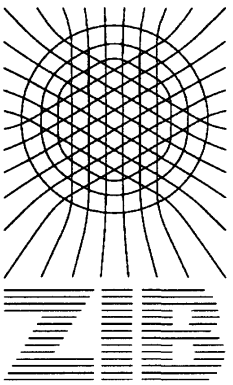
Konrad-Zuse-Zentrum für Informationstechnik Berlin



H. C. Hege

Datenabhängigkeitsanalyse und Programmtransformationen auf CRAY–Rechnern mit dem FORTRAN–Präprozessor **fpp**

Technical Report TR 90–4 (Februar 1990)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Rabe KG Buch- und Offsetdruck Berlin

ISSN 0933-789X

Hans-Christian Hege

Datenabhängigkeitsanalyse und Programmtransformationen auf CRAY-Rechnern mit dem FORTRAN-Präprozessor **fpp**

Abstract

The FORTRAN preprocessor **fpp** in the newly introduced Autotasking System of CRAY Research allows automatic vectorization and parallelization on basis of a data dependence analysis. An introduction into data dependence analysis is given, showing how data dependence graphs unveil opportunities for program transformations like vectorization and concurrentization. The report contains a complete description of the preprocessor's functionality, its options and directives for increasing the effectiveness of the dependence analyzer and steering the code transformations. Finally, some advice is given for the practical use of **fpp** on CRAY computers.

INHALT

Vorwort

Mit dem Autotasking-System stellt CRAY erstmals ein autovektorisierendes und autoparallelisierendes Übersetzersytem zur Verfügung. Es ist größtenteils aus schon bekannten Komponenten zusammengesetzt, nur der FORTRAN-Präprozessor **fpp** stellt eine wirkliche Neuerung dar. Ziel ist dieser Schrift ist, grundlegende Sachverhalte zum Thema "Vektorisierung und Parallelisierung" darzustellen und konkrete Vorstellungen über die von **fpp** durchgeführten Analysen und Programmtransformationen zu vermitteln, um darauf aufbauend eine umfassende Benutzeranleitung zu **fpp** zu geben.

Die mit dem ersten Release verfügbare Herstellerdokumentation zum Autotasking-System (Handbuch [1] und man pages unter UNICOS 4.0) läßt hinsichtlich ihrer Korrektheit und Vollständigkeit einige Wünsche offen. Eine Reihe von Eigenschaften des Präprozessors mußte beim Verfassen dieser Schrift empirisch geklärt werden. Mit Erscheinen der Version CFT77 3.1, die ab UNICOS 5.0 lauffähig ist, besserte sich die Dokumentationslage [2]. Doch scheint es weiterhin sinnvoll, eine umfassende Beschreibung des **fpp** zu geben, die neben einer allgemeinen Einführung in die Thematik auch Hinweise für die praktische Arbeit mit dem Übersetzersystem enthält. Teile dieses Reports stehen seit Oktober 1989 im Online-Dokumentationssystem des ZIB zur Verfügung.

Vorausetzung für ein Verständnis ist Kenntnis des Konzepts der Pipeline-Verarbeitung in Vektorrechnern, wie auch des Problems der Vektorisierung von Programmen. Eine gute Einführung in dieses Gebiet findet sich in [19].

Vektorisierung und Parallelisierung werden in der vorliegenden Schrift als Programmtransformationen aufgefaßt, die die Reihenfolge von Anweisungen verändern. Diese Gemeinsamkeit hat zur Folge, daß Vektorisierung und Parallelisierung gleichartige Analysen und ähnliche Programmumformungen erfordern. Die gemeinsamen Aspekte der Vektorisierung und Parallelisierung werden im zweiten Kapitel relativ ausführlich behandelt. In den nachfolgenden Kapiteln liegt die Betonung etwas mehr bei der Vektorisierung.

Die Beschreibung ist in drei Teile gegliedert. Im ersten Teil (Kapitel 1 und 2) werden die Stellung und Funktion des **fpp** innerhalb des Autotasking-Systems erläutert und grundlegende Betrachtungen zu den Themen Datenabhängigkeitsanalyse, Vektorisierung und Parallelisierung angestellt. Im zweiten Teil (Kapitel 3 bis 5) wird eine ausführliche Beschreibung des **fpp** gegeben. Der dritte Teil (Kapitel 6 und 7) enthält praktische Hinweise und eine Besprechung derzeit bestehender Probleme.

Zum Thema Parallelisierung werden in absehbarer Zeit weitere On-Line-Dokumentationen bereitgestellt: eine Darstellung der von **fpp** erzeugten und von **fmp** genutzten Parallelisierungs-Direktiven [10], eine Beschreibung des Kommandos **cf77**, mit dem sich alle Komponenten des Autotasking-Übersetzersystems aufrufen lassen [11] und generelle Empfehlungen zum Thema Multitasking [12].

Lesehinweis

Für einen ersten Überblick genügt die Kenntnis der Kapitel 1, 3 und 6. Beim praktischen Einsatz des FORTRAN-Präprozessors sollte auch Kapitel 7 beachtet werden.

Kapitel 2 ermöglicht ein besseres Verständnis der von **fpp** durchgeführten Analysen und Programmumformungen. Die Kapitel 4 und 5 sind zum Nachschlagen gedacht und enthalten eine detaillierte Darstellung *aller* Optionen und Direktiven des Präprozessors.

1. Allgemeines zum Autotasking-Übersetzersystem

Der Präprozessor **fpp** bildet, wie auch **fmp** und **cft77**, einen Teil des FORTRAN-Übersetzersystems **cf77**, das eine automatische Parallelisierung ("Autotasking") ermöglicht. Zur Transformation sequentieller Programme in auf mehreren Prozessoren parallel ausführbare ist eine Analyse der Datenabhängigkeiten nötig. Auf Grundlage der hierbei gewonnenen Resultate kann eine automatische Parallelisierung vorgenommen werden, aber auch eine Verbesserung der Vektorisierung, sowie weitergehende skalare Optimierung erzielt werden. Die automatische Parallelisierung ist optional. Somit läßt sich das Übersetzungssystem auch zur Erzeugung optimierter, vektorisierter, nicht parallel ablauffähiger Programme nutzen. Dies macht das Übersetzersystem, insbesondere den Teil **fpp**, interessant für *jeden Anwender* mit FORTRAN-Programmen.

In den folgenden Kapiteln soll nur diese Komponente des Übersetzersystems beschrieben werden. Zur besseren Orientierung nun noch ein paar Worte zum Gesamtkonzept des Autotasking-Systems.

Die Grundlage des Übersetzer-Systems bildet eine Syntax zur Beschreibung von Parallelität in Programmen, die in einer imperativen Programmiersprache (zur Zeit nur FORTRAN) verfaßt sind [1]. Die Syntax ist (leider) herstellerspezifisch, da es noch keine entsprechenden Standards gibt. Der Programmtext wird zur Beschreibung vorhandener Parallelität, entweder automatisch oder vom Anwender selbst, durch Kommentare ergänzt, die dieser Syntax genügen. Das Verfahren ermöglicht einen modularen Aufbau des Übersetzersystems; darüber hinaus erlaubt es dem Anwender, den Übersetzungsprozeß in den verschiedenen Phasen zu verfolgen und durch Zugabe von Informationen gezielt zu unterstützen.

Das Übersetzersystem **cf77** besteht aus separaten Einheiten (vgl. Abb.1, Seite 6)

- zur Makro-Substitution, bedingten Übersetzung und Inklusion von Dateien
  (C preprocessor **cpp**, hier angewandt auf FORTRAN-Programme)
- zum Aufspüren der Parallelität und Umstrukturieren des Codes
  (FORTRAN preprocessor **fpp**)
- zum Erzeugen zusätzlichen FORTRAN-Codes für eine parallele Ausführung
  (FORTRAN multitasking translator **fmp**)
- zum Erzeugen des Objektcodes
  (CRAY FORTRAN compiler **cft77**)
- zum Erzeugen des ausführbaren Programms
  (link editor oder loader **ld** und segment loader **segldr**).

Der Präprozessor **fpp**, das eigentlich neue Produkt [2] im Autotasking-System, leistet im wesentlichen folgendes:

- Erzeugung von DO-Schleifen aus IF-Schleifen
- Ersetzung bestimmter Anweisungsfolgen durch Aufrufe hochoptimierter Bibliotheksroutinen
- Inline-Expandierung von Unterprogrammen

[1] Imperative Programmiersprachen bereiten etwa im Gegensatz zu funktionalen Programmiersprachen besondere Probleme bei der algorithmischen Aufdeckung von Parallelität in Programmen.

[2] Zum Präprozessor **fmp** gibt es schon im Microtasking-System, dem Vorläufer des Autotasking-Systems, ein Pendant (**premult**) ähnlicher Funktionalität.

- Vorbereitungen für eine verbesserte Vektorisierung
- Vorbereitungen für eine automatische Parallelisierung

fpp strukturiert den FORTRAN-Code zunächst um und spürt dann durch Analyse der Datenabhängigkeiten die innerhalb einzelner Programmeinheiten vorhandene Parallelität auf. Dabei beschränkt sich die Analyse auf den in numerischen Anwendungen häufigsten Typus von Parallelität: solche, die sich in DO-Schleifen manifestiert. Auch in unübersichtlichen Fällen wird für jede DO-Schleife festgestellt, ob die Datenabhängigkeiten eine Vektorisierung zulassen, oder ob die Schleifeniterationen auf unabhängigen Feldelementen arbeiten und somit parallelisierbar sind. Die Ergebnisse dieser Analyse werden zu weiteren Umstellungen im Code genutzt und schließlich in Form von Vektorisierungs- und Multitasking-Direktiven in der modifizierten FORTRAN-Quelle niedergelegt.

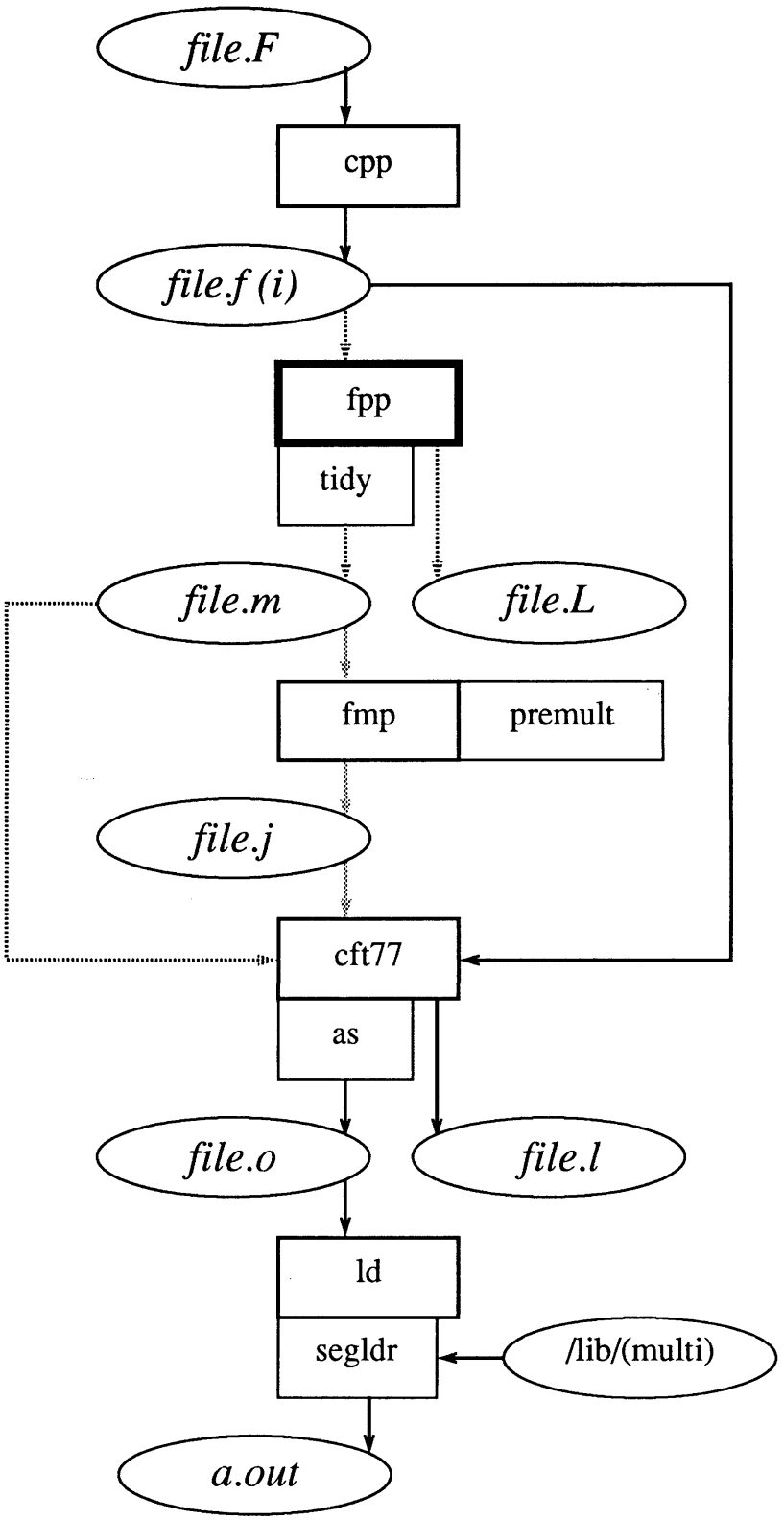
Bei den Programmumformungen wird aufgrund der spezifischen Architekturmerkmale von CRAY-Multiprozessor-Systemen der Vektorisierung Vorrang gegenüber der Parallelisierung eingeräumt. Alle Schleifen werden, sofern möglich, vektorisiert. Bei geschachtelten Schleifen wird, falls die Abhängigkeiten es erlauben, die innerste Schleife vektorisiert und die äußerste parallelisiert. Die Auswahl der zu vektorisierenden und zu parallelisierenden Schleife sowie die manchmal erforderliche Vertauschung von Schleifen erfolgt automatisch. Auf besonderen Wunsch werden auch vektorisierbare Schleifen mit hoher Iterationszahl in eine vektorisierte innere und parallelisierte äußere Schleife zerlegt. Auch nicht vektorisierbare innere Schleifen können in einer äußeren Schleife parallel abgearbeitet werden. Alle diese Transformationen lassen sich vom Anwender durch fpp-Direktiven steuern. Kann **fpp** aufgrund unzureichender statischer Informationen die zur Ausführungszeit sich einstellenden Datenabhängigkeiten nicht vorhersehen, hat der Anwender die Möglichkeit, über Direktiven gezielt weitere statische Information zur Verfügung zu stellen.

Das vom **fpp** erzeugte FORTRAN-Programm kann mit dem FORTRAN-Compiler **cft77** direkt zu Objektcode compiliert werden. Damit wird allein die durch Codeumstellungen und Einfügen von Compiler-Direktiven erzielte Verbesserung der Vektorisierung genutzt. Soll das Programm darüber hinaus parallelisiert werden, ist zuvor **fmp** auf die modifizierte FORTRAN-Quelle anzuwenden: damit werden vom **fpp** oder auch vom Anwender eingesetzte Autotasking-Direktiven in Aufrufe maschinenabhängiger Bibliotheksroutinen und in weiteren FORTRAN-Code umgesetzt und so eine parallele Ausführung des Programms ermöglicht.

Im folgenden wird zunächst eine allgemeine Einführung in die Thematik der Vektorisierung und Parallelisierung gegeben. Danach werden alle Optionen und Direktiven zur Steuerung des **fpp** erläutert. Besonders betont werden die für die Verbesserung der Vektorisierung relevanten Aspekte.

Eine generelle Empfehlung zur weitaus komplexeren Parallelisierung kann beim derzeitigen Stand der Software, wie auch aus betrieblichen Gründen (insgesamt verringerter Jobdurchsatz) nicht gegeben werden. Ausnahmen bilden Programme, die so viel Hauptspeicher benötigen (> 2 Mw auf der Berliner CRAY), daß häufig kein anderes der zur gleichen Zeit in Ausführung befindlichen Programme in den noch verbleibenden Teil des Hauptspeichers paßt und somit nur eine der CPUs beschäftigt wird: Programme, die aufgrund ihrer Hauptspeicheranforderungen CPUs "stillegen", sollten auf jeden Fall parallelisiert werden, um diese Verschwendung von Ressourcen zu vermeiden.

Abb.1: Aufbau und interner Datenfluß des Autotasking-Systems **cf77** (ohne Option →, mit Vektorisierungsoption -Zv ⋯▸ und mit Parallelisierungsoption -Zp ⋯▸)



2. Das Konzept der Datenabhängigkeit ([14] - [18])

2.1 Vektorisierung und Parallelisierung

In imperativen Programmiersprachen wird durch die Formulierung selbst eine Reihenfolge der Operationen vorgegeben, die bei der Ausführung der Programme im wesentlichen eingehalten wird. Zwar darf ein FORTRAN-Compiler die Reihenfolge der Operationen innerhalb einzelner Anweisungen beliebig wählen (unter Berücksichtigung der arithmetischen Vorrangregeln und der Klammerung), hinsichtlich der Abfolge mehrerer Anweisungen ist er aber an die im Programmtext vorgegebene Reihenfolge gebunden [13]. Rein theoretisch müßte diese Reihenfolge zur korrekten Bearbeitung von Programmen nicht unbedingt eingehalten werden: jede Reihenfolge liefert gleiche Resultate, vorausgesetzt die Vorrangbedingungen des Algorithmus bleiben erfüllt [1]. Die Vorrangbedingungen sind äußerst einfach: jede Operation darf erst dann ausgeführt werden, wenn die Werte aller Input-Variablen berechnet sind. Zu diesen, allein aus dem Algorithmus folgenden Abhängigkeiten kommen bei der Formulierung des Algorithmus in einer konventionellen Programmiersprache weitere hinzu. Solche "künstlichen" Datenabhängigkeiten können bei der Ausführung des Programmes im Prinzip ignoriert werden, allein die aus dem Algorithmus folgenden Ordnungsrelationen sind einzuhalten. Dieses Prinzip ist in sogenannten Datenfluß-Architekturen, bei denen die Programmausführung nicht "programmgetrieben", sondern "datengetrieben" ist, konsequent verwirklicht.

Bei der Ausführung von Programmen auf Multiprozessor-Vektorrechnern, wie der CRAY-XMP, spielt die Reihenfolge der Operationen eine ganz zentrale Rolle. Effiziente Nutzung der speziellen Rechnerarchitektur erfordert Vektorisierung und unter Umständen Parallelisierung der Programme. Diese Programmtransformationen bewirken eine *Veränderung der Operationsreihenfolge* gegenüber der im Sprachstandard definierten und vom (sequentiell denkenden) Programmierer intendierten Reihenfolge. Die Schleife

```
      DO 10 J = 1, N
      A(J)  = A(J) + X(J)
10    CONTINUE
```

wird zwar sowohl skalar wie auch vektoriell in der Reihenfolge

```
A(1)  = A(1)  + X(1)
A(2)  = A(2)  + X(2)
        . . .
```

bearbeitet, doch gibt es auf unterer Ebene einen Unterschied: bei vektorieller Ausführung werden erst alle rechts stehenden Operanden aus dem Hauptspeicher geladen, bevor irgendeine Abspeicherung der links stehenden Ergebnisse vorgenommen wird [2]. Die

[1] Wir sehen hier ab von der Ungültigkeit des Assoziativgesetzes bei Gleitpunktoperationen durch die Diskretisierung des Zahlenbereichs und die Rundungsoperationen; weiteres dazu in Abschnitt 4.2 und 5.3.

[2] In der Realität werden, wegen der begrenzten Länge der Vektorregister, die Vektoren nicht als Ganzes geladen, bearbeitet und abgespeichert, sondern jeweils in Teilen von maximal der Länge der Vektorregister (auf CRAY-Rechnern in Segmenten von maximal 64 Elementen).

Veränderung der Operationsreihenfolge bedeutet in diesem einfachen Beispiel keinen semantischen Unterschied und ist daher erlaubt. Das ist nicht generell der Fall: wird in einer Iteration ein Wert berechnet und in einer späteren Iteration wiederverwendet (Rekurrenz), so führt eine analoge Veränderung der Operationsreihenfolge zu anderen Resultaten, das heißt zu solchen Operationen gibt es keine semantisch äquivalenten Vektoroperationen. Beispielsweise läßt sich die sequentielle DO-Schleife

```
      DO 10 J = 1, N
      A(J + 1) = A(J) + X(J)
10    CONTINUE
```

nicht in die Vektoroperation (ab jetzt in FORTRAN-8X-Schreibweise dargestellt)

```
A(2:N+1) = A(1:N) + X(1:N)
```

übersetzen, da diese als Operanden nur die alten Werte von A verwenden würde. Die Veränderung der Operationsreihenfolge als Charakteristikum der Programmtransformationen Vektorisierung und Parallelisierung wird noch offensichtlicher, wenn die Schleife mehrere Anweisungen enthält. Beispielsweise wird die Schleife

```
      DO 10 J = 1, N
      B(J)  = A(J) + X(J)
      A(J)  = B(J) + C(J) * D(J)
10    CONTINUE
```

bei skalarer Ausführung in der Reihenfolge

```
B(1)  = A(1) + X(1)
A(1)  = B(1) + C(1) * D(1)

B(2)  = A(2) + X(2)
A(2)  = B(2) + C(2) * D(2)

   . . .
```

und bei vektorieller Ausführung hingegen in der Reihenfolge

```
B(1)  = A(1) + X(1)
B(2)  = A(2) + X(2)
   . . .
A(1)  = B(1) + C(1) * D(1)
A(2)  = B(2) + C(2) * D(2)
   . . .
```

ausgeführt [1]. Parallele Bearbeitung der DO-Schleife bedeutet auf der CRAY Verteilung

[1] Genauer: wegen der begrenzten Länge L der Vektorregister werden die beiden Vektorinstruktionen $\lceil N/L \rceil$ mal ausgeführt und zwar in der Reihenfolge B(1:64)=... , A(1:64)=... , B(65:128)=... , usw.

der einzelnen Schleifeniterationen auf verschiedene Prozessoren. Bei zwei Prozessoren etwa

Prozessor 1	Prozessor 2
B(1) = A(1) + X(1)	B(2) = A(2)+ X(2)
A(1) = B(1) + C(1) * D(1)	A(2) = B(2)+ C(2) * D(2)
B(3) = A(3) + X(3)	B(4) = A(4)+ X(4)
A(3) = B(3) + C(3) * D(3)	A(4) = B(4)+ C(4) * D(4)
...	...

Im Fall genügend vieler Prozessoren können die Rechenschritte somit für jeden Wert der Indexvariablen J auf einem anderen Prozessor durchgeführt werden. In der Praxis ist die Zahl der Iterationen jedoch meist größer als die Zahl verfügbarer Prozessoren, so daß jeder Prozessor mehrere Iterationen durchzuführen hat. Wichtig ist, daß die Verteilung der Aufgaben an die verschiedenen Prozessoren zur Laufzeit in Abhängigkeit vom Systemzustand vorgenommen wird. Beginn und Ende der Bearbeitung je einer Iteration hängen auch vom Systemzustand ab. Das heißt, bei paralleler Bearbeitung ist die zeitliche Abfolge der Iterationen, genereller, der parallelisierten Programmteile, *nicht deterministisch* [1], man spricht von *paralleler, asynchroner Ausführung*.

Das angegebene Beispiel ist eigentlich untypisch für Multiprozessor-Vektorrechner, da jede Iteration im Skalarmodus durchgeführt wird; in der Praxis werden solche Schleifen aus Effizienzgründen stets vektorisiert, nicht parallelisiert. Parallele Bearbeitung ist wegen des damit verbundenen Overheads nur für größere Programmteile sinnvoll, z.B. für Iterationen einer äußeren Schleife (CRAY-Terminologie: Auto- und Microtasking), oder für ganze Unterprogramme (Macrotasking) [2]. Realistische Beispiele für sinnvollerweise zu parallelisierende Programmteile würden mindestens eine Doppelschleife enthalten.

Nun ist klar, welche Schritte bei der Vektorisierung oder Parallelisierung, sei sie automatisch oder vom Anwender vorgenommen, durchzuführen sind: aus dem vorgegebenen Programmtext ist zu ermitteln, welche Datenabhängigkeiten vorliegen und ob sie die für eine Vektorisierung oder Parallelisierung notwendigen Veränderungen der Operationsreihenfolge zulassen. **fpp** nimmt diese Analysen vor, vermerkt Vektorisierungs- und Parallelisierungsmöglichkeiten im Programmtext und informiert über Datenabhängigkeiten, die diese Programmumformungen verhindern (sogenannte Abhängigkeitskonflikte).

Die Ermittlung der Datenabhängigkeiten in den angegebenen Beispielen ist trivial. Im allgemeinen kann sie jedoch sehr komplex werden und auch maschinell nicht immer durchführbar sein. Es muß dazu herausgefunden werden, ob an verschiedenen Stellen der Ausführung des Codes auf dasselbe Datenwort (lesend oder schreibend) zugegriffen wird.

[1] Dies gilt für Parallelbearbeitung auf CRAY-Multiprozessor-Systemen und ähnlichen Rechnerarchitekturen, jedoch *keineswegs generell*. Systolische Rechner arbeiten beispielsweise streng synchron.

[2] Man unterscheidet Parallelität *feiner, mittlerer* und *grober Granularität*, ungefähr gleichzusetzen mit Parallelität auf Anweisungs-, Schleifen- und Unterprogrammebene. In CRAY-Multiprozessor-Systemen wird beim Auto- und Microtasking hardwaremäßig (mittels Semaphor-Registern) und somit relativ schnell synchronisiert. Dies erlaubt Parallelisierung mittlerer Granularität; weiteres in [12].

Bei der Analyse von DO-Schleifen mit indizierten Feldzugriffen ist insbesondere zu ermitteln, ob verschiedene Indexkombinationen gleiche Elemente eines Feldes adressieren. Für lineare Indexausdrücke läuft dies auf die Lösung diophantischer Gleichungen hinaus, für nichtlineare Adressierung gibt es bisher kaum algorithmische Handhabe. Neben zahlentheoretischen Methoden sind auch solche der Symbolik zu verwenden, da die Werte der relevanten Variablen und Ausdrücke (z.B. Indexgrenzen und Inkremente) zur Übersetzungszeit im allgemeinen nicht bekannt sind.

Ein Hauptproblem der Abhängigkeitsanalyse ist mangelnde statische Information (siehe Abschnitt 2.4). In **fpp** sind daher Möglichkeiten vorgesehen, den Programmtext durch weitere Informationen in Form von Direktiven zu ergänzen. Können die Abhängigkeiten zur Übersetzungszeit nicht aufgeklärt werden, so ist doch oftmals erkennbar, daß nur wenige unterschiedliche Abhängigkeitsstrukturen eintreten können. Der Übersetzer kann in solchen Fällen für jede dieser Alternativen Code erzeugen, zusammen mit einem Test, der zur Laufzeit unter den Codevarianten auswählt. Sind solche Alternativen nicht erkennbar oder zu zahlreich, kann nur konservativ verfahren werden und Abhängigkeit angenommen werden.

Die ermittelten beziehungsweise sicherheitshalber angenommenen Abhängigkeiten lassen sich in Form gerichteter Graphen darstellen. Nach Aufbau des Abhängigkeitsgraphen reduziert sich die Prüfung der Vektorisierbarkeit und Parallelisierbarkeit von Programmteilen im wesentlichen auf graphen- und zahlentheoretische Fragestellungen.

2.2 Der Übersetzungsprozeß

Die Zielsprache des **fpp** ist FORTRAN-77. Der Präprozessor kann daher keinen vektorisierten oder parallelisierten Code erzeugen, sondern nur Information über Möglichkeiten zur Vektorisierung und Parallelisierung in Form von Kommentaren (Compiler-Direktiven und Autotasking-Direktiven, eingeleitet durch die Kürzel CDIR$ beziehungsweise CMIC$) im Quellprogramm niederlegen.

Der gesamte Übersetzungsprozeß läßt sich folgendermaßen gliedern [1]:

- Scanner-Parser-Phase (Übersetzung des eingegebenen Programms in einen abstrakten Syntaxbaum)
- Transformationsphase I (einfache Programmumformungen, z.B. Übersetzung von IF-GOTO-Schleifen in DO-Schleifen und Index-Standardisierung) [2]
- Abhängigkeitsanalyse
- Transformationsphase II (Vektorisierung und Parallelisierung)
- Code-Generierung

[1] Der interne Ablauf des **fpp** ist in der CRAY-Dokumentation nicht explizit beschrieben; er wurde aus der Funktionsbeschreibung und Fachliteratur zur Theorie parallelisierender Compiler erschlossen.

[2] Die vorbereitenden Transformationen machen sich nach außen hin nur dann bemerkbar, wenn **fpp** eine Umformung der Schleife für erforderlich hält und für diese neuen Code generiert.

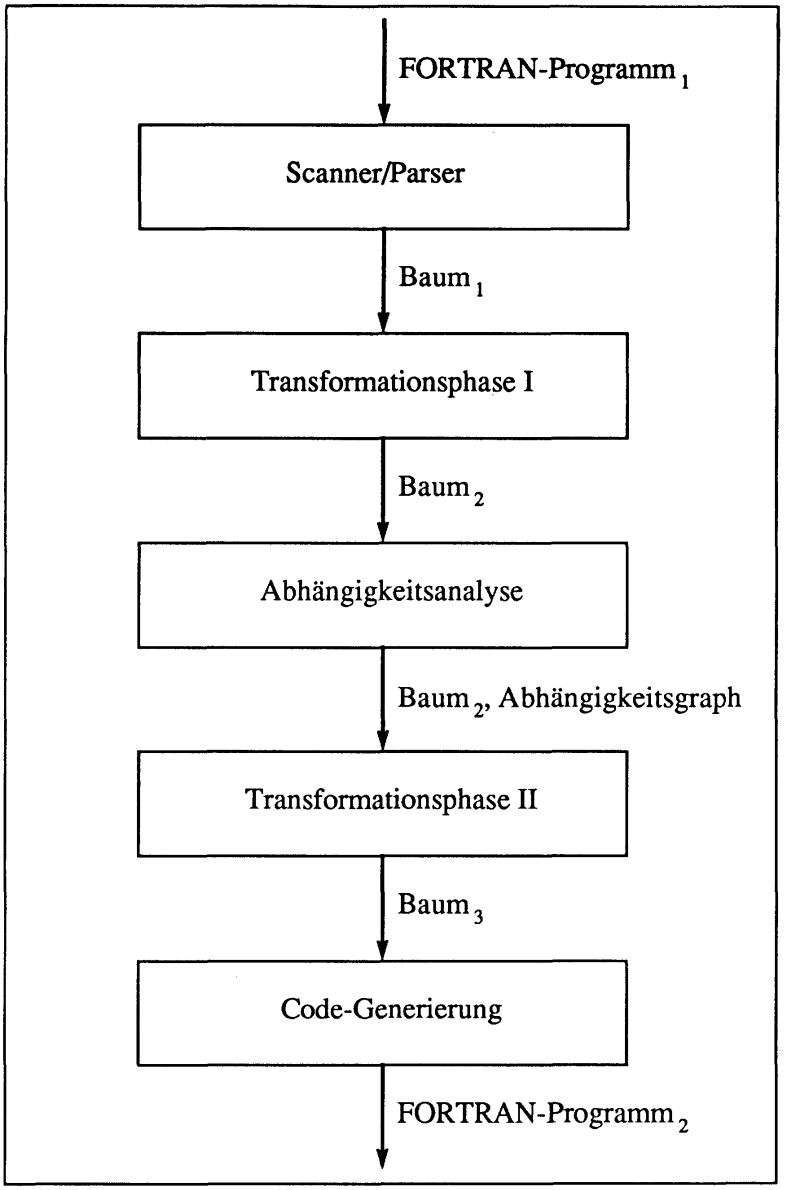


Abb.2 : Interner Ablauf im Präprozessor **fpp**

Ein Großteil der in numerischen Programmen vorhandenen Parallelität findet sich in Schleifen. **fpp** beschränkt sich im wesentlichen auf die Analyse von Schleifen. In der vorbereitenden Transformationsphase werden IF-GOTO-Schleifen in DO-Schleifen verwandelt und alle DO-Schleifen in kanonische Form gebracht, so daß die Laufvariablen von eins beginnend bis zu einer Obergrenze in Schritten von eins laufen. Weitere Induktionsvariablen (ganzzahlige Variablen, deren Werte bei Durchlauf der Schleife eine arithmetische Folge bilden) werden eliminiert. Dazu werden Ausdrücke, die solche Variablen inkrementieren, in lineare Ausdrücke der Laufvariablen und Schleifenkonstanten umgeformt und in die Feldzugriffe eingesetzt. Erst nach diesen Vorbereitungen wird eine Analyse der Abhängigkeiten in DO-Schleifen durchgeführt.

Obwohl die Datenabhängigkeitsanalyse und die Programmumformungen vom **fpp** automatisch vorgenommen werden, sind Grundkenntnisse zu den Themen "Datenabhängigkeiten" und "Programmtransformationen" nützlich. Einerseits ermöglichen sie ein Verständnis der von **fpp** ausgegebenen Meldungen zu den einzelnen Schleifen (die ja zum Teil Anlaß geben sollen zu weiteren Hilfestellungen durch den Anwender), andererseits eröffnen sie die Chance, auf höherem, heutigen Compilern noch nicht zugänglichem Niveau möglichst schon beim Entwurf oder der Auswahl von Algorithmen, günstige Vorbedingungen für eine Vektorisierung und Parallelisierung zu schaffen.

Zukünftige Softwaretools zur Programmierung paralleler Rechner (denen im Superrechnerbereich unbestritten die Zukunft gehört) werden das Konzept der Datenabhängigkeit an ganz zentraler Stelle verwenden.

2.3 Datenabhängigkeitstypen

Zur Feststellung aller Datenabhängigkeiten eines Programmteils genügt es, die Datenabhängigkeiten zwischen je zwei Anweisungen zu bestimmen. Mit jeder Anweisung werden die beiden Mengen

in(A) :== Menge der Input-Variablen der Anweisung A
out(A) :== Menge der Output-Variablen der Anweisung A

assoziiert. Somit sind bei der Betrachtung zweier Anweisungen A_i und A_j vier Mengen im Spiel; deren Schnittmengen bestimmen das Abhängigkeitsverhältnis der beiden Anweisungen. Von den vier möglichen Schnittmengen

$\cap$	$in(A_i)$	$out(A_i)$
$in(A_j)$	O	X
$out(A_j)$	X	X

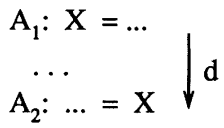
Tab. 1: Schnittmengen, die eine Datenabhängigkeit zweier Anweisungen definieren.

weisen alle auf eine Datenabhängigkeit hin. In unserem Zusammenhang sind nur die in Tab.1 mit X bezeichneten Schnittmengen relevant: *ist mindestens eine dieser drei Schnittmengen nicht leer, besteht eine Datenabhängigkeit zwischen den Anweisungen* A_i *und* A_j.

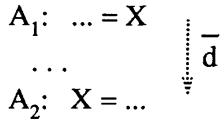
Zur Unterscheidung der drei Typen von Datenabhängigkeit definiert man [14]:

Definition: Kann der Kontrollfluß innerhalb eines Programms die Anweisung A_2 nach Passieren der Anweisung A_1 erreichen, dann *hängt A_2 von A_1 ab*, wenn

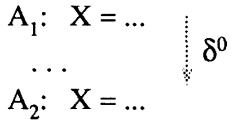
(1) $out(A_1) \cap in(A_2)$ nichtleer ist, d.h. wenn A_2 Ausgabedaten von A_1 benutzt. Dieser Abhängigkeitstyp heißt *(echte) Abhängigkeit*, geschrieben $A_1 \delta A_2$; er läßt sich wie folgt illustrieren



(2) $in(A_1) \cap out(A_2)$ nichtleer ist, d.h. wenn A_2 Eingabedaten von A_1 überschreibt. Dieser Abhängigkeitstyp heißt *Anti-Abhängigkeit*, geschrieben $A_1 \bar{\delta} A_2$; er läßt sich wie folgt illustrieren



(3) $out(A_1) \cap out(A_2)$ nichtleer ist, d.h. wenn A_2 Ausgabedaten von A_1 überschreibt. Dieser Abhängigkeitstyp heißt *Ausgabe-Abhängigkeit*, geschrieben $A_1 \delta^0 A_2$; er läßt sich wie folgt illustrieren



In allen drei Fällen ist die Reihenfolge der beiden Anweisungen nicht ohne weiteres vertauschbar, denn in

(1) benötigt A_2 zur Ausführung Eingabedaten, die von A_1 berechnet werden

(2) verwendet A_1 zur Ausführung Eingabedaten, die von A_2 überschrieben werden

(3) berechnet A_1 Größen, die von A_2 überschrieben werden.

Abhängigkeiten vom Typ 1 spiegeln die algorithmischen Vorrangrelationen wieder [1]. Nur dieser Abhängigkeitstyp ist Gegenstand der konventionellen Datenflußanalyse, die ermittelt, welche Anweisungen ausgeführt sein müssen, damit andere Anweisungen korrekte Eingabedaten erhalten.

[1] Versteht man unter Algorithmus eine Rechenvorschrift, die auch die Verwendung von temporären Variablen festlegt (etwa zur Vermeidung der Mehrfachberechnung gemeinsamer Teilausdrücke), sind algorithmische Vorrangrelationen und echte Abhängigkeiten *identisch*. Dürfen bei der Implementation des Algorithmus jedoch nach Belieben temporäre Variablen eingeführt werden (das ist der Normalfall), stellen die im Code vorhandenen echten Abhängigkeiten eine *Obermenge* der algorithmischen Vorrangrelationen dar.

Anti- und Ausgabeabhängigkeit entstehen nicht dadurch, daß Daten von einer Anweisung an eine andere übergeben werden, sondern durch mehrfache Verwendung eines Speicherplatzes für unterschiedliche Daten. Abhängigkeiten dieses Typs sind nicht durch den Algorithmus vorgegeben, sondern durch die Art der Programmierung und die Semantik der Programmiersprache[1]; sie werden daher auch als *Pseudoabhängigkeiten* bezeichnet . Durch Änderung von Variablennamen oder Kopieren von Daten lassen sich Pseudoabhängigkeiten oft eliminieren; bei der Vektorisierung und Parallelisierung müssen sie aber berücksichtigt werden.

Manchmal wird auch eine nichtleere Schnittmenge $in(A_1) \cap in(A_2)$ als Datenabhängigkeit (*Input-Abhängigkeit*) betrachtet [14]. Im Zusammenhang mit den Programmtransformationen für die Vektorisierung und Parallelisierung ist diese Art der Abhängigkeit jedoch weniger bedeutsam und bleibt im folgenden daher unberücksichtigt [2].

2.4 Abhängigkeitsanalyse

Die Verfahren zur Abhängigkeitsanalyse lassen sich streng formal abhandeln - ein Weg, der hier nicht beschritten wird. Glücklicherweise ist ein Verständnis der allgemeinen Vorgehensweise wie auch der dabei auftretenden Probleme anhand einfacher Beispiele möglich.

Von besonderem Interesse sind, wie schon mehrfach betont, die Datenabhängigkeiten in DO-Schleifen. Die *Vertreter* (instances) von Anweisungen in einzelnen Iterationen werden zur besseren Unterscheidung durch einen Index (Multiindex bei geschachtelten Schleifen) gekennzeichnet, der sich aus dem Wert der Laufvariablen ergibt. Die Indizes sind Elemente des sogenannten Iterationsraumes $\subset N^d$, wobei d die Schachtelungstiefe der Schleifen ist. Zu untersuchen sind Datenabhängigkeiten zwischen Vertretern einer Iteration wie auch unterschiedlicher Iterationen.

In dem Programmteil

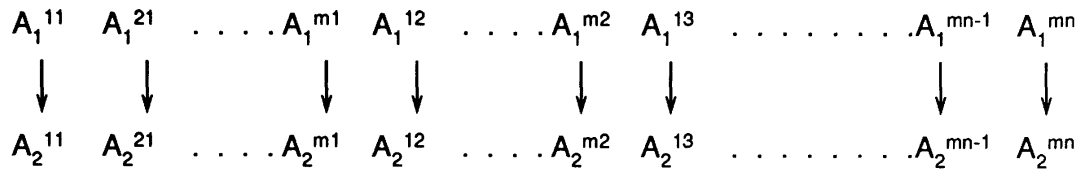
```
        DO 10 J = 1,N
        DO 10 I = 1,M
A1:     X(I,J) = Y(I,J) * C
A2:     Z(I,J) = X(I,J) + R
  10    CONTINUE
```

besteht der Abhängigkeitsgraph aus einzelnen Kanten, die keine unterschiedlichen Punkte im Iterationsraum verbinden, da Datenabhängigkeiten nur zwischen Vertretern

[1] Neben den konventionellen (Kontrollfluß-)Sprachen gibt es andere Sprachklassen, z.B. funktionale Programmiersprachen, die keine Pseudoabhängigkeiten zulassen und bei denen die Reihenfolge der Auswertung das Resultat nicht beeinflußt (wohl aber die Laufzeit). Solche auf dem Lamda-Kalkül von Church beruhende Sprachen sind vom theoretischen Standpunkt für eine Parallelisierung deutlich besser geeignet, in der Praxis aber mit anderen Problemen behaftet.

[2] Auch Input-Abhängigkeiten können eine Koordination erfordern: erfolgen mehrere lesende Zugriffe auf einen Speicherplatz so ist dabei normalerweise ein zeitlicher Mindestabstand einzuhalten. Die bei gleichzeitigem mehrfachem Lesen entstehenden "Konflikte" werden meist hardwaremäßig zur Laufzeit gelöst.

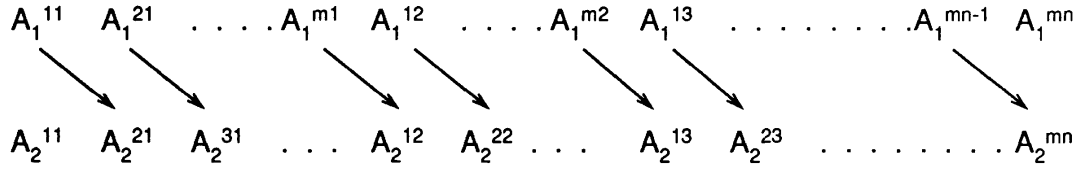
gleicher Iterationen bestehen:



Im leicht veränderten Beispiel

```
          DO 10 J = 1,N
          DO 10 I = 1,M
A1:       X(I,J) = Y(I,J) * C
A2:       Z(I,J) = X(I -1,J) + R
   10     CONTINUE
```

bestehen Datenabhängigkeiten zwischen Vertretern der beiden Anweisungen aus verschiedenen Iterationen:



Die Formulierung von Datenabhängigkeiten in Schleifen kann also durch gerichtete Graphen (Digraphen) geschehen, deren Knoten Vertreter von Anweisungen sind. Nicht immer wird solche detaillierte Information benötigt: für die Vektorisierung und Parallelisierung genügt es meist zu wissen, ob sich Abhängigkeiten über verschiedene Iterationen erstrecken und wie sie relativ zu der im Iterationsraum durch die sequentielle Ausführung vorgegebenen Richtung orientiert sind.

Im ersten Beispiel gilt für jede Iteration $A_1\ \delta\ A_2$. Da sich die Abhängigkeit nicht über verschiedene Iterationen erstreckt, schreibt man $A_1\ \delta_{=}\ A_2$. Auch im zweiten Beispiel gilt $A_1\ \delta\ A_2$, jedoch für Vertreter aus verschiedenen Iterationen: jeder Vertreter A_2^{ij} benötigt Eingabedaten, die in der vorigen Iteration von $A_1^{i-1\ j}$ berechnet wurden (mit der Ausnahme von A_2^{1j}, wo auf den "alten" Wert in X(0,J) zugegriffen wird). Da die Abhängigkeit von Iteration i -1 zu Iteration i reicht, schreibt man $A_1\ \delta_{<}\ A_2$. Im dritten Beispiel

```
          DO 10 J = 1,N
          DO 10 I = 1,M-1
A1:       X(I,J) = Y(I,J) * C
A2:       Z(I,J) = X(I +1,J) + R
   10     CONTINUE
```

benötigt A_2^{ij} ein Element von X, das von $A_1^{i+1\,j}$ überschrieben wird, das heißt, es liegt eine Antiabhängigkeit $A_2\ \bar{\delta}_{<}\ A_1$ vor.

Die Relationen < , = und > werden als *Datenabhängigkeitsrichtung* bezeichnet, da sie die Richtung der Abhängigkeitsrelation im Iterationsraum angeben. Bei geschachtelten Schleifen gibt es für jede Schleife eine Richtung; zusammengefaßt bilden sie den sogenannten *Datenabhängigkeits-Richtungsvektor*. Beispielsweise gelten in den Schleifen

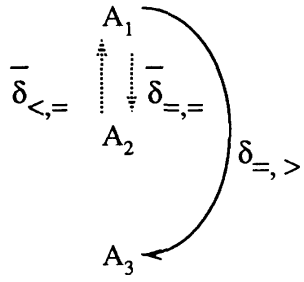
```
          DO 10 J = 2,N
          DO 10 I = 1,M
A1:       X(I,J) = Y(I,J) * A(I,J-1)
A2:       Y(I,J) = X(I+1,J) + R
A3:       A(I,J) = B + C
   10     CONTINUE
```

die Abhängigkeitsrelationen $A_2\ \bar{\delta}_{<,=}\ A_1$

$A_1\ \bar{\delta}_{=,=}\ A_2$

$A_1\ \delta_{=,>}\ A_3$.

Der Abhängigkeitsgraph der Anweisungen (nicht der einzelnen Vertreter) erhält damit die kondensierte Form



Durch Angabe von Richtungsvektoren können die Datenabhängigkeiten ohne wesentlichen Informationsverlust zwischen den Anweisungen selbst, das heißt ohne Aufschlüsselung nach Vertretern, formuliert werden. Oft läßt sich jedoch weitergehende Information nutzen: statt Vektoren, deren Komponenten nur die Richtung angeben, werden Abstandsvektoren zwischen den abhängigen Vertretern im d-dimensionalen Iterationsraum angegeben, sofern sie konstant sind [1].

[1] Generell sollten folgende Fälle unterschieden werden, da sie jeweils unterschiedliche Typen von Programmtransformationen zulassen:
(i) abstandserhaltende Abhängigkeiten (d.h. Abhängigkeiten mit einem für alle Iterationen konstanten Abstandsvektor im Iterationsraum)
(ii) richtungserhaltende Abhängigkeiten (d.h. Abhängigkeiten mit einem Abstandsvektor, dessen Komponenten im Laufe der Iterationen ihr Vorzeichen nicht wechseln)
(iii) allgemeine Abhängigkeiten mit beliebig variablem Abstandsvektor

Zur Bestimmung von Datenabhängigkeiten muß festgestellt werden, ob mehrmals auf ein und denselben Speicherplatz zugegriffen wird. Der Typ der Abhängigkeit ist dadurch bestimmt, welche Anweisung schreibend und welche lesend zugreift. Abhängigkeiten aller drei Typen können daher mit den gleichen Tests gefunden werden. Im folgenden wird daher nur der Fall echter Abhängigkeiten betrachtet.

Die Untersuchung von Datenabhängigkeiten in Schleifen erfordert eine Analyse der Indexausdrücke. Wir betrachten zunächst den Fall von DO-Schleifen mit nur einer Anweisung.

Die folgende Schleife

```
          DO 10 I = 1, N
A1:       X(I) = X(I) + C
  10      CONTINUE
```

ist vektorisierbar und parallelisierbar, da sich die Eingabevariable X(I) immer auf alte Werte an dem entsprechenden Speicherplatz bezieht (formal $A_1\ \delta_{=}\ A_1$). Anders bei

```
          DO 10 I = 1, N
A1:       X(I + 1) = X(I) + C
  10      CONTINUE
```

wo das Eingabedatum X jeder Iteration i+1 der in Iteration i berechnete Wert ist (Rekurrenz, $A_1\ \delta_{<}\ A_1$), was eine Vektorisierung und Parallelisierung verhindert [1]. Allgemein interessiert der Fall

```
          DO 10 I = 1, N
A1:       X( f (I) ) = F( X( g (I) ) )
  10      CONTINUE
```

wobei f und g beliebige Indexausdrücke und F eine beliebige Funktion eines Elementes von X sind. Ist F Funktion mehrerer Elemente von X, können diese bei der Abhängigkeitsanalyse nacheinander betrachtet werden. Die Fragen bei Bestimmung der Abhängigkeiten sind:

(i) Ist $in(A_1) \cap out(A_1)$ nichtleer, d.h. überlappen sich die links und rechts des Gleichheitszeichens angesprochenen Indexbereiche ?

(ii) Wie ist die Richtung der Datenabhängigkeit im Iterationsraum ?

Oder algebraisch: hat die *Abhängigkeitsgleichung*

$$f(i_1) - g(i_2) = 0$$

ganzzahlige Lösungen (i_1,i_2) in den Bereichen $1 \le i_1 < i_2 \le N$ oder $1 \le i_2 \le i_1 \le N$?

[1] Es gibt auch Vektorrechner, die eine Lösung von Rekurrenzen 1.Ordnung ($x_i = ax_{i-1} + c_i$ mit i=1,2,3,... und x_0=0) hardwaremäßig unterstützen; im allgemeinen ist dies jedoch nicht der Fall.

Besonders wichtig ist der Fall $1 \le i_1 < i_2 \le N$, da die Existenz von Lösungen in diesem Bereich eine Vektorisierung verhindert (siehe letztes Beispiel). Nur dieser Fall wird nun weiter betrachtet.

Für beliebige Funktionstypen ist dies ein schwieriges zahlentheoretisches Problem. Sind die Indexfunktionen jedoch linear mit ganzzahligen Koeffizienten (was glücklicherweise den häufigsten Anwendungsfall darstellt)

$$\mathbf{f}(i) = a_0 + a_1\, i$$
$$\mathbf{g}(i) = b_0 + b_1\, i$$

wird das Problem behandelbarer: gesucht sind Lösungen der diophantischen Gleichung

$$a_1\, i_1 - b_1\, i_2 = b_0 - a_0$$

im Bereich $1 \le i_1 < i_2 \le N$. Ein einfaches Ergebnis der Zahlentheorie besagt, daß diese Gleichung für ganzzahlige Koeffizienten genau dann ganzzahlige Lösungen hat, wenn $ggT(a_1,b_1)$ ein Teiler von $b_0 - a_0$ ist. Das Resultat liefert eine notwendige, aber keine hinreichende Bedingung für eine Abhängigkeit, da es keinen Aufschluß über die Existenz von Lösungen in dem eingeschränkten Bereich $1 \le i_1 < i_2 \le N$ gibt. Verfahren zur Untersuchung der Existenz von ganzzahligen Lösungen in einem eingeschränkten Bereich führen zu extrem aufwendigen Abhängigkeitstests. Daher werden meist einfachere, konservative Methoden verwendet, die notwendige Bedingungen für die Existenz *reeller* Lösungen in dem eingeschränkten Bereich testen.

Betrachtet man Schleifen mit mehreren Anweisungen

```
        DO 10  I = 1, N
                . . .
Ak:     X( f (I) ) = . . .
                . . .
Al:     . . .      = F( X( g (I) ) )
                . . .
10      CONTINUE
```

so interessiert zur Entscheidung der Vektorisierbarkeit neben der Existenz einer Lösung

$(i_1, i_2),\ 1 \le i_1 < i_2 \le N,$ mit $\mathbf{f}(i_1) = \mathbf{g}(i_2)$ *(iterationsüberschreitende Abhängigkeit)*

auch die einer Lösung

$i,\ 1 \le i \le N,$ mit $\mathbf{f}(i) = \mathbf{g}(i)$ *(iterationsgleiche Abhängigkeit).*

Die wichtigsten Verfahren lassen sich auf die Untersuchung von Schleifen mit mehreren

Anweisungen und auf geschachtelte Schleifen ausdehnen; für detailliertere Angaben zu den Algorithmen muß wieder auf die Fachliteratur verwiesen werden.

Zur Ermittlung der Datenabhängigkeiten ist auch der Kontrollfluß, d.h. alle möglichen Durchlaufwege und die sie bedingenden Anweisungen, zu analysieren. Beispielsweise ist in

```
      IF (M.GE.0) THEN
         DO 10 J = 1, N
         A(J) = A(J + M) . . .
10       CONTINUE
      ENDIF
```

nur bei Berücksichtigung der IF-Bedingung zu erkennen, daß keine Rekurrenz vorliegt.

Ein weiteres Problem ist die Ausschließung sogenannter *symbolischer Datenabhängigkeiten.* Das sind Fälle, bei denen die Indexausdrücke zur Compilationszeit nicht in Ausdrücke von Konstanten zerlegt werden können. Zu deren Behandlung sind Methoden der Computeralgebra heranzuziehen.

Das Problem mangelnder statischer Information läßt sich außer durch Verwendung konventioneller Techniken wie Konstanten-Propagation insbesondere durch eine *interprozedurale Abhängigkeitsanalyse* lindern. Beim Präprozessor **fpp** wird von einer einfacheren Möglichkeit Gebrauch gemacht, der *Inline-Expandierung*, das heißt der textuellen Ersetzung von Unterprogrammaufrufen durch das jeweilige Unterprogramm selbst. Dieses Verfahren hat folgende Vorteile:

- Indexausdrücke, Schleifengrenzen und Inkremente werden unter Umständen zu arithmetischen Ausdrücken, die nur Konstanten enthalten
- das Problem des Aliasing wird gelöst (bei der üblichen Vorgehensweise, nämlich der Sichtung jeweils nur eines Unterprogrammes, ist zur Compilationszeit nicht feststellbar, ob zwei als Parameter übergebene Felder identisch sind oder sich überlappen)
- es besteht die Möglichkeit, daß Schleifen mit Unterprogrammaufrufen vektorisierbar werden.

Nachteile der Inline-Expandierung sind ein unter Umständen stark anwachsender Codeumfang, sowie längere Compilationszeiten (weiteres zur konkreten Realisierung der Inline-Expandierung im **fpp** in Kap. 5.4).

Nach Aufbau des Datenabhängigkeitsgraphen wird dessen transitiver Abschluß gebildet. Der Graph gibt nun Auskunft über die Möglichkeit aller die Reihenfolge von Operationen verändernden Programmtransformationen, wie etwa Vertauschung von Anweisungen, Schleifen oder Schleifeniterationen zur Vektorisierung beziehungsweise Parallelisierung und viele andere optimierende Programmumformungen.

Für den Anwender ist es wichtig zu wissen, daß der von **fpp** aufgebaute Abhängigkeitsgraph aus besprochenen Gründen (z.B. mangelnder statischer Information oder Verwendung von Algorithmen, die nur hinreichende, aber nicht notwendige Bedingungen für Abhängigkeiten testen) nicht immer minimal ist. Das Defizit an statischer Information kann durch Verwendung von CFPP$-Direktiven ausgeglichen werden. Reicht dies nicht aus, so können in den von **fpp** erzeugten FORTRAN-Code auch Autotasking- und Compiler-Direktiven eingesetzt werden.

2.5 Programmtransformationen

Vektorisierung

Für die Vektorisierung einer Schleife sind zwei Teilfragen zu klären:

a) Gibt es in der Schleife Datenabhängigkeiten, die eine Vektorisierung verhindern ?

b) Lassen sich die in den Anweisungen der Schleife auftretenden Operationen durch Vektorinstruktionen der Zielmaschine ausdrücken ?

Die erste Frage wird im Autotasking-System vom **fpp** entschieden [1]:
- Schleifen ohne hinderliche Datenabhängigkeiten werden mit CDIR$ IVDEP gekennzeichnet
- Schleifen, die aufgrund von Datenabhängigkeiten definitiv nicht vektorisierbar sind, werden mit CDIR$ NOVECTOR gekennzeichnet
- Schleifen, bei denen die Datenabhängigkeiten nicht hinreichend gut geklärt werden können, werden nicht gekennzeichnet.

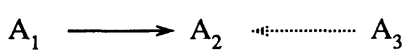
Die Frage der Umsetzung einzelner Anweisungen in Vektorinstruktionen entscheidet im Autotasking-System der **cft77**. Auf CRAY-Rechnern können alle elementaren und gängigen höheren Rechenoperationen, wie auch fast alle FORTRAN intrinsic functions (bis auf Funktionen zur Zeichenverarbeitung) durch Vektorbefehle ausgedrückt werden. Die tatsächliche Realisierbarkeit einzelner Operationen durch Vektorinstruktionen ist in numerischen Anwendungen daher kein Problem und wird im folgenden außer acht gelassen.

Eine hinreichende, aber nicht notwendige Bedingung für die Vektorisierbarkeit ist, daß der Datenabhängigkeitsgraph keine Rückwärtskanten besitzt (bei lexikalischer Orientierung). Dieses einfache Kriterium greift jedoch in vielen Fällen nicht. Deutlich weiter führt folgende, auch nicht notwendige Bedingung: *enthält ein Abhängigkeitsgraph keine einfachen Kreise* [2], *so kann die Schleife vektorisiert werden.*

Beispielsweise gehört zur Schleife

```
          DO 10  I = 1, N
A1:       A(I ) = X(I)
A2:       B(I ) = A(I) + Y(I)
A3:       C(I ) = B(I+1)
   10     CONTINUE
```

der Abhängigkeitsgraph



der zwar eine Rückwärtskante, aber keinen Kreis enthält. Somit kann die Schleife vollständig vektorisiert werden. Wegen der rückwärts gerichteten Abhängigkeit muß beim

[1] Aus historischen Gründen ist ein Teil dieser Funktionalität auch in den eigentlichen FORTRAN-Compilern cft77 und cft enthalten.

[2] Einfacher Kreis: geschlossener Kantenzug aus unterschiedlichen Kanten und Punkten.

Erzeugen von Vektorbefehlen die Reihenfolge der Anweisungen verändert werden: A_3 muß vor A_2 ausgeführt werden:

```
A1:     A(1:N) = X(1:N)
A3:     C(1:N) = B(2:N+1)
A2:     B(1:N) = A(1:N) + Y(1:N)
```

Gibt es Kreise im Abhängigkeitsgraphen, ist herauszufinden, ob sie sich aufbrechen lassen (wird später erklärt), oder ob sie einen bekannten Typ einer Reduktions- oder Rekurrenzoperation darstellen, die durch eine spezielle Folge von Vektorinstruktionen oder einen Unterprogrammaufruf ersetzt werden kann. **fpp** ersetzt beispielsweise bestimmte Formen linearer Rekursionen und bedingter Reduktionen durch Aufrufe optimierter SCILIB-Routinen. Die folgende Schleife

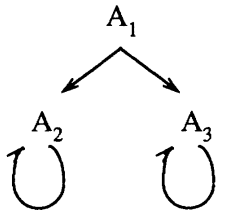
```
        DO 10 I = 1, N
A1:     S = S + A(I) * A(I)
   10   CONTINUE
```

hat als Abhängigkeitsgraph einen einfachen Kreis mit nur einem Knoten: $A_1 \; \delta \; A_1$. Die Reduktionsoperation wird sowohl vom **fpp** wie auch von den FORTRAN-Compilern erkannt. Der **cft77** ersetzt sie durch eine spezielle Folge von Vektorinstruktionen, der **cft** erzeugt einen Aufruf einer SCILIB-Routine (in beiden Fällen vektorisiert durch Teilsummenbildung).

Enthält der Abhängigkeitsgraph mehrere Kreise mit nur einem Knoten, ist die Schleife weiterhin vektorisierbar - falls es die einzelnen Anweisungen sind. Zur Schleife

```
        DO 10 I = 1, N
A1:     A(I) = X(I) * Y(I)
A2:     S = S + A(I)
A3:     AMAX = MAX ( AMAX, A(I) )
   10   CONTINUE
```

gehört der Abhängigkeitsgraph



Die Kreise werden als bekannte Reduktionsoperationen erkannt und Vektorcode der Art

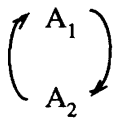
```
A1:     A(1:N) = X(1:N) + Y(1:N)
A3:     S = S + SUM(A(1:N))
A2:     AMAX = MAX ( AMAX, MAXVAL(A(1:N))
```

erzeugt (MAXVAL symbolisiere die Vektorfunktion "maximale Vektorkomponente").

Erstreckt sich ein *Kreis* in einem Abhängigkeitsgraphen über *mehrere Anweisungen*, kann die entsprechende DO-Schleife *nicht vektorisiert* werden. Beispielsweise gehört zur Schleife

```
          DO 10  I = 1, N
A1:       A(I) = X(I) + B(I -1)
A2:       B(I) = A(I) + S
   10     CONTINUE
```

ein Abhängigkeitsgraph bestehend aus einem Kreis mit zwei Knoten:

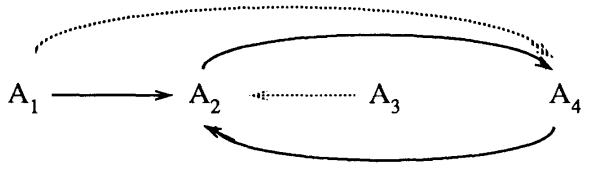


Die Anweisung A_2 benötigt in jeder Iteration als Eingabedatum das von A_1 berechnete Eingabedatum A(I), während A_1 ab der zweiten Iteration den von A_2 berechneten und B(I-1) zugewiesenen Wert benötigt. Die Schleife kann daher nur skalar ausgeführt werden.

Schleifen, die neben Anweisungen in Abhängigkeitskreisen noch weitere Anweisungen enthalten, lassen sich oft *teilweise vektorisieren*. Beispielsweise hat die Schleife

```
          DO 10  I = 1, N
A1:       A(I) = X(I)
A2:       B(I) = A(I) + X(I-1)
A3:       C(I) = B(I+1)
A4: 10    X(I) = B(I) + S
```

den Abhängigkeitsgraphen



in dem A_2 und A_4 einen Kreis bilden, also skalar bearbeitet werden müssen; die Anweisungen A_1 und A_3 lassen sich hingegen vektorisieren:

```
A1:       A(1:N) = X(1:N)
A3:       C(1:N) = B(2:N+1)
          DO 10  I = 1, N
A2:       B(I) = A(I) + X(I-1)
A4: 10    X(I) = B(I) + S
```

Das Beispiel zeigt, daß die maximalen Kreise[1)] im Abhängigkeitsgraphen als getrennte Schleifen behandelt und separat auf Vektorisierbarkeit untersucht werden können. Das

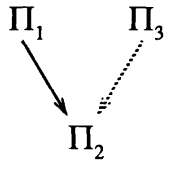
allgemeine Vorgehen ist also folgendes:

- Finde die maximale Kreise Π_1, Π_2, ... im (gerichteten) Datenabhängigkeitsgraphen.
- Bilde den zugehörigen kreisfreien (gerichteten) Abhängigkeitsgraphen; er induziert eine Halbordnung der maximalen Kreise (Blöcke)
- Untersuche die einzelnen Π-Blöcke auf Vektorisierbarkeit und erzeuge für sie Code in einer Reihenfolge, die die Halbordnung respektiert.

Im letzten Beispiel lauten die Blöcke

$$\Pi_1=\{A_1\} \quad \Pi_2=\{A_2, A_4\} \quad \Pi_3=\{A_3\}$$

und bilden den kreisfreien Abhängigkeitsgraphen

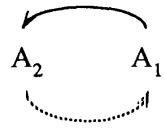


Skalarer bzw. vektorisierter Code ist in der Reihenfolge Π_1, Π_3, Π_2 oder Π_3, Π_1, Π_2 zu erzeugen.

Bisher wurde davon ausgegangen, daß sich Anweisungen, die Abhängigkeitskreise bilden, nicht vektorisieren lassen (bis auf wenige Formen der Reduktion und Rekursion). Tatsächlich lassen sich Pseudoabhängigkeiten, also Anti- und Ausgabeabhängigkeiten (vgl. Abschnitt 2.3) eliminieren, d.h. Abhängigkeitskreise, die solche Abhängigkeiten enthalten, lassen sich aufbrechen. Folgende Methoden finden dabei Verwendung:

- Indexmengen-Aufteilung
  In der nachfolgende Schleife

```
          DO 10  I = 1, 100
A1:       A(I) = B(101 - I) + S
A2:       B(I) = C(I)
   10     CONTINUE
```

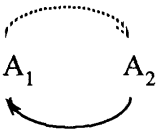


läßt sich durch Aufteilung der Indexmenge in zwei disjunkte Teilmengen aufbrechen:

[1] Jeder Knoten, der nicht in einem Kreis enthalten ist, bilde mit sich selbst einen maximalen Kreis.

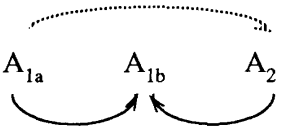
- Knoten-Aufteilung
  In gewissen Fällen läßt sich durch Einführung von Hilfsvariablen eine Knoten aufteilen und damit ein Abhängigkeitskreis aufbrechen. Beispiel:

```
          DO 10  I = 1, N
A1:       A(I) = ( B(I) + B(I+1) ) * 0.5
A2:       B(I+1) = C(I)
   10     CONTINUE
```

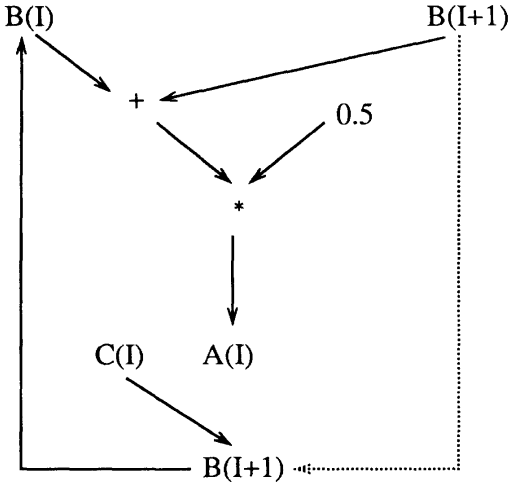


  Durch Kopieren des Feldes B in einen Hilfsvektor T ergibt sich:

```
          DO 10  I = 1, N
A1a:      T(I) = B(I+1)
A2:       B(I+1) = C(I)
A1b:      A(I) = ( B(I) + T(I+1) ) * 0.5
   10     CONTINUE
```



  Bei Vektorrechnern, die mit Vektorregistern ausgestattet sind, wie den CRAY-Rechnern, muß der temporäre Vektor nicht explizit erzeugt werden: die hinzugefügte Anweisung stellt einen Befehl zum Laden eines Vektorregisters dar - eine Operation, die sowieso durchgeführt wird. Die Transformation erfordert somit keinen zusätzlichen Aufwand.
  Tatsächlich handelt es sich von vornherein um ein Scheinproblem: betrachtet man statt des Abhängigkeitsgraphen auf Anweisungsebene einen auf Befehlsebene, so enthält dieser von vornherein keinen Kreis:



Das Beispiel zeigt, daß zur exakten Aufdeckung der Abhängigkeitsverhältnisse eigentlich die Abhängigkeitsgraphen auf Operationsebene betrachtet werden müßten. In der Praxis erweist sich jedoch, daß, abgesehen von Spezialfällen, die Analyse auf Anweisungsebene schon zu ausreichend genauen Abhängigkeitsgraphen führt.

Parallelisierung

Die Umformung von sequentiellem Code in parallel ausführbaren muß unter noch stärkerer Beachtung der Rechnerarchitektur der Zielmaschine geschehen, als die Vektorisierung. Hier soll keine Einordnung der von CRAY Research gewählten parallelen Rechnerarchitektur oder der Ablaufsteuerung zur Vergabe von Betriebsmitteln (scheduling) an einzelne parallel bearbeitbare Aufträge (tasks) gegeben werden. Nur die wichtigsten, bei der Parallelisierung von Programmen (für CRAY-Rechner) zu beachtenden Punkte seien nochmal zusammengefaßt:

- Feinkörnige Parallelität (d.h. solche auf Anweisungsebene) läßt sich mit CRAY-Rechnern wegen der hohen Interferenz beim Zugriff auf den gemeinsamen Speicher und wegen des hohen Kommunikations-Overheads nicht effizient nutzen.
- Grobkörnige Parallelität (d.h. solche auf Unterprogrammebene) kann mit CRAY-Rechnern gut genutzt werden (z.B. durch Macrotasking), erfordert aber eine interprozedurale Datenabhängigkeitsanalyse, die heute noch nicht maschinell durchgeführt werden kann. Ein weiteres Problem bei der parallelen Ausführung von Unterprogrammen ist die unterschiedliche Auslastung der Prozessoren: oft erweist es sich als schwierig, die Arbeit in gleich großen Portionen an die Prozessoren zu verteilen (load balancing) und zu verhindern, daß manche Prozessoren früher fertig werden und untätig auf andere warten.
- Im Auto- und Microtasking wird daher mittelkörnige Parallelität genutzt: bei geschachtelten Schleifen wird die am weitesten außen liegende, parallelisierbare Schleife parallel bearbeitet; diese Art von Parallelität ist in numerischen Codes häufig anzutreffen. Im allgemeinen steht der Kommunikations- sowie Synchronisationsaufwand in einem vernünftigen Verhältnis zum Gesamtaufwand. Weiterhin läßt sich die Datenabhängigkeitsanalyse wie auch die Erzeugung des parallel ablauffähigen Codes automatisch vornehmen (Autotasking). Die Vergabe von Iterationen der parallel zu bearbeitenden Schleife an die Prozessoren geschieht zur Laufzeit in Abhängigkeit vom jeweiligen Systemzustand und unterliegt nicht der Kontrolle des Anwenders. Der Scheduler ist damit in der Lage, 'idle times' von Prozessoren für Tasks zu nutzen (im günstigsten Fall werden mehr idle times genutzt, als für die Kommunikation und Synchronisation verbraucht wird - dies würde den Jobdurchsatz der Maschine sogar erhöhen) und erzielt durch die gute Portionierung der Tasks gutes Load Balancing.
- Wichtigste Ziele bei der Erzeugung parallelen Codes sind:
  Nutzung der spezifischen Stärken der Maschine, d.h. bei CRAY-Rechnern, der Vektorfunktionseinheiten; daher hat Vektorisierung Vorrang vor Parallelisierung: es muß bei der Code-Erzeugung als erstes versucht werden (eventuell nach Vertauschung von Schleifen), die innerste Schleife zu vektorisieren; per Voreinstellung wird nur die am weitesten außen liegende, parallelisierbare Schleife parallelisiert. Der Code ist so zu organisieren, daß Synchronisation vermieden, oder wenigstens Wartezeiten minimiert werden.
- Parallele Ausführung auf CRAY-Multiprozessorsytemen bedeutet asynchrone, nichtdeterministische Bearbeitung der Einzelaufträge.

Es läßt sich leicht einsehen, daß *asynchrone Bearbeitung* von Schleifeniterationen voraussetzt, daß *alle Datenabhängigkeiten dieser Schleife =-Richtung* haben, d.h. daß keine

iterationsüberschreitende Abhängigkeiten existieren. Dies garantiert, daß keine Daten zwischen den Prozessoren ausgetauscht werden müssen.

Beispiel: die Abhängigkeitsrelationen für die Schleife

```
          DO  30 I = 1, L
            DO 20  J = 1, M
              DO 10 K = 1, N
A1:             A(I,J,K-1) = B(I,J-1,K) + S
A2:             B(I,J,K) = A(I,J,K) + T
   10     CONTINUE
   20     CONTINUE
   30     CONTINUE
```

erlauben vektorielle Bearbeitung der innersten Schleife, parallele Bearbeitung der äußersten Schleife und sequentielle Bearbeitung der mittleren Schleife.

Alle Abhängigkeitsrelationen mit < oder >-Richtung müssen beim Parallelisieren explizit erfüllt werden. Wie bei der Vektorisierung können manche Abhängigkeiten aufgebrochen werden. Zusätzlich gibt es Methoden, Abhängigkeiten, die über Iterationen hinweg reichen, zu eliminieren. Abhängigkeiten, die nicht eliminiert werden können, erfordern eine Synchronisation der Prozessoren, die die verschiedenen Iterationen bearbeiten. Es gibt verschiedene Typen der Synchronisation; beim Autotasking kommt nur ein Typ, die Verwendung von kritischen Abschnitten (critical regions) vor. Kritische Abschnitte sind Codeblöcke, die zu jedem Zeitpunkt nur von einem Prozessor bearbeitet werden dürfen. Im Prinzip läßt sich mit ihnen jede Abhängigkeit erfüllen - im ungünstigsten Fall erstreckt sich der kritische Abschnitt jedoch auf das gesamte Codesegment.

Beispiel:

```
          SUM = 0.0
          DO  20 I = 1, L
            DO 10 J = 1,M
            B(J,I) = . . .
               . . .
   10       CONTINUE
            A(I) = . . .
            SUM = SUM + A(I)
   20     CONTINUE
```

Die äußere Schleife läßt sich parallelisieren, indem jede Task eine eigene Kopie der Variablen SUM erhält und zusätzlicher Code erzeugt wird, der nach Durchlauf der Schleife die Werte der verschiedenen Tasks aufsummiert. Das parallel bearbeitete Codesegment schließt diesen zusätzlichen Code mit ein; zur korrekten Aufsummation wird er als kritischer Abschnitt deklariert.

Eine ausführlichere Besprechung der Parallelisierung durch Autotasking findet sich in [10] und [12].

3. Aufruf des fpp

Der Präprozessor **fpp** kann entweder als Teil des **cf77** oder direkt aufgerufen werden. Der direkte Aufruf ist aus mehreren Gründen, insbesondere der einfacheren Bedienung wegen, vorzuziehen. Er hat die Form

fpp [options] [-o outputfile] [-l listingfile] file.f

Die Datei *file.f* enthält den zu bearbeitenden FORTRAN-Code. Die Ausgabe des von **fpp** modifizierten FORTRAN-Codes erfolgt nach *stdout*, wenn nicht mit -o eine Datei angegeben wurde. Ergebnisse der Abhängigkeitsanalyse, d.h. Informationen zur Vektorisierbarkeit und Parallelisierbarkeit, wie auch eine Gegenüberstellung des ursprünglichen und des modifizierten FORTRAN-Codes, werden in der mit -l angegebenen Datei abgelegt. Die restlichen Optionen *options* lassen sich gruppieren in solche zur

- Optimierung (-e, -d)
- Formatierung von *outputfile* durch das Programm **TIDY** (-r, -n)
- Ausgabe in *listingfile* (-p, -q)
- und sonstige (-C, -I, -M, -S, -T).

Die Laufzeiten des Präprozessors sind relativ gering (ca. 1.3 - 2.5 sec/1000 Zeilen Code). Allerdings benötigt er mindestens, d.h. auch zur Bearbeitung kleinster Programme, ca. 1.5 Mw Hauptspeicher.

4. SWITCH-Direktive und Optionen

4.1 SWITCH-Direktive

Die Optionen der ersten drei nachfolgend besprochenen Gruppen, also zur Optimierung, Ausgabesteuerung und Formatierung, können mittels der SWITCH-Direktive auch im Quelltext untergebracht werden und ermöglichen so eine lokale Steuerung. Die Direktive hat die Form

CFPP$ SWITCH, key1=string1 [,key2=string2]...

wobei die Schlüsselworte *key* die Werte OPTON, OPTOFF, LSTON, LSTOFF, TDYON und TDYOFF annehmen können und den Optionen -e, -d, -p, -q, -r und -n entsprechen. Ein weiteres Schlüsselwort, SPACE, erlaubt die Festlegung des von **fpp** zur Aufnahme temporärer Vektoren erzeugten COMMON-Blocks (Voreinstellung: SPACE=65535

Worte). Die Zeichenketten *string* setzen sich zusammen aus Schaltern ("switches"). Leerzeichen sind nicht signifikant, Schlüsselwörter und Schalter können entweder groß- oder kleingeschrieben werden.

Fast alle in den Abschnitten 4.2 und 4.5 angegebenen Schalter bzw. Optionen sind relevant für die Verbesserung der Vektorisierung; ausschließlich der Steuerung der Parallelisierung dienen die Optionen -e/-d 0, -e/-d i, -C und -T.

4.2 Optionen zur Optimierung

-d.., -e..	Die Optionen aktivieren (-e) bzw. unterdrücken (-d) verschiedene Optimierungen. Manche der nachfolgend angegebenen Schalter (a, c, d, e, r, u, v und 7) haben die gleiche Wirkung wie Direktiven (beispielsweise ist -d d äquivalent zu CFFP$ NODEPCHK F, s. Kap.5); diese Schalter können mittels der Direktive CFPP$ SWITCH innerhalb einer Routine mehrfach ein- und ausgeschaltet werden. Die anderen Schalter (b, k, l, o, s, 6 und 0) können nur einen gültigen Wert pro Routine haben; werden sie mehrfach gesetzt, so gilt jeweils der letzte Wert. Die Schalter x und 8 sind in der SWITCH-Direktive nicht erlaubt.	
	Schalter, (Voreinstellung)	Beschreibung des Schalters (beginnend mit dem Fall -e)

a (-e) Erlaubt assoziative Transformationen (Vertauschen der Reihenfolge von Operationen) beim Erzeugen vektorisierten und parallelen Codes; mit -d a werden assoziative Umformungen unterdrückt (äquivalent zur NOASSOC-Direktive mit globaler Gültigkeit). Das Assoziativgesetz ist in der Numerik durch die Diskretisierung des Zahlbereichs und die Rundungsoperationen im allgemeinen nicht gültig. Bei rundungsfehler-sensitiven Algorithmen können assoziative Transformationen zu veränderten Resultaten führen.

b (-d) Erzeugt Aufrufe von SCILIB-Routinen (FOLR, FOLR2, FOLRP, FOLR2P, FOLRC, SOLR und SOLR3) für bestimmte Formen linearer Rekursionen 1. und 2. Ordnung, s. [9].

c (-e) Automatische Parallelisierung (Autotasking) eingeschaltet; -dc ist äquivalent zur NOCONCUR-Direktive mit globaler Gültigkeit.

d (-e) Potentielle Datenabhängigkeiten (d.h. solche, über die **fpp** aufgrund der in der betreffenden Routine vorliegenden statischen Informationen nicht entscheiden kann) werden bei der Code-Generierung

Schalter, (Voreinstellung)	Beschreibung des Schalters (beginnend mit dem Fall -e)

nicht berücksichtigt. Wird diese Option ausgeschaltet (äquivalent zur NODEPCHK-Direktive mit globaler Gültigkeit), so muß sichergestellt sein, daß potentielle Abhängigkeiten keine wirklichen sind; dies kann die Performance mancher Programme verbessern.

e (-e) Bei der Abhängigkeitsanalyse werden EQUIVALENCE-Anweisungen berücksichtigt; -de ist äquivalent zur NOEQVCHK-Direktive mit globaler Gültigkeit.

i (-d) Innere Schleifen werden auf Parallelisierungsmöglichkeiten untersucht, sofern es keine äußeren Schleifen gibt. Normalerweise werden nur äußere Schleifen und innere, die offensichtlich genügend Rechenoperationen enthalten, parallelisiert. Bei inneren Schleifen mit hoher Iterationszahl und vielen Anweisungen kann -ei die Performance verbessern.

j (-e) Ersetzt (in der üblichen Weise programmierte) Matrix-Matrix- und Matrix-Vektor-Multiplikationen durch Aufrufe von SCILIB-Routinen (MXM, MXMA, SGEMM, DGEMM, CGEMM, SGEMV, DGEMV und CGEMV, s. [9]).

k (-e) Dient zur Kompatibilität mit anderen Compilern: Zeilen mit dem Buchstaben 'D' in der ersten Spalte werden als Kommentar aufgefaßt; bei -dk wird das Zeichen wie ein Leerzeichen behandelt.

l (-e) Transformiert mit IF und GOTO programmierte Schleifen in DO-Schleifen.

m (-e) Für Schleifen mit potentiellen Abhängigkeiten (siehe d-Schalter) werden alternativer Code (skalar/vektoriell bzw. seriell/parallel) und ein Laufzeittest erzeugt; bei -dm werden solche Schleifen nicht optimiert.

o (-d) Die minimale Anzahl von Schleifendurchläufen ist nicht 0, sondern 1 (für ANSI '66 FORTRAN Code).

q (-e) Beendet den **fpp**-Lauf mit Fehlerstatus, wenn Fehler gefunden werden.

r (-d) Alle Aufrufe von Anwender-Unterprogrammen (SUBROUTINES und FUNCTIONS) werden aus Schleifen herausgenommen und in separate, zusätzlich erzeugte Schleifen verlagert (äquivalent zur

Schalter, (Voreinstellung)	Beschreibung des Schalters, beginnend mit dem Fall -e

SPLIT-Direktive mit globaler Gültigkeit). Von der Verwendung dieser Option ist abzuraten, da ein Aufruf nur dann aus einer Schleife herausgenommen werden darf, wenn das betreffende Unterprogramm keine rekursiven Seiteneffekte hat. Empfohlen wird stattdessen die Verwendung der SPLIT-Direktive mit lokaler Wirkung - aber nur in sicheren Fällen und wenn die neu entstehende Schleife ohne Unterprogrammaufruf vektorisierbar wird.

s (-e) Erlaubt die Auftrennung von Schleifen in vektorisierbare Schleifen und solche, die rekursive und damit nicht vektorisierbare Code-Teile enthalten.

u (-e) Bei der Transformation von Schleifen entfällt manchmal die Verwendung im Original-Code enthaltener temporärer, skalarer Variablen, z.B. Induktionsvariablen. Wird der Wert solch einer Variablen nach Durchlauf der Schleife weiterverwendet, erzeugt **fpp** eine der modifizierten Schleife folgende Anweisung, die den Wert rekonstruiert. In manchen Fällen kann **fpp** nicht herausfinden, daß der Wert der eliminierten Variablen nicht benötigt wird und generiert dann überflüssigen Code. Dies kann durch eine NOLSTVAL-Direktive mit lokaler Wirkung unterbunden werden; die Verwendung von -du, äquivalent zur gleichen Direktive, aber mit globaler Wirkung, ist *nicht* zu empfehlen.

v (-e) Aktiviert die Verbesserung der Vektorisierung (geeignete Umstrukturierung von Schleifen und Kennzeichnung mit IVDEP-Direktiven). Bei -dv, äquivalent zur NOVECTOR-Direktive mit globaler Gültigkeit, haben die b-, m-, p-, r- und s-Schalter keinen Sinn.

x (-e) Erzeugt eine optimierte Quelle; ist x ausgeschaltet, so liefert **fpp** keine transformierte Quelle, sondern nur die üblichen Meldungen im Listingfile (anzugeben mit der Option -l).

y (-e) Nur die umstrukturierten Schleifen werden bei der Ausgabe der modifizierten Quelle neu formatiert; bei -dy wird die gesamte Quelle vom Programm **TIDY** neu formatiert (s. Abschnitt 4.4).

0 (-e) Wenn möglich, entscheidet **fpp** schon beim Compilieren, ob sich eine Parallelisierung lohnt, d.h. ob in einem parallelisierbaren Programmteil genügend Rechenoperationen vorhanden sind, um den mit paralleler Ausführung verbundenen Overhead zu kompensier-

Schalter, (Voreinstellung)	Beschreibung des Schalters, beginnend mit dem Fall -e

en. Ist diese Entscheidung zur Compilationszeit nicht möglich, erzeugt **fpp** Code für beide Fälle und für einen Test, der die Entscheidung zur Laufzeit trifft. Mit -e0 ist die Generierung dieses Laufzeittests eingeschaltet.

6 (-d) Unterprogramme, die gewisse Kriterien erfüllen (s. Abschnitt 5.5), werden "inline expandiert", d.h. die Aufrufe der Routinen werden durch die Unterprogramme selbst ersetzt. Im Gegensatz zur Option -e7 sollten hierbei keine Fälle auftreten, die zu inkorrektem FORTRAN-Code führen.

7 (-d) Automatische "inline-Expandierung", jedoch mit gegenüber der Option -e6 abgeschwächten Kriterien. In seltenen Fällen (im Zusammenhang mit variabler Felddimensionierung) wird dabei falscher Code erzeugt. -e7 ist äquivalent zur AUTOEXPAND-Direktive mit globaler Gültigkeit.

8 (-d) Die Datei mit dem FORTRAN-Code des Anwenders wird in einem vorgeschalteten Durchlauf nach expandierbaren Routinen durchsucht. Bei der Voreinstellung -d8 werden Routinen in gleichnamigen, *klein*-geschriebenen Dateien gesucht, z.B. eine Routine x oder X in der Datei x.f. Für Alternativen vgl. die (AUTO)EXPAND- und SEARCH-Direktiven in Abschnitt 5.5.

4.3 Optionen zur Ausgabesteuerung

-p.., -q..	Diese Optionen aktivieren (-p) bzw. unterdrücken (-q) die Ausgabe verschiedener Informationen im Listingfile. Voraussetzung für die Erstellung eines Listingfiles ist die Angabe eines Dateinamens mit der Option -l.

Schalter, (Voreinstellung)	Beschreibung der Option -p
b (-p)	Listet die Zeilennummern des Input-Files in den Spalten 73-80 der FORTRAN-Output-Zeilen im Listing; nur wirksam, wenn Schalter n aktiviert ist.
c (-p)	Listet die Datenabhängigkeits-Konflikte.
d (-q)	Listet von **fpp** hinzugefügte Deklarationen.
e (-p)	Listet eine Zusammenfassung zu jeder Routine.
f (-p)	Listet Fehlermeldungen.
g (-p)	Listet diagnostische Übersetzermeldungen.
h (-p)	Listet das Eingabe-Quellenfile.
i (-p)	Listet über INCLUDE dazugenommene Zeilen (durch einen Bindestrich nach der Zeilennummer gekennzeichnet).
l (-p)	Erzeugt ein Listing; -ql ist äquivalent zur NOLIST-Direktive.
n (-p)	Listet den modifizierten Quellencode.
p (-p)	Listet Kommentare zu den DO-Schleifen am Ende jeder Routine.
s (-q)	Listet nur die Zusammenfassung; falls s eingeschaltet ist, sollten die Schalter c, d, e, f, g, h, l, n, o, p, w und y nicht verwendet werden.
t (-p)	Listing für ein Terminal (80 Spalten); bei -qt wird die Liste 132 Spalten breit, erhält eine Seitennumerierung und Zeichen zur Steuerung des Zeilendruckers.

Schalter, (Voreinstellung)	Beschreibung der Option -p

u (-q) Zeigt Umfang und Anordnung von DO-Schleifen.

w (-p) Listet Warnungen.

y (-p) Listet Syntax-Fehler.

4.4 Optionen zur Formatierung

-r.., -n..	Diese Optionen aktivieren (-r) bzw. unterdrücken (-n) verschiedene Formatier-Optionen für die Ausgabe des FORTRAN-Output-Files. Die transformierte Quelle wird vom Programm TIDY transformiert. Wir geben an dieser Stelle nur kurz die Optionen zur Steuerung des Formatiervorganges an; eine ausführlichere Beschreibung der Optionen und der hier nicht besprochenen Steuerungsparameter findet sich in Ref. [2].

Schalter, (Voreinstellung)	Beschreibung der Option -r
a (-r)	Zeilenkommentare werden vor die Anweisung gezogen, wenn sie nach Formatierung nicht mehr in die Zeile passen.
b (-n)	Setzt hinter das Komma eines Index ein Leerzeichen, z.B. A(I, J).
c (-r)	Setzt hinter Kommata von Parameter-, Eingabe- und Ausgabelisten Leerzeichen, z.B. CALL X(A, B, N).
e (-r)	Umgibt Gleichheitszeichen mit Kommata.
f (-n)	Verlagert alle FORMAT-Anweisungen an das Ende der Routinen vor die END-Anweisung.
h (-r)	Anweisungsnummern erscheinen nur in CONTINUE- und FORMAT-Anweisungen; Sprungziele und Abschlußanweisungen von Schleifen werden umgewandelt in CONTINUE-Anweisungen mit Anweisungsnummern.
j (-n)	** und //-Operatoren werden mit Leerzeichen umgeben.
k (-n)	Nur die Operatoren .AND., .OR. werden von Leerzeichen umgeben; auf IF-Anweisungen haben die k-, o- und q-Schalter unterschiedliche Wirkung; nur einer davon sollte jeweils eingeschaltet sein.
l (-n)	Der gesamte FORTRAN-Output, bis auf die Deklarationen, wird in Kleinschreibung ausgegeben. Auch **fpp**-Deklarationen werden kleingeschrieben; Anwender-Deklarationen werden unverändert übernommen.

Schalter, (Voreinstellung)	Beschreibung der Option -r

m (-r) Zweizeilige Anweisungen werden, sofern möglich, unter Vernachlässigung der Leerzeichen-Regeln, auf eine Zeilenlänge komprimiert.

n (-n) Klammern, die keine Indizes umschließen, werden mit Leerzeichen umgeben.

o (-n) Alle logische Operatoren werden mit Leerzeichen umgeben (siehe Schalter k und q).

p (-r) + und - werden von Leerzeichen umgeben.

q (-r) Alle logische Operatoren, die allein in einer IF-Anweisung auftreten, werden mit Leerzeichen umgeben, andernfalls nur die logischen Operatoren .AND., .OR., .EQV. und .NEQV. (siehe auch Schalter k und o).

r (-n) Erzeugt für jede Programm-Einheit Kommentare zu allen externen Referenzen und fügt diese vor der ersten ausführbaren Anweisung ein.

s (-n) Leerzeichenregeln für Operatoren (vgl. j-, p- und t-Schalter) werden innerhalb von Klammern befolgt.

t (-n) Die Operatoren * und / werden mit Leerzeichen umgeben.

u (-n) Ein- und Ausgabeanweisungen werden mit den Schlüsselworten UNIT= und FMT= versehen.

v (-n) Gleichheitszeichen in Schlüsselwort-Listen werden mit Leerzeichen umgeben.

x (-n) Setzt RENUMB = 100:10, FORMAT = 900:10, TDYON=R. RENUMB und FORMAT sind Formatierparameter ohne äquivalente Kommandozeilen-Option; TDYON=R entspricht -r r.
Weitere Informationen in [2].
Von einer Verwendung der Option -rx ist derzeit noch abzuraten, sie führt in einzelnen Fällen zu fehlerhaft formatierten Quellen.

y (-r) Nur transformierte Programmblöcke werden formatiert, der Rest des Programms wird unverändert wiedergegeben.

4.5 Sonstige Optionen

Die nachstehenden Optionen werden erst ab UNICOS 5.0 mit der CFT77-Version 3.1 verfügbar.

Option und Argumente	Beschreibung der Option
-C r1,r2,..	Angabe parallel aufrufbarer Routinen r1, r2, ...
-I r1,r2,..	Angabe von Unterprogrammen r1, r2, ..., die inline-expandiert werden sollen.
-M lines.	Definiert die maximale Anzahl von Programmzeilen der Routinen, die inline-expandiert werden sollen; Voreinstellung ist 50.
-S name1, name2 ...	Angabe von Dateien, die Unterprogramme zum Inline-Expandieren enthalten.
-T threshold	Legt den Schwellenwert für den Umfang an Rechenoperationen fest, ab dem Autotasking, d.h. automatische Parallelisierung durchgeführt wird (Voreinstellung: *threshold*=800). Es werden nur solche Schleifen parallelisiert, die genügend Rechenoperationen enthalten, um den mit der parallelen Abarbeitung verbundenen Overhead zu kompensieren. Zur Durchführung eines entsprechenden Tests zählt **fpp** die Operationen in der für Parallelisierung vorgesehenen Schleife und gewichtet die Zahl der Operationen mit den Faktoren 1 (+, -, *), 2 (indirekte Adressierung), 3 (mod, /) und 12 (math. Funktionen). Ist die Anzahl der Schleifendurchläufe größer als der Quotient aus der so gewonnenen Zahl und dem Schwellenwert, so wird Autotasking durchgeführt. Kennt **fpp** die Anzahl der Schleifeniterationen, wird der Test sofort ausgewertet, andernfalls wird ein Laufzeittest erzeugt. Der Test wird nur vorgenommen, wenn Autotasking eingeschaltet ist (Voreinstellung -ec) und die Erzeugung des Tests nicht durch -d0 oder die Direktive CFPP$ SELECT(CONCUR) unterdrückt ist. Die Schwellenwerte können für einzelne Schleifen nicht separat festgelegt werden, jedoch ist es mit CFPP$ SWITCH,THRESH=threshold möglich, Schwellenwerte für einzelne Routinen zu definieren.

5. Beschreibung der fpp-Direktiven

5.1 Typen von Direktiven

fpp stehen nur die zur Übersetzungszeit bekannten und daraus deduzierbaren statischen Informationen zur Verfügung. Weiterhin betrachtet **fpp** zu jedem Zeitpunkt nur einzelne Programmeinheiten. Durch CFPP$-Direktiven hat der Anwender die Möglichkeit, **fpp** zusätzliche Informationen zu geben. Die CFPP$-Direktiven enthalten Erklärungen, die zur Aufdeckung von Parallelität nützlich sein können und ermöglichen eine Steuerung der von **fpp** durchgeführten Transformationen.

fpp interpretiert und erzeugt zur Vektorisierung CDIR$-Compiler-Direktiven und (optional) zur Parallelisierung CMIC$-Direktiven. Auch der Anwender hat die Möglichkeit, durch CMIC$-Direktiven mitzuteilen, wo Parallelität vorliegt und wie sie genutzt werden soll.

Im Zusammenhang mit der Anwendung von **fpp** treten also insgesamt drei Typen von Direktiven auf, deren Wirkung und Funktion in der nachfolgenden Tabelle aufgelistet sind:

Typ	**fpp-**	Wirkung, Funktion des Direktiventyps	Beschreibung
CFPP$	Input	Werden von **fpp** interpretiert.	Kap.5.2 und Refs.[1], [2]
CDIR$	Input	Folgende Direktiven werden von **fpp** interpretiert: CDIR$ IVDEP wirkt wie CFPP$ NODEPCHK L CDIR$ VECTOR " CFPP$ VECTOR R CDIR$ NOVECTOR " CFPP$ NOVECTOR R CDIR$ SHORTLOOP " CFPP$ NOINNER L Die anderen CDIR$-Direktiven werden von **fpp** nicht interpretiert und bleiben unangetastet - mit einer Ausnahme: CDIR$-Direktiven in Unterprogrammen gehen bei der Inline-Expansion dieser Module verloren !	Ref.[4]
	Output	Zusätzlich zu den schon vorhandenen CDIR-Direktiven setzt **fpp** weitere ein: alle als vektorisierbar erkannten Schleifen werden durch CDIR$ IVDEP gekennzeichnet und definitiv nicht-vektorisierbare Schleifen mit CDIR$ NOVECTOR- und CDIR$ VECTOR-Direktiven geklammert.	
CMIC$	Input	Schon vorhandene, vom Anwender eingefügte CMIC$-Direktiven veranlassen **fpp**, die Analyse der betreffenden Routine abzubrechen, also keine Optimierung durchzuführen. Autotasking, Microtasking und Macrotasking können in einem Programm koexistieren, jedoch nicht in einer Unterprogramm-Einheit.	Ref. [10] und [1], [2], [7], [8]
	Output	Die Syntax zur Formulierung von Parallelität ist mittels CMIC$-Direktiven realisiert. Dazu werden ein Teil der älteren Microtasking-Direktiven neben neu eingeführten Direktiven, den sogenannte *Autotasking-Direktiven*, verwendet. Die von **fpp** gewonnene Information zur (außer durch Vektorisierung) nutzbaren Parallelität wird in Form von CMIC$-Direktiven im Code niedergelegt.	

Tab.2: Wirkung der im Zusammenhang mit **fpp** auftretenden Direktiventypen.

5.2 Format der CFPP$-Direktiven

Die CFPP$-Direktiven haben, bis auf die schon behandelte SWITCH-Direktive und einige nachfolgend beschriebenen Direktiven, das Format

CFPP$ Direktive [X]

Das C in der ersten Spalte macht die Anweisung zu einem Kommentar für alle anderen FORTRAN-Übersetzer, erhält also die Portabilität der Programme. Der Geltungsbereich der Direktive kann durch Angabe des Parameters X gesteuert werden:

Wert	Bedeutung	Geltungsbereich
F	File	bis zum Ende der Eingabe-Datei
R	Routine	bis zum Ende der aktuellen Routine
L	Loop	in der nächsten Schleife
keine Angabe		in der nächsten Schleife

Tab. 3: Angabe des Geltungsbereichs bei CFPP$-Direktiven

Bei manchen Direktiven ist der Bereichs-Parameter nicht sinnvoll und wird ignoriert; Direktiven zur Beeinflussung von IF-Schleifen müssen R oder F-Geltungsbereich haben. Die Funktion der meisten Direktiven läßt sich durch Voranstellen der Silbe NO umkehren. Die Direktiven PERMUTATION, RELATION, SELECT, SEARCH, SWITCH und EXPAND haben eine abweichende Syntax (wird an jeweiliger Stelle angegeben).

Die CFPP$-Direktiven lassen sich klassifizieren in solche zur direkten Beeinflussung der Programmtransformationen, zur Angabe von Datenabhängigkeiten, zur Steuerung des Listings und die SWITCH-Direktive zum Setzen oder Ändern von Optimierungs-, Ausgabe- und Formatier-Schaltern.

5.3 CFPP$-Direktiven zur Beeinflussung der Transformationen

Nachfolgende Direktiven dienen zur Steuerung der an Schleifen vorgenommenen Umformungen.

Direktive	Voreinstellung	Funktion	Bereich
VECTOR NO...	VECTOR	Aktiviert bzw. unterdrückt die Verbesserung der Vektorisierung. Nutzbar zur gezielten Unterdrückung der Vektorisierung einzelner Schleifen, die **fpp** aufgrund statischer Informationen als vektorisierbar kennzeichnet, obwohl sie skalar schneller ablaufen.	LRF
CONCUR NO...	CONCUR	Aktiviert bzw. unterdrückt die Analysen und Transformationen zur automatischen Parallelisierung.	LRF
SKIP	keine	Wirkt wie die Kombination NOVECTOR und NOCONCUR.	LRF
INNER NO...	NOINNER	Ermöglicht die Parallelisierung innerer Schleifen.	LRF
CNCALL	keine	Teilt mit, daß im Geltungsbereich keines der in Schleifen aufgerufenen Unterprogramme rekursive Seiteneffekte hat und somit zur parallelen Durchführung der Schleifeniterationen von verschiedenen Prozessoren aufgerufen werden darf.	LRF
CHOP_HERE	keine	Weist **fpp** an, die Schleife an der Stelle dieser Direktive zweizuteilen (für das Fine-Tuning von Schleifen).	---
ALTCODE NO...	ALTCODE	Weist **fpp** an, für Schleifen, über deren Datenabhängigkeiten nicht entschieden werden kann, eine skalare und eine vektorielle Version zu erzeugen, sowie einen Laufzeittest, der zwischen diesen Versionen auswählt. Für parallelisierte Schleifen bewirkt ALTCODE die Erzeugung eines Schwellenwerttests in der IF-Abfrage der CMIC$-Direktiven DO ALL oder DO PARALLEL (vgl.	LRF

Direktive	Voreinstellung	Funktion	Bereich
		Option -T in 4.5 und Ref. [2]). Die Option -dm ist äquivalent zu CFPP$ NOALTCODE F. Bei ALTCODE kann, durch mindestens ein Leerzeichen getrennt, ein Parameter angegeben werden. Ist dieser eine ganzzahlige Größe, so erzeugt **fpp** einen Test, der die Zahl der Schleifendurchläufe mit der abgegebenen Konstanten vergleicht; im anderen Fall wird die angegebene Zeichenkette als logischer Ausdruck im erwähnten IF-Test verwendet.	
ASSOC NO...	ASSOC	Erlaubt beim Erzeugen vektorisierten und parallelisierten Codes Transformationen, die die Reihenfolge von Operationen vertauschen (vgl. Option -ea in Abschnitt 4.2).	LRF
SPLIT NO...	NOSPLIT	Teilt **fpp** mit, daß Unterprogrammaufrufe keine Seiteneffekte haben, die eine Verwendung von Resultaten aus einer Schleifeniteration in einer nachfolgenden bedingen. Solche Aufrufe können in eine separate Schleife verlagert werden und die verbleibende Schleife kann optimiert werden.	LRF
SELECT	keine	Weist **fpp** an, von mehreren geschachtelten Schleifen die nächste zu vektorisieren oder zu parallelisieren, gesteuert durch ein optionales Argument, VECTOR oder CONCUR; Voreinstellung ist VECTOR. Beim Auswählen der zu parallelisierenden Schleife aus mehreren geschachtelten Schleifen gewichtet **fpp** nach einem heuristischen Algorithmus die Faktoren Iterationszahl, Datenabhängigkeiten und Umfang der Rechenarbeit in der Schleife. Da zur Compilationszeit nicht immer alle erforderlichen Information vorhanden ist, trifft **fpp** manchmal eine ungünstige Wahl. Die SELECT-Direktive ermöglicht dem Anwender, diesen Auswahlvorgang zu steuern.	L
LSTVAL NO...	LSTVAL	Weist **fpp** an, Werte aller Variablen, die beim Optimieren aus Schleifen eliminiert wurden, nach dem letzten Schleifendurchlauf zu rekonstruieren (vgl. Option -eu in Abschnitt 4.2).	LRF

5.4 CFPP$-Direktiven zur Steuerung der Abhängigkeitsanalyse

Zur Vektorisierung oder Parallelisierung einer Schleife muß **fpp** für Feldelemente, deren Wert in der Schleife geändert wird, feststellen, wie die Speicheradressen dieser Elemente relativ zu den anderen Zugriffen auf dieses Feld in der Schleife liegen. Überlappen sich die Adressen nicht, so kann die Schleife parallel abgearbeitet werden. Die nachfolgenden Direktiven unterstützen **fpp** bei der Analyse der Datenabhängigkeiten in einer Schleife.

Direktive	Voreinstellung	Funktion	Bereich
DEPCHK NO...	DEPCHK	NODEPCHK teilt **fpp** mit, daß alle potentiellen Abhängigkeiten, d.h. Abhängigkeiten, über die zur Compilationszeit nicht entschieden werden kann, keine echten Abhängigkeiten sind. Damit wird jedoch nicht die Vektorisierung unzweifelhaft rekursiver Schleifen erzwungen. NODEPCHK mit globaler Gültigkeit ist äquivalent zur Option -dd. Die DEPCHK-Direktive dient zum Zurückschalten auf die Voreinstellung.	LRF
SYNC NO...	SYNC	NOSYNC ist die Verallgemeinerung von NODEPCHK für Parallelisierung, bei der auch die Reihenfolge der Iterationen nicht mehr erhalten ist.	LRF
EQVCHK	EQVCHK	NOEQCHK weist **fpp** an, bei der Datenabhängigkeitsanalyse die Abhängigkeiten zwischen Variablen durch EQUIVALENCE-Anweisungen zu ignorieren. NOEQVCHK mit globaler Gültigkeit ist äquivalent zur Option -de.	LRF
PERMUTATION	keine	Die Direktive dient zur Erklärung, daß ein INTEGER-Feld keinen Wert zweifach enthält. Dies ist nützlich, wenn Elemente des Feldes als Index eines anderen Feldes verwendet werden ("indirekte Adressierung"). Die Anweisung CFPP$ PERMUTATION (ia1,ia2,...,ian) konstatiert den Sachverhalt für die INTEGER-Felder *ia1, ia2, ... ian.* Mit dieser Zusatzinformation gelingt es **fpp** oftmals herauszufinden, daß mit Zugriffen auf das betreffende Feld kein Feedback (Verwendung von Werten aus früheren Iterationen der Schleife) verbunden ist.	R R

Direktive	Voreinstellung	Funktion	Bereich
RELATION	keine	Format: CFPP$ RELATION (i1.rel.i2) Dabei sind *i1,i2* : INTEGER-Größen, wobei eine von beiden eine Konstante sein kann. *rel* : GT, LT, GE, LE, EQ, NE Vergleichsoperatoren mit FORTRAN-Bedeutung. Die Anweisung teilt **fpp** mit, daß die angegebene Relation zwischen den beiden INTEGER-Größen vorliegt (zur Entscheidungshilfe für **fpp** bei der Abhängigkeitsanalyse). Die Direktive ist der unspezifischen NODEPCHK-Direktive, die alle Verantwortung dem Benutzer überläßt, vorzuziehen.	

5.5 CFPP$-Direktiven zur Angabe von Listing-Optionen

Direktive	Voreinstellung	Funktion	Bereich
LIST NO...	LIST	Zum selektiven Unterdrücken von Teilen des Listings. Ist NOLIST (oder die Option -ql) bei einer END-Anweisung des FORTRAN-Programmes in Kraft, wird der Rest des Listings unterdrückt (es sei denn, mit CFPP$ SWITCH LSTON = ... wird wieder explizit umgeschaltet).	F

5.6 CFPP$-Direktiven zur Inline-Expandierung

Die Direktiven dienen zur Angabe der zu expandierenden Routinen, sowie der sie enthaltenden Dateien. Es sind zwei Modi der Inline-Expandierung möglich:

expliziter Modus: nur Routinen, die in einer Direktive angegeben sind, werden expandiert.

automatischer Modus: alle Routinen werden expandiert,
- die aus weniger als 50 Anweisungen bestehen (vgl. Option -M in Abschnitt 4.5)
- die keine anderen Routinen aufrufen
- deren Expandierung nicht durch einen der nachfolgenden Gründe verhindert ist.

Folgende Eigenschaften verhindern die Expandierung von Unterprogrammen:
- die zu expandierende Routine ist nicht aufzufinden
- sie enthält Syntaxfehler
- die Argumente in der Parameterliste stimmen nicht mit denen im Aufruf überein
- gemeinsame COMMON-Blöcke der rufenden und der zu expandierenden Routine stimmen nicht überein
- ein in der zu expandierenden Routine auftretender Unterprogrammname wird in der aufrufenden Routine nicht als Unterprogrammname, sondern als anderer Name (z.B. Variablenname) verwendet.

Direktive	Voreinstellung	Funktion	Bereich
AUTOEXPAND NO...	AUTOEXPAND	Schaltet den automatischen Modus ein (AUTOEXPAND mit globalem Geltungsbereich ist äquivalent zur Option -e7).	LRF
EXPAND	keine	Mit CFPP$ EXPAND [(list)] wird die explizite Expandierung aktiviert. "list" ist eine Liste von Routinen, die in der aktuellen Programmeinheit zu expandieren sind; wird "list" nicht angegeben, so werden in der nächsten Anweisung aufgerufene Unterprogramme (SUBROUTINE und FUNCTIONs) expandiert. Diese Direktive ermöglicht auch die Expandierung geschachtelter Routinen.	F
SEARCH	siehe Text	Dient zur Angabe der Dateien, die nach zu expandierenden Unterprogrammen zu durchsuchen sind. Format: CFPP$ SEARCH (files) wobei *files* eine Liste durch Kommata getrennter Dateinamen ist. Hat *files* den Wert *.f, so werden die Routinen in gleichnamigen, kleingeschriebenen Dateien gesucht, z.B. die Routine x oder X in der Datei x.f. Dies ist zugleich das voreingestellte Suchverfahren. Befinden sich die zu expandierenden Routinen in der Datei, die auch die aufrufenden enthält (auch indirekt durch INCLUDE), so ist die Option -e8 zu verwenden (vgl. Abschnitt 4.2).	F

Anmerkungen zur Inline-Expandierung

a) Überprüfung von Aufrufen

Vor dem Expandieren werden Aufruf und Parameterliste der betreffenden Routine hinsichtlich der Übereinstimmung von Anzahl und Typ der Parameter untersucht. Nichtübereinstimmung wird gemeldet, d.h. mit **fpp** können Fehler gefunden werden können, ähnlich wie mit dem UNIX-Werkzeug *lint* in C-Programmen.

b) Fehlerhafter Code

Inline-Expandierung ist mit Vorsicht zu verwenden, mit dem ersten Release gibt es noch eine Reihe von Problemen (vgl. Kap.7); unter Umständen wird gar fehlerhafter Code erzeugt.

c) Getrennte Übersetzung

Wird das Unterprogramm B in A expandiert, so muß nach einer Änderung von B auch das Unterprogramm A neu übersetzt werden. Bei getrennter Übersetzung ist es daher wichtig zu wissen, welche Routinen wo expandiert wurden. Aus diesem Grund erzeugt **fpp** für jede durchgeführte Expandierung eine Warnung.

d) Suchverfahren

Per Voreinstellung werden zu expandierende Routinen in gleichnamigen Dateien gesucht. Falls Sie davon Gebrauch machen, beachten Sie den nichtlokalen Charakter dieses Verfahrens: nicht nur die angegebenen Quelldateien, sondern alle Dateien mit Suffix .f im aktuellen Directory kommen als Quelldateien in Betracht.

e) Größe des erzeugten Codes

Werden sehr viele oder sehr häufig aufgerufene Routinen expandiert, kann der übersetzte Code einen nicht mehr akzeptablen Umfang annehmen. Durch Verwendung des expliziten Modus ist der Umfang an Expandierungen steuerbar.

f) Fehlersuche

Laufzeitfehlermeldungen beziehen sich auf den von **fpp** erzeugten FORTRAN-Code, nicht den originalen Quellcode; es empfiehlt sich daher, beim Übersetzen mit **fpp** eine entsprechende Liste zu erzeugen und diese aufzubewahren.

g) Inline-Expandierung mit dem **cft77**

Auch beim cft77 ist Inline-Expandierung möglich (-I Option); die Expandierung geschieht jedoch nicht in FORTRAN, sondern in einem von cft77 erzeugten Zwischencode, so daß der expandierte Code dem Anwender nur in Form von Assemblercode zugänglich wird.

6. Praktische Hinweise und Aufruf-Beispiele

Wir beschränken uns hier auf die Verwendung des **fpp** zur Verbesserung der Vektorisierung (Parallelisierungsbeispiele werden in Ref. [10] und [12] angegeben). Die erzielbaren Beschleunigungsfaktoren hängen sehr von den verwendeten Algorithmen, der Art der Programmierung und der Güte bisher vorgenommener Optimierungen ab, so daß dazu keine generelle Vorhersage gemacht werden kann.

Die Laufzeiten des **fpp** sind relativ gering (ca. 1.3 - 2.5 sec/1000 Zeilen Code), allerdings werden mindestens 1.5 Mw Hauptspeicher benötigt, auch zur Bearbeitung kleinster Programme, so daß in vielen Fällen ein separater Übersetzungslauf zu empfehlen ist.

Der FORTRAN-Output des **fpp** wird von allen Versionen des **cft77** wie auch der Version 1.16 des **cft** akzeptiert, vorausgesetzt, der **fpp**-Input genügt den entprechenden Sprachdefinitionen.

Soll nicht automatisch parallelisiert werden, empfiehlt es sich, die Erzeugung von CMIC$-Direktiven durch die Option -dc auszuschalten. Der einfachste Aufruf zur Verbesserung der Vektorisierung ist somit

fpp -dc prog.f > prog.m

oder äquivalent

fpp -dc -o prog.m prog.f

Wir folgen hier der im **cf77**-Übersetzersystem geltenden Konvention, die von **fpp** erzeugten FORTRAN-Quellen durch das Suffix .m zu kennzeichnen. Nicht ganz konsequent ist die Namensgebung der Compiler in diesem Zusammenhang: aus Dateien mit Suffix .m werden, wenn nicht explizit anders angegeben, solche mit Suffix .m.o statt .o erzeugt.

Wird zusätzlich eine Liste (prog.L) mit **fpp**-Diagnosen gewünscht, so ist diese mit der Option -l anzugeben:

fpp -dc -o prog.m -l prog.L prog.f

Zur weiteren Optimierung ist, insbesondere bei Algorithmen der Linearen Algebra, die Ersetzung linearer Rekursionen durch SCILIB-Aufrufe zu empfehlen (Option -eb).

Inline-Expandierung bringt durch Wegfall von Unterprogrammaufrufen Laufzeit-Verbesserungen von nur wenigen Prozent (höchstens). Werden durch die Expandierung jedoch weitere Schleifen vektorisierbar (was von einer den CRAY-Rechnern wenig angepaßten Programmierung zeugt), sind weitaus höhere Beschleunigungsfaktoren möglich. Zur Inline-Expandierung ist die Option -e6 zu empfehlen, -e7 erzeugt nicht immer richtigen Code. Befinden sich die zu expandierenden Routinen in der Datei, die auch die aufrufenden Routinen enthält, muß zusätzlich die Option -e8 angegeben werden.

Der Aufruf zur Nutzung aller (sicheren) Optimierungsmöglichkeiten, ohne Parallelisierung und von den Voreinstellungen abweichenden Ausgabe- oder Formatierwünsche, hat somit die Form

fpp -dc -eb68 -o prog.m -l prog.L prog.f

Nimmt der Code durch zu häufige Inline-Expandierung einen nicht mehr akzeptablen Umfang an, ist auf den expliziten Modus mit selektiver Expandierung umzuschalten (siehe Abschnitt 5.5).

Generell ist beim derzeitigen Stand der Transformationssoftware zu empfehlen, Inline-Expandierung nicht "blind" zu verwenden. Beachten Sie auf jeden Fall die Anmerkungen in Abschnitt 5.6 und die derzeit noch bestehenden Probleme (Kap.7).

fpp überprüft die Syntax nicht systematisch und geht davon aus, daß sie vorher kontrolliert wurde. Nach Änderungen im Quellcode sollte daher dem **fpp**-Aufruf sicherheitshalber ein Syntaxcheck vorangestellt werden:

cft77 -dB -em prog.f || { cat prog.l; exit 1; }
fpp ...

In mehreren Fällen erzeugt **fpp** Code, der zwar semantisch korrekt ist, bei Feldgrenzenüberwachung zur Laufzeit aber auf vermeintliche Fehler führt (vgl. Kap.7). Die Verwendung von **fpp** während der Programmentwicklung ist daher problematisch und kann nur für ausgetestete Programme kurz vor Beginn der Produktionsphase empfohlen werden.

7. Probleme

Mit dem ersten Release des Autotasking-Systems sind noch eine Reihe von Problemen verbunden. Folgende Fehler bzw. Mängel des **fpp** sind uns bekannt:

- Routinen, zu deren korrekter Übersetzung die -o zeroinc Option beim **cft77** erforderlich ist, in denen also Schleifeninkremente von Null vorkommen können, werden vom Autotasking-System nicht richtig behandelt (vermutlich auch ein Problem von **fpp**).
- Die Inline-Expandierung durch **fpp** ist noch nicht perfekt. Folgende Schwachpunkte ergaben sich bei unseren Beobachtungen:
  - CDIR$-Compiler-Direktiven (des **cft77** und **cft**) in Unterprogrammen des Input-FORTRAN-Codes gehen beim Expandieren dieser Module verloren; handelt es sich um Direktiven, die nicht nur die Performance, sondern auch die Rechnung beeinflussen (z.B. CDIR$ INTEGER=64), so kann dies zu falschen Resultaten führen.
  - In expandiertem Code gibt es Formatierungsfehler (fehlende Leerzeichen), die zu FORTRAN-Syntaxfehlern führen.
  - In gewissen Fällen findet **fpp** den zu expandierenden Quellcode nicht.
  - Beim Expandieren werden in bestimmten Fällen überflüssige Variablen deklariert.
- In manchen Fällen werden zur Optimierung mehrere Feldindizes zu einem Index zusammengefaßt, etwa wenn mehrere Schleifen in einer vereinigt werden, oder wenn die Laufvariable in mehr als einer Felddimension auftritt, z.B. beim Adressieren von Diagonalelementen einer Matrix. Zwar wird dabei semantisch korrekter Code erzeugt,

doch führt Feldgrenzenüberwachung während der Laufzeit auf vermeintliche Fehler [1].

- Der Subprocessor **TIDY** enthält noch Fehler, die in Einzelfällen dazu führen können, daß im generierten FORTRAN-Code Zeichen fehlen, z.B. bei Verwendung der Option -rx Bestandteile von Anweisungsnummern.
- Bei manueller Parallelisierung oder gemeinsamer Verwendung von Autotasking und Macrotasking werden temporäre Felder, die durch Auftrennung von Schleifen zusätzlich erforderlich werden, von **fpp** fälschlicherweise als 'shared' statt 'private' erklärt. Es empfiehlt sich in solchen Fällen, die Auftrennung von Schleifen durch -ds zu unterdrücken.
- Bei Schleifen mit temporären Feldern hat **fpp** Probleme, die inhärente Parallelität zu erkennen.

Danksagung

Für die kritische Durchsicht früherer Versionen des Manuskripts und wertvolle Hinweise danke ich H.Busch, J.Gottschewski und W.Vortisch. Mein besonderer Dank gilt W.Vortisch für hilfreiche Diskussionen und D. Kehl für die Durchsicht des gesamten Manuskripts, sowie viele Korrekturvorschläge.

[1] Das Problem ließe sich lösen durch Einführung eines zusätzlichen Feldes, bei dem mehrere Dimensionen zu einer zusammengefaßt sind und das mittels einer EQUIVALENCE-Anweisung über das ursprüngliche Feld gelegt ist.

Literatur

Die Original-Dokumentation zum Thema Autotasking unter UNICOS 4.0 befindet noch sich auf einem ungenügenden Stand. Erst zum (unter UNICOS 5.0 lauffähigen) CFT77 3.1-Release gibt es halbwegs zufriedenstellende Beschreibungen vom Hersteller. Je nach Version des Betriebssystems und des FORTRAN-Compilers sind verschiedene Beschreibungen geeignet (Tab. 4).

UNICOS	CFT77	CFT	Auto-tasking	Kommandos cf77, fpp, fmp	FORTRAN-Compiler CFT77 CFT	Multi-tasking
4.0	3.0	1.16	[1]	--	[4]	[7]
5.0	3.0	1.16	[1]	[3]	[4]	[8]
5.0	3.1	1.16	[2]	[2],Anhang A	[4], [5]	[8]

Tab.4: Hersteller-Dokumentation zum Autotasking für die verschiedenen Betriebssystem- und Compiler-Versionen.

[1] UNICOS Autotasking User's Guide, SN-2088, CRAY Research Inc., Oct. 1988
Achtung: Die Beschreibung im Hauptteil gibt die heutige Implementation der Autotasking-Kommandos unter UNICOS 4.0 nur ungefähr und mit vielen Fehlern behaftet wieder. Anhang A mit den unter UNICOS 4.0 verfügbaren "man pages" zum Thema Autotasking (= Prerelease Draft des UNICOS 5.0 User Commands Reference Manual), ist unbrauchbar; die dortigen Beschreibungen der Auto-Tasking-Kommandos beziehen sich offensichtlich auf vorläufige, schon unter UNICOS 4.0 veraltete Implementierungen.

[2] UNICOS Autotasking User's Guide, SN-2088 CFT77 3.1, CRAY Research Inc., August 1989 (völlig neu überarbeitete Version von [1])

[3] UNICOS 5.0 User Commands Reference Manual, SR-2011 5.0, CRAY Research Inc., March 1989

[4] CFT77 Reference Manual, SR-0018 C, CRAY Research Inc., Oct.1988

[5] CFT77 3.1 Release Notice C7-03.1-BAD-RN, CRAY Research Inc, August 1989

[6] Fortran (CFT) Reference Manual, SR-0009 M, CRAY Research Inc., May 1989

[7] CRAY Y-MP, CRAY X-MP EA and CRAY X-MP Multitasking Programmer's Manual, SR-0222E, Cray Research Inc., June 1988

[8] CRAY Y-MP, CRAY X-MP EA and CRAY X-MP Multitasking Programmer's Manual, SR-0222F, Cray Research Inc., March 1989

[9] Programmer's Library Reference Manual, SR-0113C, Cray Research, July 1988

[10] Online-Dokumentation DOC,CRYFORT,FMP (in Arbeit)

[11] Online-Dokumentation DOC,CRYFORT,CF77 (in Arbeit)

[12] Online-Dokumentation DOC,CRAY,MULTI (in Arbeit)

[13] American National Standard Programming Language FORTRAN, ANSI X3.9-1978, American National Standards Institute, Inc., Apr. 1978, New York

[14] Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B. and Wolfe, M., "Dependence Graphs and Compiler Optimizations", Proceedings of the 8th ACM Symposium on Principles of Programming Languages, (Williamsburg, Va., Jan. 1981), New York, 1981, pp. 207-218.

[15] Wolfe, M.J., "Optimizing Supercompilers for Supercomputers", Ph.D. Dissertation, Univ. Illinois, Urbana-Champaign, DCS Rep. UIUCDCS-R-82-1105, Oct. 1982

[16] Burke, M., Cytron, R., "Interprocedural Dependence Analysis and Parallelization", Proc. of the Sigplan '86 Symposium on Compiler Construction, Sigplan Notices, vol. 21, no.7, July 1986, pp. 162-175, New York. ACM

[17] Polychronopoulos, D., "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design", IEEE Trans.Comput. vol.37, no.8 , Aug. 1988, pp. 991-1004

[18] Polychronopoulos, D., Girkar, M., Haghighat, M.R., Lee, C.L., Leung, B., Schouten, D., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Int. Journ. High Speed Comput., vol.1., no.1, 1989, pp. 45-72

[19] CFT Optimization Guide, SG-0115 1/88, Cray Research, 1988

Veröffentlichungen des Konrad-Zuse-Zentrum für Informationstechnik Berlin
Preprints Februar 1990

SC 86-1. P. Deuflhard; U. Nowak. *Efficient Numerical Simulation and Identification of Large Chemical Reaction Systems.* (vergriffen) In: Ber. Bunsenges. Phys. Chem., vol. 90, 1986, 940-946
SC 86-2. H. Melenk; W. Neun. *Portable Standard LISP for CRAY X-MP Computers.*

SC 87-1. J. Anderson; W. Galway; R. Kessler; H. Melenk; W. Neun. *The Implementation and Optimization of Portable Standard LISP for the CRAY.*
SC 87-2. Randolph E. Bank; Todd F. Dupont; Harry Yserentant. *The Hierarchical Basis Multigrid Method.* (vergriffen) In: Numerische Mathematik, 52, 1988, 427-458.
SC 87-3. Peter Deuflhard. *Uniqueness Theorems for Stiff ODE Initial Value Problems.*
SC 87-4. Rainer Buhtz. *CGM-Concepts and their Realizations.*
SC 87-5. P. Deuflhard. *A Note on Extrapolation Methods for Second Order ODE Systems.*
SC 87-6. Harry Yserentant. *Preconditioning Indefinite Discretization Matrices.*

SC 88-1. Winfried Neun; Herbert Melenk. *Implementation of the LISP-Arbitrary Precision Arithmetic for a Vector Processor.*
SC 88-2. H. Melenk; H. M. Möller; W. Neun. *On Gröbner Bases Computation on a Supercomputer Using REDUCE.* (vergriffen)
SC 88-3. J. C. Alexander; B. Fiedler. *Global Decoupling of Coupled Symmetric Oscillators.*
SC 88-4. Herbert Melenk; Winfried Neun. *Parallel Polynomial Operations in the Buchberger Algorithm.*
SC 88-5. P. Deuflhard; P. Leinen; H. Yserentant. *Concepts of an Adaptive Hierarchical Finite Element Code.*
SC 88-6. P. Deuflhard; M. Wulkow. *Computational Treatment of Polyreaction Kinetics by Orthogonal Polynomials of a Discrete Variable.* (vergriffen)
SC 88-7. H. Melenk; H. M. Möller; W. Neun. *Symbolic Solution of Large Stationary Chemical Kinetics Problems. I*
SC 88-8. Ronald H. W. Hoppe; Ralf Kornhuber. *Multi-Grid Solution of Two Coupled Stefan Equations Arising in Induction Heating of Large Steel Slabs.*
SC 88-9. Ralf Kornhuber; Rainer Roitzsch. *Adaptive Finite-Element-Methoden für konvektionsdominierte Randwertprobleme bei partiellen Differentialgleichungen.*
SC 88-10. S -N. Chow; B. Deng; B. Fiedler. *Homoclinic Bifurcation at Resonant Eigenvalues.*

SC 89-1. Hongyuan Zha. *A Numerical Algorithm for Computing the Restricted Singular Value Decomposition of Matrix Triplets.*
SC 89-2. Hongyuan Zha. *Restricted Singular Value Decomposition of Matrix Triplets.*
SC 89-3. Wu Huamo. *On the Possible Accuracy of TVD Schemes.*
SC 89-4. H. Michael Möller. *Multivariate Rational Interpolation: Reconstruction of Rational Functions.*
SC 89-5. Ralf Kornhuber; Rainer Roitzsch. *On Adaptive Grid Refinement in the Presence of Internal or Boundary Layers.*
SC 89-6. Wu Huamo; Yang Shuli. *MmB-A New Class of Accurate High Resolution Schemes for Conservation Laws in Two Dimensions.*
SC 89-7. U. Budde; M. Wulkow. *Computation of Molecular Weight Distributions for Free Radical Polymerization Systems.*
SC 89-8. Gerhard Maierhöfer. *Ein paralleler adaptiver Algorithmus für die numerische Integration.*
SC 89-9. Harry Yserentant. *Two Preconditioners Based on the Multi-Level Splitting of Finite Element Spaces.*
SC 89-10. Ronald H. W. Hoppe. *Numerical Solution of Multicomponent Alloy Solidification by Multi-Grid Techniques.*

SC 90-1. M. Wulkow; P. Deuflhard. *Towards an Efficient Computational Treatment of Heterogeneous Polymer Reactions.*
SC 90-2. Peter Deuflhard. *Global Inexact Newton Methods for Very Large Scale Nonlinear Problems.*

Veröffentlichungen des Konrad-Zuse-Zentrum für Informationstechnik Berlin
Technical Reports Februar 1990

TR 86-1. H. J. Schuster. *Tätigkeitsbericht (vergriffen)*

TR 87-1. Hubert Busch; Uwe Pöhle; Wolfgang Stech. *CRAY-Handbuch. - Einführung in die Benutzung der CRAY.*
TR 87-2. Herbert Melenk; Winfried Neun. *Portable Standard LISP Implementation for CRAY X-MP Computers. Release of PSL 3.4 for COS.*
TR 87-3. Herbert Melenk; Winfried Neun. *Portable Common LISP Subset Implementation for CRAY X-MP Computers.*
TR 87-4. Herbert Melenk; Winfried Neun. *REDUCE Installation Guide for CRAY 1 / X-MP Systems Running COS Version 3.3*
TR 87-5. Herbert Melenk; Winfried Neun. *REDUCE Users Guide for the CRAY 1 / X-MP Series Running COS. Version 3.3*
TR 87-6. Rainer Buhtz; Jens Langendorf; Olaf Paetsch; Danuta Anna Buhtz. *ZUGRIFF - Eine vereinheitlichte Datenspezifikation für graphische Darstellungen und ihre graphische Aufbereitung.*
TR 87-7. J. Langendorf; O. Paetsch. *GRAZIL (Graphical ZIB Language).*

TR 88-1. Rainer Buhtz; Danuta Anna Buhtz. *TDLG 3.1 - Ein interaktives Programm zur Darstellung dreidimensionaler Modelle auf Rastergraphikgeräten.*
TR 88-2. Herbert Melenk; Winfried Neun. *REDUCE User's Guide for the CRAY 1 / CRAY X-MP Series Running UNICOS. Version 3.3.*
TR 88-3. Herbert Melenk; Winfried Neun. *REDUCE Installation Guide for CRAY 1 / CRAY X-MP Systems Running UNICOS. Version 3.3.*
TR 88-4. Danuta Anna Buhtz; Jens Langendorf; Olaf Paetsch. *GRAZIL-3D. Ein graphisches Anwendungsprogramm zur Darstellung von Kurven- und Funktionsverläufen im räumlichen Koordinatensystem.*
TR 88-5. Gerhard Maierhöfer; Georg Skorobohatyj. *Parallel-TRAPEX. Ein paralleler, adaptiver Algorithmus zur numerischen Integration ; seine Implementierung für SUPRENUM-artige Architekturen mit SUSI.*

TR 89-1. *CRAY-HANDBUCH. Einführung in die Benutzung der CRAY X-MP unter UNICOS.*
TR 89-2. Peter Deuflhard. *Numerik von Anfangswertmethoden für gewöhnliche Differential-*gleichungen.
TR 89-3. Artur Rudolf Walter. *Ein Finite-Element-Verfahren zur numerischen Lösung von Erhaltungsgleichungen.*
TR 89-4. Rainer Roitzsch. *KASKADE User's Manual.*
TR 89-5. Rainer Roitzsch. *KASKADE Programmer's Manual.*
TR 89-6. Herbert Melenk; Winfried Neun. *Implementation of Portable Standard LISP for the SPARC Processor.*
TR 89-7. Folkmar A. Bornemann. *Adaptive multilevel discretization in time and space for parabolic partial differential equations.*
TR 89-8. Gerhard Maierhöfer; Georg Skorobohatyj. *Implementierung des parallelen TRAPEX auf Transputern.*

TR 90-1. Karin Gatermann. *Gruppentheoretische Konstruktion von symmetrischen Kubaturformeln.*
TR 90-2. Gerhard Maierhöfer; Georg Skorobohatyj. *Implementierung von parallelen Versionen der Gleichungslöser EULEX und EULSIM auf Transputern.*