NEUE VERFAHREN ZUR OPTIMISIERUNG UND PARALLELISIERUNG VON PROGRAMMEN

G.Urschler, IBM Labor Wien

ÜBERSICHT

Ziel dieses Vortrages ist es, die Zweckmäßigkeit der Strukturierung (Modularisierung) von Programmen für deren automatische Verarbeitung aufzuzeigen. Dies geschieht an Hand der Vorführung von 2 neuartigen Verfahren zur Optimierung und Parallelisierung von Flussdiagrammen, die beide von einer jeweils spezifischen Modularisierung der zu verarbeitenden Programme ausgehen. Es wird gezeigt, wie sich beispielsweise redundante Ausdrücke in Flussdiagrammen vollständig - auch in datenabhängigen Situationen - eliminieren lassen mittels einer symbolischen Interpretation genannten Verfahrens. Dieses zielt auf möglichst kurze Programmausführungen ab, tut dies jedoch auf Kosten der Programmgröße.

Es wird ferner gezeigt, dass sich Flussdiagramme derartig in eine variablenfreie und gleichfalls modulare Form übersetzen lassen, dass sich dabei maximale Parallelität ergibt. Diese erlaubt beispielsweise die parallele Ausführung von voneinander unabhängigen Iterationen oder Iterationsteilen auf Grund eines die Parallelität dynamisch erkennenden Kontrollmechanismus.

1. EINLEITUNG

Unter "Programmen" werden im folgenden einfache Flussdiagramme (vgl. Floyd /3/) verstanden. Der Einfachheit halber wird angenommen, dass jedes Programm nur einen Anfang (bezeichnet mit ∇), nur ein Ende (bezeichnet mit Δ) und nur zweiwertige (ja/nein) Entscheidungen aufweist, und dass sämtliche Ausdrücke nur aus einfachen (nicht strukturierten) Variablen aufgebaut sind. Abb. 1 zeigt in bekannter Weise die Grundstruktur eines solchen Flussdiagramms. Boole'sche Ausdrücke sind darin mit p , q , ... bezeichnet und A , B , C , ... stehen für "elementare Programmblöcke", worunter maximale Sequenzen von Zuweisungen und Eingabe/Ausgabeoperationen verstanden werden.

Ein Programm wird optimisiert durch die Eliminierung von Redundanzen, jedoch unter Beibehaltung der ursprünglichen Kontrollstruktur, während Parallelisierung den Übergang zu einer Kontrollstruktur bezeichnet, die die gleichzeitige Ausführung von voneinander unabhängigen Operationen erlaubt.

Ohne weitere Begründung sei die These vorweggenommen, dass sowohl die Optimierung als auch die Parallelisierung von Programmen automatisch, d.h. auf Grund geeigneter Algorithmen durchgeführt werden soll. Der Grundgedanke dieses Vortrages besteht dann darin zu zeigen, dass die Durchführung dieser Programmtransformationen wesentlich erleichtert wird, wenn den gegebenen Programmen vorher eine geeignete Struktur aufgeprägt wird. Dieses "Aufprägen einer geeigneten Struktur" besteht darin, die gegebenen Programme in eine modulare, auf Prozeduraufrufen aufbauende Form überzuführen (wie sie manche Sprachen, z.B. VDL - siehe /5/ - von vornherein garantieren).

Die Modularisierung sieht für Optimierung und Parallelisierung verschieden aus und wird im nächsten Abschnitt beschrieben werden. Von einem modularen Programmaufbau ausgehend wird im Kapitel 3 dann eine symbolische Interpretation benannte, neuartige Optimierungstechnik an Hand der Eliminierung von redundanten Ausdrücken beschrieben und ein Formalismus zur Durchführung dieser Methode dargelegt

werden. Diese Optimierungstechnik ist in erster Linie bestrebt minimale Programmausführungen, dies jedoch auf Kosten der Programmgrösse, zu erhalten. Im Kapitel 4 wird abschliessend gezeigt werden, dass die geeignete Modularisierung von Programmen eine wesentliche Voraussetzung für die Ausnützung von maximaler Parallelität ist.

2. MODULARISIERUNG

Bereits im Jahre 1960 wurde von McCarthy (vgl. /6/) ein Verfahren angegeben, dass es gestattete, goto-Programme in bedingte Ausdrücke zu übersetzen und within diese Programme in gewissem Sinne zu strukturieren. Das Verfahren sei an Hand des in Abb. 1 gegebenen Programmbeispiels illustriert. Als erstes wird jedem Verzweigungspunkt (jeder Entscheidung) und jedem Verschmelzungs (Ziel)punkt ein Name (bezeichnet mit α , β , γ ...) gegeben. Dies sei wie in Abb. 2 angegeben durchgeführt.

Jedem dieser Namen wird ein Modul (ein Programnteil) zugeordnet, der in Form einer Rekursionsgleichung beschrieben wird und wie folgt aussieht. Ist der Modul unbedingt (geht er aus einem Verschmelzungspunkt hervor), dann gibt die Gleichung an mit welchem Programmblock der Modul beginnt und mit welchem Modul der Block anschliessend fortgesetzt wird. Die Gleichung $\beta = A\gamma$ beschreibt beispielsweise den Modul β . Ist der Modul bedingt (liegt ihm ein Verzweigungspunkt zugrunde), dann sind für diesen Modul 2 Alternativen zu beschreiben, denen der Boolesche Ausdruck, der die Auswahl der Alternativen regelt, vorangesetzt wird. In selbsterklärender Syntax ergibt sich für den Modul α des Beispiels: $\alpha = (p) \{ \beta \mid \gamma \}$. Für das gesamte Programm der Abb. 2 erhält man das folgende Gleichungssystem (wird künstlich als "Hauptmodul" eingeführt):

$$\begin{aligned}
 \nabla &= \alpha \\
 \alpha &= \{p\} \{\beta \mid \gamma\} \\
 \beta &= A \gamma \\
 \gamma &= B \delta \\
 \delta &= \{a\} \{\beta \mid \quad\}
 \end{aligned}$$

Das Charakteristische an diesem Modularisierungsverfahren ist, dass in ihm der "Bereich" eines jeden Moduls (die zu ihm gehörenden Anweisungen) maximal ist, was impliziert, dass wenn immer ein Modul zu Ende ausgeführt ist auch das Programmende angelangt ist. Diese Eigenschaft zeigt zumindestens die Unnatürlichkeit der aufgeprägten Struktur an wodurch diese Technik auch bis heute keine praktische Bedeutung erlangt hat. In Abschnitt 3 wird jedoch gezeigt werden, dass sie sich für das dort angegebene Optimisierungsverfahren in idealer Weise eignet.

Zu einer natürlichen, d.h. die Programmlogik aufzeigenden, Modularisierung kommt man, wenn die Modulbereiche "minimalisiert" werden. Dieses Verfahren ist in /11/ näher beschrieben. Es beruht auf der seit langem bekannten Tatsache (vgl. /4/, /8/), dass es zu jedem Verzweigungspunkt in einem Programm genau einen ihm zugeordneten Programmpunkt gibt, wo die entsprechenden Alternativen das erste Mal wieder zusammenlaufen. Dieser "unmittelbare Nachdominator" gibt damit die (selbst bereits ausserhalb liegende) Grenze des betreffenden Modulbereiches an.

Von Abb. 2 lässt sich beispielsweise ablesen, dass γ der unmittelbare Nachdominator von α ist und A der von δ . Für die effektive Bestimmung von Nachdominatoren stehen bekannte Verfahren zur Auswahl (siehe /9/). Durch die (zwangsläufig eindeutige) Kettenbildung von unmittelbaren Nachdominatoren - wobei diese Ketten aber auf den Bereich des Moduls den sie beschreiben, beschränkt bleiben müssen - ergibt sich ein zweites Modularisierungsverfahren, dass im 4. Abschnitt seine Bedeutung für die Parallelisierung zeigen wird. Für das Beispiel von Abb. 2 ergibt sich folgendes Gleichungssystem (Λ bezeichnet den leeren Programmblock):

$$\begin{aligned}\nabla &= \alpha \ B \ \delta \ \Delta \\ \alpha &= (p) \ \{A \ | \ \sqcup\} \\ \delta &= (q) \ \{A \ B \ \delta \ | \ \sqcup\}\end{aligned}$$

(Die Einführung der unbedingten Module β und γ könnte gemacht werden, ist aber für das gegebene Beispiel nur sinnvoll, wenn die Elementarblöcke A und B in die entsprechenden Anweisungen aufgegliedert werden.)

Dass durch dieses Modularisierungsverfahren die Programmlogik zutage kommt, illustriert eine nähere Betrachtung der Gleichung für beispielsweise ∇ . Sie zeigt, dass jeder Programmablauf (unabhängig von den Daten) in einer Ausführung des Moduls α , gefolgt von einer Ausführung des Blockes B, gefolgt von einer Ausführung des Moduls δ besteht. Damit ist bereits die Grundstruktur des gegebenen Programmes herausgearbeitet. Wenn man sich näher für einen der vorkommenden Module, beispielsweise δ interessiert, zeigt die entsprechende Gleichung, dass entweder A gefolgt von B auszuführen und anschliessend δ zu wiederholen ist, oder dass δ abgeschlossen werden kann. Damit ist eine Programmstruktur erreicht, die ein schrittweises und nur bis zum gewünschten Detail gehendes Verständnis eines gegebenen Programmes gewährleistet und mithin der Grundanforderung für eine sinnvolle Programmstruktur Genüge tut.

3. SYMBOLISCHE INTERPRETATION

Jede Programmausführung lässt sich als Folge der nacheinander ausgeführten Programmteile beschreiben. Für das gegebene Beispiel würde beispielsweise die Folge $\nabla \ p \ B \ q \ A \ B \ q \ A$ eine Programmausführung beschreiben. Alle möglichen Programmausführungen (die im allgemeinen eine echte Übermenge der wirklichen Programmausführungen darstellen) lassen sich als unendlicher Baum mit Wurzel ∇ darstellen, wie in Abb.3 angegeben. Dieser Baum hat nur endlich viele verschiedene Unterbäume, die genau durch die Gleichungen des ersten

Modularisierungsverfahrens beschrieben werden. Ein besonderer Vorteil dieser Programmdarstellung ist es, dass der (potentiell) unendliche Baum aller Programmausführungen durch die schrittweise Expansion von Modulen aus dem Hauptmodul ∇ hergeleitet werden kann.

Der Grundgedanke der symbolischen Interpretation ist es, die durch den unendlichen Baum gegebenen Programmausführungen schrittweise durchzugehen, dabei Optimierungsinformation zu sammeln und von dieser Information später dadurch Gebrauch zu machen, dass Modifizierungen im Baum durchgeführt werden, die einerseits die Logik der Ausführung ungeändert lassen, andererseits aber die Ausführungszeit verkürzen. Damit dieses Transformationsverfahren endlich bleibt, ist ein entsprechendes Konvergenzkriterium bereitzustellen, das nach endlich vielen Schritten den Abbruch der Transformation bewirkt.

Im Folgenden wird das Verfahren an Hand der Elimination von redundanten Ausdrücken skizziert. Eine genauere Beschreibung findet sich in /14/. Die Grundidee bei der Eliminierung von redundanten Ausdrücken ist es, von der Verfügbarkeit von Ausdrücken Gebrauch zu machen. Ein Ausdruck wird dabei als verfügbar angesehen, wenn er zu diesem Zeitpunkt in einer Variablen gespeichert ist und wenn alle Variablen, die zur Auswertung dieses Ausdruckes benötigt wurden, seit diesem Zeitpunkt nicht mehr neu definiert ("getötet") wurden.

In Abb. 4 ist das Beispiel von Abb. 1 mit speziellen Anweisungen wiedergegeben. Es zeigt, dass der Ausdruck $g(a)$ in Block B redundant ist, dies aber nur falls vorher Block A ausgeführt wurde. In Abb. 5 ist der Versuch gemacht, den Vorgang der symbolischen Interpretation informal zu veranschaulichen. Was dabei passiert ist, dass als 1. Schritt der unendliche Baum von oben nach unten in allen seinen Verzweigungen durchgegangen wird und dabei "offensichtliche" Redundanzen (strichlierte Linien) beseitigt werden. Der 2. Schritt besteht darin, im Resultatbaum gleiche Unterbäume zu identifizieren (punktierte Linien) und damit wieder zu einer endlichen Programmdarstellung zu gelangen.

Um diesen Vorgang formal durchzuführen, wird der Begriff einer Zustandsbeschreibung eingeführt, der, bei der gegebenen Problemstellung, aufzeigt welche Ausdrücke in welchen Variablen verfügbar sind. Seinen Ausgang nimmt das Verfahren indem die leere (keine Information enthaltende) Zustandsbeschreibung σ_0 auf den Hauptmodul ∇ "angewandt" wird, in Zeichen $\sigma_0.\nabla$. Gemäss dem Rekursionsgleichungssystem von Kapitel 1 ist dies gleichbedeutend mit $\sigma_0.\alpha$. Werden Boole'sche Ausdrücke der Einfachheit halber nicht in das Optimisierungsverfahren einbezogen, ergibt sich weiter $\sigma_0.\alpha = \{\sigma_0.\beta \mid \sigma_0.\gamma\}$ (eine Zustandsbeschreibung wird auf einen Modul angewandt, indem sie auf seine Alternativen angewandt wird). $\sigma_0.\beta$ ist wieder gleichbedeutend mit $\sigma_0.A\gamma$. Interpretation von A in Bezug auf σ_0 bewirkt zunächst die Aufnahme der Information $a \rightarrow f(b,c)$ ($f(b,c)$ ist in a verfügbar) in die Zustandsbeschreibung, unmittelbar darauf wird diese Eintragung jedoch wieder durch die Neudefinition von b gelöscht und dafür $b \rightarrow g(a)$ in die Zustandsbeschreibung aufgenommen. Formal: $\sigma_0.A\gamma = A \sigma_1.\gamma$ wobei $\sigma_1 = \{b \rightarrow g(a)\}$. Ferner $\sigma_1.\gamma = \sigma_1.B\delta$ Wenn B in Bezug auf σ_1 interpretiert wird, ergibt sich, dass $g(a)$ bereits in b verfügbar ist, sodass sich $c := g(a)$ durch $c := b$ ersetzen liesse. Man kann jedoch einen Schritt weiter gehen (wie im Folgenden getan) und die Anweisung $c := g(a)$ gänzlich eliminieren und dafür die Information $b \sim c$ (b und c sind äquivalent mit der Bedeutung, dass spätere Vorkommnisse von c als Vorkommnisse von b zu interpretieren sind) in der Zustandsbeschreibung speichern.

In Zeichen wiederum: $\sigma_1.B\delta = \sigma_2.\delta$, $\sigma_2 = \{b \rightarrow g(a); b \sim c\}$ Ferner $\sigma_2.\delta = \{\sigma_2.\beta \mid \sigma_2.A\}$, $\sigma_2.\beta = \sigma_2.A\gamma$ und schliesslich $\sigma_2.A\gamma = A \sigma_1.\gamma$ wobei durch die Unterstreichung von A angedeutet wird, dass darin, gemäss der in σ_2 enthaltenen Äquivalenzinformation, $f(b,c)$ durch $f(c,c)$ ersetzt ist. Zur Zustandsbeschreibung σ_1 kommt man nach der neuerlichen Interpretation von A zurück, da durch die Neudefinition von a und b sämtliche in σ_2 enthaltene Information wiederum vernichtet wird. Analog ergibt sich:

$$\begin{aligned}\sigma_0.\gamma &= \sigma_0.B\delta = B\sigma_3.\delta, \quad \sigma_3 = \{c \rightarrow g(a)\} \\ \sigma_3.\delta &= \{\sigma_3.\beta \mid \sigma_3.\Delta\} \\ \sigma_3.\beta &= \sigma_3.A\gamma = A\sigma_1.\gamma\end{aligned}$$

Das Interpretationsverfahren endet, sobald ein Modul in Bezug auf eine Zustandsbeschreibung zu interpretieren ist in Bezug auf die er bereits interpretiert wurde. Die Endlichkeit des Verfahrens ergibt sich dabei automatisch aus der Endlichkeit der Anzahl von Modulen und aus der Endlichkeit der möglichen Information, die in die Zustandsbeschreibung eingeht.

Da eine Zustandsbeschreibung angewandt auf einen Modul wieder einen Modul ergibt, steht es frei, dafür einen neuen Namen zu wählen. Wird dies für obiges Beispiel gemacht, ergibt sich das folgende Rekursionsgleichungssystem als Resultatprogramm:

$$\begin{aligned}\nabla_0 &= \alpha_0 \\ \alpha_0 &= (p) \{A \gamma_1 \mid B \delta_3\} \\ \gamma_1 &= (q) \{A \gamma_2 \mid \Delta_2\} \\ \delta_3 &= (q) \{A \gamma_1 \mid \Delta_3\}\end{aligned}$$

Diesem System entspricht das in Abb. 6 wiedergegebene Flussdiagramm. Es enthält keine Redundanzen mehr.

Das oben wiedergegebene Eliminationsverfahren für redundante Ausdrücke wurde insoweit vereinfacht, als darin die Namen von Variablen unverändert blieben. Um redundante Ausdrücke vollständig zu eliminieren (wie in /14/ beschrieben) ist es an und für sich notwendig, Variablen durch Äquivalenzklassen von Variablen zu ersetzen (eine Idee, die im wesentlichen vom value-numbering Algorithmus von /1/ übernommen wurde), um dem Umstand Rechnung zu tragen, dass in einem Programm verschiedene Variablen den gleichen Namen tragen können und dass umgekehrt verschiedene Namen die gleiche Variable bezeichnen können.

Um ein besseres Verständnis für die Möglichkeit des angegebenen Verfahrens zu geben, werden im Folgenden noch 2 Beispiele gebracht allerdings ohne Ausführung des eigentlichen Interpretationsprozesses. Das erste Beispiel betrifft die

Optimisierung des folgenden Programmteiles (einem unveröffentlichten Artikel von J.C.Beatty entnommen) :

```
DO I=1 TO N;
  IF K(I) > 0 THEN M(I)=L(J);
END;
```

Mit herkömmlichen Optimisierungsverfahren (code motion) lässt sich der nach der ersten Ausführung redundante Ausdruck L(J) nicht eliminieren, da es sonst zu das Programm falsifizierenden Seiteneffekten (unsafe code motion) kommen könnte. Abb. 7 zeigt obiges Beispiel in geeigneter Form als Flussdiagramm kodiert. Abb. 8 zeigt das durch Optimierung erhaltene Resultatprogramm. Der Vollständigkeit halber sei allerdings hinzugefügt, dass dabei in Anschluss an das symbolische Interpretationsverfahren noch eine Reduktion durchgeführt wurde (wie aus der Syntaxtheorie für gerichtete, benannte Graphen bekannt), in der unnötig duplizierter Code wiederum zusammengelegt wurde. Aus Abb. 8 ist ersichtlich, dass die ursprünglich einfache Schleife durch Duplizierung in 2 Schleifen aufgelöst wurde und darin der Ausdruck L(J) wie gewünscht höchstens einmal ausgeführt werden kann.

Das zweite Beispiel soll zeigen, wie das angegebene Optimisierungsverfahren durch den Einschluss von Wahrheitswerten und der Propagierung von Wahrheitswerten erweitert werden kann. Nach jeder Entscheidung ist im Interpretationsverfahren der entsprechende Wahrheitswert ja bekannt. Einschluss dieser Information in die Zustandsbeschreibung (da es nur 2 Wahrheitswerte und nur endlich viele Bocle'sche Variable gibt ist wiederum keine Divergenzgefahr gegeben) und nachträgliche Verwendung davon erlaubt es, in gewissen Fällen überflüssige Entscheidungen zu erkennen und damit beispielsweise Schleifen zu diagnostizieren, die selbst geübten Programmierern manchmal verborgen bleiben. Abb. 9 gibt ein solches Programmfragment. Das Resultatprogramm in Abb. 10 zeigt, dass das Programm für die ersten beiden Schleifendurchläufe noch sinnvoll ist, und dass es erst nach dem dritten Schleifendurchlauf unbedingt zu schleifen beginnt. Im Formalismus der symbolischen Interpretation ist dieser

Zustand durch das Auftreten eines unbedingten, rekursiven Moduls leicht zu erkennen.

4. MAXIMALE PARALLELISIERUNG

Parallelisierung ist ähnlich der Optimierung ein von Natur aus eigentlich dynamisches (nur zur Programmaufzeit zu lösendes) Problem. Im letzten Abschnitt wurde gezeigt, dass sich für die Optimierung ein Teil dieser Dynamik bereits in einem zur Übersetzungszeit durchgeführten symbolischen Programmablauf vorwegnehmen lässt. Im Folgenden wird gezeigt werden, dass sich, zumindestens für Flussdiagramme, das Parallelisierungsproblem vollständig - was eine Erkennung der Parallelität zur Laufzeit impliziert - lösen lässt. Das Verfahren beruht auf der in Kapitel 2 angegebenen zweiten Modularisierungstechnik und ist in /10/ und /13/ näher beschrieben. Es wird illustriert an Hand des in Abb. 11 gezeigten Flussdiagramms. Das Programm berechnet die Ausdrücke $f(1)$, $f(2)$, ..., $f(100)$ (wobei der Operator f unbestimmt gelassen wird) und summiert diese Ausdrücke auf. Von der Logik her ist klar, dass die Auswertung aller dieser Ausdrücke voneinander unabhängig ist und mithin parallel zueinander erfolgen kann. Zu zeigen ist, wie dieser "dynamische" Parallelismus (der im allgemeinen nicht von der Programmgrösse sondern von der Länge des Programmablaufs abhängt) ermittelt werden kann.

Gemäss der in Abschnitt 2 angegebenen Modularisierungstechnik ergeben sich die folgenden das Programm beschreibenden Rekursionsgleichungen (die einzelnen Anweisungen werden nur in symbolischer Weise als A, B, C, ... angeführt, wie in Abb. 11 angedeutet):

$$\begin{aligned} \nabla &= A \ B \ C \ \alpha \ G \ \Delta \\ \alpha &= \{p\} \ \{D \ E \ F \ C \ \alpha \ | \ \perp\} \end{aligned}$$

Der grosse Vorteil dieser Programmdarstellung für die Parallelisierung ist, dass durch sie genau die Kontrollabhängigkeiten - das sind die Abhängigkeiten einer Anweisungsausführung von vorangehenden Entscheidungen - aufgezeigt werden. Wann immer eine Anweisung im Bereich eines Moduls liegt, d.h. entweder direkt in einer seiner Alternativen vorkommt oder im Bereich eines darin enthaltenen weiteren Moduls liegt, dann hängt die Ausführung dieser Anweisung von der Ausführung des betreffenden Modulaufrufes ab. Um die Abhängigkeit einer Anweisungsausführung von den zu ihrer Ausführung benötigten Daten herauszuarbeiten ist eine Datenflussanalyse unumgänglich. Des leichteren Verständnisses halber wird diese Analyse im Folgenden in graphischer Form durchgeführt, basierend auf dem Konzept eines Datenflussgraphen (siehe /7/), wobei in Erweiterung dieses Konzeptes zu den Basisoperationen (Zuweisung und Eingabe/Ausgabeoperationen), die von Variablen lesen, bzw. nach Variablen schreiben, noch Expansionsoperationen (durch Doppelkreis gekennzeichnet) hinzukommen, die bei Ausführung einen Knoten in einen weiteren (durch eine Alternative gegebenen) Datenflussgraphen expandieren.

Ein Datenflussgraph wird gezeichnet, indem für eine Operation des betreffenden Moduls nach der anderen die Lese- und Schreiboperationen in Pfeilform gezeichnet werden, wobei aber jede Variable (in Kästchenform angegeben) höchstens einmal beschrieben werden darf, sodass bei wiederholtem Auftreten ein und derselben Variablen Kopien anzufertigen sind (die durch fortlaufende Numerierung voneinander unterschieden werden) und nur die letzte dieser angefertigten Kopien "Aktualitätswert" besitzt, d.h. in darauf folgenden Operationen verwendet werden darf. Für Expansionsoperationen ist der Lesevorgang von der (für Auslösung der Expansion allein massgebenden) Entscheidungsvariable durch einen Doppelpfeil angegeben. Einfachpfeile zu oder von Expansionsoperationen sind potentielle Datenlinien, die erst bei Ausführung der Expansion in tatsächliche Datenlinien übergehen können.

Für den Hauptmodul ∇ ergibt sich dabei das in Abb. 12 dargestellte Bild. Der Nebenmodul α hat 2 Alternativen, deren

entsprechende Datenflussgraphen Abb. 13 zeigt. Damit auch wirklich jede dieser Alternativen bei einer Expansion zum Zuge kommen kann, ist für ein gleichförmiges "interface" Sorge zu tragen (gleich Anzahl von Eingabe und Ausgabevariablen für jede Modulalternative). Eine tiefergehende (hier nicht wiedergegebene aber in /13/ beschriebene) Analyse zeigt ferner, dass im Falle eines Aufrufes der zweiten Modulalternative nur die Variable *s* benötigt wird, deren Bereitstellung jedoch durch Einführung einer eigenen Kopieroperation (*s := s*) zu gewährleisten ist.

Die Datenflussgraphen lassen, zumindestens für einen menschlichen Betrachter (und für den sind sie ausschliesslich bestimmt), bereits den Kontrollmechanismus der parallelen Ausführung erkennen. Operationen müssen zuerst einmal erzeugt werden. Dies geschieht durch Expansion, beginnend mit der vorgegebenen Expansionsoperation ∇ . Sobald durch Expansion ein eigenes Datenflussnetz hergestellt ist, sind darin Basisoperationen ausführbar, sobald ihre Eingabewerte vorhanden sind. Bei deren Ausführung wird der entsprechende Wert an die Ausgabevariable geschrieben und die Operation selbst kann im Graphen gelöscht werden.

Expansionsoperationen werden ausführbar, sobald ihre Entscheidungsvariable definiert ist. Der Wert dieser bestimmt welche Alternative ausgewählt werden soll, und eine Kopie des entsprechenden Datenflussgraphen wird in das bestehende Netz anstelle der Expansionsoperation eingesetzt. Zu jedem Ausführungszeitpunkt stellt das Netz den "precedence Graphen" (vgl. /2/) dar, der sämtliche momentan erlaubten Ausführungsmöglichkeiten beschreibt. Die Ausführung endet mit dem Verschwinden des Netzes (die eigentlichen Auswirkungen der Programmausführung, das Schreiben zu externen Medien, steht ausserhalb der Betrachtungen). Die maximal mögliche Parallelität ergibt sich dabei bei Programmausführung von selbst, einfach durch das Vorhandensein oder Nichtvorhandensein von Datenlinien. Für das gegebene Programm möge sich der Leser selbst überzeugen, dass sich dabei die wiederholten Ausführungen der Anweisung D als voneinander unabhängig erweisen.

So anschaulich die graphische Darstellung sein mag, so ungeeignet ist sie für die Verwendung durch eine Maschine. Als nächster Schritt wird die graphische Darstellung daher einmal linearisiert und in die Form einer "single-assignment" Sprache gebracht. Dies geschieht indem jede Modulalternative als Menge (nicht als Liste) der in dem entsprechenden Graphen auftretenden Operationen geschrieben wird, wobei Modulaufrufe mit einer Parameterliste versehen werden, die die durch Semikolon voneinander getrennten Eingabe und Ausgabevariablen aufzeigt. Für das angeführte Beispiel ergibt sich damit in selbsterklärender Form:

$$\nabla = \{s_0 := 0; n_0 := 1; p_0 := n_0 \leq 100; \alpha(p_0)[s_0, n_0; s_1, n_1]; \\ \text{write } s_1\}$$

$$\alpha(1) = [s_0, n_0; s_2, n_2] \{r_0 := f(n_0); s_1 := s_0 + r_0; n_1 := n_0 + 1; \\ p_0 := n_1 \leq 100; \alpha(p_0)[s_1, n_1; s_2, n_2]\}$$

$$\alpha(2) = [s_0, -; s_2, -] \{s_2 := s_0\}$$

Die Reihenfolge der Anweisungen innerhalb einer Modulalternative hat keinen Einfluss auf die Programmlogik, da die Ausführung von Anweisungen lokal durch das Eintreffen der benötigten Eingangswerte gesteuert wird. Bei Modulaufrufen ist eine Parameterübergabe per Name angenommen (per Wert wäre jedoch genauso gut).

Der Nachteil der obigen Programmrepräsentation ist, dass sie immer noch nicht genügend maschinenorientiert ist, da die Ausführbarkeit einer Operation zwar implizit eindeutig beschrieben ist, in einem echten Kontrollmechanismus die entsprechenden Signale jedoch explizit gemacht werden müssen. Dieses Explizitmachen erfolgt in einem letzten Transformationsschritt, wobei die single assignment Form in eine variablenfreie Programmstruktur umgeschrieben wird. Der Grundgedanke dieser Transformation ist es, die durch die Operation ermittelten Werte nicht indirekt in bestimmten Datenfeldern zu speichern und später bei Bedarf von dort wieder abzurufen, sondern diese Werte direkt in den Eingabestellen von Operationen einzusetzen, wo sie benötigt werden. Zu diesem Zweck

ist es zunächst einmal notwendig eine Eins-zu-Eins Beziehung zwischen Schreibe- und Leseoperationen herzustellen, was durch die Einführung von "Verteileranweisungen" (die Mehrfachzuweisungen darstellen) erfolgt. Zusätzlich hat es sich jedoch noch notwendig gezeigt an all den Stellen "Bufferanweisungen" (die Kopieranweisungen der Art $a := b$ sind) einzuführen, wo eine nicht triviale Operation sonst nur deswegen nicht ausführbar wäre, weil die Wertzuweisungsstelle erst in einem späteren erfolgenden Expansionsschritt erzeugt wird.

Sobald diese Paarung von Lese- und Schreiboperationen erfolgt ist, kann dazu übergegangen werden, die Variablen zu eliminieren und durch Adressen zu ersetzen. Liegen z.B. Anweisungen der Art $a := f(b)$ und $c := g(a)$ vor, dann gehen diese in $\vec{a} := f(b)$ und $\vec{c} := g(a)$ über, wobei $\vec{a}$ die Adresse von a (d.h. die eindeutig bestimmte Stelle im Programm, wo a sonst noch vorkommt) angibt. a selbst ist dabei nicht mehr als Variable, sondern als platzhaltende Leerstelle anzusehen. Für das Beispielsprogramm ergibt sich in variablenfreier Form z.B. die folgende Darstellung ($\vec{}$ bezeichnet die Leeradresse):

$$\begin{aligned} \nabla &= \{s_{00} := 0; \vec{n}_{00}, \vec{n}_{01} := 1; \vec{p}_0 := n_{00} \leq 100; n_{02} := n_{01}; \\ &\quad \alpha(\vec{p}_0) [\vec{s}_{00}, \vec{n}_{02}; \vec{s}_1, \vec{}]; \text{write } s_1\} \\ \alpha(1) &= [\vec{s}_0, \vec{n}_0; \vec{s}_2, \vec{n}_2] \{ \vec{n}_{00}, \vec{n}_{01} := n_0; \vec{r}_0 := f(n_{00}); \\ &\quad \vec{s}_{10} := s_0 + r_0; s_{11} := s_{10}; \vec{n}_{10}, \vec{n}_{11} := n_{01} + 1; \vec{p}_{00} := n_{10} \leq 100; \\ &\quad n_{12} := n_{11}; \alpha(\vec{p}_{00}) [\vec{s}_{11}, \vec{n}_{12}; s_2, n_2] \} \\ \alpha(2) &= [\vec{s}_0, \vec{}; \vec{s}_2, \vec{}] \{ s_2 := s_0 \} \end{aligned}$$

Eine Basisoperation wird ausgeführt, sobald ihre Eingabestellen Werte und sobald ihre Ausgabestellen Adressen darstellen. Ihre Ausführung erfolgt durch Auswertung des an der "rechten" Seite stehenden Ausdruckes und der Zuweisung des erhaltenen Wertes an die angegebenen Adressen. Expansionsoperationen werden ausführbar, sobald der für die Entscheidung notwendige Wert vorhanden ist und alle übrigen Parameter Adressen darstellen. Eine Expansionsoperation wird ausgeführt, indem zunächst die entsprechende Modulalternative ausgewählt, eine Kopie davon

angefertigt ("Expansion") und diese Kopie schliesslich "geladen" wird, wobei die im Modul vorhandenen relativen Adressen dynamisch in Absolutadressen umgewandelt werden. Der letzte Schritt in der Ausführung besteht schliesslich im Herstellen der entsprechenden Datenanschlüsse. Dies geschieht durch geeignete Adressübergabe zwischen der "aktuellen" Adressliste, die vom Modulaufruf kommt und der "formalen" Adressliste, die vom Modul selbst her übernommen wurde.

Dieser Ausführungsmechanismus sei im Folgenden an Hand eines kurzen Ausführungsbeispiels illustriert. Jede Ausführung beginnt mit der Expansionsoperation ∇ . Diese geht bei Ausführung über in:

$$s_{00}:=0; \vec{n}_{00}, \vec{n}_{01}:=1; \vec{p}_0:=n_{00} \leq 100; n_{02}:=n_{01}; \\ \alpha(p) [\vec{s}_{00}, \vec{n}_{02}; \vec{s}_1, \vec{}]; \text{ write } s_1.$$

Von allen diesen Operationen ist nur $\vec{n}_{00}, \vec{n}_{01}:=1$ ausführbar. Exekution ergibt:

$$s_{00}:=0; \vec{p}_0:=1 \leq 100; n_{02}:=1; \alpha(p) [\vec{s}_{00}, \vec{n}_{02}; \vec{s}_1, \vec{}]; \\ \text{ write } s_1$$

Wiederum ist nur eine Operation ausführbar und führt zu (der Wahrheitswert "wahr" wird durch 1 und "falsch" durch 2 repräsentiert):

$$s_{00}:=0; n_{02}:=1; \alpha(1) [\vec{s}_{00}, \vec{n}_{02}; \vec{s}_1, \vec{}]; \text{ write } s_1$$

Nunmehr ist die Expansionsoperation α ausführbar. Der sich ergebende formale Adressvektor ist $[\vec{s}_0, \vec{n}_0; \vec{s}_2, \vec{n}_2]$. $\vec{s}_0, \vec{n}_0$ sind darin jene Stellen im Modul, an die die "ausssen" erzeugten Eingabewerte zu liefern sind. $\vec{s}_0$ ist daher an die Adresse $\vec{s}_{00}$ zu setzen und $\vec{n}_0$ nach $\vec{n}_{02}$. Für die Ausgabewerte verläuft die Übergabe in umgekehrter Richtung. Die Adresse $\vec{s}_1$ kommt nach $\vec{s}_2$ und die Leeradresse $\vec{}$ nach $\vec{n}_2$. Zusammen ergibt dies:

$$\vec{s}_0:=0; \vec{n}_0:=1; \vec{n}_{00}, \vec{n}_{01}:=n_0; \vec{r}_0:=f(n_{00}); \vec{s}_{10}:=s_0+r_0; \\ s_{12}:=s_{10}; \vec{n}_{10}, \vec{n}_{11}:=n_{01}+1; \vec{p}_{00}:=n_{10} \leq 100; n_{12}:=n_{11}; \\ \alpha(f_{00}) [\vec{s}_{11}, \vec{n}_{12}; \vec{s}_2, \vec{}]; \text{ writes } s_1$$

Nunmehr sind die ersten drei Operationen simultan ausführbar und setzt sich die Programmausführung in analoger Weise fort. Je eher und je mehr Expansionsoperationen ausgeführt werden umso grösser ist im allgemeinen die mögliche Parallelität.

Die zuletzt behandelte Sprache ist selbstverständlich nicht als Programmiersprache, sondern als Maschinensprache zu verstehen, deren Aufbau bereits die Struktur der zu ihrer Interpretation benötigten Maschine erkennen lässt. In /12/ findet sich eine Beschreibung der Logik dieser Maschine.

Die Vorteile der erhaltenen Programmstruktur sind mannigfaltig. Neben der (in /10/ nachgewiesenen) erhaltenen maximalen Parallelität sind vor allem zu erwähnen: dynamische Speicherzuweisung (Daten existieren genau so lange, als sie benötigt werden), direkte Datenverarbeitung (kein interner Datenspeicher), dynamische Adressberechnung, einfaches Pagen (durch den look-ahead Effekt der Expansion) und eine natürliche Einbaumöglichkeit in ein multitasking Environment.

Als Nachteile sind anzuführen der zusätzliche Overhead sowohl zur Übersetzungs- als auch zur Laufzeit und vor allem das Fehlen eines physisch existenten Interpretierers.

Appendix A: ABBILDUNGEN

Abb. 1 FLUSSDIAGRAMM

Abb. 2 FLUSSDIAGRAMM MI
MODULNAMEN

Abb. 3 BAUM DER PROGRAMMAUSFÜHRUNGEN

Abb. 4 DATENABHÄNGIGE REDUNDANZ

Abb. 5 SYMBOLISCHE INTERPRETATION - INFORMAL

Abb. 6 RESULTATPROGRAMM

Abb. 7 UNSAFE CODE MOTION

Abb. 8 RESULTATPROGRAMM

Abb. 9 PROGRAMM MIT ENDLOSSCHLEIFE

Abb. 10 RESULTATPROGRAMM

Abb. 11 PROGRAMM MIT PARALLEL AUSFÜHRBAREN ITERATIONSTEILEN

Abb. 12 DATENFLUSSGRAPH FÜR ∇ Abb. 13 DATENFLUSSGRAPHEN FÜR α

ANHANG B

LITERATUR

- /1/ J.Cocke, J.T.Schwartz, "Programming Languages and Their Compilers", Courant Institute of Mathematical Sciences, New York University, 1969.
- /2/ J.B.Dennis, "Computational Structures", Lecture Notes, 1970.
- /3/ R.W.Floyd, "Assigning Meaning to Programs", Proc.of Symposia in Appl. Math., Vol.19, 1967, pp.19-32.
- /4/ E.S.Lowry, C.W.Medlock, "Object Code Optimisation", CACM, Vol.12, 1969.
- /5/ P.Lucas, K.Walk, "On the Formal Description of PL/I", Ann.Reviews in Automatic Progr., Vol. 6/3, 1969.
- /6/ J.McCarthy, "Towards a Mathematical Science of Computation", Proc. of IFIPs Congr., 1962, pp.21-28.
- /7/ R.E.Miller, J.D.Rutledge, "Generating a Data Flow Model of a Program", IBM Techn.Disclosure Bulletin, Vol.8/11, 1966, pp.1550-1553.
- /8/ R.T.Prosser, "Application of Boolean Matrices to the Analysis of Flow Diagrams", 1959 Proc.of the EJCC, pp. 133-138.
- /9/ R.E.Tarjan, "Finding Dominators in Directed Graphs", Proc. 7th Ann.Princeton Conf. in Form.Sciences and Systems, 1973.
- /10/ G.Urschler, "The Inherent Parallelism of Flow Diagrams", IBM Lab Vienna, TR 25.129, 1972.

- /11/ G.Urschler, "Automated Functional Programming", Paper available from author, 1973.

- /12/ G.Urschler, "Patentschrift GE 972 502", IEM Österreich GmbH, Wien, 1973.

- /13/ G.Urschler, "The Transformation of Flow Diagrams into Maximally Parallel Form", Proc. of Sagamore Computer Conf. on Parallel Processing, Syracuse University, N.Y., 1973.

- /14/ G. Urschler, "Complete Redundant Expression Elimination in Flow Diagrams", IBM T.J.Watson Research Center, Yorktown Heights, N.Y., 1974.