicherplätze zu sparen. Optimierende Compiler sind heute zwar in der Lage, mit entsprechendem Aufwand die meisten derartigen Pseudoabhängigkeiten zu entflechten, um dadurch eine parallele Verarbeitung zu ermöglichen. Da, wie erwähnt, in einem Datenflusskonzept keine Speicherplätze belegenden Variablen existieren, entfällt die Notwendigkeit derartiger Speicheroptimierungen und somit die Verwendung gleicher Variablennamen. Compiler werden dadurch wesentlich einfacher, und Parallelverarbeitungen werden nicht mehr unnötig eingeschränkt.

3. Einfachzuweisungen (Single Assignment)

Datenflussprogramme müssen in eindeutige, gerichtete Graphen umgewandelt werden können. Konsequenterweise dürfen Variablennamen auf der linken Seite von Zuweisungen in-

nerhalb eines zusammenhängenden Programmbereichs nur *einmal* vorkommen.

4. Spezielle Notationen zur Programmierung von Iterationen

Damit in Iterationen Loop-Zähler der Art i:=i +1 realisiert werden können, wird eine spezielle Notation der Art new i :=i +1 eingeführt (z.B. [11]). Dies ermöglicht dem Compiler, eine Entflechtung der einzelnen Loop-Durchgänge vorzunehmen und zudem Hilfsinstruktionen in den Programmgraphen einzufügen, welche in den Token Tags (Fig. 9) die entsprechenden Iteration-Count-Felder inkrementieren, damit eine parallele Abarbeitung mehrerer Loop-Instanzen möglich wird.

Ausblick, offene Fragen

Datenflussrechner sind nach wie vor Gegenstand intensiver Forschungsarbeiten. Trotz den zahlreichen positiven Aspekten kann deren Erfolg noch nicht endgültig abgeschätzt werden. Die wichtigsten, zurzeit noch nicht befriedigend gelösten Probleme betreffen die Last- und Datenverteilungsstrategien sowie die technologischen Aspekte für eine kosteneffiziente Implementation. Die Frage der Lastverteilung ist nicht nur datenflussspezifisch, sie muss in jedem Parallelprozessorkonzept für technisch-wissenschaftliche Anwendungen allgemeiner Art (General Purpose) gelöst werden. Obwohl beim Datenflusskonzept durch die Generierung eines Programmgraphen wichtige Informationen, wie z.B. der Grad der Parallelität im Applikationsprogramm usw., in vielen Fällen explizit vorliegen und durch den Compiler/Optimizer wichtige Eingangswerte für eine Lastverteilungsstrategie gewonnen werden können, müssen trotzdem während der Laufzeit Entscheide über die Zuweisung von Programmteilen zu Prozessoren vorgenommen werden. Kriterien dazu sind etwa die momentane Last einzelner Prozessoren oder die zusätzliche Belastung durch den neuen Auftrag. Hier treten aber nichttriviale Probleme auf, da die Belastung eines Prozessors kaum exakt, allenfalls sogar nur grob angenähert werden kann.

Intensive Vertiefung erfordert auch die Organisation und Zuweisung von Datenstrukturen in den Speicherblökken der verschiedenen Prozessoren. Dazu gehören Strategien, welche mittels Anlegens von mehreren Kopien einzelner Datenstrukturen die Effizienz eines Rechners erhöhen sollen, sowie die Datenduplikationen, welche durch den funktionalen Programmierstil erzwungen werden.

Ebenfalls Gegenstand von Untersuchungen ist die Frage, ob Datenfluss mit sehr feiner Granularität (Fine-Grain Parallelism) tatsächlich effizient ist oder ob ein grobkörniges Konzept (Coarse-Grain Parallelism) [13] verfolgt werden soll, bei dem im Datenflussgraphen gemäss Figur 1 anstelle von einzelnen einfachen Instruktionen in den Knoten Blöcke von Instruktionen (Codeblocks) vorgesehen werden sollen, welche in den einzelnen Prozessoren von-Neumann-artig abgearbeitet werden sollen.

Eine Übersicht über laufende Forschungsprojekte auf dem Gebiet Datenflussrechner, vertiefende Diskussionen über die verschiedenen Aspekte dieser neuen Rechnergeneration sowie umfassende Literaturreferenzen können z.B. in [4; 8; 14; . . .;17] gefunden werden.

Literaturverzeichnis

[1] *D.D. Gajski* a.o.: A second opinion on data flow machines and languages. IEEE Computer 15(1982)2, p. 58...69.

[2] *J.R. Gurd, C.C. Kirkham* and *I. Watson:* The Manchester prototype dataflow computer. Communications of the ACM 28(1985)1, p. 34...52.

[3] *Arvind* a.o.: The tagged token dataflow architecture. Memo of the Computation Structures Group/Laboratory for Computer Science. Cambridge/Mass., Massachusetts Institute of Technology (MIT), 1983.

[4] *P.C. Treleaven* and *I.G. Lima:* Future computers: Logic, data flow, ..., control flow? IEEE Computer 17(1984)3, p. 47...58.

[5] *T. Agerwala* and *Arvind:* Data flow systems. IEEE Computer 15(1982)2, p. 10...13.

[6] *T. Gunzinger:* Synchroner Datenflussrechner zur Echtzeitbildverarbeitung. Mustererkennung 1986. 8. DAGM-Symposium (Deutsche Arbeitsgemeinschaft für Mustererkennung). Informatik-Fachberichte 125. Berlin u.a., Springer-Verlag, 1986; S. 123...128.

[7] *A. Kündig:* Parallelverarbeitung in elektronischen Systemen - eine Übersicht. Bull. SEV/VSE 78(1988)7, S. 338...343.

[8] *P.C. Treleaven, D.R. Brownbridge* and *R.P. Hopkins:* Data-driven and demand-driven computer architectures. ACM Computing Surveys 14(1982)1, p. 93...143.

[9] *H.J. Siegel:* Interconnection networks for large-scale parallel processing. Theory and case studies. Lexington/Mass./Toronto, Lexington Books, 1985.

[10] *W.B. Ackermann:* Data flow languages. IEEE Computer 15(1982)2, 15...25.

[11] *R.S. Nikhil, K. Pingali* and *Arvind:* Id nouveau. Memo 265 of the Computation Structures Group/Laboratory for Computer Science. Cambridge/Mass., Massachusetts Institute of Technology (MIT), 1986.

[12] *J. McGraw:* SISAL - Streams and iteration in a single assignment language. Language reference manual. Livermore/California, Livermore National Laboratory, 1983.

[13] *R. Buehrer* and *K. Ekanadham:* Incorporating data flow ideas into Von Neumann processors for parallel execution. IEEE Trans. on Computers 36(1987)12, p. 1515...1522.

[14] Data flow systems. Special issue. IEEE Computer 15(1982)2.

[15] *J.-L. Gaudiot:* Structure handling in dataflow systems. IEEE Trans. on Computers 35(1986)6, p. 489...502.

[16] *V.P. Srini:* An architectural comparison of dataflow systems. IEEE Computer 19(1986)3, p. 68...88.

[17] *A.H. Veen:* Dataflow machine architecture. ACM Computing Surveys 18(1986)4, p. 365...396.

[18] *J.B. Dennis:* Data flow supercomputers. IEEE Computer 13(1980)4, p. 48...56.