Christoph W. Keßler

Automatische Parallelisierung numerischer Programme durch Mustererkennung

Dissertation
zur Erlangung des Grades
Doktor der Naturwissenschaften
der Technischen Fakultät
der Universität des Saarlandes

Saarbrücken

1994

Dekan:	Prof. Dr. G. Hotz
Vorsitzender der Prüfungskommission:	Prof. Dr. G. Weikum
Erster Berichterstatter:	Prof. Dr. W. J. Paul
Zweiter Berichterstatter:	Prof. Dr. R. Wilhelm
Beisitzer:	Dr. T. Rauber
Tag des Promotionskolloquiums:	19.10.1994

Danksagungen

Ich danke Herrn Prof. Dr. W.J. Paul für nützliche und anregende Diskussionen über diese Arbeit und dafür, daß ich ein selbstgewähltes Thema bearbeiten durfte, das *nicht* der aktuellen Hauptstoßrichtung des Lehrstuhls (PRAM–Projekt) entsprach.

Den Herren Professoren Dr. A.K. Louis, Dr. R. Wilhelm und Dr. H. Zima danke ich für Diskussionen und Vorschläge zu dieser Arbeit.

Desweiteren danke ich meinen Kollegen am Lehrstuhl Prof. Paul, namentlich Dr. Arno Formella, Dr. Silvia M. Müller, Dr. Thomas Rauber, sowie den ehemaligen Kollegen Anne Dierstein und Roman Hayer für fachliche Gespräche und Zusammenarbeit. Gleicher Dank gebührt meinen Kollegen vom Lehrstuhl Prof. Wilhelm, namentlich Martin Alt und Christian Ferdinand. Martin Alt und Christian Ferdinand danke ich auch für das Durchlesen des Manuskripts.

Die vorliegende Arbeit entstand in den Jahren 1991–94 am Fachbereich Informatik der Universität des Saarlandes. Von April 1991 bis März 1994 war ich Mitglied im Graduiertenkolleg Informatik an der Universität des Saarlandes, das von der Deutschen Forschungs–Gemeinschaft eingerichtet wurde. Ich danke allen Teilnehmern des Graduiertenkollegs für die Zusammenarbeit. Vor allem möchte ich mich bedanken bei dem Sprecher des Graduiertenkollegs, Herrn Prof. Dr. J. Buchmann, und der Sekretärin des Graduiertenkollegs, Frau Monika Fromm. Beiden danke ich insbesondere für die Unterstützung bei der Durchführung des Internationalen Workshops *AP93* über Automatische Parallelisierung, den ich im März 1993 im Rahmen des Graduiertenkollegs organisieren konnte.

Ferner danke ich meinen auswärtigen Kollegen, die durch Hinweise und Vorschläge zu dieser Arbeit beigetragen haben, insbesondere Dr. Thomas Brandes, Barbara Chapman, Dr. Thomas Fahringer, Dr. Michael Gerndt, Rolf Hänisch, Dr. Kathy Knobe und Ulrich Kremer.

Vor allem aber danke ich meiner Frau Susanne für ihr Verständnis für meine Arbeit, und meinen Eltern für ihre langjährige Unterstützung mit Rat und Tat.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Supercomputer–Architekturen und Supercomputer–Übersetzer 1

1.2 Automatische Parallelisierung durch Mustererkennung 5

1.3 Überblick über diese Arbeit 6

2 Sequentielle Quellsprache und Zwischendarstellung **7**

2.1 Vereinfachte sequentielle Quellsprache für numerische Programme 7

2.2 Zwischendarstellung 9

2.3 Syntaktische Prädikate 10

3 Grundlagen der Automatischen Parallelisierung **12**

3.1 Datenabhängigkeiten 12

3.1.1 Ausführungsreihenfolge und Datenabhängigkeiten 12

3.1.2 Verfahren zum Testen von Datenabhängigkeit 14

3.1.3 Richtungsvektoren 15

3.1.4 Linearisierung 15

3.2 Transformationen 15

3.2.1 Prozedur-Expansion (procedure inlining) 16

3.2.2 Konstantenpropagation (constant propagation) 16

3.2.3 Eliminieren von Induktionsvariablen 16

3.2.4 Eliminieren von temporären Variablen 17

3.2.5 Vereinfachung von Feldern (array simplification) 18

3.2.6 Entfernung von totem oder nutzlosem Code (dead code elimination) 18

3.2.7 Aufgliederung bedingter Anweisungen (`IF` distribution) 18

3.2.8 Normalisierung von Schleifen (loop normalization) 18

3.2.9 Schleifen-Blockung (loop blocking, strip mining, loop tiling) 19

3.2.10 Abwickeln einzelner Iterationen (loop peeling) 19

3.2.11 Abrollen von Schleifen (loop unrolling) 20

3.2.12 Schleifenvertauschung (loop interchange) 20

3.2.13 Schleifenaufgliederung (loop distribution) 20

3.2.14 Vektorisierung 21

3.2.15 Vektorisierung bedingter Anweisungen (`IF` conversion) 22

3.2.16 Skalar–Expansion (scalar expansion) 22

3.2.17 Linearisierung von Schleifen (loop collapsing) 23

3.3 Parallelisierung für SMS 23

3.4 Halbautomatische Parallelisierung für DMS 24

3.4.1 Splitting, Adaptierung und Standardoptimierungen 24

3.4.2 Lokale Iterations–, Index– und Kommunikationsmengen 26

3.4.3 Laufzeitparallelisierung für indirekte Feldzugriffe . . . 27
3.5 Datenaufteilungen . . . 28
3.5.1 Alignment . . . 30
3.5.2 Partitionierung . . . 32
3.5.3 Statische Umverteilung . . . 32
3.6 Bestimmung von Kommunikationsanweisungen höherer Ordnung . . . 33
3.7 Laufzeitvorhersage . . . 34

4 Grundlagen zur syntaktischen Mustererkennung in Programmen 38
4.1 Treepatternmatching . . . 39
4.1.1 Bottom–Up–Verfahren zum Tree Pattern Matching–Problem . . . 39
4.1.2 Top–Down–Verfahren zum Tree Pattern Matching–Problem . . . 41
4.1.3 Nichtlineare Treepattern . . . 42
4.1.4 Vergleich . . . 42
4.2 Baumgrammatiken, Treeparsing und Codeerzeugergeneratoren . . . 43
4.2.1 Treepatternmatching und endliche Baumautomaten . . . 44
4.3 Baumtransformationssysteme . . . 45
4.4 Grenzen syntaktischer Mustererkennung . . . 46
4.5 Unsere Lösung . . . 47

5 Muster in numerischen Programmen 49
5.1 Muster, Inkarnationen, Instanzen, Implementierungen . . . 49
5.2 Prinzipien für das Muster–Design . . . 50
5.3 Basis–Muster–Bibliothek . . . 52
5.3.1 Skalare Muster (Ordnung 0) . . . 52
5.3.2 Muster der Ordnung 1 . . . 57
5.3.3 Muster der Ordnung 2 . . . 65
5.3.4 Muster der Ordnung 3 . . . 73
5.3.5 Muster der Ordnung 4 . . . 74
5.3.6 Instabile Muster . . . 75
5.3.7 Geplante Erweiterungen . . . 75
5.4 Analyse von Quellprogrammen . . . 76
5.4.1 Typische Indizierungen in numerischen Programmen . . . 76
5.4.2 Benchmark Suites: Livermore Loops und Purdue Set . . . 76
5.4.3 Array–Primitive in Fortran 90 und HPF . . . 77
5.4.4 Numerische Standardsoftware–Pakete . . . 78
5.4.5 Numerikpakete der Supercomputer–Hersteller . . . 79
5.4.6 Sonstige . . . 80
5.5 Diskussion . . . 81

6 Erkennung von Mustern in numerischen Programmen **82**

6.1 Vorabtransformationen . . . 83

6.2 Inkarnationen, Schablonen und Musterhierarchiegraph . . . 83

6.2.1 Komplexitätsbetrachtungen (vertikale Mustererkennung) . . . 85

6.2.2 Der Grundalgorithmus zur Mustererkennung (vertikal) . . . 85

6.2.3 Beispiel: Matrix–Matrix–Multiplikation . . . 86

6.2.4 Beispiel: Eliminieren semantisch invarianter Bedingungen . . . 87

6.2.5 Beispiel: Ausschluß einzelner Werte von Schleifenvariablen . . . 88

6.2.6 Beispiel: Entblocken von Schleifen . . . 88

6.2.7 Beispiel: $\mathsf{MULTIADD}^{(0)}$ und $\mathsf{MULTIMUL}^{(0)}$ . . . 89

6.2.8 Beispiel: Differenzensterne (stencils) . . . 89

6.3 Ein kompakter Zugriffsdeskriptor für Felder . . . 90

6.3.1 Motivation . . . 90

6.3.2 Definition des Deskriptors . . . 91

6.3.3 Berechnung des Deskriptors . . . 93

6.3.4 Operationen auf Deskriptoren . . . 94

6.3.5 Anwendung des Gebietsanzeigerkonzeptes auf Tensoren höherer Ordnung . 95

6.3.6 Andere Arbeiten über Deskriptoren . . . 96

6.4 Berechnung von Datenfluß–Querkanten (cross edges) . . . 97

6.5 Algorithmus (vertikale und horizontale Mustererkennung) . . . 99

6.5.1 Komplexitätsbetrachtungen (vertikale und horizontale Mustererkennung) . 99

6.6 Spezielle Schablonen bei Querkanten . . . 100

6.6.1 Aufrollen von Schleifen . . . 100

6.6.2 Umbenennen von Ergebnissen . . . 103

6.7 Erkennung von Datenstrukturkonzepten . . . 104

6.8 Spezielle Transformationen bei der Mustererkennung . . . 105

6.8.1 Aufgliederung bedingter Anweisungen (IF–distribution) . . . 105

6.8.2 Skalar–Expansion . . . 105

6.8.3 Aufgliederung von Schleifen (loop distribution) . . . 106

6.8.4 Einordnung der Transformationen . . . 107

6.9 Transformationen nach der Mustererkennung . . . 108

6.9.1 Entfernen von nutzlosem Code . . . 108

6.9.2 Zerfall instabiler Muster . . . 108

6.9.3 Konstruktion von Vektor–DAGs . . . 108

6.10 Korrektheit . . . 108

6.11 Implementierung und Ergebnisse . . . 109

6.11.1 Warum kein Treepatternmatcher–Generator von der Stange? . . . 109

6.11.2 Stand der Implementierung . . . 109

6.11.3 Ergebnisse . . . 111

6.12 Diskussion . . . 111

6.13 Verwandte Arbeiten . . . 111

7 Muster–gesteuerte Code–Erzeugung für DMS 113
7.1 Algorithmen–Ersetzung und Transformationen . . . 113
7.1.1 Code–Erzeugung durch Algorithmenersetzung . . . 113
7.1.2 Beispiele für parallele Implementierungen . . . 117
7.1.3 Diskussion . . . 121
7.2 Mustergesteuerte Datenaufteilung . . . 122
7.3 Mustergesteuerte synthetische Laufzeitvorhersage . . . 123
7.4 Codeerzeugung für nicht erkannte Programmteile . . . 124
7.5 Implementierung . . . 124
7.6 Verwandte Arbeiten . . . 124

8 Zusammenfassung 127

A Details zu *geqdescr*, *disjdescr*, und *neighbdescr* 129
A.1 Einschluß von Deskriptoren: *geqdescr* . . . 129
A.2 Disjunktheit von Deskriptoren: *disjdescr* . . . 130
A.3 Nachbarschaft von Deskriptoren: *neighbdescr* . . . 131

B Schablonen–Übersicht 132
B.1 Schablonen zu skalaren Mustern . . . 132
B.2 Schablonen zu Mustern der Ordnung 1 . . . 136
B.3 Schablonen zu Mustern der Ordnung 2 . . . 144
B.4 Schablonen zu Mustern der Ordnung 3 . . . 152
B.5 Schablonen zu Mustern der Ordnung 4 . . . 154
B.6 Musterhierarchiegraph . . . 155
B.6.1 Vertikale Musterhierarchie . . . 155
B.6.2 Horizontale Musterhierarchie . . . 157

C Weitere Quellen und Ergebnisse 159
C.1 CG–Verfahren . . . 159
C.2 Livermore Loops . . . 160
C.3 Sonstige Testprogramme . . . 170
C.4 EISPACK–Routine `tred2` . . . 173
C.5 Perfect Club Benchmark–Routine EFLUX . . . 176

D Muster für Operationen auf dünnbesetzten Matrizen 179
D.1 Datenstrukturen für dünnbesetzte Matrizen . . . 179
D.2 Muster für Operationen mit indirekten Feldzugriffen . . . 180
D.3 Analyse der SPARSE-BLAS–Routinen . . . 184

1 Einführung

1.1 Supercomputer–Architekturen und Supercomputer–Übersetzer

In diesem Abschnitt geben wir einen kurzen Überblick über Supercomputer–Architekturen und deren Einfluß auf die Technologie optimierender Übersetzer. Wir beschränken uns dabei auf die Themen, die uns im Zusammenhang mit dieser Arbeit von Bedeutung erscheinen.

Als Übersicht über die derzeit verfügbaren Superrechner verweisen wir auf [HB84], [TW91] und [Got92].

Supercomputer — Parallelrechner

Die Nachfrage nach Rechenleistung aus Naturwissenschaften, Technik, Medizin, Wetter- und Klimaforschung und anderen Anwendungsgebieten, repräsentiert durch die sogenannten "grand challenge"–Anwendungen, steigt rapide an — schneller noch, als technologische Fortschritte seitens der Computerhersteller diese Lücke schließen können. Als Ausweg aus dieser Situation bietet die Parallelverarbeitung (*parallel processing*) einen — zumindest theoretisch — technologieunabhängigen Geschwindigkeitsgewinn (*speed–up*) gegenüber Einprozessorarchitekturen, und zwar — im Idealfall — skaliert mit der Anzahl der beteiligten Prozessoren (linearer speed–up).

In den 1970er Jahren wurde mit den *Vektorrechnern*[1] ein erster Schritt in Richtung Parallelverarbeitung getan. Sie arbeiten nach dem *SIMD* (Single Instruction, Multiple Data)–Prinzip. Da diese Rechner Vektoroperationen (*datenparallele Operationen*) sehr schnell ausführen können, sind sie skalaren Rechnern diesbezüglich beträchtlich überlegen. Trotzdem sind sie technologischen Beschränkungen unterworfen: Sie sind nicht *skalierbar*, d.h. die Anzahl der Pipeline–Stufen kann nicht beliebig gesteigert werden; sie bieten oft nicht genug Flexibilität, und sie können nicht effektiv eingesetzt werden für unregelmäßige oder nicht–datenparallele Programme.

Systolische Felder von Prozessoren arbeiten in einer streng synchronen Weise — Daten werden durch das Prozessorfeld „gepulst" und dabei geeignet miteinander verknüpft. Auch hier ist pipelining das grundlegende Verfahren.

VLIW (Very Long Instruction Word) – Architekturen besitzen mehrere serielle funktionale Einheiten, die in synchroner Weise parallel genutzt werden können. Die Steuerung erfolgt über ein langes Befehlswort, dessen Teilabschnitte je eine funktionale Einheit einzeln ansprechen und mit Operanden versorgen.

Seit den frühen 1980er Jahren wurden mehr und mehr „echte" Parallelrechner entworfen und gebaut, zunächst überwiegend Mehrprozessorsysteme mit gemeinsamem Speicher (*shared memory multiprocessor (SMS)*, die aus einer kleinen Anzahl von Prozessoren bestehen, die mit dem gemeinsamen Arbeitsspeicher durch ein schnelles *Verbindungsnetzwerk* verbunden sind. Beispiele sind Cray X-MP oder Alliant FX/8. Jeder dieser Prozessoren bearbeitet sein eigenes „Knotenprogramm", was dem MIMD (Multiple Instructions, Multiple Data)–Prinzip entspricht: Zusätzlich zum Datenparallelismus kann nun auch Kontrollflußparallelismus (*task parallelism, parallel threads*) ausgenutzt werden. SMS sind für den Programmierer handlich, weil sie die globale Sicht des Arbeitsspeichers erhalten, die der Programmierer vom sequentiellen Programm her gewohnt

[1] Ein Vektorrechner enthält eine oder mehrere arithmetische Verarbeitungseinheiten, die in mehrere aufeinanderfolgende Teilschritte (Stufen) aufgeteilt wurden, sodaß in allen Stufen gleichzeitig, allerdings auf verschiedenen Daten, gerechnet werden kann („*pipelining*"). Dadurch ist ein Vektorrechner für die Verarbeitung von Sequenzen gleichartiger Operationen (Vektoroperationen) auf Operandenfeldern (Vektoren) geeignet. Für skalare Rechnungen ergibt sich kein Geschwindigkeitsgewinn. Beispiele für Vektorrechner sind Cray 1, Fujitsu VP-100 oder die Vektorknoten der Thinking Machines CM-5.

ist. Er muß sich auch nicht mit Fragen wie Einbettung, Kommunikationsbandbreite oder Puffergrößen herumschlagen. Allerdings sind die „klassischen“ SMS nicht skalierbar, weil das Netzwerk als Engpaß die derzeit realisierbare Prozessoranzahl auf eine Größenordnung von etwa 32 beschränkt.

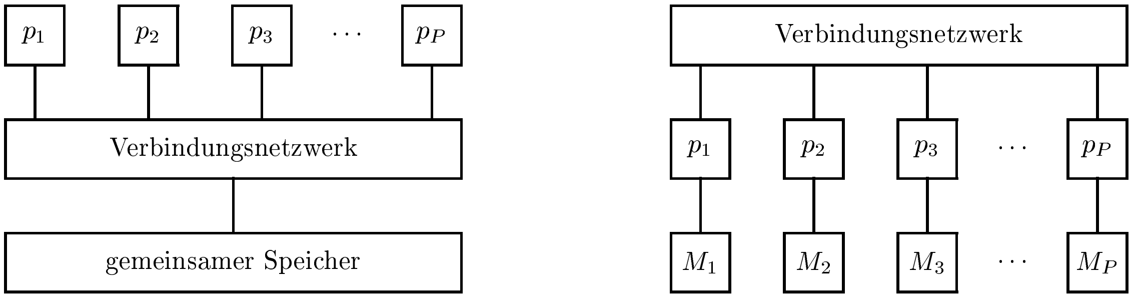


Abbildung 1.1 Schematische Darstellung von SMS (links) und DMS (rechts)

Seit der Mitte der 1980er Jahre bis zum heutigen Tag finden wir einen deutlichen Trend in Richtung skalierbarer massiv–paralleler Architekturen, die aus Tausenden von Prozessoren bestehen. Diese können gegenwärtig nur als Maschinen mit (physikalisch) verteiltem Speicher (*distributed–memory, DMS*) realisiert werden, entweder in der Form eines homogenen Multiprozessors (wie zum Beispiel Intel iPSC/860 [INT90], Intel Paragon, NCUBE-2, Thinking Machines CM-5 [PS92]) oder als Cluster von mehr oder weniger lose gekoppelten Workstations (wie zum Beispiel IBM 9076 SP1). Das DMS–Konzept stützt sich auf skalierbare Netzwerke wie Hyperwürfel (*hypercubes*), bandbreitenverstärkte Bäume (*fat trees*) oder Gitter und Tori, welche die Prozessoren *untereinander* verbinden; ferner auf preisgünstige Speicherbausteine und auf preisgünstige (weil massenproduziert) Standard–Prozessoren wie Intel i860 oder die Inmos Transputer. Da es bei den DMS keinen globalen Arbeitsspeicher mehr gibt, müssen die Datenobjekte (insbesondere, Felder) auf die einzelnen lokalen Speichermodule der Prozessoren aufgeteilt werden. Ein Zugriff auf nichtlokale Daten ist nur durch Interprozessor–Kommunikation (*message-passing*) möglich, welche, als Betriebssystemroutinen implementiert, extrem zeitaufwendig im Vergleich zu den schnellen arithmetischen Operationen ist. Daher zählen die DMS zu den sogenannten NUMA–Architekturen (Non–Uniform Memory Access).

Zwei weitere Architekturmodelle werden derzeit erprobt, um Skalierbarkeit durch Verwendung verteilter Speichermodule zu erreichen und trotzdem dem Benutzer eine globale Sicht des Arbeitsspeichers zu ermöglichen.

Das erste ist der *virtually shared memory* (VSM) – Ansatz, der einen globalen Arbeitsspeicher auf einem DMS-System durch *caching* simuliert. Will ein Prozessor auf Daten zugreifen, die sich nicht in seinem lokalen Cache befinden, so müssen die Speicherseiten, die diese Daten enthalten, über das Verbindungsnetzwerk in den Cache geladen werden. Nach diesem Prinzip arbeitet zum Beispiel der Kendall Square Research KSR-1.

Der zweite Ansatz befindet sich noch im Forschungsstadium: Die Realisierung [AKP90] des theoretischen skalierbaren SMS–Modells *Parallel Random Access Machine (PRAM)* basiert auf der randomisierten Bestimmung möglichst kollisionsarmer Wege durch das Netzwerk (*random routing*) und der kosteneffektiven Gestaltung der Wartezeit (*latency hiding*) durch ein ausgeklügeltes gepipelinetes und doch skalierbares Netzwerk, das die Prozessoren mit den Speichermodulen verbindet. Während die PRAM ein echter Allzweck–Parallelrechner ist, wird ihre erwartete Leistung auf numerischen Anwendungen hoher Lokalität durch DMS vergleichbarer Technologie deutlich übertroffen.

Insbesondere VSM, aber auch die PRAM verbergen viele Implementierungsdetails vor dem Benutzer und, leider, auch vor dem Übersetzer. Während beide Technologien die implizite Parallelisierung unterstützen, wird eine maschinenspezifische Optimierung des parallelisierten Codes erschwert.

Betrachtet man die Entwicklung der „Supercomputer“, so läßt sich feststellen, daß die Lücke zwischen der (theoretisch) erreichbaren Spitzenleistung (*peak performance*) und der auf realen

Anwendungen tatsächlich erreichbaren Leistung (*sustained performance*) immer größer wird, und daß es immer schwieriger wird, die Anwendung in bezug auf die Hardware zu optimieren.

Laufzeitvorhersage für Parallelrechner

Nicht nur die effiziente Programmierung wird schwieriger. Bei den heutigen Parallelrechnern ist es alles andere als einfach, die Leistung dieser Maschinen (Laufzeit) für ein nichttriviales paralleles Programm vorherzusagen. Dies wird verursacht durch im allgemeinen schlecht dokumentierte Eigenarten der Hardware, wie Caches, komplexe Nachrichtenprotokolle, Puffergrößen oder Routing–Anomalien.

Für die Beurteilung der Qualität einer bestimmten Datenaufteilungskonfiguration bei DMS oder einer bestimmten Programmtransformation ist eine zuverlässige automatische *Laufzeitvorhersage* parallelisierter Programme unabdingbar. Da die Hersteller von Supercomputern kaum detaillierte Datenblätter ihres Architekturdesigns veröffentlichen, muß die Leistungs– bzw. Laufzeitvorhersage *indirekt* erfolgen. Dazu mißt man die Laufzeiten auf geeigneten Meßprogrammen (*benchmarks*), wie zum Beispiel den *Livermore Loops* [McM86]. Mit diesem Datenmaterial versucht man, ein einfaches Architekturmodell für die Zielmaschine zu verifizieren und die charakteristischen Maschinenparameter herzuleiten, die wiederum später die Basis für die Laufzeitvorhersage bilden.

Wurde für das Meßprogramm ein Übersetzer und/oder Betriebssystemroutinen benutzt, so wird auch deren Güte mitgemessen und geht ins Maschinenmodell ein. Da das vereinfachte Modell niemals die Zielmaschine exakt wiedergeben kann, kann auch die so erzeugte Laufzeitvorhersage im allgemeinen nicht exakt sein.

Code–Erzeugung für Parallelrechner

Die meisten numerischen Anwendungen enthalten eine Vielzahl von Gitteroperationen oder Routinen der Linearen Algebra. Schleifen, die große Felder indizieren, bieten darin das größte Potential für (Daten-)Parallelismus. Das Extrahieren von Parallelismus bedeutet hier das Umsetzen serieller Schleifen in parallele Schleifen, ohne dadurch die Semantik des Programms durch die Verletzung von Datenabhängigkeiten zu verändern. Eine detaillierte Diskussion der Theorie der Datenabhängigkeitsanalyse ist in [Ban88] zu finden.

Weitere wichtige Eigenschaften von Gitteroperationen und den meisten Routinen der Linearen Algebra sind die *Lokalität* der Feldzugriffe und ein niedriges Verhältnis von erforderlichem Kommunikationsumfang zum Volumen der auszuführenden arithmetischen Operationen. Ein Programm, das diese beiden Eigenschaften nicht besitzt, wird im allgemeinen nicht effektiv auf heutigen DMS-Supercomputern laufen.

Geeignete Programmtransformationen können die Datenabhängigkeitsanalyse erleichtern und dadurch das Potential für die Extraktion von Parallelismus erhöhen. Obwohl heute viele Transformationen bekannt und größtenteils intensiv untersucht worden sind, war die *automatische Führung* in der geeigneten Anwendung dieser Transformationen ein bislang ungelöstes Problem. Die komplexeste Programmtransformation, die man überhaupt ausführen kann, ist die komplette (lokale) Ersetzung einer sequentiellen Implementierung durch einen parallelen Algorithmus gleicher Funktionalität (*algorithm replacement*). In den letzten Jahrzehnten wurden viele parallele Algorithmen entwickelt, aber im allgemeinen gehen sie eben *nicht* aus ihren sequentiellen Gegenstücken durch einfache Transformation und das Parallelisieren der Schleifen hervor.

Die Erzeugung von parallelem Code für SMS geschieht gewöhnlich durch das Zerlegen des Programms in einzelne Teilaufgaben (*tasks*), die Planung von deren Abarbeitung durch jeweils einen Prozessor in einer korrekten Reihenfolge (*scheduling*) und das Einfügen von Synchronisationspunkten (barrier synchronization points) an den Nahtstellen, um zu gewährleisten, daß alle Prozessoren auf die jeweils gültigen Versionen der Programmdaten zugreifen. Eine gute Übersicht über Übersetzertechniken für SMS bietet [Pol88].

Die wichtigsten Übersetzeroptimierungen für SMS betreffen die gleichmäßige Verteilung der Rechenlast (*load balancing*) durch das Aufteilen des Programms in Teilaufgaben von nahezu gleicher Größe, und die Minimierung der Anzahl der Synchronisationspunkte.

DMS sind schwieriger zu programmieren. Die Aufteilung des Programms in Teilaufgaben für die einzelnen Prozessoren wird fast immer durch das SPMD (single program, multiple data) –Paradigma gelöst: Alle Prozessoren haben das gleiche Programm, arbeiten aber auf verschiedenen Datenbereichen. Damit wird durch die Datenaufteilung auch die Code–Aufteilung in Teilaufgaben induziert. Ein Zugriff auf nichtlokale Daten resultiert in Interprozessor–Kommunikation. Wenn Daten vom lokalen Speicher eines Prozessors zu dem eines anderen transportiert werden müssen, muß dies explizit als korrespondierende Paare von SEND und RECEIVE Anweisungen programmiert werden (message–passing). Bei dieser langweiligen und fehleranfälligen Vorgehensweise kann dem Anwender auf zwei verschiedene Arten geholfen werden: zum einen durch die Benutzung von maschinenunabhängigen Kommunikationsroutinen für message–passing, und zum anderen durch (halb-)automatische Parallelisierung.

Sammlungen von maschinenunabhängigen Message–passing–Routinen (wie p4 [BL92], PVM [?], EXPRESS [Par], PARMACS [CHHW94], MPI [CGH94], LINDA [CGMS94]) enthalten einfache (SEND/RECEIVE) und höhere (z.B. globale Operationen auf Feldelementen) Kommunikationsprimitive. Die Benutzung dieser Routinen gewährleistet die Portabilität des parallelen Programms auf vielen message-passing DMS-Hardware–Plattformen. Nichtdestoweniger muß der Parallelismus explizit durch den Benutzer programmiert werden, zum Beispiel in der Form paralleler Schleifen.

Eine Alternative bietet die *halbautomatische Parallelisierung*: Der Benutzer bestimmt (manuell), wie die Daten (Felder) über die Prozessorspeicher aufgeteilt werden sollen. Dies kann technisch entweder durch Kommandos in einem interaktiven System wie z.B. SUPERB [ZBG88, Ger89], ASPAR [IFKF90] oder MIMDizer [SWW91], oder durch Direktiven bzw. entsprechende DISTRIBUTE–Anweisungen in datenparallelen, DMS–spezifischen Programmiersprachen wie C* [RS87], High Performance Fortran [HPF93], Fortran D [HKT91a], Vienna Fortran [CMZ92], oder Modula-2* [PT91] realisiert werden.

Ferner ist es fast immer ratsam, an dieser Stelle (manuell) einige geeignete Optimierungen durchzuführen, wie zum Beispiel Schleifentransformationen, Umstellung von Anweisungen oder gar die teilweise Ersetzung von Algorithmen, um die Eigenarten der Hardware (wie z.B. Cache-Größen oder Prozessortopologien) auszunutzen.

Danach wird das Programm automatisch *adaptiert*, d.h. an die vorgegebene Datenaufteilung angepaßt, indem vor jede Anweisung eine *Maske* (Bedingung) gesetzt wird, um zu gewährleisten, daß ein jeder Prozessor genau die Feldelemente berechnet, die in seinem lokalen Speicher residieren (*owner–computes rule*). Darüberhinaus werden die bei Zugriffen auf nichtlokale Daten erforderlichen SEND– und RECEIVE–Anweisungen in den Code eingefügt. Das auf diese Weise generierte parallele Programm ist im allgemeinen nicht effizient. Es muß weiter optimiert werden, insbesondere durch die Straffung der Kommunikation und durch die Vereinfachung der Masken. Sind die Knotenprozessoren selbst wiederum Vektorrechner (z.B. bei der CM-5), so sollte das parallelisierte Knotenprogramm nachfolgend vektorisiert werden, um dieses zusätzliche Leistungspotential zu nutzen.

Warum vollautomatische Parallelisierung?

Obwohl die halbautomatische Parallelisierung für den Benutzer die Generierung der Kommunikationsanweisungen übernimmt, bleibt dieser dennoch mit den anspruchvollsten Teilaufgaben auf sich alleine gestellt: die Bestimmung einer geeigneten Datenaufteilung (ein NP-vollständiges Problem) und die Anwendung einer geeigneten Folge von optimierenden Transformationen auf das Quellprogramm. Beide Faktoren bestimmen entscheidend die Effizienz des erzeugten parallelen Programms. Schlimmstenfalls kann eine ungeeignete Datenaufteilung sogar zu einer Verlangsamung des parallelisierten Programms im Vergleich zu seinem sequentiellen Äquivalent führen (*speed–down*).

Sofern der Programmierer über hinreichendes Wissen über seine Datenstrukturen und die Zielmaschine verfügt, und sofern er außerdem bereit ist, eine Menge Zeit in die manuelle Transformation seiner oft mehrtausendzeiligen Anwendung zu investieren, so mag dieser Stand der Technik für ihn genügen. Im allgemeinen ist aber der typische Supercomputer–Anwender gar kein Parallelrechner–Spezialist und möchte sich lieber auf seine primären Ziele konzentrieren, anstatt viel Zeit und Nerven in das manuelle Umschreiben seines Codes für eine Maschine aufzuwenden, die schon wenige Jahre später „reif für's Museum“ ist. Was er will, ist, dem Compiler sein sequentielles Quellprogramm zu präsentieren und wenig später dann die optimierte parallelisierte Version „in Händen zu halten“. Darüberhinaus gibt es einen immensen Bestand an Altprogrammen (*dusty decks*), die ebenfalls auf neue Generationen von Supercomputern portiert werden könnten. Selbst für neu zu schreibende Programme verhalten sich die Anwendungsprogrammierer eher reserviert gegenüber den in jüngster Zeit wie Pilze aus dem Boden sprießenden parallelen Programmiersprachen, da sich noch keine davon als *die* Standardsprache durchgesetzt hat (auch nicht High Performance Fortran).

Vollautomatische Parallelisierung — eine Utopie?

Auch wenn in den letzten Jahren mehrere interessante Ansätze zum Problem der automatischen Datenaufteilung erschienen sind, ist doch die halbautomatische Parallelisierung noch immer der aktuelle Stand der Technik. *Vollautomatische Parallelisierung* erscheint noch als Utopie, und manche glauben sogar, daß sie es auch für immer bleiben werde. Es gibt eine ganze Reihe ungelöster Probleme, zum Beispiel undurchsichtiges und schlecht dokumentiertes Verhalten von Parallelrechnern, seltsame oder veraltete Eigenschaften der Quellsprache, unregelmäßige Berechnungsstrukturen, wichtige zur Übersetzungszeit nicht bekannte Werte (z.B. Schleifengrenzen), unauflösbare Alias– oder Pointerzugriffe, und bislang nur verfügbare Algorithmen exponentieller Laufzeit für die Bestimmung *exakter* Feld–Datenflußinformation sowie für optimales Feld–Alignment und Datenaufteilung.

Da schnelle und umfassende Lösungsverfahren zu diesen Problemen in nächster Zukunft nicht erreichbar scheinen, konzentriert sich die aktuelle Forschung eher auf VSM und auf Übersetzer für parallele Programmiersprachen; Forschung über automatische Methoden erscheint eher an deren Rande. Nichtdestoweniger stellt die automatische Parallelisierung eine wichtige Schlüsseltechnologie für den zukünftigen Einsatz von Parallelrechnern dar: *Der Supercomputer–Anwender will nicht parallel programmieren — er will nur mehr Rechenleistung.*

1.2 Automatische Parallelisierung durch Mustererkennung

Da derzeit keine Strategie zur vollautomatischen und effizienten Parallelisierung für allgemeine Anwendungen existiert, liegt der Gedanke nahe, sich auf eine wichtige, aber in bezug auf die DMS–Parallelisierung doch relativ „gutartige“ Teilklasse von Anwendungen zu beschränken, die etwa auf Vektoren oder Matrizen arbeiten.

In dieser Arbeit beschreiben wir das PARAMAT–System zur vollautomatischen DMS–Parallelisierung, das auf diesem Gedanken aufbaut. Unserem PARAMAT–Ansatz liegen drei Hauptideen zugrunde:

- Zunächst untersuchen wir repräsentative Beispiele aus der betrachteten Klasse von Anwendungsprogrammen. Wir stellen fest, daß es eine relativ kleine Anzahl (etwa 150) typischer Muster (einfache semantische Operationen auf Skalaren, Vektoren oder Matrizen) gibt, die in diesen Programmen häufig vorkommen, insbesondere in den zeitkritischen inneren Schleifen. Wir sammeln diese Muster in einer Musterbibliothek und notieren auch typische Modifikationen (syntaktische Erscheinungsformen bzw. semantikerhaltende Transformationen) dieser Muster.

- Ein ausgeklügeltes und speziell auf die Musterbibliothek zugeschnittenes Mustererkennungssystem wird auf das sequentielle Quellprogramm angesetzt und ersetzt alle erkannten Vorkommen von Mustern aus der Musterbibliothek durch eine Musterinstanz, die einem Aufruf einer extern definierten Laufzeitroutine vergleichbar ist.
- Durch die Musterinstanzen sowie einige weitere von der Mustererkennung beschaffte Informationen erfahren wir, was das Programm auf lokaler Ebene macht. Dieses Wissen eröffnet uns den Zugang zu einer Menge von Hintergrundwissen (Expertenwissen) über mathematische Eigenschaften und effiziente parallele Implementierungen der Muster, über lokal geeignete Datenaufteilungen und das Laufzeitverhalten dieser Implementierungen auf der Zielmaschine. PARAMAT benutzt dieses Expertenwissen für die Generierung effizienten parallelen Codes.

1.3 Überblick über diese Arbeit

Nach dieser Einführung und Motivation unserer Arbeit stellen wir in Kapitel 2 die von PARAMAT verwendete sequentielle Quellsprache sowie unsere Zwischendarstellung vor.

Anschließend geben wir in Kapitel 3 eine kurze Einführung in die Grundlagen der automatischen Parallelisierung. Insbesondere gehen wir dabei auf Techniken zur Codeerzeugung für DMS sowie auf Datenaufteilungen und Laufzeitvorhersage ein.

Kapitel 4 referiert über grundlegende Techniken der syntaktischen Mustererkennung (tree pattern matching und endliche Baumautomaten), die wir für das PARAMAT–Mustererkennungssystem auf syntaktischer Ebene teilweise verwenden.

Kapitel 5 enthält nach einigen grundlegenden Definitionen eine Aufstellung aller bisher vorgesehenen Muster aus der Musterbibliothek. Ferner werden dort numerische Codes, Algorithmen und Standard–Software–Pakete auf ihren Gehalt an Mustern aus der Musterbibliothek hin untersucht.

In Kapitel 6 beschreiben wir das PARAMAT–Mustererkennungssystem. Wir legen dabei besonderes Gewicht auf die effiziente Ausnutzung der semantischen Hierarchie der Muster, auf die Mustererkennung entlang von Datenflußkanten sowie auf inkrementelle, normalisierende Transformationen der Zwischendarstellung während der Mustererkennung.

Kapitel 7 enthält die Konzeption der „wissensbasierten" Codeerzeugung aus den Musterinstanzen.

Die wichtigsten Ergebnisse und Beiträge dieser Arbeit werden in Kapitel 8 nochmals zusammengefaßt.

2 Sequentielle Quellsprache und Zwischendarstellung

2.1 Vereinfachte sequentielle Quellsprache für numerische Programme

Numerische Anwendungen werden derzeit noch immer vorwiegend in `FORTRAN` (77) geschrieben. Daneben hat sich die Programmiersprache C den zweiten Platz gesichert. Weitere imperative Programmiersprachen wie z.B. Modula-2 oder PASCAL spielen in der Numerik kaum eine Rolle. Funktionale Programmiersprachen wie FP werden allenfalls im akademischen Bereich für kleine isolierte Algorithmen benutzt. Der Einsatz logischer Programmiersprachen erscheint im numerischen Bereich nicht sinnvoll.

`FORTRAN` und C stimmen in vielen Eigenschaften überein; ja, `FORTRAN` scheint sich sogar auf C zuzubewegen[1]. Die Kontrollstrukturen sind sehr ähnlich. Viele im Sprachumfang von `FORTRAN` enthaltene Standardfunktionen, die im numerischen Anwendungsbereich von Interesse sind, kommen auch in C vor, und umgekehrt.

Allerdings gibt es auch signifikante Unterschiede. `FORTRAN77` hantiert eher mit statisch definierten Speicherblöcken (z.B. durch die `COMMON`-Anweisung) und kann so direkten Einfluß auf die Speicherung von Datenstrukturen nehmen, während C eher auf den Datenstrukturen selbst operiert. Dynamische Datenstrukturen kommen in `FORTRAN77` nicht vor, sind aber in numerischen Anwendungen nur selten notwendig. Während es beispielsweise in `FORTRAN` bei entsprechender Gestaltung des `COMMON`-Blocks erlaubt ist, auch auf Speicherzellen jenseits der deklarierten Grenzen eines Feldes zuzugreifen, kann dies in C zu einem Programmabsturz führen. Die Überdeckung zweier `COMMON`-Blöcke kann zu schwer identifizierbaren Alias-Beziehungen zwischen verschiedenen Datenobjekten führen.

In `FORTRAN` werden mehrdimensionale Felder immer in einer genau festgelegten Reihenfolge in den linearen Speicher abgebildet, auf die der Programmierer sich verlassen kann. So kann eine Matrix — grundsätzlich spaltenweise[2] abgelegt — auch als linearisierter Vektor adressiert[3] werden. In C wird zeilenweise linearisiert; man kann hier jedoch — mit Hilfe von Pointern — auch erreichen, daß die minderdimensionalen Teilfelder *unzusammenhängend* im Speicher abgelegt werden. Solch implizite Vereinbarungen ebenso wie explizite Kunstgriffe stören jedoch die Datenabhängigkeitsanalyse, die Ausrichtung von Feldachsen (Alignment) und die Datenaufteilung ganz erheblich. Aus diesem Grunde vereinbaren wir für unsere vereinfachte sequentielle Quellsprache, daß ein d-dimensionales Feld auch immer als solches angesprochen werden muß, d.h. als ein Feldzugriff mit d Indexausdrücken. Damit bleibt die Wahl der endgültigen Speicherabbildung dem Code–Erzeuger vorbehalten.

`FORTRAN` stellt als weiteren Basistyp den Datentyp `complex` zur Verfügung, der in C mit Hilfe zweier Gleitkommazahlen oder mit Strukturen explizit simuliert werden muß.

Wir haben uns entschlossen, als Quellsprache eine Teilmenge von C zuzulassen, erweitert um den Datentyp `complex`. Ausgeschlossen wurden:

- Unterprogramme (außer `main()` und den C–Standardfunktionen) und lokale Variablen
- Pointer und Strukturen (außer `complex`)
- Feldzugriffe über die Dimensionsgrenzen hinweg

[1] Zum Beispiel gibt es in `Fortran 90` auch Pointer und Strukturen wie in C.

[2] Im allgemeinen variiert beim Durchlaufen eines so linearisierten mehrdimensionalen Feldes der Index der ersten Dimension am schnellsten und der der letzten Dimension am langsamsten.

[3] Diese Flexibilität der Speicherabbildung ist in der Programmiersprache APL besonders stark ausgeprägt [Ive62].

- Mehrfachauswahl (`switch`-Anweisung)
- die Anweisungen `goto`, `break`, `continue`

Unterprogramme erschweren wichtige Transformationen, die Datenabhängigkeitsanalyse und die automatische Datenaufteilung[4]. Da numerische Programme fast nie rekursive Funktionen enthalten, kann diese Forderung auch durch *procedure inlining* erreicht werden. Nach Beobachtungen im PACE–Projekt [Hä94] und eigenen Untersuchungen zufolge wird der Programmcode bei numerischen Programmen im Durchschnitt nur um einen relativ kleinen Faktor (ca. 3) aufgebläht, was durchaus vertretbar ist.

Zur Übersetzungszeit eines Programms ist selten bekannt, wohin ein Pointer zur Laufzeit zeigen wird. In Ausnahmefällen kann dies durch eine eingehende Analyse des Quellprogramms herausgefunden oder stark eingegrenzt werden (Pointer Unaliasing, z.B. [HN90b]). Pointer ins „Ungewisse“ machen aber jegliche Datenabhängigkeitsanalyse hinfällig. Da Strukturen fast immer im Zusammenhang mit Pointern auftreten (z.B. bei Graphenalgorithmen), werden auch sie ausgeschlossen. Sowohl Pointer als auch Strukturen kommen jedoch in numerischen Programmen äußerst selten vor, sodaß auch diese Einschränkung nicht wesentlich ist.

Die Mehrfachauswahl (`switch`-Anweisung) tritt in numerischen Programmen ebenfalls sehr selten auf; sie kann auch leicht mit Hilfe der `if`-Anweisung simuliert werden.

Unstrukturierter Kontrollfluß, auch „Spaghetti-Code“ genannt, ist sowohl in `FORTRAN` als auch in C durch Verwendung von Labels und `GOTO`'s möglich. Damit wird jedoch eine Programmanalyse sehr erschwert. Da es in `FORTRAN77` noch keine strukturierten Äquivalente zur `if-then-else`-Anweisung und zur `while`-Anweisung von C gibt, werden diese dort notgedrungen mit `GOTO`'s programmiert, die sich aber oft einfach als strukturierte Kontrollanweisungen umschreiben lassen.

Wir sehen, daß die genannten Einschränkungen für numerische Programme keine große Rolle spielen. Wir fassen nachfolgend den Sprachumfang unserer „vereinfachten sequentiellen Quellsprache für numerische Programme“ zusammen, die wir LATINUS[5] nennen:

- Kontrollstrukturen: `for`-Schleife[6], `while`-Schleife, `if-then` und `if-then-else`-Anweisung. Die Semantik ist wie in C definiert.
- Datentypen: `integer`, `double`, `complex`, Felder von `integer`, `double` und `complex` beliebiger Dimensionalität.
- Arithmetische und Boolesche Operatoren: `+,-,*,/,++,--,=,==,!=,<,>,<=,>=,&&,||,!`
- Standardfunktionen:
  - Ein-/Ausgabe: `read` (wie `scanf()` ohne Adreßoperator) und `write` (wie `printf()`)
  - mathematisch, unär: `sin`, `cos`, `tan`, `log`, `exp`, `abs`, `sqrt`, `floor`, `ceil`, `round`
    mathematisch, binär: `max`, `min`, `pow` (Exponentiation a^b)

Ein Feldzugriff besteht aus Feldname und *Indizierung*. Die Indizierung wiederum besteht aus einem oder mehreren *Indexausdrücken*, jeweils in eckigen Klammern. Sei `A[idx1][idx2]...[idxd]` ein Feldzugriff auf das `d`-dimensionale Feld `A` mit den `d` Indexausdrücken `idx1` bis `idxd`. Tritt in einem der Indexausdrücke wiederum ein Feldzugriff auf, so spricht man von einem *indirektem* Feldzugriff. Indirekte Feldzugriffe stellen für die Datenabhängigkeitsanalyse ein Problem dar, da erst zur Laufzeit bekannt ist, welche Feldelemente referenziert werden. Sie sind aber leider für gewisse Probleme in der Numerik unverzichtbar, zum Beispiel beim Zugriff auf kompakt gespeicherte dünnbesetzte Matrizen.

[4] Auch die richtungsweisenden Arbeiten [GB90], [CKKM94] und [DHR94] haben Unterprogramme ausgeschlossen.

[5] LATINUS steht für „LAnguage To Implement NUmerical codes with Simplifications“

[6] Die `for`-Schleife sieht zwar syntaktisch so aus wie in C, muß jedoch — in Analogie zur `DO`-Schleife in `FORTRAN` — als Parameter gerade die Initialisierung, die Iterationsbedingung und die In– bzw. Dekrementierung derselben Schleifenvariable enthalten. Der Wert der Schleifenvariablen nach dem Verlassen der `for`-Schleife ist in LATINUS undefiniert.

2.2 Zwischendarstellung

Als Zwischendarstellung des Quellprogramms haben wir die des abstrakten Syntaxbaumes gewählt, die von unserem Frontend erzeugt wird (vgl. [ASU86]). Die deklarierten Variablen, einschließlich Typ und ggf. deklarierter Feldgrößen, werden in einer Symboltabelle abgelegt. Da wir keine Prozeduren verarbeiten müssen, brauchen wir uns um Sichtbarkeitsbereiche von Variablen und lokale Symboltabellen nicht zu kümmern.

Als Konstanten sind ganze Zahlen, Gleitkommazahlen und komplexe Gleitkommazahlen zugelassen, als Variablen ganzzahlige Variablen, Gleitkommavariablen, komplexwertige Variablen und beliebigdimensionale[7] Felder dieser drei Typen.

Ein *Bereichsausdruck* (*range*) `i=[lb:ub:st]` besteht aus einer ganzzahligen Variablen `i`, die alle Werte aus der Menge

$$range_i = \{\mathtt{lb}, \mathtt{lb}+\mathtt{st}, \mathtt{lb}+2\cdot\mathtt{st}, \ldots, \mathtt{lb}+k\cdot\mathtt{st}\} \quad \text{mit } \mathtt{lb}+k\cdot\mathtt{st} \leq \mathtt{ub} < \mathtt{lb}+(k+1)\mathtt{st}$$

annehmen kann, wobei `lb`, `ub` und `st` ganzzahlig (aber nicht notwendig konstant) sein müssen. Ein Bereichsausdruck faßt die Wirkung einer `for`–Schleife auf die Schleifenvariable `i` zusammen.

Eine `for`–Schleife heißt *indizierend*, wenn ihre Schleifenvariable in mindestens einer Feldreferenz innerhalb ihres Rumpfes vorkommt.

Eine *Feldreferenz* besteht, wie bereits erwähnt, aus dem Feldnamen und einem oder mehreren Indexausdrücken, die jeweils in eckigen Klammern eingeschlossen sind.

Eine ganzzahlige Variable, die im Rumpf einer Schleife inkrementiert wird und darin außerdem in mindestens einem Indexausdruck vorkommt, selbst aber keine Schleifenvariable ist, heißt *Induktionsvariable*. Diese Induktionsvariablen können, wie wir noch später erwähnen werden, durch geeignete Transformationen eliminiert werden, sodaß nur Schleifenvariablen „laufen“.

Eine Schleifenvariable `i` einer indizierenden Schleife heißt *gebunden*, wenn die zugehörige `for`–Schleife im betrachteten Teil–Syntaxbaum enthalten ist.

Eine Feldreferenz, in deren Indexausdrücken *keine* gebundene Variablen vorkommen, heißt *Skalar*. Eine Feldreferenz, bei der in *genau* einem Indexausdruck eine oder mehrere gebundene Schleifenvariablen vorkommen, heißt *Vektor*. Eine Feldreferenz, bei der in *genau* zwei Indexausdrücken eine oder mehrere gebundene Schleifenvariablen vorkommen, heißt *Matrix*[8]. Allgemein heißt eine Feldreferenz, bei der in *genau* $D > 0$ Indexausdrücken eine oder mehrere gebundene Schleifenvariablen vorkommen, *Tensor* D–ter Ordnung[9]. Zu Vektoren, Matrizen und höherdimensionalen Tensoren trägt der Mustererkenner bei Bedarf die Dimensionsnummern der Indexausdrücke, in denen gebundene Variablen vorkommen, sowie die entsprechenden gebundenen Variablen ein.

Arithmetische oder Boole'sche *Ausdrücke* werden, wie allgemein üblich, als Ausdrucksbäume dargestellt. Eine Ausnahme betrifft unäre Operationen: Die wichtigsten unären Operatoren `INV` (Inversion), `ABS` (Absolutbetrag) und `NEG` (Negation) werden (in dieser Reihenfolge absteigender Priorität) als gleichnamige Bits (*flags*)[10] in der Wurzel des ihrem Operanden entsprechenden Unterbaumes gesetzt, anstatt einen eigenen Knoten für den unären Operator zu konstruieren und den Operanden als „Sohn“ daranzuhängen[11]. Dies macht den Ausdruck kompakt und erleichtert später die Mustererkennung.

[7] Die maximal zulässige Anzahl von Dimensionen kann in der Konstanten `MAXDIM` beliebig voreingestellt werden.

[8] Man beachte, daß z.B. eine Diagonalmatrix, obwohl prinzipiell eindimensional, hier als spezielle Matrix und nicht als Vektor behandelt wird.

[9] Entsprechend werden bei dreidimensionalen Tensoren („Quadern“) prinzipiell ein- oder zweidimensionale Strukturen wie „Raumdiagonalen“ oder „Diagonalflächen“ nicht als Vektoren bzw. Matrizen, sondern als spezielle Quader behandelt.

[10] Weitere Flag–Bits sind für die Typkonversionen `FLOAT`, `FLOOR`, `CEIL` und `ROUND` vorgesehen.

[11] Diese Bits werden bereits vom Frontend entsprechend gesetzt. Als Konsequenz gibt es auch nicht mehr die binären Operationen Subtraktion und Division.

Typ \ Eintrag	val	lson	rson	e[1]	e[2]	e[3]	... bis e[`MAXDIM`]
`ival, dval, cval`	c						int/double/complex–Konstante `c`
`ivar, dvar, cvar`	x						nichtindizierte Variable `x`
`arref`	`A`			`i1`	`i2`	`i3`	skalare Feldreferenz `A[i1][i2][i3]`...
`vector`	”			”	”	”	eine gebundene Dimension
`matrix`	”			”	”	”	zwei gebundene Dimensionen
`rng` (range)		`i`	`xi`	`lb`	`ub`	`st`	Bereichsausdruck
BinOp (`plus`, `mult`,...)		`opnd1`	`opnd2`				binäre Operation
UnOp (`sin`, `cos`,...)		`opnd`					unäre Operation
`Sdo`		`stmt`	`i`	`lb`	`ub`	`st`	`for (i=lb;i<=ub;i++) stmt;`
`Swhile`		`stmt`		`cond`			`while (cond) stmt;`
`Sif`		`stmt1`	`stmt2`	`cond`			`if (cond) stmt1; else stmt2;`
`assign`		`x`	`e`				Zuweisung `x = e;`

Tabelle 2.1 Die wichtigsten Einträge eines Knotens im Syntaxbaum. Für jeden Ausdrucksknoten gibt es die Flag-Bits `INV`, `NEG` und `ABS`. Vektoren und Matrizen enthalten außerdem Einträge für die gebundenen Dimensionen und Verweise auf die Bereichsausdrücke der gebundenen Variablen. Eine Erweiterung für höherdimensionale Tensoren ist bei Bedarf möglich. Für jede Anweisung gibt es horizontale Verkettungsmöglichkeiten. In einem Bereichsausdruck kann optional ein ausgenommener Wert `xi` der Laufvariablen `i` spezifiziert werden. Auf weitere Einträge wird bei Bedarf im Text eingegangen.

Die zugelassenen *Anweisungen (statements)* sind die Zuweisung, die `for`–Schleife, die `while`–Schleife und die bedingte Anweisung `if-then` bzw. `if-then-else`. Die „vertikale“ Verkettung dieser Anweisungsknoten (z.B. Schleifenkopf ↔ Rumpf) folgt den üblichen hierarchischen Regeln [ASU86], wie in Tabelle 2.1 dargestellt. Ein Block von Anweisungen wird als doppelt verkettete Liste „horizontal“ verbunden. Ein spezieller Knoten mit Aufschrift `main` bildet die Wurzel des Syntaxbaumes für das gesamte Quellprogramm.

Tabelle 2.1 zeigt, wie die einzelnen Operanden intern abgespeichert werden.

2.3 Syntaktische Prädikate

Bei Analyse und Transformation des Quellprogramms ist es nützlich, für häufig benutzte Tests *syntaktische Prädikate* bereitzustellen.

Seien im folgenden `e`, `e1` und `e2` Ausdrücke, `v` eine Variable, `i` und `j` ganzzahlige Variablen.

- `isconst(e)` liefert TRUE g.d.w. `e` vom Typ `ival`, `dval` oder `cval` ist.
- `isvar(e)` liefert TRUE g.d.w. `e` vom Typ `ivar`, `dvar`, `cvar`, `arref`, `vector`, `matrix` oder `quader` ist.
- `isscalar(e)` liefert TRUE g.d.w. `isconst(e)` oder `e` vom Typ `ivar`, `dvar`, `cvar` oder `arref` ist.
- `isleaf(e)` liefert TRUE g.d.w. `isconst(e)` oder `isvar(e)`.
- Für zwei Variablenvorkommen `i`, `j` evaluiert `eqsym(i,j)` zu TRUE g.d.w. `i` und `j` Vorkommen der gleichen Variablen sind; die flag–Bits spielen jedoch keine Rolle (z.B. `eqsym(i,-1/i)` liefert TRUE).
- `eqfex(e,e1)` liefert TRUE, falls aus den syntaktischen Strukturen von `e` und `e1` geschlossen werden kann, daß für alle Belegungen der Variablen, die in `e` und `e1` vorkommen, der Wert von `e` zur Laufzeit stets gleich dem von `e1` ist.
- `eqex(e,e1)` arbeitet wie `eqfex(e,e1)`, wobei die flag–Bits beim Vergleich aber keine Rolle spielen.

- `geqex(e,e1)` liefert TRUE, falls aus den syntaktischen Strukturen von `e` und `e1` geschlossen werden kann, daß für alle Belegungen der Variablen, die in `e` und `e1` vorkommen, der Wert von `e` zur Laufzeit stets größer oder gleich dem von `e1` ist. `leqex( e, e1)` tut dasgleiche für den Fall „kleiner oder gleich". `gtex(e,e1)` entspricht `geqex(e,e1) AND NOT eqex(e,e1)`, `ltex(e,e1)` analog `leqex(e,e1) AND NOT eqex(e,e1)`.
- `occurin( e, e1 )` liefert TRUE g.d.w. `e1` syntaktisch ein Unterausdruck von `e` ist[12]. Man beachte, daß auch Indexausdrücke Unterausdrücke einer Feldreferenz sind, und daß für Anweisungen die Unteranweisungen (z.B. Rumpf einer Schleife, Bestandteile einer bedingten Anweisung) Unterausdrücke sind. Flags werden nicht berücksichtigt.
- `occuridx( e, e1 )` evaluiert für eine Feldreferenz, einen Vektor, eine Matrix oder einen höherdimensionalen Tensor `e` zu TRUE g.d.w. `e1` in mindestens einem Indexausdruck von `e` als Unterausdruck vorkommt.
- `isoffset( e )` liefert TRUE g.d.w. `e` von der Form `c + i` ist, wobei `c` eine ganzzahlige Konstante und `i` eine ganzzahlige Variable ist.
- `isstandard( e )` liefert TRUE g.d.w. `e` von der Form `c1 + c2*i` ist, wobei `c1` und `c2` ganzzahlige Konstanten sind (eventuell auch `0`) und `i` eine ganzzahlige Variable.
- `issimpleidx( e, i, rangingvars )` liefert TRUE g.d.w. `i` vom Typ `ivar` ist, `occurin(e1, i)` für genau einen Indexausdruck `e1` von `e` erfüllt ist, für diesen `isstandard(e1)` gilt[13], und `i` nicht in der Variablenliste `rangingvars` vorkommt.
- `isdoubleidx( e, i, rangingvars )` arbeitet wie `issimpleidx()`, wobei hier `occurin(e1, i)` für genau zwei Indexausdrücke `e1, e2` von `e` erfüllt sein muß.
- Falls zwei (verschiedene) ganzzahlige Variablen `i` und `j` in jeweils einem einzigen (aber nicht im selben) Indexausdruck einer Feldreferenz `e` vorkommen, so evaluiert `precedes( e, i, j )` zu TRUE g.d.w. der Indexausdruck, in dem `i` vorkommt, in der Indizierung von `e` vor demjenigen steht, der `j` enthält.

Die folgenden Operationen liefern einen Ausdruck als Ergebnis:

- `eval( e, v, e1 )` liefert eine Kopie von `e`, in der alle Vorkommen der Variablen `v` durch Kopien von `e1` ersetzt sind. Dabei entstehende konstante Teilausdrücke werden sofort ausgewertet.
- `negex( e )` negiert `e` symbolisch, `invex( e )` invertiert `e` symbolisch, `plusex( e, e1 )` addiert `e` und `e1` symbolisch, und `multex( e, e1 )` multipliziert `e` und `e1` symbolisch. Die Rückgabewerte sind jeweils neue Ausdrücke. Konstante Ausdrücke werden sofort ausgewertet.

[12] Dies entspricht in der Darstellung von `e` als abstraktem Syntaxbaum der Übereinstimmung eines Unterbaumes von `e` mit dem abstrakten Syntaxbaum von `e1`.

[13] Die aktuelle Implementierung des Prototypen verlangt hier noch `eqex(e1,i)`.

3 Grundlagen der Automatischen Parallelisierung

3.1 Datenabhängigkeiten

Die Semantik sequentieller Programmiersprachen wie `FORTRAN77` oder C impliziert eine genau festgelegte lineare Reihenfolge für die Ausführung von Anweisungen. Diese Reihenfolge kann durch Transformationen wie das Vertauschen von Anweisungen, Vertauschen von Schleifenköpfen usw., aber auch durch das parallele Ausführen von Anweisungen geändert werden. Wenn dabei Datenabhängigkeiten verletzt werden, wird sich i.allg. die Semantik des Programms verändern. Um dies zu verhindern, müssen alle Datenabhängigkeiten vor Anwendung solcher Transformationen bekannt sein. Eine *exakte* Berechnung der Datenabhängigkeiten (Datenfluß) ist im allgemeinsten Fall nicht (Halteproblem) oder nur mit unvertretbarem Aufwand möglich; man versucht daher, eine (möglichst kleine) Obermenge der tatsächlich vorhandenen Datenabhängigkeiten zu berechnen, um eine konservative Abschätzung zu erhalten.

Wir können uns eine explizite Berechnung von *Kontrollabhängigkeiten* ersparen, da nach Voraussetzung das Quellprogramm in strukturierter Form vorliegt und `GOTO`–Sprünge ausdrücklich verboten wurden — der Kontrollfluß geht unmittelbar aus der hierarchischen Programmstruktur (Schleifen, Bedingungen) hervor. Möchte man Sprunganweisungen zulassen, so muß man den Kontrollabhängigkeitsgraphen des Programms bestimmen (siehe [FOW87, ASU88]). Datenfluß (z.B. *reaching definitions*) bzw. Datenabhängigkeiten ergeben sich dann in Abhängigkeit des Kontrollflusses bzw. der Kontrollabhängigkeiten.

Wir lehnen uns in diesem Abschnitt weitgehend an die Terminologien der klassischen Arbeiten [KKP+81, BC86, Ban88] sowie an [ZC90] an, erlauben aber beliebige ganzzahlige Schrittweiten in `for`–Schleifen anstatt `+1`.

3.1.1 Ausführungsreihenfolge und Datenabhängigkeiten

Betrachten wir zunächst einen Grundblock[1] von Zuweisungen. Im folgenden bezeichne $S_1 \prec S_2$ (S_1 *vor* S_2) die lineare, dynamische Reihenfolge für die (sequentielle) Ausführung zweier Anweisungen S_1 und S_2. Für eine Anweisung S bezeichne $DEF(S)$ die Menge aller Variablen, die durch S geschrieben werden, und $USE(S)$ die Menge aller Variablen, die durch S gelesen werden. Falls $DEF(S)$ oder $USE(S)$ nicht exakt berechnet werden können, so muß eine Obermenge berechnet werden.

Definition 3.1 Datenabhängigkeiten Seien S_1, S_2 Anweisungen mit $S_1 \prec S_2$.

Falls $DEF(S_1) \cap USE(S_2) \neq \emptyset$, so besteht eine *echte* oder *Fluß–Abhängigkeit* (*true dependence*) von S_1 nach S_2, bezeichnet mit $S_1 \delta^t S_2$.

Falls $USE(S_1) \cap DEF(S_2) \neq \emptyset$, so besteht eine *inverse* oder *Anti–Abhängigkeit* (*anti dependence*) von S_1 nach S_2, bezeichnet mit $S_1 \delta^a S_2$.

Falls $DEF(S_1) \cap DEF(S_2) \neq \emptyset$, so besteht eine *Ausgangs–Abhängigkeit* (*output dependence*) von S_1 nach S_2, bezeichnet mit $S_1 \delta^o S_2$. □

Falls $S_1 \delta^t S_2$, $S_1 \delta^a S_2$ oder$S_1 \delta^o S_2$, so kann dies auch durch $S_1 \delta S_2$ abgekürzt werden. $S_1 \delta S_2$ besagt, daß zwischen S_1 und S_2 eine wirkliche Datenabhängigkeit bestehen könnte, aber wegen i.allg. nicht exakter Berechnung von *DEF* und *USE* nicht unbedingt bestehen, sondern nur sicherheitshalber angenommen werden muß. Falls $S_1 \delta S_2$, darf die Reihenfolge der Ausführung von S_1 und S_2 nicht verändert werden, um die Semantik des Programms zu erhalten.

[1] Ein Grundblock (*basic block*, vgl. [ASU86]) ist eine Folge von Zuweisungen, innerhalb derer keine Verzweigung oder Vereinigung des Kontrollflusses stattfindet.

Der *Datenabhängigkeitsgraph* eines Grundblocks besteht aus allen seinen Anweisungen als Knoten, und eine Kante verläuft von S_1 nach S_2 genau dann wenn $S_1 \delta S_2$. Die Kante wird mit dem Typ der Datenabhängigkeit (δ^t, δ^a oder δ^o) beschriftet.

Die Schleifenvariable einer Schleife `for (i=lb; i<=ub; i+=st)`[2] kann $N_i = \lfloor$`(ub-lb+1)/st`$\rfloor$ verschiedene Werte $i =$`lb`, `lb+st`, ... annehmen. Für k verschachtelte Schleifen

```
for (i1=lb1; i1<=ub1; i1+=st1)
  for (i2=lb2; i2<=ub2; i2+=st2)
    ....
      for (ik=lbk; ik<=ubk; ik+=stk)
        S;
```

wird die Anweisung S insgesamt bis zu $N_{\mathtt{i1}} \cdot N_{\mathtt{i2}} \cdot ... \cdot N_{\mathtt{ik}}$ mal[3] ausgeführt. Eine jede dieser Ausführungen bezeichnen wir mit $S(i_1, i_2, ..., i_k)$, wobei i_q den jeweiligen Wert der Schleifenvariablen `iq` bezeichnet, $1 \leq q \leq k$. Einen solchen S indizierenden Vektor $\vec{i} = (i_1, ..., i_k)$ nennen wir *Iterationsvektor*.

Aus dem Iterieren jeder Schleife gemäß den Konventionen der jeweiligen Programmiersprache ergibt sich eine feste lineare Reihenfolge $\lhd$ von Iterationsvektoren und damit auch für die verschiedenen Ausführungen von S.

Falls beispielsweise alle Schleifen vorwärts laufen, gibt die lexikalische Ordnung der Iterationsvektoren gerade die Ausführungsreihenfolge der $S(i_1, ..., i_k)$ an, d.h. für $\vec{i} <_{lex} \vec{j}$ gilt mit $<_{lex} = \lhd$ auch $S(\vec{i}) \prec S(\vec{j})$.

Für zwei verschiedene Anweisungen S_1 und S_2 im Rumpf derselben Schleifenschachtelung gilt $S_1(\vec{i}) \prec S_2(\vec{j})$ genau dann, wenn $(\vec{i} \lhd \vec{j}) \vee ((\vec{i} = \vec{j}) \wedge (S_1 \prec S_2))$.

Definition 3.2 Befinden sich zwei (nicht notwendig verschiedene) Anweisungen S_1 und S_2 im Rumpf von k verschachtelten Schleifen, $k \geq 1$, so besteht eine Datenabhängigkeit $S_1 \delta S_2$ genau dann, wenn es zwei (nicht notwendig verschiedene) Iterationsvektoren $\vec{i}$, $\vec{j}$ gibt mit $S_1(\vec{i}) \delta S_2(\vec{j})$, d.h. falls $DEF(S_1(\vec{i})) \cap USE(S_2(\vec{j})) \neq \emptyset$ oder $USE(S_1(\vec{i})) \cap DEF(S_2(\vec{j})) \neq \emptyset$ oder $DEF(S_1(\vec{i})) \cap DEF(S_2(\vec{j})) \neq \emptyset$.

Die Datenabhängigkeit heißt *schleifenunabhängig* (loop independent), falls $\vec{i} = \vec{j}$. Wir notieren dies als $S_1 \delta_\infty S_2$.

Die Datenabhängigkeit heißt *schleifengetragen* (loop carried) *in Höhe* h, falls $i_q = j_q$ für $1 \leq q \leq h-1$ und $i_h \neq j_h$. h heißt die *Tiefe* der Datenabhängigkeit. Wir notieren dies als $S_1 \delta_h S_2$. □

Eine schleifengetragene Datenabhängigkeit würde verschwinden, wenn man die sie tragenden Schleifenköpfe entfernen würde.

Definition 3.3 Der Datenabhängigkeitsgraph G_h der Tiefe h zu einer Schleifenverschachtelung der Gesamttiefe k besteht aus den Anweisungen im Rumpf der Schleife h, und eine Kante (S_1, S_2) verbindet zwei (nicht notwendig verschiedene) Anweisungen S_1, S_2, genau dann, wenn $S_1 \delta_q S_2$ für mindestens ein q mit $h \leq q \leq k$. □

Eine Datenabhängigkeit muß angenommen werden, sofern sie nicht für alle Paare $(\vec{i}, \vec{j})$ von Iterationsvektoren sicher ausgeschlossen werden kann. Sie wird bereits angenommen, falls sie für nur ein solches Paar bestehen kann. Damit können eventuell trotzdem vorhandene Möglichkeiten zur parallelen Abarbeitung anderer, eigentlich unabhängiger Ausführungen nicht genutzt werden, falls es nicht möglich ist, gerade die die Abhängigkeit(en) verursachenden Iterationsvektorpaare explizit zu berechnen.

Es ist das Bestreben der *exakten Datenflußanalyse für Felder*, dieses Manko abzustellen. Hierzu ist aber im allgemeinen ein beträchtlicher Aufwand erforderlich, sofern die Berechnung exakten Datenflusses überhaupt möglich ist. Wir werden auf dieses Problem in Abschnitt 6.3 zurückkommen.

[2] Falls `ub<lb` und `st`< 0, muß die Wiederholungsbedingung `i>=ub` lauten.

[3] Dieser Wert wird gerade erreicht, falls die Schleifenunter– und –obergrenzen alle voneinander unabhängig sind, d.h. keine Schleifenvariablen übergeordneter Schleifen enthalten.

3.1.2 Verfahren zum Testen von Datenabhängigkeit

Beim Testen der Möglichkeit, alle Datenabhängigkeiten zwischen zwei Anweisungen S_1, S_2 in einer Schleifenverschachtelung von n Schleifen sicher ausschließen zu können, zieht man sich i.allg. auf den relativ einfachen und häufig vorkommenden Fall zurück, daß die Indexausdrücke aller Feldreferenzen $A[e_1]...[e_{d_A}]$ lineare Funktionen $e_s = a_{s,0} + \sum_{r=1}^{n} a_{s,r} i_r$, $1 \leq s \leq d_A$, in den Schleifenvariablen $i_1, ..., i_n$ und alle $a_{s,r}$ konstant sind.

Seien für $S_1 \prec S_2$ z.B. $A[e_1]...[e_{d_A}] \in \mathit{DEF}(S_1)$ und $A[f_1]...[f_{d_A}] \in \mathit{USE}(S_2)$. Seien die Indexausdrücke $e_s = a_{s,0} + \sum_{r=1}^{n} a_{s,r} i_r$, $1 \leq s \leq d_A$ und $f_s = b_{s,0} + \sum_{r=1}^{n} b_{s,r} i_r$, $1 \leq s \leq d_A$.

Damit ergibt sich aus obengenannter Definition der Datenabhängigkeit zwischen S_1 und S_2 die folgende *Abhängigkeitsgleichung*:

$$a_{s,0} + \sum_{r=1}^{n} a_{s,r} i_r \quad = \quad b_{s,0} + \sum_{r=1}^{n} b_{s,r} j_r, \qquad 1 \leq s \leq d_A \tag{3.1}$$

Eine Datenabhängigkeit zwischen S_1 und S_2 aufgrund dieser Vorkommen von A besteht genau dann, wenn diese Abhängigkeitsgleichung ganzzahlige Lösungen $(\vec{i}, \vec{j})$ besitzt, die auch

$$i_r \in \mathit{range}_{i_r} \quad \wedge \quad j_r \in \mathit{range}_{j_r}, \quad 1 \leq r \leq n \tag{3.2}$$

erfüllen.

Falls $\vec{i} \trianglelefteq \vec{j}$, so handelte es sich um eine Flußabhängigkeit, andernfalls um eine Anti–Abhängigkeit.

Entsprechendes gilt für die Fälle $A[e_1]...[e_{d_A}] \in \mathit{USE}(S_1)$ und $A[f_1]...[f_{d_A}] \in \mathit{DEF}(S_2)$ bzw. $A[e_1]...[e_{d_A}] \in \mathit{DEF}(S_1)$ und $A[f_1]...[f_{d_A}] \in \mathit{DEF}(S_2)$.

Zur approximativen Lösung der Abhängigkeitsgleichung, einer linearen Diophantischen Gleichung, gibt es mehrere Möglichkeiten:

- *ggT–Test*: Gleichung 3.1 besitzt eine ganzzahlige Lösung $(\vec{i}, \vec{j})$, falls

  $$\mathrm{ggT}(a_{s,1} - b_{s,1}, a_{s,2} - b_{s,2}, \ldots, a_{s,n} - b_{s,n}) \mid (a_{s,0} - b_{s,0}) \ \text{ für alle } s, \ \ 1 \leq s \leq d_A \tag{3.3}$$

  Aus der damit festgestellten Existenz einer Lösung kann aber nicht geschlossen werden, ob diese auch Gleichung 3.2 erfüllt.

- *Banerjee–Test*: Im Gegensatz zum ggT–Test bestimmt der Banerjee–Test [Ban88], ob es eine *rationale* Lösung $(\vec{i'}, \vec{j'})$ von Gleichung 3.1 im Bereich 3.2 gibt. Der Banerjee–Test bestimmt mit Hilfe von Gleichung 3.2 die untere Schranke $LB_{s,r}$ und die obere Schranke $UB_{s,r}$ jedes Terms $a_{s,r} i_r - b_{s,r} j_r$, $1 \leq r \leq n$, $1 \leq s \leq d_A$. Falls $LB_{s,r} > b_{s,0} - a_{s,0}$ oder $UB_{s,r} < b_{s,0} - a_{s,0}$, kann Gleichung 3.1 keine rationale Lösung haben. Die Existenz einer rationalen Lösung besagt jedoch nicht, daß es auch eine ganzzahlige Lösung gibt, d.h. der Banerjee–Test ist eine hinreichende, aber keine notwendige Bedingung für Unabhängigkeit.

- *Separabilitätstest*: Der Separabilitätstest ist ein exakter Test, der aber nur für einen Spezialfall angewendet werden kann, wenn nämlich in Gleichung 3.1 in jeder Dimension s, $1 \leq s \leq d_A$, nur eine einzige Schleifenvariable vorkommt, d.h. die $a_{s,r}$, $b_{s,r}$ für jeweils höchstens ein r, $1 \leq r \leq n$, ungleich Null sind [ZC90].

ggT–Test und Banerjee–Test sind die beiden grundlegenden Verfahren zum Testen auf Unabhängigkeit. Durch Kombination und Verfeinerung beider Verfahren ergeben sich weitere Abhängigkeitstests (I–Test [KKP91], λ–Test [LYZ90], hierarchischer Test [BC86], Power–Test [WT92]).

Alternativ kann man die Gleichungen 3.1 und 3.2 als Instanz eines ganzzahligen Linearen Programmier–Problems (*integer linear programming*) interpretieren. Dazu existieren bekannte Lösungsverfahren wie z.B. auf Basis des Simplex–Algorithmus, der Shostak–Algorithmus [Sho81][BC86] oder Fourier–Motzkin–Elimination [WT92], die zwar eine exakte Lösung liefern, dafür aber unter Umständen exponentielle Laufzeit haben können und sich daher nur für interaktive Systeme zum Experimentieren mit Datenabhängigkeiten und Transformationen an kleinen Beispielen eignen, wie z.B. TINY [Wol91].

3.1.3 Richtungsvektoren

Definition 3.4 Sei $(\vec{i}, \vec{j})$ eine Lösung der Abhängigkeitsgleichung 3.1. Ein Vektor $\theta = \theta_1, ..., \theta_n$ mit $\theta_r \in \{<, >, =\}$ heißt *Richtungsvektor* zu $(\vec{i}, \vec{j})$, falls für alle r, $1 \leq r \leq n$ gilt[4]:

$\theta_r =$ '<' $\iff$ $i_r \lhd j_r$,
$\theta_r =$ '>' $\iff$ $i_r \rhd j_r$, und
$\theta_r =$ '=' $\iff$ $i_r = j_r$. □

Zusätzlich kann man die Symbole '$\leq$', '$\geq$' und '$*$' einführen, die entsprechende Disjunktionen zweier oder aller drei Richtungssymbole bedeuten.

Man kann den Banerjee–Test so umformulieren, daß er sich auf den Beweis der Unabhängigkeit bei gleichzeitiger Vorgabe eines bestimmten Richtungsvektors beschränkt. Damit kann man bei mehrfachen Schleifen ($n > 1$) einen *hierarchischen Abhängigkeitstest* [BC86] formulieren, sodaß man beim Scheitern der Standardtests für den allgemeinsten Fall $\theta = (*, *, ..., *)$ anschließend für die spezielleren Fälle $\theta = (<, *, ..., *)$, $(=, *, ..., *)$ und $(>, *, ..., *)$ testet, notfalls in der zweiten, dritten usw. Schleifenvariable rekursiv weiter verfeinert, bis schließlich für alle Zweige des so generierten Entscheidungsbaumes Unabhängigkeit gezeigt oder für bestimmte Richtungsvektoren Abhängigkeit nicht ausgeschlossen werden kann.

3.1.4 Linearisierung

ggT–Test und Banerjee–Test können bei mehrdimensionalen Feldern in zwei Varianten angewendet werden:

1. auf die in Gleichung 3.1 beschriebene Variante, bei der alle Dimensionen des Feldes A separat auf Abhängigkeit geprüft werden, und

2. auf die *linearisierte* Variante, die simultan alle Dimensionen auf einmal testet.

Beide Verfahren haben Vor– und Nachteile. Beispielsweise könnte die zweite Variante nicht feststellen, daß die Feldzugriffe `A[i][i]` und `A[i][i+1]` für beliebiges `i` niemals dasselbe Feldelement adressieren können, wohingegen die linearisierte Variante des ggT–Tests die Unabhängigkeit beweisen könnte. Auf der anderen Seite können beispielsweise in manchen Dimensionen vorkommende unbekannte symbolische Variablen bei Linearisierung den gesamten Test unbrauchbar machen, obwohl für einzelne Dimensionen Unabhängigkeit gezeigt werden könnte.

Für Gleichung 3.2 müssen wir sogar die linearisierte Version (*Banerjee–Ungleichung*, vgl. [Ban88]) heranziehen, falls in mindestens einer Schleifenunter– bzw. –obergrenze eine Schleifenvariable einer übergeordneten Schleife vorkommt.

3.2 Transformationen

Programmtransformationen verändern in der Regel die Ausführungsreihenfolge von Anweisungen, ohne die Semantik des Programms durch Mißachtung von Datenabhängigkeiten zu verletzen.

Man kennt heute über hundert verschiedene Programmtransformationen. Die wichtigsten Transformationen, die für uns im Rahmen dieser Arbeit von Bedeutung erscheinen, werden nachfolgend aufgeführt und kurz erläutert. Für weitergehende Untersuchungen allgemeiner Art verweisen wir auf [Lov77], [PW86], [Pol88], [Wol89] und [ZC90].

[4] Hier entspricht $\vec{i}$ dem Iterationsvektor einer Definition und $\vec{j}$ dem einer Benutzung desselben Feldes (Fluß–Abhängigkeit). Bei Anti–Abhängigkeiten ergeben sich die Richtungsvektoren aus den umgekehrten Fällen $i_r \rhd j_r$ bzw. $i_r \lhd j_r$ (vgl. [BC86]).

Viele Transformationen besitzen inverse bzw. Umkehrtransformationen. [WS90] beschreibt die Wechselwirkungen zwischen einigen Standardtransformationen. Im allgemeinen ist das Problem der optimalen bzw. sinnvollen Auswahl und Anwendungsreihenfolge von Transformationen ungelöst. Wir beschreiben in Abschnitt 7.1, wie unser System dieses Problem anpackt.

3.2.1 Prozedur-Expansion (procedure inlining)

Numerische Anwendungsprogramme enthalten äußerst selten rekursive Funktionen. Daher kann man die Aufrufe einer Funktion durch deren Rumpf ersetzen, wenn man die formalen Parameter durch die aktualen Parameter des Aufrufs ersetzt und die lokalen Variablen der Funktion (ggf. nach Umbenennung) in die aufrufende Funktion integriert.

In numerischen Applikationen werden Funktionen typischerweise weniger zur Kompaktierung des Programmcodes als vielmehr zur optischen und gedanklichen Gliederung des Programmtextes verwendet. Obwohl sich theoretisch Programme konstruieren lassen, deren Länge durch Prozedur–Expansion exponentiell wächst, bläht das Expandieren von Funktionen, wie wir bereits in Abschnitt 2.1 erwähnten, typische Anwendungsprogramme im Mittel nur um einen kleinen konstanten Faktor auf[5].

Der Vorteil expandierter Funktionen ist, daß die beschriebenen Datenabhängigkeitstests nun mehr Information zur Verfügung haben, und daß optimierende Transformationen auf größeren Codestücken ein größeres Betätigungsfeld finden, anstatt an den Aufrufgrenzen haltzumachen.

3.2.2 Konstantenpropagation (constant propagation)

Man betrachte das folgende Beispiel:

```
N = 1024;
....
NP1 = N + 1;
....
```

Wir nehmen für dieses Beispiel an, daß es keine weitere, anderslautende Zuweisung an `NP1` gebe.

Derartige Konstrukte werden häufig benutzt, um z.B. den Übersetzer zu zwingen, für `NP1` ein Register zu benutzen. Dennoch ist `NP1` eine schon zur Übersetzungszeit bekannte Konstante und kann durch ihren Wert `1025` ersetzt werden. Wird sie es nicht, so könnten unter Umständen bei Datenabhängigkeitstests aufgrund von worst–case–Annahmen für `NP1` als symbolische Variable gewisse Datenabhängigkeiten nicht ausgeschlossen werden und somit Möglichkeiten zur Parallelisierung verschenkt werden.

3.2.3 Eliminieren von Induktionsvariablen

Induktionsvariablen sind ganzzahlige Variablen, die benutzt werden, um Feldindexausdrücke abzukürzen. Man versucht, dadurch die Zeit zur Adreßberechnung zu optimieren und/oder den Übersetzer zu veranlassen, den Wert des Indexausdrucks in einem Register zu halten. Im Gegensatz zum vorhergehenden Beispiel werden Induktionsvariablen in jeder Iteration einer Schleife (genau einmal) geschrieben bzw. inkrementiert (daher der Name), ohne selbst Schleifenvariable zu sein.

Beispiel 3.1

[5] Die resultierende Vergrößerung des Programmcodes ist dabei weniger schwerwiegend als die durch das Einfügen von message–passing–Anweisungen entstehende Programmaufblähung um einen Faktor 3 bis 10 [HKT91b].

```
   DO 20 J=2,NC
    JF=J+J
    DO 10 I=2,NC
     IF1=I+I
     FC(I,J)=2*(FF(IF1,JF)-4.0*UF(IF1,JF)+UF(IF1,JF-1)
  *           + UF(IF1-1,JF)+UF(IF1+1,JF)+UF(IF1,JF+1))
10  CONTINUE
20 CONTINUE
```

Die Einführung der Induktionsvariablen `JF` und `IF1` erspart etliche Multiplikationen mit 2, aber macht das Programm auch unleserlicher; außerdem wird die Abhängigkeitsanalyse behindert, die davon ausgeht, daß nur Schleifenvariablen inkrementiert werden. Daher ersetzt man die Induktionsvariablen durch entsprechende Ausdrücke, die nur noch von Schleifenvariablen abhängen, z.B. hier durch `2*J` bzw. `2*I` (*scalar forward substitution*). ◇

Falls die Induktionsvariablen in jeder Iteration nicht einfach um eine Konstante inkrementiert werden, sondern vielmehr um eine sich in der Schleife ändernde Variable, wird das Hinzuziehen von Mustererkennungstechniken und Hintergrundwissen erforderlich, um eine explizite Formel (*closed form*) für die Folge der Werte der Induktionsvariablen in Abhängigkeit der Werte der Schleifenvariablen anzugeben [Pol88, EHLP91, AH90]:

```
j = 0;
for (i=1; i<=n; i++) {  j = j + i;   a[i] = a[i] + b[j];  }
```

Durch Analyse finden wir heraus, daß in der `i`–ten Iteration der Schleife die Invariante $\mathtt{j} = \sum_{k=1}^{\mathtt{i}} k$ gilt. Wir kennen, z.B. aus einer Formelsammlung, eine äquivalente nichtrekursive Berechnungsformel, womit sich das Programm wie folgt umformulieren läßt:

```
for (i=1; i<=n; i++) {  a[i] = a[i] + b[(i*(i+1))/2];  }
```

Ferner kann es vorkommen, daß die Induktionsvariable auch auf höheren Schleifenebenen inkrementiert wird. In diesem Fall bilden die von der Induktionsvariable angenommenen Werte zwar keine arithmetische Folge, aber trotzdem läßt sich oft zur Übersetzungszeit eine explizite Formel angeben [EHLP91].

3.2.4 Eliminieren von temporären Variablen

Temporäre Variablen benutzt ein Programmierer für gewöhnlich, um gemeinsame Teilausdrücke zu kennzeichnen, um die Verwendung eines Registers zu veranlassen oder um einfach die Leserlichkeit des Programms zu erhöhen.

Man betrachte folgendes Beispiel einer Matrix–Matrix–Multiplikation:

Beispiel 3.2

```
do i=1,64
 do j=1,64
  temp = 0.0
  do k=1,64
   temp = temp + a(i,k) * b(k,j)
  enddo
  c(i,j) = temp
 enddo
enddo
```

Die Variable `temp` wird als temporär erkannt[6] und kann daher durch `c(i,j)` ersetzt werden. ◇

[6] Wir nehmen in diesem Beispiel an, daß die Variable `temp` nach diesem Programmstück nicht vor einer eventuellen Neuzuweisung nochmals benutzt wird.

3.2.5 Vereinfachung von Feldern (array simplification)

Ein (eventuell mehrdimensionales) Feld, dessen d–te Dimension eine Größe (extent) E_d^A aufweist, die kleiner ist als ein festgesetzter Schwellenwert (z.B. 8), wird in E_d^A Einzelfelder $A1, \ldots, AE_d^A$ mit jeweils Dimensionalität $d-1$ aufgespalten. Schleifen, die die d–te Dimension von A indizieren, müssen dabei vollständig abgerollt (d.h. $E_d^A - 1$ mal repliziert) werden, und alle Zugriffe auf $A[e_1]...[e_d]...[e_k]$ müssen durch $Ae_d[e_1]...[e_{d-1}][e_{d+1}]...[e_k]$ ersetzt werden. Tauchen dabei in e_d Variablen auf, deren Wert zur Übersetzungszeit nicht konstant oder nicht bekannt ist, kann diese Transformation nicht durchgeführt werden.

3.2.6 Entfernung von totem oder nutzlosem Code (dead code elimination)

Toter Code sind Anweisungen, die vom Kontrollfluß niemals erreicht werden. Im ursprünglichen Programm natürlich selten, kann toter Code doch im Zuge von Programmtransformationen entstehen. Wenn sich zum Beispiel durch Konstantenpropagation eine Bedingung zur Übersetzungszeit als konstant wahr oder falsch herausstellt, sind die Bedingung und ggf. die im „falschen“ Zweig liegenden Anweisungen überflüssig und können entfernt werden.

Nutzlose Anweisungen berechnen Daten, die niemals benutzt werden. Nutzlose Anweisungen sind auch Zuweisungen einer Variable an sich selbst: In Beispiel 3.2 wird die Zuweisung `c(i,j) = temp` nach dem Ersetzen aller Vorkommen von `temp` durch `c(i,j)` umgewandelt in die nutzlose Zuweisung `c(i,j) = c(i,j)`, die ersatzlos gestrichen werden kann.

3.2.7 Aufgliederung bedingter Anweisungen (`IF` distribution)

Nützlich z.B. bei der Vektorisierung ist das Aufgliedern einer bedingten Anweisung, deren `then`– oder `else`–Teil aus mehr als einer Anweisung besteht:

```
if (cond) { S1; S2; ... Sn; }
else      { T1; T2; ... Tm; }
```

Falls in keiner der Anweisungen `S1`,...,`Sn`,`T1`,...,`Tm` eine Variable geschrieben wird, die in `cond` vorkommt, darf die Bedingung `cond` über alle Anweisungen wie folgt repliziert werden:

```
if (cond) S1;
if (cond) S2;
...
if (cond) Sn;
if (!(cond)) T1;
if (!(cond)) T2;
...
if (!(cond)) Tm;
```

3.2.8 Normalisierung von Schleifen (loop normalization)

Unter Normalisierung von Schleifen versteht man das Umformen von Schleifengrenzen und Schleifenrumpf, sodaß sich eine Schrittweite von `+1` ergibt (cf. [ZC90]). Viele Datenabhängigkeitstests (z.B. in [KKP+81, BC86, Ban88, Wol89, ZC90, GKT91, WT92]) erwarten eine Schleifenschrittweite von `+1`, die durch Normalisierung bereitgestellt werden muß. Mancherorts wird auch als Untergrenze `1` gefordert [AK87]. Da diese Transformation die äußere Erscheinung einer Schleife stark verfremden kann, haben wir bewußt darauf verzichtet. Aus diesem Grunde haben wir bei der Definition von Datenabhängigkeiten in Schleifen beliebige ganzzahlige Inkremente ausdrücklich zugelassen.

3.2.9 Schleifen-Blockung (loop blocking, strip mining, loop tiling)

Beim Blocken von Schleifen wird der Iterationsraum in (abgesehen von Randstücken) gleich große Blöcke aufgeteilt. Dabei kann entweder die Anzahl oder die Größe der Blöcke vorgegeben werden. Wird eine einzelne Schleife geblockt, so wird der Iterationsraum in Streifen aufgeteilt (*strip mining*):

Beispiel 3.3

```
for (i=1; i<=n; i++)
  temp = temp + D[i];
```

Durch Blockung der Schleife mit der Blockgröße 4 erhält man

```
for (i=1; i<=n; i+=4)
  for (ii=i; ii<=min(n,i+3); ii++)
    temp = temp + D[ii];
```

◇

Werden mehrere Schleifen gleichzeitig geblockt, so wird der Iterationsraum in entsprechend vielen Dimensionen von wie Fliesen aneinandergrenzenden Blöcken überdeckt (*loop tiling*).

Beispiel 3.4

```
for (i=1; i<=n; i++)
  for (j=1; j<=m; j++)
    A[i][j] = B[i][j] + C[i][j];
```

Blockt man die äußere Schleife mit dem Faktor 16 und die innere mit dem Faktor 32, so erhält man

```
for (i=1; i<=n); i+=16)
  for (j=1; j<=m; j+=32)
    for (ii=i; ii<=min(n,i+15); ii++)
      for (jj=j; jj<=min(n,j+31); jj++)
        A[ii][jj] = B[ii][jj] + C[ii][jj];
```

◇

Sinnvoll ist diese Beschränkung der „Fliesengröße" (hier: $16 \cdot 32$) vor allem, wenn die Operanden jedes Blocks bei dieser Größe gerade noch in die Vektorregister oder Datencaches der Zielmaschine passen ([KPR92], [WL91]). Man kann aber auch durch geeignete Wahl der Blockgröße bei Abhängigkeitszyklen mit Distanzen größer als 1 zusätzlichen Parallelismus gewinnen (*cycle shrinking*, [Pol88]).

Die Umkehrtransformation nennen wir *Entblocken* von Schleifen (*loop unblocking*).

3.2.10 Abwickeln einzelner Iterationen (loop peeling)

Möchte man sich in Beispiel 3.3 die Berechnung des Terms `min(n,i+3)` in jeder Iteration der `i`–Schleife ersparen, kann man die Iterationen für das „Randstück" als separate Schleife schreiben.

Die Umkehrtransformation ist das Zusammenziehen benachbarter Schleifen.

3.2.11 Abrollen von Schleifen (loop unrolling)

Ist wie in Beispiel 3.3 die Blockgröße zur Übersetzungszeit bekannt, so kann man die durch das Blocken entstandene innere Schleife abrollen, indem man für jeden Wert der inneren Schleifenvariablen eine Kopie des Schleifenrumpfes anlegt und diese Werte explizit einsetzt:

```
for (i=1; i<=n; i+=4) {
    temp = temp + D[i];
    temp = temp + D[i+1];
    temp = temp + D[i+2];
    temp = temp + D[i+3];
}
```

Wir bezeichnen dies als Abrollen auf Anweisungsebene. Alternativ könnte man die vier Anweisungen zu einer einzigen zusammenfassen:

```
for (i=1; i<=n; i+=4)
    temp = temp + D[i] + D[i+1] + D[i+2] + D[i+3];
```

Dies bezeichnen wir als Abrollen auf Ausdrucksebene (*jamming*). Das Abrollen von Schleifen wird in sequentiellen Programmen häufig angewendet, um den Schleifen–Overhead (Inkrementieren der Schleifenvariable, Ausgangstest) zu reduzieren.

Die Umkehrtransformation ist das *Aufrollen* von Schleifen (*loop rerolling*) auf Anweisungs– bzw. Ausdrucksebene.

3.2.12 Schleifenvertauschung (loop interchange)

In einer vollkommenen Schleifenverschachtelung (Anweisungen befinden sich nur in der innersten Schleife) dürfen zwei Schleifenköpfe vertauscht werden, wenn die daraus resultierende Änderung in der Ausführungsreihenfolge der Anweisungen im gemeinsamen Rumpf nicht der Richtung der ursprünglichen Datenabhängigkeiten entgegensteht.

Beispiel 3.5 In

```
for (i=1; i<=n; i++)
   for (j=1; j<=m; j++)
S:    A[i+1][j] = A[i][j+1] * B[i][j];
```

dürfen aufgrund von von beiden Schleifen getragenen Datenabhängigkeiten mit „ungünstigen“ Richtungsvektoren die `i`– und die `j`–Schleife nicht vertauscht werden. Ansonsten würde beispielsweise $S(2,1)$ vor $S(1,2)$ ausgeführt, obwohl von $S(1,2)$ nach $S(2,1)$ eine Datenabhängigkeit besteht. ◇

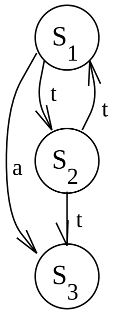
Generell ist Schleifenvertauschung unzulässig, falls es zwischen zwei (nicht notwendig verschiedenen) Anweisungen des Rumpfes eine Datenabhängigkeit gibt, in deren Richtungsvektor, wie in obigem Beispiel, sowohl '$<$' als auch '$>$' vorkommen [Wol89]. Eine allgemeine Diskussion der Schleifenvertauschung findet man in [Wol89] und [AK84].

3.2.13 Schleifenaufgliederung (loop distribution)

Schleifenaufgliederung (*loop distribution*) entspricht dem Anwenden des Distributivgesetzes in der Arithmetik:

Beispiel 3.6

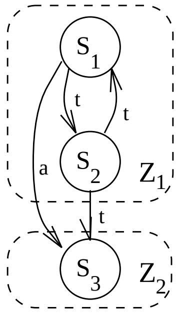
```
   for (i=1; i<=n; i++) {
S1:  A[i+1] = B[i-1] + C[i];
S2:  B[i] = A[i] * K;
S3:  C[i] = B[i] - 1;
   }
```



Wir sehen $S_1\delta^t S_2$ (Distanz 1), $S_2\delta^t S_1$ (Distanz 1), $S_1\delta^a S_3$ (Distanz 0, d.h. schleifenunabhängig) und $S_2\delta^t S_3$ (Distanz 0). Die Anweisungen S_1 und S_2 bilden im Datenabhängigkeitsgraphen eine starke Zusammenhangskomponente Z_1 (hier: Zyklus), während S_3 eine einzelne Zusammenhangskomponente Z_2 bildet.

Dann dürfen wir die Schleife wie folgt aufgliedern:

```
   for (i=1; i<=n; i++) {   /* aus SZK Z1 */
S1:  A[i+1] = B[i-1] + C[i];
S2:  B[i] = A[i] * K;
   }
   for (i=1; i<=n; i++)     /* aus SZK Z2 */
S3:  C[i] = B[i] - 1;
```



Das Ausgliedern von S_3 aus der Schleife war gestattet, da die Datenabhängigkeiten von Z_1 nach Z_2 in derselben Richtung verliefen. Hingegen darf Z_1 nicht weiter aufgegliedert werden, weil sonst Datenabhängigkeiten zwischen S_1 und S_2 mißachtet würden. ◇

Allgemein darf für eine Schleifenverschachtelung der Tiefe k die Schleife h, $1 \leq h \leq k$, über die starken Zusammenhangskomponenten (auch PI–Blöcke genannt) $Z_1, Z_2, ...$ des Datenabhängigkeitsgraphen G_h der Tiefe h aufgegliedert werden, wobei die Reihenfolge der Z_j durch die Datenabhängigkeiten zwischen den Z_j (azyklische Kondensation von G_h) zumindest partiell vorgegeben ist [AK87].

Die Berechnung der starken Zusammenhangskomponenten kann sehr effizient mit Tarjan's Algorithmus [Tar72] erfolgen; die dazu benötigte Zeit ist proportional zur Größe von G_h. Anschließend werden die Z_j gemäß der azyklischen Kondensation von G_h topologisch sortiert; dies geht mittels Tiefensuche (depth first search) ebenfalls in Linearzeit [Tar72].

Die Umkehrtransformation zur Schleifenaufgliederung heißt *Schleifenverschmelzung* (loop fusion).

3.2.14 Vektorisierung

Betrachten wir nochmals Beispiel 3.6 nach der Aufgliederung der Schleife. Anweisung S_3 bildet eine einelementige starke Zusammenhangskomponente Z_2. Wir können mittels der z.B. in `FORTRAN 8x` und anderen vektoriellen Programmiersprachen üblichen *Arraysyntax* die Schleife um S_3 wie folgt schreiben:

```
C[1:n] = B[1:n] - 1;
```

Damit drücken wir aus, daß die Vektoren `C` und `B` elementweise miteinander verrechnet werden, wobei die Reihenfolge der einzelnen Elementaroperationen keine Rolle spielt, weil es keine schleifengetragenen Datenabhängigkeiten gibt.

Allgemein darf man in einer Schleifenverschachtelung der Tiefe k alle vollständig verschachtelten Schleifen der Tiefe h, $h+1$,...,k vektorisieren, falls (1) der Rumpf dieser Schleifen aus nur einer Anweisung S besteht, und (2) keine dieser Schleifen eine Datenabhängigkeit $S\delta_q S$ mit $h \leq q \leq k$ und Richtung '>' im q–ten Eintrag des Richtungsvektors zu $S\delta S$ trägt.

Die Vektorisierung kann durch eine Reihe von Transformationen erleichtert werden. Die Schleifenaufgliederung ist sehr nützlich zur Erfüllung der ersten Voraussetzung (einelementige starke Zusammenhangskomponenten). Die Schleifenvertauschung bietet zum einen der Schleifenaufgliederung zusätzliche Betätigungsmöglichkeiten; zum anderen ist sie geeignet, vektorisierbare Schleifen ins Innere der Schleifenverschachtelung zu transferieren.

3.2.15 Vektorisierung bedingter Anweisungen (`IF` conversion)

Für bedingte vektorisierbare Anweisungen, wie etwa im folgenden Beispiel

```
for (i=1; i<=n; i++)
  if (C[i] != 0)
     A[i] = B[i] / C[i];
```

kann durch Einführung eines boole'schen Vektors (*Maskenvektor*) `MASK[1:n]` in eine *maskierte Vektoroperation* der Form

```
MASK[1:n] = (C[1:n] != 0);
A[1:n] = (MASK[1:n])?  B[i] / C[i]  :  A[i];
```

transformiert werden [AK87][ZC90].

3.2.16 Skalar–Expansion (scalar expansion)

Beispiel 3.7 Die Schleife

```
    for (i=1; i<=n; i++) {
S1:   temp = a[i] + b[i];
S2:   c[i] = temp * (K - temp);
    }
```

kann nicht vektorisiert werden, da wegen $S_1\delta^t_\infty S_2$ und $S_2\delta^a_1 S_1$ aufgrund der skalaren Variable `temp` die Schleife nicht aufgegliedert werden darf.

Es bietet sich an, `temp` zu einem Vektor `temp[1:n]` zu expandieren. So kann auf Kosten zusätzlichen Speicherplatzes die Schleife

```
    for (i=1; i<=n; i++)
S1':   temp[i] = a[i] + b[i];
    for (i=1; i<=n; i++)
S2':   c[i]    = temp[i] * (K - temp[i]);
```

aufgegliedert und vektorisiert werden:

```
S1'':   temp[1:n] = a[1:n] + b[1:n];
S2':'   c[1:n]    = temp[1:n] * (K - temp[1:n]);
```

◇

Ein solcher Dimensionswechsel vom Skalar zum Vektor kann auch in höheren Dimensionen vorkommen, z.B. vom Vektor zur Matrix usw. Allgemein entspricht dies einem Wechsel des Unter–Iterationsraumes (*subspace change*). Eine allgemeine Diskussion hierüber findet man z.B. in [KD94].

3.2.17 Linearisierung von Schleifen (loop collapsing)

Das explizite Linearisieren mehrdimensionaler Schleifen ist eine problematische Transformation, weil sie keine Umkehrtransformation besitzt. Sie geht außerdem meist von einer bestimmten Speicherungskonvention für mehrdimensionale Felder aus und ist in manchen FORTRAN–Programmen angewendet worden, um z.B. mehr Entscheidungsfreiheit beim Blocken von Schleifen zu haben:

Beispiel 3.8

```
for (i=1; i<=n; i++)
  for (j=1; j<=m; j++)
    A[i][j] = ... ;
```

kann bei FORTRAN–Speicherungskonvention linearisiert werden zu

```
nm = n * m;
for (ij=1; ij<=nm; ij++)
  A[ij] = ... ;
```

oder bei Dimensionserhalt auf Kosten komplizierterer Adressierung zu

```
nm = n * m;
for (ij=1; ij<=nm; ij++)
  A[i/m+1][j%m+1] = ... ;
```

◇

3.3 Parallelisierung für SMS

Während die Vektorisierung die Iterationen der inneren Schleifen einer Schleifenverschachtelung parallel (d.h. als Vektoroperationen) auszuführen versucht und die äußeren Schleifen sequentiell beläßt, bemüht sich die Parallelisierung, die Iterationen beliebiger Schleifen (vorwiegend der äußeren Schleifen, sofern möglich), unabhängig voneinander auf verschiedenen Prozessoren ausführen zu lassen, während die übrigen Schleifen weiterhin sequentiell abgearbeitet werden. Damit ist die Parallelisierung im Prinzip mächtiger als die Vektorisierung.

Für die Parallelisierung auf SMS werden die Schleifen, die parallel ausgeführt werden können, als *parallele Schleife* gekennzeichnet [ACK87]. Je nachdem, ob diese Schleife Datenabhängigkeiten trägt oder nicht, nennen wir sie DOACROSS oder DOALL (forall) [ACK87]. Im Fall von DOACROSS [Cyt86] müssen Synchronisierungsanweisungen eingefügt werden, um die Unverletzlichkeit der Datenabhängigkeiten zu garantieren. Dabei wird gewöhnlich nach jeder Anweisung $S(\vec{i})$, die eine Quelle einer Datenabhängigkeit ist, eine Nachricht geschickt (SIGNAL($S(\vec{i})$)), daß $S(\vec{i})$ beendet ist, während vor jeder Anweisung $S'(\vec{i'})$, die Ziel einer Datenabhängigkeit $S(\vec{i})\delta S'(\vec{i'})$ ist, auf solch eine Nachricht von $S(\vec{i})$ gewartet wird (WAIT($S(\vec{i})$)). Bei Datenabhängigkeiten zwischen einer Anweisung in einem sequentiell auszuführenden Bereich und einer Anweisung in einer parallelen Schleife muß zwischen diesen beiden Bereichen eine globale Synchronisation (BARRIER–Synchronisationspunkt) erfolgen. Bei Datenabhängigkeiten zwischen Anweisungen in zwei verschiedenen parallelen Schleifen kann solch ein Synchronisationspunkt unter Umständen entfallen. [ACK87] gibt einen Algorithmus an, der parallelen Code mit minimaler Anzahl von Barrieren generiert.

Die DOACROSS–Schleife degeneriert zu einer sequentiellen Schleife, falls das Minimum m der Absolutbeträge aller Distanzen von Abhängigkeiten, die diese Schleife trägt, gleich eins ist. Ansonsten können jeweils m Iterationen der DOACROSS–Schleife überlappend ausgeführt werden; folglich ist der durch DOACROSS erreichbare Speedup durch m nach oben beschränkt. Bei zur Übersetzungszeit bekanntem $m > 1$ kann mittels der Transformation *cycle shrinking* [Pol88] jede DOACROSS–Schleife durch eine äußere sequentielle und eine innere DOALL–Schleife ersetzt werden.

Loop scheduling Prinzipiell kann für jede Iteration einer parallelen Schleife ein eigener paralleler Prozeß (einem virtuellen Knotenprozessorprogramm entsprechend) erzeugt werden. Um den Verwaltungsaufwand zu verringern, werden gewöhnlich mehrere Iterationen zu sogenannten *tasks* zusammengestellt, deren Ausführungsdauer — je nach Art der Schleifenstruktur — unterschiedlich sein kann. Daher werden, zumal die Anzahl der Prozessoren gegenüber der Anzahl der tasks gewöhnlich relativ klein ist, die tasks nach einem bestimmten Verfahren auf die Prozessoren verteilt (*scheduling*) — entweder zur Übersetzungszeit (statisches scheduling) oder zur Laufzeit (dynamisches scheduling), wo die ausführbaren tasks in einer Warteschlange gehalten werden, wo sich jeder freiwerdende Prozessor seine nächste task abholen kann [Pol88].

3.4 Halbautomatische Parallelisierung für DMS

Die Parallelisierungsstrategie für DMS unterscheidet sich grundlegend von der in Abschnitt 3.3 vorgestellten SMS–Parallelisierung. Kernbestandteil der üblichen DMS–Parallelisierungsstrategien ist die *owner–computes–Regel*, die dafür sorgt, daß jede Variable und insbesondere jedes Feldelement von dem Prozessor berechnet wird, in dessen lokalem Speicher es sich befindet.

Voraussetzung der DMS–Parallelisierung ist eine Vorgabe von Datenaufteilungen für die im Programm vorkommenden Felder auf die lokalen Speicher der Prozessoren. Wird diese Vorgabe vom Anwender — sei es in Form von Kommandos in einem interaktiven Parallelisierungssystem oder als Sprachkonstrukt oder Übersetzerdirektive in einer datenparallelen Programmiersprache — geliefert, spricht man von *halbautomatischer* Parallelisierung; wird die Datenaufteilung auch vom Parallelisierer bestimmt, so nennt man dies *vollautomatische* Parallelisierung. Man beachte, daß dies nichts über die Effizienz des generierten Codes aussagt!

Fast immer wird DMS–Code als SPMD–Programm (Single Program, Multiple Data) erzeugt, sodaß auf jedem Knotenprozessor eine Kopie desselben Knotenprogramms vorliegt. Dabei ist nur durch die individuellen Prozessornummern festgelegt, welcher Prozessor welche Anweisungen ausführt.

3.4.1 Splitting, Adaptierung und Standardoptimierungen

Die DMS–Parallelisierung, wie in [CK88, ZBG88, ZC90] beschrieben, erfolgt dann in drei Schritten:

1. *Splitting*: Wenn die Zielmaschine einen Host–Prozessor besitzt, der die I/O[7] von bzw. nach außen übernimmt, so wird das sequentielle Quellprogramm aufgespalten in ein *Host–Programm*, das alle I/O–Anweisungen sowie Anweisungen zum Starten der Knotenprogramme enthält, und in ein *Restprogramm*, aus dem später die *Knotenprogramme* erzeugt werden, und das jetzt nur noch Rechenbefehle enthält. Das Host–Programm wird ferner mit Anweisungen versehen, die die von außen gelesenen Operanden zum Restprogramm versenden, sowie mit Anweisungen, die die auszugebenden Daten vom Restprogramm empfangen. Das Restprogramm erhält den Host–Sendeanweisungen entsprechende Empfangsanweisungen sowie den Host–Empfangsanweisungen entsprechende Sendeanweisungen anstelle der früheren I/O–Anweisungen.

2. *Adaptierung*: Das Knotenprogramm wird aus dem Restprogramm wie folgt erzeugt:

   Jede Anweisung wird mit einer von der Prozessornummer und der Datenaufteilung der geschriebenen Variable abhängenden *Maske* (Bedingung) versehen, die gewährleistet, daß jeder Knotenprozessor diese Anweisung genau dann ausführt, wenn er laut Datenaufteilung Eigentümer der zu beschreibenden Variable ist.

[7] I/O = griechische Göttin der Ein– und Ausgabe `;-)`, gemeint sind `read` und `write`.

Anschließend werden ggf. erforderliche Kommunikationsanweisungen generiert. Für jede von einem Prozessor p geschriebene (also lokale) Variable v, die eventuell von einem anderen Prozessor $p' \neq p$ nachfolgend als (nichtlokaler) Operand benötigt wird, muß eine Anweisung `SEND`(v, p') unmittelbar nach der Berechnung von v eingefügt werden. Für jede von Prozessor p benötigte, aber eventuell nichtlokale Variable v muß eine Anweisung `RECEIVE`(v) direkt vor der Benutzung von v eingefügt werden.

Beispiel 3.9 Sei folgendes Restprogramm gegeben:

```
for (i=1; i<=n; i++)
  A[i] = B[i] + B[i+1];
```

Nehmen wir an, für das Feld `A` gebe die Funktion `owned(A[i])` in Abhängigkeit der Prozessornummer p und der Schleifenvariablen `i` die Datenaufteilung an, d.h. `owned(A[i])` evaluiert zu TRUE auf dem Prozessor, dem `A[i]` gehört. Nehmen wir ferner an, die Aufteilung von `B` sei mit der von `A` identisch. Die Maskierung produziert

```
for (i=1; i<=n; i++)
  if (owned(A[i]))   A[i] = B[i] + B[i+1];
```

und mit der Einfügung der Kommunikationsanweisungen erhalten wir

```
for (i=1; i<=n; i++) {
  EXCH(B[i+1]);
  if (owned(A[i]))  A[i] = B[i] + B[i+1];
}
```

wobei die `EXCH(B[i+1])`–Anweisung schematisch wie folgt aussieht:

```
EXCH(B[i+1]):
if owned(B[i+1]) then  SEND(B[i+1]) an alle Prozessoren p',
                        die B[i+1] lesen,
                        deren Maske TRUE ergibt, und
                        für die owned(B[i+1]) zu FALSE evaluiert;
else                   if  Maske für p ergibt TRUE
                       then  RECEIVE(B[i+1]);
```

◇

Gewöhnlich sind die hierbei verwendeten `SEND`–Anweisungen des Zielrechners nichtblockierend. Sofern blockierende `RECEIVE`–Anweisungen verwendet werden, ist keine weitere explizite Synchronisierung (wie bei der SMS–Parallelisierung) mehr erforderlich.

3. *Optimierung*: Ein so erzeugtes Programm ist zwar bereits auf der Zielmaschine lauffähig und zu dem sequentiellen Programm semantisch äquivalent, aber in der Regel alles andere als effizient. Wir unterscheiden drei grundlegende Optimierungstechniken [Ger89], die anwendbar sind, sofern die Datenaufteilung zur Übersetzungszeit explizit bekannt ist:

   - *Verrechnung der Masken mit den Schleifengrenzen:* Anstatt in jeder Iteration der Schleife in Beispiel 3.9 die Maskenbedingung auswerten zu müssen, kann bei einer Standard–Blockverteilung für `A` die Maskenbedingung in die Schleifengrenzen integriert werden, sodaß jeder Prozessor p von vornherein nur die Iterationen `i` ausführt, für die `owned(A[i])` erfüllt ist:

     Beispiel 3.10 (Fortsetzung von Beispiel 3.9): Seien `A` und `B` in P Streifen der Länge n/P aufgeteilt (Annahme: $P|n$). Dann ergibt sich für Prozessor p, $1 \leq p \leq P$:

```
LEFT = (p-1)*n/P + 1;
RIGHT = p*n/P;
for (i=LEFT; i<=RIGHT; i++) {
  EXCH(B[i+1]);
  A[i] = B[i] + B[i+1];
}
```

◇

- *Overlap–Konzept:* In manchen Anwendungen, z.B. Gitterrelaxationen, bilden die von einem Knotenprozessor p benötigten nichtlokalen Feldelemente bei üblichen Block–Aufteilungen einen einfach strukturierten Randbereich um die lokalen Feldelemente von p herum. Mit der Kenntnis dieses Überlappungsbereichs wissen wir genau, welche Elemente kommuniziert werden und wie man sie zu Blöcken zusammenfassen kann. Ferner können wir den Speicherplatz für den Überlappbereich mit den lokalen Feldelementen gleich mitreservieren.

  Beispiel 3.11 In Beispiel 3.10 entspricht der Überlappbereich von Prozessor p dem Feldelement `B[p*n/P+1]`. ◇

- *Vektorisierung der Kommunikationsanweisungen:* Viele einzelne `SEND`–Anweisungen sind ineffektiv, weil bei jeder einzelnen die Startup–Zeit der Kommunikation anfällt, die bei derzeitigen DMS sehr hoch ist. Faßt man jedoch alle Feldelemente, die vom selben Quellprozessor zum selben Zielprozessor geschickt werden sollen, zu Blöcken zusammen, fällt die Startupzeit pro Block nur einmal an. Dabei wendet man Schleifenaufgliederung und Vektorisierung auf die aus den `EXCH`–Makros hervorgehenden Kommunikationsanweisungen an.

Eine ausführliche Betrachtung der hier beschriebenen halbautomatischen DMS–Parallelisierung und geeigneter Optimierungen findet man z.B. in [Ger89] und [Die93].

Vergleichbare Ansätze zur halbautomatischen Parallelisierung enthalten neben den interaktiven Systemen SUPERB [ZBG88, Ger89] und FORGE/MIMDIZER [SWW91] auch die Übersetzer für die (daten)parallelen Programmiersprachen BLAZE [MR87], BOOSTER/V-CAL [Paa92, PSvG91], C^* [RS87], DINO [RSW91], FORTRAN-D [HKT91a] und sein Vorläufer [CK88], ID NOUVEAU [RP89], KALI [KMR90], PANDORE [APT90] und VIENNA FORTRAN [CMZ92].

Das Knotenprogramm benutzt auch nach Adaptierung und ggf. Optimierung noch immer den globalen Adreßraum des sequentiellen Quellprogramms. Manche Parallelrechner bieten einen virtuellen globalen Adreßraum an. Wo das nicht der Fall ist, müssen die globalen Adressen (Feldindizierungen) in lokale Adressen umgesetzt werden, und zwar für Zuweisungen ebenso wie für Kommunikationsanweisungen. Die Adreßumsetzung $g \mapsto (p, l)$ bildet eine globale Indizierung g bijektiv auf ein Paar aus Prozessornummer p und lokaler Indizierung l ab. Zu ihrer Berechnung muß die Datenaufteilung des indizierten Feldes explizit vorliegen.

3.4.2 Lokale Iterations–, Index– und Kommunikationsmengen

Die *lokale Iterationsmenge* $Exec(p)$ für einen Prozessor p und eine Zuweisung

$$S: \quad \texttt{A[}f(\vec{i})\texttt{] = ... B[}g(\vec{i})\texttt{] ...}$$

innerhalb einer Schleifenverschachtelung ist die Menge der Werte der Schleifenindizes $\vec{i}$, für die $owned($`A[`$f(\vec{i})$`]`$)$ auf p zu TRUE evaluiert. *Exec* ist die Umkehrfunktion der Indizierung f des Vorkommens von `A` auf der linken Seite der Zuweisung, angewandt auf die Datenaufteilung von `A`.

Man kann also die umständliche Maskenbedingung `if owned(A[...])` leicht optimieren, falls es gelingt, eine explizite Darstellung (closed form) für $Exec(p)$ zu finden. Die oben beschriebene Einarbeitung der Masken in die Schleifengrenzen ist ein Spezialfall. [CGL+93] und [GKM+93] geben weitere explizite Darstellungen für Spezialfälle (eindimensionale Felder und block-zyklische Datenaufteilungen).

Die *Lokalreferenziterationsmenge* $Ref(p)$ für einen Prozessor p und eine Schleifenverschachtelung um eine Anweisung S ist die Menge der Werte der Schleifenindizes $\vec{i}$, sodaß für *alle* Feldvorkommen `B[`$g(\vec{i})$`]` in S die Maske *owned*(`B[`$g(\vec{i})$`]`) auf p zu TRUE evaluiert. *Ref* ist die Umkehrfunktion der Indizierung g des Vorkommens von `B`, angewandt auf die Datenaufteilung von `A`.

Analog ist man auch an expliziten Darstellungen für die Mengen der von jedem Prozessor zu versendenden bzw. zu empfangenden Elemente (*communication sets*) interessiert. Einen Spezialfall haben wir bereits mit der oben beschriebenen Vektorisierung der Kommunikation kennengelernt.

Zu diesem Zweck bestimmt man aus den gegebenen Feldaufteilungen die in der Prozessornummer p parametrisierten *lokalen Indexmengen* $L_A = f(Exec(p))$ für das Feld A auf der linken Seite, und *referenzierte Indexmengen* $M_B = g(Ref(p))$, d.h. explizite Darstellungen der Mengen von Indexwerten, die die Indizierung des jeweiligen Feldvorkommens aufgrund der sie indizierenden umgebenden Schleifen annehmen kann.

Ferner definieren wir entsprechend lokale Indexmengen $L_B = g(Ref(p) \cap Exec(p))$ für alle Feldvorkommen B auf der rechten Seite der Zuweisung. Für diese Feldvorkommen B sind die Indexmengen $L_B(p)$ auf p lokal und die Indexmengen $M_B(p) \setminus L_B(p)$ auf p nichtlokal.

Für jedes Feldvorkommen B auf der rechten Seite und jeden Prozessor $p' \neq p$ kann die *Kommunikationsmenge* $Comm_B(p,p') = (M_B(p) \setminus L_B(p)) \cap L_B(p')$ der von p nach p' zu transferierenden Feldelemente von B berechnet werden. Um weiterrechnen zu können, muß $Comm_B(p,p')$ explizit vorliegen.

Entsprechend sei $Comm(p,p') = \cup_B Comm_B(p,p')$ die Kommunikationsmenge für alle vorkommenden Felder B. Mit einer expliziten Darstellung von $Comm(p,p')$ kann man die oben beschriebene `EXCH`–Prozedur effizienter gestalten.

Sind lokale Iterationsmengen und Kommunikationsmengen für eine Schleifenschachtelung um eine Zuweisung S berechnet, formuliert sich das Knotenprogramm der Schleifenschachtelung um S für jeden Prozessor p wie folgt:

1. für alle Prozessoren $p' \neq p$:
   if $Comm(p,p') \neq \emptyset$ **then** versende Nachrichten gemäß der Listen $Comm(p,p')$
2. für alle Iterationen $\vec{i} \in Exec(p) \cap Ref(p)$:
   führe $S(\vec{i})$ aus
3. für alle Prozessoren $p'' \neq p$:
   if $Comm(p'',p) \neq \emptyset$ **then** empfange Nachrichten gemäß der Listen $Comm(p'',p)$
4. für alle Iterationen $\vec{i} \in Exec(p) - Ref(p)$:
   führe Iterationen in $Exec(p) - Ref(p)$ aus.

Im Vergleich zur `EXCH`–Kommunikationsroutine ermöglicht diese Variante eine explizite Überlappung von Kommunikation und lokalen Rechnungen.

3.4.3 Laufzeitparallelisierung für indirekte Feldzugriffe

Das bisher Gesagte erstreckt sich auf den Fall, daß die zur Adaptierung benötigte Information zur Übersetzungszeit verfügbar ist.

In Gegenwart indirekter Feldzugriffe sind die Adaptierung sowie weitere Optimierungen nicht möglich, da die *owned*–Masken erst zur Laufzeit ausgewertet werden können. Damit ist auch erst zur Laufzeit bekannt, welcher Prozessor an wen welche Daten zu schicken hat.

Beispiel 3.12 [Ger93] Matrix–Vektor–Multiplikation für eine dünnbesetzte Matrix:

```
for (j=1; j<=nnp; j++)
   for (i=1; i<=eintrz; i++)
      axvec[j] = axvec[j] + cmatrx[j][i] * vec[ind[j][i]];
```

Das Indexfeld `ind` liefert den globalen Index eines Matrixelementes. Diese Matrix ist komprimiert im Vektor `vec` gespeichert. `cmatrx` speichert den zu multiplizierenden Vektor so, daß jeweils mit dem richtigen Matrixelement multipliziert wird.

Auch hier wird parallelisiert, indem die Felder aufgeteilt werden. Im allgemeinen ist aber eine gleichmäßige Datenverteilung hier weniger sinnvoll [Ger93]. Nehmen wir an, daß `axvec` und `vec` sowie die erste Dimension von `cmatrx` und `ind` gleichmäßig aufgeteilt werden. Dadurch kann die `j`–Schleife parallelisiert werden. ◇

Ob ein Zugriff nichtlokal ist, kann hier erst zur Laufzeit erkannt werden. Statt der `EXCH`–Kommunikationsanweisung generiert man zur Übersetzungszeit zwei Routinen: den *inspector* und den *executor* [KMR90].

Der *inspector* wird von jedem Prozessor p ausgeführt. Er bestimmt (zur Laufzeit) den lokalen Iterationsbereich $\mathit{Exec}(p)$, sammelt die von ihm für $\mathit{Ref}(p)$ benötigten indirekten Feldzugriffe (`vec[ind[i][j]]`) und baut daraus in einer globalen Kommunikationsphase seine lokalen Sende– und Empfangslisten $\mathit{Comm}(p,*)$ und $\mathit{Comm}(*,p)$ auf. Ferner werden die globalen Indizes der indirekten Feldzugriffe in lokale Indizes umgesetzt.

Der *executor* jedes Prozessors führt dann, soweit erforderlich, die für den jeweiligen Prozessor notwendige Kommunikation auf der Basis der Sende– und Empfangslisten durch. Anschließend führt er, sofern vorhanden, die Iterationen aus dem lokalen Iterationsbereich des Prozessors aus.

Der mit dem *inspector* verbundene Verwaltungsaufwand ist beträchtlich. Für Spezialfälle gibt es einige Optimierungsmöglichkeiten, wie z.B. das Herausziehen des *inspectors* aus umgebenden Schleifen, um die vom *inspector* berechneten Informationen mehrfach verwenden zu können [ESH93, LZ93].

Mehr zu *inspector* und *executor* entnehme man z.B. [KMR90] oder [SCMB90].

3.5 Datenaufteilungen

Wir haben im vorangehenden Abschnitt festgestellt, daß eine akzeptable Optimierung (Masken, Kommunikation) des adaptierten DMS–Programms nur möglich ist, falls die Datenaufteilungen der beteiligten Felder zur Übersetzungszeit explizit gegeben sind.

In diesem Abschnitt wollen wir verschiedene Datenaufteilungsstrategien vorstellen, die für die Verteilung von Feldern in Betracht kommen. Eine Datenaufteilung besteht aus der eigentlichen *Partitionierung* des Feldes in disjunkte, nicht notwendig zusammenhängende Teilfelder (Partitionen), die virtuellen Prozessoren entsprechen, und der *Prozessorabbildung*, einer Surjektion, die dann jeder Partition einen physikalischen Prozessor zuordnet. Die Prozessorabbildung sollte eigentlich von der Struktur des Verbindungsnetzwerkes abhängig sein; in der Realität hat die direkte Prozessor–Nachbarschaft eingebetteter 'benachbarter' Partitionen bei gängigen DMS–Rechnern jedoch nur einen untergeordneten Einfluß auf die Gesamtkommunikationszeit[8].

Zunächst unterscheiden wir die *Standardaufteilungen*, die nur rechteckige Partitionen gleicher Größe zulassen, von den *Nichtstandardaufteilungen*, die auch andere geometrische Strukturen (z.B. Sechsecke [HA90] oder unregelmäßige Vierecke [Hin92]) oder Partitionen unterschiedlicher Größen ermöglichen. Während die Nichtstandardaufteilungen entweder für sehr spezielle Anwendungen (Sechsecke für Gitterrelaxationen wegen geringerem Umfang–Flächen–Verhältnis) oder für bessere Lastverteilung bei im aufzuteilenden Feld lokal stark abweichendem Rechenaufwand geeignet sind, bieten die Standardaufteilungen wesentlich bessere und einfachere Möglichkeiten zur statischen Optimierung von Masken und Kommunikation.

Die Standardaufteilungen wiederum zerfallen in *zusammenhängende* und *zyklische* Aufteilungen sowie deren Kombinationen für alle Dimensionen des Feldes. Ferner sind über die Prozessorabbildung Replikationen von Partitionen über mehrere physikalische Prozessoren möglich [GB90, Gup92].

[8] Vgl. hierzu Abschnitt 3.7, Formel 3.5.

Viele heutige DMS bieten eine zweidimensionale Netzwerktopologie. Darauf lassen sich problemlos Felder einbetten, die in einer oder zwei Dimensionen aufgeteilt werden. Felder, die in mehr als zwei Dimensionen massiven Parallelismus durch Aufteilung bieten, sind in realen Anwendungen selten.

Intuitiv ist es nur dann sinnvoll, eine Dimension d eines Feldes A aufzuteilen, falls alle Schleifen, die den d–ten Indexausdruck in einem Vorkommen von A indizieren, keine Datenabhängigkeit tragen, da sonst zum einen solche Schleifen sequentialisiert würden und zum anderen an den Partitionsgrenzen auch noch Kommunikationsverzögerungen auftreten würden — dies ergäbe einen Speed-down, sofern nicht eine weitere Aufteilung in einer anderen Dimension von A diese Nachteile wieder kompensiert). Daher sollten schleifengetragene Datenabhängigkeiten *internalisiert* werden. Allerdings steht ein Feldvorkommen nicht alleine im Programm. Aufgrund anderer Vorkommen mit anderen Indizierungen können sich Präferenzen ergeben, die eine lokal ungünstige Aufteilung sogar erfordern, um die Gesamtlaufzeit zu minimieren. Daher müssen wir alle Felddimensionen (bei schleifengetragenen Datenabhängigkeiten von hinreichend großem Extent, z.B. 16) als aufteilbar in Betracht ziehen.

Formal [GB90, Gup92, Who91] betrachten wir zunächst ein abstraktes DMS mit einem D–dimensionalen Gitter ($D = \max_A \dim(A)$) von $N = N_1 \cdot N_2 \cdots N_D$ virtuellen Prozessoren, das dann durch die Prozessorabbildung auf ein physikalisches Prozessorgitter der Abmessungen $P = P_1 \cdot P_2 \cdots P_K$, mit $K \leq D$, eingebettet wird. Jeder virtuelle Prozessor wird durch ein D–tupel $(v_1, \ldots, v_D)$ spezifiziert, wobei entweder $0 \leq n_i \leq N_i - 1$ oder $n_i = X$. Ein X in Dimension i steht für alle n_i, $0 \leq n_i \leq N_i - 1$ (Replikation).

Skalare Variablen und kleine Felder werden grundsätzlich repliziert. Für jedes Feld A wird die k–te Dimension A_k, $1 \leq k \leq \dim(A)$, injektiv auf eine Dimension $map(A_k) \in \{1, ..., D\}$ abgebildet. Ist $N_{map(A_k)} = 1$, so ist die Dimension A_k *sequentialisiert*, d.h. alle Feldelemente, die sich nur in dieser Dimension unterscheiden, werden auf denselben (virtuellen) Prozessor abgebildet. Andernfalls heißt die Dimension A_k *aufgeteilt*.

Die *Aufteilungsfunktion* $dist_k(A_k, t)$ liefert die k–te Komponente des Prozessortupels, das Eigentümer des Feldbereichs $A(\cdot, t, \cdot)$ ist (d.h. der k–te Indexausdruck ist t, die übrigen beliebig). Sie hat in [GB90] die allgemeine Form

$$dist_k(A_k, t) = \begin{cases} \lfloor \frac{t - offset}{block} \rfloor [\bmod\ N_{map(A_k)}] & \text{falls } A_k \text{ aufgeteilt} \\ X & \text{falls } A_k \text{ repliziert} \end{cases} \tag{3.4}$$

wobei der Term $\bmod\ N_{map(A_k)}$ optional ist. Sie gibt an, (1) ob die Dimension A_k repliziert oder aufgeteilt ist, (2) ob die Aufteilung blockweise oder zyklisch (mod –Term) ist, (3) die Gitterdimension, auf die die k–te Felddimension abgebildet wird, (4) die Blockgröße der Aufteilung, d.h. die Anzahl Feldelemente, die als zusammenhängender Block im lokalen Speicher eines (virtuellen) Prozessors stehen, und (5) eine Zahl *offset*, die zur Bestimmung der Prozessorkomponente auf den Indexwert addiert wird.

Die im vorangehenden Abschnitt eingeführte *owned*–Funktion ergibt sich dann als Konkatenation aus der Prozessorabbildung, angewandt auf die Aufteilungsfunktion.

Wir halten den *offset*–Wert für unwesentlich und lassen ihn im folgenden weg.

Für einen Vektor wird die Aufteilungsfunktion sehr einfach: Neben der Totalreplikation gibt es nur noch die Möglichkeit blockweiser oder zyklischer Aufteilung.

Eine Matrix kann in jeder einzelnen Dimension oder total repliziert werden. Bei zusammenhängende Aufteilung nur der ersten Dimension erhalten wir eine *zeilenweise*, nur der zweiten eine *spaltenweise* Aufteilung der Matrix. Werden beide Dimensionen zusammenhängend aufgeteilt, so ergeben sich rechteckige Blöcke entsprechender Gestalt. Zyklische Verteilungen dienen dem Lastausgleich (z.B. bei der LR–Zerlegung); eine zyklische Aufteilung in mehr als einer Dimension scheint allerdings selten sinnvoll. Einige Beispiele gibt Abbildung 3.1.

Die Anzahl der Freiheitsgrade bei Datenaufteilungen wächst exponentiell mit der Felddimension. Jeder Parameter kann die Gesamtlaufzeit des parallelisierten Programms beträchtlich beeinflussen. Zugleich müssen in der Regel die Aufteilungen mehrere Felder gleichzeitig betrachtet werden, die gewöhnlich in komplexen Zusammenhängen zueinander stehen und Konflikte bilden.

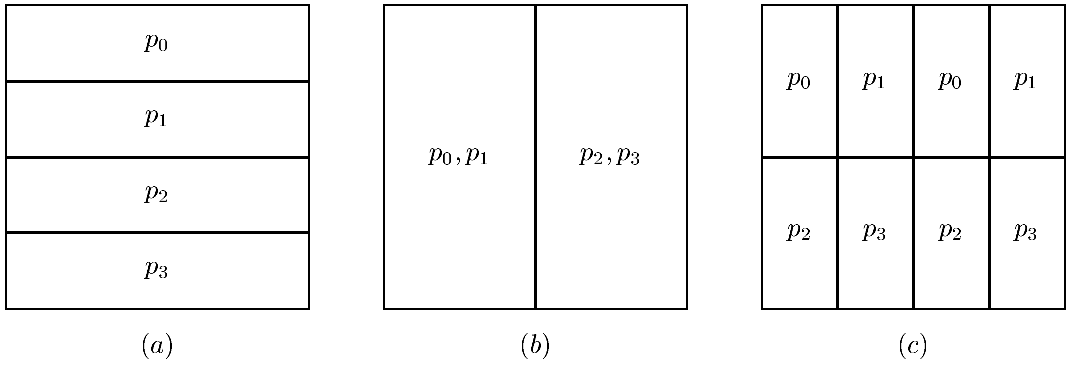


Abbildung 3.1 Drei Beispiele für Datenaufteilungen eines zweidimensionalen Feldes A mit Extents 16 x 16 (in Anlehnung an [Gup92]): (a) $N_1 = 4$, $N_2 = 1$, $dist(A_1, t) = \lfloor \frac{t-1}{4} \rfloor$, $dist(A_2, t) = 0$ (zeilenweise, zusammenhängend); (b) $N_1 = 2$, $N_2 = 2$, $dist(A_1, t) = X$, $dist(A_2, t) = \lfloor \frac{t-1}{8} \rfloor$ (spaltenweise, teilrepliziert); (c) $N_1 = 2$, $N_2 = 2$, $dist(A_1, t) = \lfloor \frac{t-1}{8} \rfloor$, $dist(A_2, t) = \lfloor \frac{t-1}{4} \rfloor \bmod 2$ (blockweise, zyklisch in der ersten Dimension).

Das ADDAP–System [Die93, Hay93, DHR94] löst dieses Problem mit Hilfe eines Branch-and-Bound–Suchverfahrens. Der Algorithmus durchmustert inkrementell einen Entscheidungsbaum, wobei in jedem Knoten eine Datenaufteilung für ein Feld festgelegt wird [Die93]. Daraus werden inkrementell die resultierenden Kommunikationskosten geschätzt [Hay93]. Jeder vollständige Pfad des Entscheidungsbaumes, dessen Länge der Anzahl der aufzuteilenden Felder entspricht, steht für eine bestimmte Kombination von Aufteilungen für all diese Felder. Pfade, deren aufgelaufene Kommunikationskosten diejenigen des bislang optimalen Pfades übersteigen, werden frühstmöglich abgebrochen. Da nur der optimale und der aktuell besuchte Pfad gespeichert werden, treten keine Platzprobleme auf. Um die Laufzeit erträglich zu gestalten, werden die wählbaren Aufteilungsarten in der Implementierung dieses Verfahrens auf nichtzyklische Aufteilungen beschränkt. Umverteilungen werden durch weitere Verzweigungen im Suchbaum berücksichtigt, doch im Hinblick auf die kombinatorische Explosion auf die oberste Schleifenebene beschränkt, was dazu führt, daß wichtige Gelegenheiten zur Umverteilung (z.B. Livermore Loop 8 — ADI–Verfahren) übersehen werden. Für einfache Beispiele mit wenigen Feldern (≤ 5) und wenigen Prozessoren (≤ 32) werden noch relativ wenige Knoten des Entscheidungsbaumes ($\leq 10^4$) besucht. Für größere reale Anwendungen erscheint dieses Verfahren jedoch nicht brauchbar.

[GB90] untersucht verschiedene Fälle von Kombinationen von Indizierungen und gibt zu diesen Fällen jeweils Empfehlungen (*constraints*) für die Wahl der Datenaufteilungen sowie eine grobe Abschätzung der Zeiteinbuße an, die durch Nichtbeachtung der Empfehlung mindestens entstehen würde. Dann bestimmt [GB90] heuristisch eine maximale Menge nicht miteinander in Konflikt stehender Empfehlungen für die einzelnen Anweisungen und Schleifenschachtelungen, sodaß die Summe aller Zeiteinbußen minimiert wird.

Um die Komplexität des Problems zu verringern, teilt man heuristisch die Bestimmung der Datenaufteilung in zwei Stufen auf: (1) das Zusammenfassen gleichartig benutzter Felddimensionen verschiedener Felder zu Bündeln (Alignment), um unnötige Kommunikation zu vermeiden, und (2) anschließend die Entscheidung für jedes Bündel, es entweder aufteilen (d.h. alle in ihm versammelten Felddimensionen), um Parallelismus zu gewinnen, oder es nicht aufzuteilen, um Datenabhängigkeiten zu internalisieren.

3.5.1 Alignment

Als *Alignment* bezeichnen wir die Ausrichtung einzelner Felddimensionen (*Feldachsen*) verschiedener Felder, die in ähnlicher Weise indiziert werden, sodaß alignte Feldelemente auf den gleichen virtuellen Prozessor abgebildet werden, also keine Interprozessorkommunikation verursachen können.

Die Alignment–Präferenzen sind zielmaschinenunabhängig und können wie folgt klassifiziert werden [LC90, KLS90, KN90]:

- Die *Eigen–Präferenz* (*identity preference*) verbindet zwei Vorkommen von Achsen desselben Feldes in verschiedenen Anweisungen. Eigenpräferenzen bilden Def–Use–Ketten von einem Vorkommen einer Achse in einem schreibenden Feldzugriff zu den Vorkommen derselben Achse in allen von dort aus erreichbaren lesenden Zugriffen desselben Feldes.

- Die *Konform–Präferenz* (*conformance preference*) verbindet zwei Vorkommen von Achsen verschiedener Felder innerhalb derselben Anweisung, die ähnliche Datenaufteilungen haben sollen.

Weitere Präferenzenarten können bei der Übernahme von Präferenzen aus der Aufteilungsphase in die Alignmentphase [KLS90] oder bei speziellen Implementierungen auftreten [PM94].

Mit jeder Präferenz notieren wir die zugehörige *Alignment–Funktion*, einer (affinen) Indextransformation, die die beiden Achsenvorkommen unifiziert.

Die Bestimmung von Alignment–Präferenzen und –funktionen zu einem gegebenen Programm kann auf mehrere Arten erfolgen: [Who91] benutzt eine an [Ble90] angelehnte imperative datenparallele Programmiersprache mit den datenparallelen Primitiven `elwise` (komponentenweise arithmetische Operationen), `reduce` (Reduktionen), `extract` (Teilfeld niedrigerer Dimension), `insert` (Ändern eines Teilfeldes) und `expand` (Aufblähen zu höherer Dimension durch Replizieren), aus der die Präferenzen direkt abgelesen werden können. [LC90] setzt eine sehr einfache funktionale Sprache voraus, aus der sich Alignmentpräferenzen und –funktionen durch einfachen Indexvergleich ergeben. Wegen der Single–Assignment–Natur der funktionalen Quellsprache sind die Def–Use–Ketten offensichtlich. [KLS90] muß im Gegensatz dazu die Bestimmung der Def–Use–Ketten selbst vornehmen. Darüberhinaus hat in Fortran D, Vienna Fortran und HPF der Programmierer die Möglichkeit, Alignment–Präferenzen explizit als Direktiven anzugeben. Teilweise benutzt man dazu ein abstraktes Array (*template*), dessen Dimensionalität gleich der maximalen vorkommenden Felddimensionalität ist, und mit dessen Achsen man die realen Feldachsen alignen lassen möchte, z.B. `ALIGN a[i] WITH T[c*i+d][j]` gibt eine Präferenz, den Vektor `a` mit der ersten Achse des Templates `T` zu alignen, wobei die Alignment–Funktion die affine Abbildung $f(i) = \mathtt{c}i + \mathtt{d}$ ist.

Diese Alignment–Präferenzen können in einem *Präferenzengraph* dargestellt werden. Dabei entsprechen die Knoten den einzelnen Feldachsenvorkommen. Für jede Def–Use–Kette von Eigenpräferenzen verbindet eine entsprechende Kette von Kanten die zugehörigen Achsenvorkommen. Für jede Konform–Präferenz zweier Achsenvorkommen verbindet eine Kante die zugehörigen Feldachsenvorkommen.

Im allgemeinen wird dieser Präferenzengraph *Konflikte* enthalten, d.h. es gibt Pfade, über die zwei verschiedene Achsen desselben Feldes miteinander verbunden sind, oder es gibt Inkonsistenzen bei zusammengesetzten Alignment–Funktionen entlang mehrerer Pfade zwischen Vorkommen gleicher Feldachsen (z.B. *stride conflict* oder *offset conflict*).

Zur Bewertung von Konflikten kann man die Kanten mit Kosten versehen, die den durch eine Mißachtung dieser Kante beim Alignment verursachten Kosten entsprechen sollten [Who91]. Mißachtet man eine Eigenpräferenz, so entspricht dies einem *Realignment* und, als Konsequenz, einer Umverteilung des Feldes an einem Punkt, an dem die Def–Use–Kette aufgetrennt wird. Die Kommunikationskosten für eine Umverteilung sind i.allg. beträchtlich und nur zu vertreten, falls dadurch anschließend mehrere andere (Konform–) Konflikte verschwinden. Mißachtet man eine Konformpräferenz, so wird, wie im vorangehenden Abschnitt beschrieben, Interprozessor–Kommunikation generiert.

Sei D die maximale vorkommende Dimension eines Feldes. Man möchte den Präferenzgraphen in D verschiedene Bündel aufteilen, sodaß (1) in jedem Bündel höchstens eine Achse jedes Feldes vorkommt, und (2) die Summe der Kosten von Kanten, die von einem Bündel zu einem anderen führen, möglichst gering wird (*minimal cost alignment*). Dieses Problem ist NP–vollständig [LC90].

Zur Lösung werden Heuristiken benutzt, z.B. das sukzessive Zusammenfassen (*greedy*) immer größerer Teilbündel in der Reihenfolge steigender Kosten [LC90, Who91]. Problematisch ist hierbei die Bestimmung einer realistischen Kostenfunktion, da auch die nachgeschaltete eigentliche Aufteilung der Bündel nicht unabhängig vom Alignment ist und ebenfalls erheblich zu den Gesamtkosten beiträgt.

3.5.2 Partitionierung

Die gefundenen Achsenbündel entsprechen gerade den einzelnen Achsen des oben genannten Templates. Anschließend werden einige (K) Achsenbündel auf die K Dimensionen des physikalischen Prozessornetzes abgebildet [Who91], also aufgeteilt. Für deren Auswahl gibt es $\binom{D}{K}$ Möglichkeiten[9]. Die übrigen $D - K$ Achsenbündel werden auf die *virtuellen* Achsen (Zeitachsen) abgebildet, also sequentialisiert. Durch geeignetes Blocken jeder der K aufgeteilten Achsen erhält man eine gleichmäßige Verteilung des Achsenextents über die Prozessoren der entsprechenden physikalischen Dimension. Allerdings können minderdimensionale Felder, die nur über einige (wenige) der physikalischen Achsen verteilt werden, dabei zu schlechter Lastbalance führen, was bei Bedarf durch Umverteilen (Realignment) ausgeglichen werden kann [Who91].

Ein weiteres Problem der beschriebenen klassischen Datenaufteilungs– und Alignment–Theorie ist das achsenweise Vorgehen. Beispielsweise wäre das Alignen eines Vektors `c[i]` (`i` gebunden) mit dem Diagonalvektor `a[i][i]` einer Matrix mit den beschriebenen Methoden nicht möglich.

3.5.3 Statische Umverteilung

Definition 3.5(Phase) [CKKM94] Eine *Phase* ist eine Schachtelung von Schleifen, sodaß alle ein Feldvorkommen im Rumpf dieser Schachtelung indizierenden Schleifen in der Phase enthalten sind. Eine Phase ist dahingehend minimal, daß sie keine anderen, nichtindizierenden Schleifen enthält. □

Phasen repräsentieren somit Codefragmente, die Operationen auf gesamten Objekten (Vektoren, Matrizen, höherdimensionale Tensoren) ausführen.

[CKKM94] liefert ein heuristisches Verfahren zur Datenaufteilung einschließlich statischer Umverteilung, welche zwischen den einzelnen *Phasen* des Programms stattfinden kann, unter der Voraussetzung, daß sich das Programm in einer Darstellung aus nichtindizierenden Schleifen über Folgen von Phasen befindet. Zum Phasenprogramm wird ein Phasengraph wie folgt erzeugt. Für jede Phase wird für jede mögliche Kombination von Datenaufteilungen der in der Phase vorkommenden Felder ein Knoten erzeugt und mit der Schätzung der Laufzeit dieser Phase bei ebendieser Kombination von Aufteilungen beschriftet. Für alle Paare von Knoten zweier aufeinanderfolgender Phasen werden Kanten erzeugt, die mit der Kostenschätzung für die Umverteilung von der ersten Kombination zur zweiten beschriftet werden. Ziel ist es nun, einen Pfad minimaler Kosten durch diesen Phasengraphen zu finden. Schleifen werden in gesonderter Weise behandelt. Prozeduren sind nicht vorgesehen.

Das Problem, eine optimale Datenaufteilung einschließlich statischer Umverteilung zu einem Phasenprogramm zu bestimmen, ist NP–vollständig [Kre93b]. Es widersetzt sich — zumindest in der Theorie — einem Ansatz auf der Basis dynamischer Programmierung wie [CKKM94] oder [LT93], da eine global optimale Kombination aus Datenaufteilungen und –umverteilungen in *jeder* Phase suboptimal sein kann.

[BKK93] formuliert das Datenauf– und –umverteilungsproblem als Instanz eines 0-1 ganzzahligen Linearen Programmierungsproblems. Mit Hilfe eines sehr schnellen Programms zur Lösung dieses 0-1-integer–programming–Problems kann eine *optimale* Lösung des Datenauf– und –umverteilungsproblems für beachtliche Problemgrößen (800 Programmzeilen) innerhalb von wenigen CPU–Sekunden gefunden werden.

[9] [GB90] beschränkt $K \leq 2$ und schreibt in diesem Fall quadratische Blöcke als Aufteilung entlang dieser beiden Dimensionen vor.

3.6 Bestimmung von Kommunikationsanweisungen höherer Ordnung

Unter einfachen Kommunikationsanweisungen verstehen wir die bisher behandelten `SEND` und `RECEIVE`–Anweisungen (*single send*, *single receive*). Durch Vektorisierung von Kommunikationsanweisungen erhalten wir `SEND`– und `RECEIVE`–Derivate für zusammenhängende Blöcke von Feldelementen.

Sei $\mathcal{P} = \{p_1, p_2, ...\}$ eine Gruppe von Prozessoren. Unter *Kommunikationsanweisungen höherer Ordnung* [FJL+88] verstehen wir folgende Operationen auf $\mathcal{P}$, die von fast allen realen DMS seitens der Hardware oder des Betriebssystems (z.B. Express [Par]) zur Verfügung gestellt werden:

- *global reduction*: Alle Prozessoren berechnen die globale Summe, das globale Produkt, UND, ODER oder eine ähnliche globale Verknüpfung einer assoziative binären Operation über je einem Skalar s_p in jedem Prozessor $p \in \mathcal{P}$. Das Ergebnis der Reduktion erscheint in einem ausgezeichneten der beteiligten Prozessoren.
- *shift*: Zyklisches Weiterschieben von einzelnen Randelementen der lokalen Feldpartitionen in $\mathcal{P}$.
- *one–to–all broadcast*: Ein Prozessor $p \in \mathcal{P}$ sendet dieselbe Nachricht an alle Prozessoren $p' \in \mathcal{P}$, $p' \neq p$.
- *gather*: Alle Prozessoren $p' \in \mathcal{P}$, $p' \neq p$, senden jeweils verschiedene Nachrichten an einen Prozessor $p \in \mathcal{P}$.
- *scatter*: Ein Prozessor $p \in \mathcal{P}$ sendet je eine (andere) Nachricht an alle Prozessoren $p' \in \mathcal{P}$, $p' \neq p$.
- *all–to–all broadcast*: Alle Prozessoren $p \in \mathcal{P}$ senden jeweils eine Nachricht an alle anderen Prozessoren $p' \in \mathcal{P}$, $p' \neq p$.

Diese vorgefertigten Routinen sind meist erheblich schneller als eine äquivalente Kombination aus einzelnen `SEND`– und `RECEIVE`–Anweisungen. Ziel ist es, für eine gegebene Anweisung Kommunikationsanweisungen möglichst hoher Ordnung zu finden, um von deren Geschwindigkeitsvorteil zu profitieren. Die Auswahl der Kommunikationsanweisungen ist abhängig von den Indizierungen der beteiligten Feldreferenzen, den indizierenden Schleifen und den gewählten Datenaufteilungen.

In [Who91] tritt dieses Problem nicht auf, da die Quellsprache die Äquivalente dieser Kommunikationsroutinen bereits explizit enthält; die Auswahl der Routinen wird dort also vom Programmierer übernommen.

Aspar [IFKF90] synthetisiert die Kommunikationsroutinen aus Express–Primitiven für eine sehr beschränkte, speziell auf die Livermore Loops [McM86] zugeschnittenen Menge von Operationen auf einem Feld A, die zu einer der folgenden vier Großgruppen gehören: (1) *independent cycles* (nirgends im Programm schleifengetragene Abhängigkeiten), (2) *combine type* (Reduktionen), (3) *concatenation type* (in der betrachteten Operation wie (1), aber es gibt Alignment–/Aufteilungskonflikte mit anderen Vorkommen dieses Feldes), und (4) *exchange type* (Gitterrelaxationen mit Austausch von Gitterrändern). Aspar kann damit für 14 der 24 Livermore Loops geeignete Express–Routinen selektieren.

[Gup92] präsentiert ein umfassendes Verfahren, das neben der schrittweisen Bestimmung guter Alignments und Datenaufteilungen auch die automatische Auswahl von Kommunikationsroutinen vorsieht. Zunächst wird für jede Dimension des zu kommunizierenden Feldes eine passende Kommunikationsroutine bestimmt. Anschließend werden diese Routinen entsprechend der Struktur der umgebenden Schleifenschachtelung zusammengesetzt.

Ähnlich werden die Kommunikationsroutinen in [LC91] bestimmt. [LC91] formuliert das Problem als Mustererkennungsproblem auf den Indizierungen der beteiligten Felder einer von indizierenden Schleifen umgebenen Zuweisung (*Referenzmuster*[10]), wobei erhebliche Einschränkungen angenommen werden: Reduktionen müssen, wie in [Who91], explizit als solche programmiert werden, und die Indexausdrücke der Felder auf der linken Seite einer Zuweisung dürfen nur entweder Konstanten oder Schleifenvariablen sein. Ein Referenzmuster R entspricht von der Funktionalität her einer `EXCH`–Anweisung und hat im wesentlichen die Form $A@(\sigma_1, ..., \sigma_{d_A}) \Rightarrow (\delta_1, ..., \delta_{d_A})$, wobei A der Name des Feldes ist, dessen Elemente verschickt werden sollen, d_A die Anzahl aufgeteilter Dimensionen von A, die σ_d die Indexausdrücke der verteilten Dimensionen von A beim Sender (eine Feldreferenz auf der rechten Seite einer Zuweisung) und die δ_d beim Empfänger (Feldreferenz auf der linken Seite einer Zuweisung) sind.

Beispiel 3.13 [LC91] Seien `A, B` in beiden Dimensionen aufgeteilt.

```
for (i=...)
  for (j=...)
    A[i][j] = B[C[i][j],j-3];
```

Das zugehörige Referenzmuster ist $B@(C[i][j], j-3) \Rightarrow (i, j)$. ◇

Die Kommunikationsroutinen erhalten (negative) Kosten, die der Laufzeitersparnis bei ihrer Verwendung (anstelle der einfachen Routinen `SEND` und `RECEIVE`) entsprechen. Im Gegensatz zu dem bottom–up–Ansatz in [Gup92] (zunächst dimensionsweises Vorgehen) wird in [LC91] top–down eine passende Routine für das gesamte Referenzmuster R gesucht: Falls man beim Durchsuchen der Liste von Kommunikationsroutinen eine genau passende Routine findet, ist man fertig. Paßt (aufgrund komplexer und mehrdimensionaler Indexausdrücke) keine der Routinen, so wählt man eine Dimension d mit einem komplexen Indexausdruck bei Sender und Empfänger und spaltet $R = A@(..., \sigma_d, ...) \Rightarrow (..., \delta_d, ...)$ in aufeinanderfolgende Teil–Referenzmuster R_1, R_2 auf[11], sodaß $R_1 = A@(..., \sigma_d, ...) \Rightarrow (..., i, ...)$ und $R_2 = A@(..., i, ...) \Rightarrow (..., \delta_d, ...)$, wobei i eine Dimension d aufspannende Schleifenvariable ist. In Beispiel 3.13 ist bei der Wahl $d = 1$ $R_1 = B@(C[i][j], j-3) \Rightarrow (i, j-3)$ und $R_2 = B@(i, j-3) \Rightarrow (i, j)$.

Anschließend wendet das Verfahren rekursiv auf R_1 und R_2 an. Ist man bei *elementaren* Referenzmustern angekommen, die sich nur noch in einer Dimension unterscheiden und sich nicht weiter aufspalten lassen, so wählt man direkt eine passende Routine aus (die einfachsten Routinen, `SEND` und `RECEIVE`, passen immer). In der Fortführung von Beispiel 3.13 für $d = 1$ sind R_1 und R_2 beide elementar; auf R_2 paßt beispielsweise die Kommunikationsroutine `SHIFT(0,+3)`. Wurde in mehrere Kommunikationsroutinen aufgespalten, so kann deren Reihenfolge noch optimiert werden [LC91].

Wir sehen, daß die Bestimmung optimaler Kommunikationsroutinen von der Wahl der Datenaufteilung abhängig ist. Um jedoch herauszufinden, welche Datenaufteilung besser ist, braucht man eine Aussage über die Laufzeit des parallelisierten Programms, insbesondere für die resultierende Kommunikation — ein Abhängigkeitszyklus. [LC91] und [Gup92] empfehlen daher, die Berechnung der Datenaufteilung in mehrere Pässe aufzuteilen und die Laufzeitvorhersage damit schrittweise zu verfeinern.

3.7 Laufzeitvorhersage

Der übliche Ansatz zur Laufzeitvorhersage bei mit der in Abschnitt 3.4 beschriebenen Methode zu gegebenen Feldaufteilungen parallelisierten DMS–Programmen ist ein bottom–up–Durchlaufen

[10] Der Name *Referenzmuster*, der in [LC91] eingeführt wird, ist irreführend. Als Mustererkennungsproblem aufgefaßt, entsprechen die Kommunikationsroutinen den Mustern, die im ‘Referenzmuster’ erkannt werden sollen. Geeigneter wäre der in [DHR94] benutzte Terminus ‘Kommunikationsverursacher’.

[11] Die Wahl dieser Dimension ist beliebig, beeinflußt aber erheblich die Qualität der später gefundenen Kommunikationsroutinen. Das Finden einer optimalen Aufspaltungsreihenfolge stellt ein NP–hartes Optimierungsproblem dar, dem [LC91] durch Dynamisches Programmieren zu Leibe zu rücken versucht.

der hierarchischen Knotenprogramm–Darstellung, wobei die geschätzten Laufzeiten für arithmetische Operationen und die geschätzten Laufzeiten für die Interprozessorkommunikation nach 'oben' propagiert werden. Wir wollen dies im folgenden kurz skizzieren.

Lastverteilung, Laufzeit der arithmetischen Operationen Die Anzahl nützlicher arithmetischer Operationen kann bei gleichmäßiger Lastverteilung als konstant für alle Prozessoren p angesehen werden: $w(p) = T_{seq}/P$, wobei T_{seq} die sequentielle Laufzeit des Restprogramms und P die Prozessorzahl bezeichne. Die Gesamtdauer $W(p)$ der von p ausgeführten arithmetischen Operationen kann jedoch $w(p)$ beträchtlich übersteigen, wenn (1) Skalare und ggf. gewisse Felddimensionen repliziert werden, wenn (2) ein minderdimensionales Feld evtl. nicht über alle Prozessoren verteilt ist, oder wenn (3) Operationen entlang eigentlich aufgeteilter Feldachsen durch Datenabhängigkeiten sequentialisiert werden, weil immer ein Prozessor auf die Ergebnisse eines anderen warten muß.

[Fah93] nimmt als Approximation für die Rechenlast eines Prozessors bei einer Zuweisung die Größe des lokalen Bereichs an, der auf der linken Seite der Zuweisung geschrieben wird. Diese Abschätzung wird dann ungenau, wenn manche Feldelemente häufiger geschrieben werden als andere, wie z.B. in `A[n+i-j] = ...`, falls `i` und `j` gebundene Schleifenvariablen sind.

Im allgemeinen aber stellt die Lastverteilung eher ein untergeordnetes Problem der Laufzeitvorhersage dar, da die Kosten für Interprozessorkommunikation erheblich teurer zu Buche schlagen.

Möchte man jedoch die genaue *Laufzeit* des Arithmetik–Anteils des (sequentiellen) Knotenprogramms berechnen, so stößt man rasch an die Grenzen analytischer Vorhersagetechnik: Für sehr einfache Testprogramme (einige Livermore Loops [McM86]) zeigt [Mac94], daß für übliche DMS–Knotenprozessoren (i860, T800) bereits Vorhersagefehler im zweistelligen Prozentbereich eher die Regel als die Ausnahme sind.

Interprozessor–Kommunikation In Abschnitt 3.4.2 haben wir beschrieben, wie für Schleifenschachtelungen der Gestalt

```
for (i=....)
  for (j=....)
    ...
       A[...]..[...] = B[...]..[...] op1 C[...]..[...] op2 ....
```

die in der Prozessornummer p parametrisierten lokalen Indexmengen L_A, die referenzierten Indexmengen M_B, M_C,... und daraus die Kommunikationsmengen $Comm_B(p,p')$, $Comm_C(p,p')$,... bestimmt werden. Durch die Parametrisierung in der Prozessornummer können bei der expliziten Darstellung dieser Mengen, sofern man sie überhaupt automatisch finden kann, u.U. komplexe symbolische Ausdrücke entstehen, deren Verarbeitung schwierig wird. In [DHR94] werden daher die lokalen Indexmengen und die daraus resultierenden Kommunikationskombinationen explizit und für jeden einzelnen Prozessor berechnet[12], was natürlich zu Lasten der Analysegeschwindigkeit geht. Jedenfalls muß, um weiterrechnen zu können, $Comm_B(p,p')$ explizit vorliegen.

Falls die zugehörige Kommunikationsanweisung vollständig vektorisiert werden kann, gibt die Anzahl der nichtleeren Mengen $Comm_B(p',p)$ für alle p' gerade die Anzahl $SN_B(p)$ der von p empfangenen Nachrichten an. Die Menge $Comm_B(p) = \cup_{p' \neq p} Comm_B(p',p)$ enthält die Menge aller Elemente von B, die p empfängt.

Die Menge $Comm(p) = Comm_B(p) \cup Comm_C(p) \cup ...$ gibt dann alle Feldelemente an, die p bei der Kommunikation vor dieser Anweisung empfängt. Die Menge der von p empfangenen Nachrichten ist $SN(p) = SN_B(p) \cup SN_C(p) \cup ....$ Durch die Vektorisierung von Kommunikationsanweisungen ist im allgemeinen $|SN(p)| \ll |Comm(p)|$.

Die Kommunikationskosten für eine aus n konsekutiven Feldelementen bestehende Nachricht haben für typische DMS im allgemeinen die idealisierte Form

[12] Dies entspricht einer Simulation des gesamten Kommunikations–Laufzeitverhaltens des parallelisierten Programms! Eine Realisierung für größere Prozessorzahlen ist daher kaum vertretbar.

$$t(p_1, p_2, n) = \sigma + \beta \cdot \rho \cdot n + \gamma \cdot dist(p_1, p_2) \tag{3.5}$$

mit der (mittleren) Kommunikations–Startupzeit σ, der Wortgröße β (in Bytes) und der (mittleren) Byteübertragungszeit ρ. Die Anzahl $dist(p_1, p_2)$ der *hops* gibt die netzwerktopologische Entfernung zweier Prozessoren p_1 und p_2 an; γ bezeichnet eine zusätzlichen Startupzeit für jeden hop. Letzterer Term kann in realen Systemen angesichts hoher Startupzeiten gewöhnlich vernachlässigt werden.

Damit haben wir das notwendige Rüstzeug zur Schätzung der Kommunikationskosten für eine vollständige Schleifenschachtelung. In der Gegenwart weiterer Schleifen oder von Bedingungen müssen diese Werte mit einem Faktor gewichtet werden, der die *Häufigkeit* der Ausführung zur Laufzeit repräsentieren soll: die *Ausführungsfrequenz* [Fah93]. Zur Berechnung der Ausführungsfrequenz benötigt man Programmparameter wie die Anzahl von Wiederholungen einer Schleife (iteration count) oder das Wahr/Falsch–Verhältnis einer Bedingung (true ratio). Hierzu kann man entweder grobe Abschätzungen vornehmen [Hay93] oder einen Profile–Lauf des (sequentiellen) Programms bei 'typischen' Eingabedaten durchführen (*Weight Finder* [Fah94]).

Sequentialisierung trotz Datenaufteilungen Jede Interprozessorkommunikation bewirkt eine partielle Synchronisierung zweier Prozesse. Daher brauchen wir noch eine Abschätzung der Verzögerung von Prozessoren aufgrund von durch Datenabhängigkeiten induzierte Kommunikation (Sequentialisierung). [DHR94] spezifiziert hierzu einen (gerichteten) *Datentransfergraphen*. Knoten sind die Prozessoren[13]; eine Kante (p_i, p_j) existiert genau dann, wenn $Comm(p_i, p_j)$ nichtleer ist aufgrund einer schleifengetragenen Datenabhängigkeit. So wären in

```
for (i=1; i<=I; i++)
  for (j=1; j<=J; j++)
    A[i][j] = A[i-1,j];
```

bei einer zeilenweisen Aufteilung von `A` auf vier Prozessoren $p_1, ..., p_4$ die Kanten (p_k, p_{k+1}) für $1 \leq k \leq 3$ im Datentransfergraphen enthalten, d.h. Prozessor p_{k+1} muß auf Nachrichten von p_k warten. Enthält der Datentransfergraph einen Zyklus, so können die im Zyklus befindlichen Prozessoren *nicht* parallel arbeiten; ja schlimmer noch, zum Verlust an Parallelismus kommen noch die Kommunikationsverzögerungen hinzu.

Nun betrachtet man die starken Zusammenhangskomponenten des Datentransfergraphen als Einheit. Die verbleibenden Kanten in der azyklischen Kondensation des Datentransfergraphen zerfallen in zwei Klassen: (1) Bei *Verzögerungskanten* (*delay edges*) muß der empfangende Prozessor warten, bis der sendende Prozessor seine Arbeit beendet hat. Erst nach dem Empfangen der Nachricht kann der Empfänger seine Arbeit aufnehmen. (2) Bei *Parallelkanten* (*parallel edges*) kann der Sender zuerst die Nachricht wegschicken und dann seine Arbeit aufnehmen, sodaß Sender und Empfänger im wesentlichen parallel arbeiten können. Eine formale Definition von Verzögerungs– und Parallelkanten findet man in [Hay93]. Wir können die Parallelkanten vernachlässigen. Aus dem resultierenden azyklischen Datentransfergraphen entnimmt man die Gesamtverzögerung als Länge des längsten Pfades, wobei sich die Länge eines Pfades P zusammensetzt aus der Summe der Prozessorarbeitsdauern $w(p)$ und der Kommunikationszeiten $|Comm(p')|$ für $(p, p') \in P$.

Diskussion Je genauer die Laufzeitvorhersage sein soll, desto mehr muß sie das Programm simulieren, desto stärker wird sie abhängig von dem idealisierten Modell der Zielmaschine, das ihr zugrundeliegt. Beispielsweise kann die Formel 3.5 durchaus komplizierter werden, z.B. durch das Einarbeiten verschiedener Protokolle für verschieden große Nachrichten, durch die — bisher vernachlässigte — Berücksichtigung der Einbettung in die Netztopologie, durch die Einarbeitung von Puffergrößen. Bei realen Rechnern ist insbesondere der Empfangspuffer kritisch: Falls ein Prozessor zum annähernd gleichen Zeitpunkt von vielen Sendern große Nachrichten erhält, kann der Puffer überlaufen — Daten gehen verloren, der Rechner bricht ggf. das Programm ab.

[13] Hierzu gelten die schon erwähnten Vorbehalte bezüglich der Einsetzbarkeit dieses Verfahrens bei größeren Prozessorzahlen.

Auch so kann es vorkommen, daß ein Prozessor eine ganze Weile untätig auf eingehende Nachrichten wartet, und er dann innerhalb kürzester Zeit einen gut gefüllten Empfangsspeicher abarbeiten muß. Ähnlich kann es 'hot spots', Staus und Kulminationspunkte, im Verbindungsnetzwerk geben, sodaß manche Nachrichten lange aufgehalten werden (*network contention*), weil ihre Routen zur gleichen Zeit dieselbe Verbindung im Netz benutzen. Laut [Bok90] können durch network contention beim Intel iPSC/860 Verzögerungen um einen Faktor bis zu 7 auftreten. Diese Verzögerungen sind aufgrund der asynchronen Arbeitsweise der (meisten) DMS kaum vorhersagbar (*chaotisches System*).

Die Überlappung von Kommunikation und Arithmetik (Pipelining) wird ebenfalls nicht vollständig berücksichtigt. Ansätze dazu findet man in [Gup92] und [Hay93].

Daher versucht man neuerdings, die oben beschriebenen analytischen Verfahren mit *Benchmarking* zu kombinieren (*synthetische Verfahren* zur Laufzeitvorhersage). Für typische arithmetische Operationen wie Vektoranweisungen, Skalarprodukt usw., aber auch für einfache und Kommunikationsanweisungen höherer Ordnung werden die Laufzeiten auf der Zielmaschine *gemessen* anstatt analytisch abgeschätzt. Damit wird man ein gutes Stück unabhängiger von dem idealisierten Zeitmodell der Zielarchitektur, und die analytisch kaum erfaßbaren Verzögerungseffekte (aber auch Beschleunigungseffekte durch Caches) werden nun besser berücksichtigt. Erste Schritte in diese Richtung weisen [BFKK91] und [FMPB94]. Es zeigt sich, daß analytische Verfahren in der Praxis große Abweichungen (z.B. durchschnittlich 50% bei den Nachrichtengrößen in [Fah93]) von den gemessenen Werten produzieren können, während die synthetischen Verfahren im allgemeinen exakter sind (z.B. $\leq 5\%$ in [FMPB94]).

4 Grundlagen zur syntaktischen Mustererkennung in Programmen

[KNE93] unterteilt das Problem der Erkennung von Mustern in Programmen und deren Transformation in andere Muster in insgesamt vier Klassen, die sich durch wachsende Komplexität unterscheiden:

1. **Textebene** (string-matching und text-level–Transformationen): Dies entspricht den in den meisten Text–Editoren verfügbaren Operationen „finde Zeichenkette s in einem Text" bzw. „ersetze ein oder alle Vorkommen der Zeichenkette s in einem Text". Effiziente Implementierungen findet man in [AC75] oder [KMP77]. Diese sehr einfachen Operationen sind für unseren Anwendungsfall nicht leistungsfähig genug.

2. **Syntaktische Ebene** (tree pattern matching / tree parsing): Das Erkennen bzw. Ersetzen von Baummustern (*tree patterns*) in einem abstrakten Syntaxbaum kann z.B. für einfache Programmtransformationen (z.B. Umbenennen von Variablen) oder für die Codegenerierung verwendet werden. In Abschnitt 4.1 stellen wir diese wichtige Klasse und die wesentlichen Techniken des Treepatternmatching[1] im Überblick vor.

3. **Semantische Ebene**: Um die Semantik eines Programmstücks verstehen zu können, muß man — im Gegensatz zu einer rein syntaktischen Betrachtung — die Semantik der Programmiersprache, in der es geschrieben ist, in Betracht ziehen. Hierzu benötigt man Informationen über Kontroll– und Datenfluß im Quellprogramm. Auf semantischer Ebene können auch *delokalisierte* Konstrukte erkannt werden, d.h. Codeteile, die zwar zusammengehören (zum selben „Berechnungsstrang" gehören), aber nicht textuell unmittelbar benachbart sind. Auf semantischer Ebene arbeiten die meisten semantik–erhaltenden Transformationen wie z.B. die oben beschriebenen Schleifentransformationen.

4. **Konzeptebene**: Transformationen auf Konzeptebene erfordern Hintergrundwissen, z.B. über Konvergenzeigenschaften und numerische Stabilität, Speicherorganisation, Datenstrukturen usw. Zum einen muß ein Fundus solchen Hintergrundwissens („abstrakte Konzepte") für das Transformationssystem auf irgendeine Weise verfügbar sein; zum andern ist *Konzepterkennung*, d.h. das Herleiten entsprechender Beziehungen aus dem Quellprogramm, erforderlich, um das Hintergrundwissen überhaupt nutzen zu können. Die in Kapitel 7.1 vorgestellte lokale Algorithmenersetzung entspricht solchen Transformationen auf Konzeptebene; der Konzepterkennung entspricht z.B. das Herleiten von Gitterhierarchien.

Wir streben an, die Konzeptebene zu erreichen. Als Grundtechnik zur Konzepterkennung benutzen wir die Mustererkennung auf syntaktischer und semantischer Ebene, die wir an geeigneter Stelle um zusätzliche Komponenten erweitern.

Wie [KNE93] bemerkt, ist es illusorisch, zu erwarten, man könne für ein nichttriviales, größeres Anwendungsprogramm ein einziges Konzept (oder Muster) finden. Daher versucht man, auf lokaler Ebene „Inseln" von erkennbaren Konzepten zu finden und das Programm soweit wie möglich durch Konzepte (Muster) zu überdecken. Dabei kann man Treepatternmatching (Klasse 2) als wichtigste Hilfstechnik verwenden (wie in [Sny82]), muß sie jedoch um weitere Techniken ergänzen.

[1] Wir benutzen im folgenden absichtlich die englischen Bezeichnungen „Treepatternmatching" bzw. „Treeparsing", die für die formal–syntaktische Erkennung (Klasse 2) stehen, um diese von dem Begriff „Mustererkennung" abzugrenzen, den wir für die Erkennung semantischer und konzeptioneller Zusammenhänge reserviert haben.

4.1 Treepatternmatching

Die klassische Treepatternmatching–Theorie versucht, das formale Baumüberdeckungsproblem auf syntaktischer Ebene zu lösen. Für den folgenden Überblick orientieren wir uns an [HO82], [WW88], [WM92] und [Fer90].

Gegeben sei ein endliches Alphabet Σ von Operatorsymbolen mit Stelligkeit, einschließlich Konstantensymbolen der Stelligkeit 0.

Definition 4.1 (Σ–Terme) Die Menge der Σ–*Terme* über dem Alphabet Σ ist wie folgt definiert:
(1) Alle Konstantensymbole b aus Σ sind Σ–Terme.
(2) Für ein Operatorsymbol $a \in \Sigma$ mit Stelligkeit q ist $a(t_1, \ldots, t_q)$ ein Σ–Term, falls die t_i auch Σ–Terme sind für alle $i \in \{1, ..., q\}$.
(3) Das sind alle Σ–Terme. □

Die Σ–Terme können als geordnete Bäume dargestellt werden, deren Knoten mit Symbolen aus Σ beschriftet sind. Beide Darstellungen sind isomorph. Daher werden die Σ–Terme auch Σ–Bäume genannt.

Entsprechend definieren wir *Pfade* in Σ–Bäumen: Ein Knoten v, der mit einem Konstantensymbol beschriftet ist, ist ein Pfad der Länge 1. Ist $(v_1, v_2, ..., v_k)$ ein Pfad der Länge k, und ist v_k Operand von v, so ist $(v_1, ..., v_k, v$ ein Pfad der Länge $k + 1$. Die *Höhe* eines Σ–Baumes ist die Länge des längsten darin vorkommenden Pfades von einem Blatt zur Wurzel.

Sei $\nu \notin \Sigma$ ein spezielles Symbol (*Variablensymbol*, „Joker") mit Stelligkeit 0, das als Platzhalter für jeden beliebigen Σ–Baum dient. Für $\Sigma_\nu = \Sigma \cup \{\nu\}$ werden die Σ_ν–Terme bzw. –Bäume analog definiert, wobei ν auch ein Σ_ν–Term ist.

Definition 4.2 (Treepattern) Ein *Treepattern* ist ein beliebiger Σ_ν–Term. □

Definition 4.3 (Treepattern matcht an einem Knoten) Ein Treepattern p mit k Vorkommen von ν *matcht* einen gegebenen Σ–Term t *am Knoten* n, falls es Σ–Bäume $t_1, \ldots, t_k$ gibt, sodaß der Σ–Baum t', der durch Ersetzen des i–ten Vorkommens von ν durch t_i für alle $i \in \{1, ..., k\}$ aus p hervorgeht, gleich ist zu dem Unterbaum von t mit Wurzel n. □

Definition 4.4 (Treepatternmatching–Problem) Eine Instanz eines *Treepatternmatching–Problems* besteht aus einer endlichen Menge $F = \{p_1, ..., p_m\}$ von Treepattern (Σ_ν–Bäume) und einem Eingabebaum (Σ–Term) t. Die Lösung des Treepatternmatching–Problems ist die Liste aller Paare (n, j), sodaß n ein Knoten in t ist und das Treepattern $p_j \in F$ den Baum t am Knoten n matcht. □

Für die Anwendung in Termersetzungssystemen werden im allgemeinen mehrere verschiedene Variablensymbole $\nu_1, \nu_2, \ldots$ benötigt, sodaß für jedes Variablensymbol nur bestimmte Terme eingesetzt werden dürfen.

Die z.B. bei Unifikation (vgl. z.B. [CM84, Han87]) erforderliche simultane Ersetzung mehrerer Vorkommen des gleichen Variablensymbols (repeated variables) durch *gleiche* Terme wird durch diese (klassische) Definition des Treepatternmatching–Problems nicht berücksichtigt. Treepattern, bei denen mehrere gleiche Variablensymbole auf diese Weise an gleiche Terme gebunden werden müssen, heißen *nichtlinear* [HS93]. Wir werden in Abschnitt 4.1.3 hierauf zurückkommen.

4.1.1 Bottom–Up–Verfahren zum Tree Pattern Matching–Problem

Der naive Ansatz zur Lösung des Treepatternmatching-Problems probiert für jeden Knoten n des Eingabebaumes alle Treepattern $p_j \in F$ durch. Die hierzu benötigte Zeit ist proportional zum Produkt aus der Größe des Eingabebaumes und der Gesamtgröße aller Treepattern.

Häufig ist jedoch die Menge F der Treepattern fest, sodaß vorab eine Vorbehandlung der Treepattern durchgeführt werden kann. Die daraus resultierende Information kann dann zum schnelleren Treepatternmatching für beliebig viele Eingabebäume verwendet werden. Man handelt also Preprocessing–Zeit gegen schnellere Match–Zeit ein. Nach diesem Prinzip arbeiten die im folgenden vorgestellten bottom–up– und top–down–Algorithmen.

Der Bottom–Up–Ansatz zur Lösung des Treepatternmatching–Problems wurde unabhängig von Kron [Kro75] und Hoffmann/O'Donnell [HO82] entwickelt.

Zunächst wird die Menge F erkennbarer Treepattern in alle darin vorkommende Unterpattern (Unterbäume) zerlegt. Die Kernidee besteht nun darin, für jeden Knoten des Eingabebaumes bottom–up die Menge derjenigen dieser Unterpattern zu berechnen, die an diesem Knoten matchen können. Diese Menge heißt *Match–Set*. Da die Treepattern spezielle Unterpattern sind, wird dabei auch die gesuchte Lösung des Treepatternmatching–Problems mitberechnet.

Definition 4.5 (Match–Set) Sei $F = \{p_1, ..., p_m\}$ eine Menge von Treepattern (Σ_ν–Bäume) und PF die Menge aller Unterpattern der p_j. Eine Teilmenge M von PF ist ein *Match–Set* für F, falls es einen Σ–Baum t gibt, sodaß (1) jedes Treepattern aus M den Baum t an der Wurzel matcht und (2) jedes Treepattern aus $PF - M$ den Baum t nicht an der Wurzel matcht. □

Man berechnet zuerst alle Match–Sets, die vorkommen können. Die folgende rekursive Formel liefert die Match–Sets $match(t)$ zu den Wurzelknoten aller Σ–Bäume t:

1. Falls $t = b$ und b hat Stelligkeit 0, dann ist $match(b) = \{b, \nu\}$, falls $b \in PF$, und $match(b) = \{\nu\}$ sonst.
2. Falls $t = a(t_1, \ldots, t_q)$ und a hat Stelligkeit $q > 0$, dann ist $match(a) = \{\nu\} \cup \{t' | t'$ hat Wurzel a, $t' \in PF$ und für alle Söhne t_i von t' gilt $t_i \in match(t_i)$, $1 \leq i \leq q$.

Der Treepattern–Wald F ist endlich und damit auch die Menge PF aller Unterpattern von F. Also gibt es auch nur endlich viele Match–Sets zu F. Σ ist endlich. Somit kann man in einem Vorlauf, der nur von F und Σ abhängt, alle Match–Sets vorberechnen, indem man die rekursive Funktion iterativ für alle Σ–Bäume der Höhen 1, 2, ... auswertet (Abschluß–Berechnung) und alle verschiedenen Match–Sets notiert. Sobald für eine Baumhöhe h^* keine neuen Match–Sets mehr hinzukommen, kann man das Iterationsverfahren abbrechen. Die gefundenen verschiedenen Match–Sets werden durchnumeriert (1,2,...).

Anschließend wird für jedes Operatorsymbol $a \in \Sigma$ mit Stelligkeit $n_a > 0$ eine Tabelle $M[a]$ der Dimension n_a angelegt, sodaß später aus $M[a](f_1, \ldots, f_{n_a})$ die Nummer des Match–Sets eines Baumes mit Wurzel a, für deren Söhne bereits die Match–Sets $f_1, \ldots, f_{n_a}$ bestimmt wurden, abgelesen werden kann.

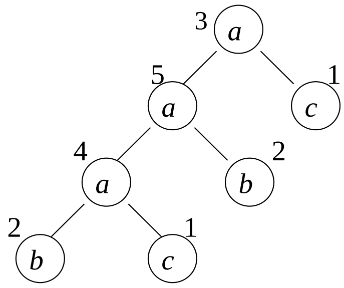
Beispiel 4.1 (vgl. [HO82]) Seien $\Sigma = \{a, b, c\}$ mit a zweistellig, b und c nullstellig, und der Treepatternwald $F = \{p_1, p_2\}$ mit $p_1 = a(a(\nu, \nu), b)$ und $p_2 = a(b, \nu)$ gegeben. Die Menge aller Unterpattern von F ergibt sich zu $PF = \{\nu,\ b,\ a(\nu, \nu),\ a(b, \nu),\ a(a(\nu, \nu), b)\}$.

Zu PF existieren folgende Match–Sets: Match–Set 1 = $\{\nu\}$,
Match–Set 2 = $\{\nu,\ b\}$,
Match–Set 3 = $\{\nu,\ a(\nu, \nu)\}$,
Match–Set 4 = $\{\nu,\ a(\nu, \nu),\ a(b, \nu)\}$ und
Match–Set 5 = $\{\nu,\ a(\nu, \nu),\ a(a(\nu, \nu), b)\}$.

Man erhält folgende Match–Tabelle $M[a]$ für a:

linker Sohn	rechter Sohn 1	2	3	4	5
1	3	3	3	3	3
2	4	4	4	4	4
3	3	5	3	3	3
4	3	5	3	3	3
5	3	5	3	3	3

Die Tabelle für b besteht aus dem einzigen Eintrag 2, die für c aus 1 (da b und c nullstellig sind, sind die Tabellen nulldimensional). Beispielsweise erklärt sich der Eintrag $M[a](3,2) = 5$ damit, daß am linken Sohn die Unterpattern ν und $a(\nu, \nu)$ aus Match–Set 3 matchen, am rechten Sohn die Unterpattern b und ν aus dem Match–Set 2.



Für den Eingabebaum $a(a(a(b,c),b),c)$ werden die Match–Sets, wie in nebenstehendem Bild angegeben, berechnet. Dabei geben die Zahlen die Nummern der für die Knoten berechneten Match–Sets an. Nummer 5 zeigt an, daß p_1 matcht, und 4, daß p_2 matcht. $\diamond$

Für eine Gesamtzahl von μ Match–Sets ist die Größe von $M[a]$ somit $O(\mu^{n_a})$. [HO82] konstruiert eine Menge von Treepattern, für das die Anzahl μ der Match–Sets exponentiell von der Anzahl der Unterpattern abhängt.

Realistische Pattern–Mengen resultieren gewöhnlich in riesigen Tabellen. Durch Tabellenkomprimierung (z.B. [BMW91],[Cha87]) kann dieses Platzproblem teilweise behoben werden.

Nach der Konstruktion der Tabellen kann in einem zweiten, eingabeabhängigen Schritt in einem postorder–Durchlauf des Eingabebaumes für jeden Knoten n mit Symbol $a \in \Sigma$ mit Hilfe der Tabelle $M[a]$ die Nummer des Match–Sets, das genau die den Knoten n matchenden Unterpattern enthält, berechnet werden. Die Match–Zeit ist linear in der Größe des Eingabebaumes.

Im Gegensatz zu diesem tabellenorientierten Ansatz aus [HO82] generiert Kron [Kro75] in der Vorlaufphase einen deterministischen endlichen bottom–up–Baumautomaten. Für jedes Symbol des Alphabets wird dabei ein Teilautomat konstruiert, der angesetzt auf einen Knoten eines Eingabebaums, der dieses Symbol trägt, in einen Endzustand übergeht, der dessen codiertem Match–Set entspricht.

Die Treepattern–Matcher von [Kro75] und [HO82] sind äquivalent, da die Automatenendzustände gerade den Match–Sets von [HO82] entsprechen und die Tabellen von [HO82] als spezielle Darstellung der Automaten–Übergangsfunktion dargestellt werden können. Die Matching–Zeiten (Phase 2) sind linear in der Größe des zu matchenden Eingabebaumes, weil für jeden Knoten des Eingabebaumes nur ein Tabellenzugriff bzw. ein Übergang im Baumautomaten ausgeführt werden muß. Der Platzbedarf bei der Implementierung ist in beiden Fällen sehr hoch und kann teilweise komprimiert werden (z.B. [BMW91]).

4.1.2 Top–Down–Verfahren zum Tree Pattern Matching–Problem

Als Alternative beschreibt [HO82] einen Top–Down–Treepatternmatcher, der zwar geringe Vorlaufzeit aufweist (linear in der Anzahl der Unterpattern), dessen Laufzeit aber im schlechtesten Fall proportional zum Produkt aus der Größe des Eingabebaumes und der Anzahl der Unterpattern ist.

Die Treepattern werden dabei in alle ihre Pfade zerlegt und diese als Zeichenketten (Pfadstrings) dargestellt. Gemeinsame Präfixe werden zusammengefaßt, sodaß sich ein Trie[2] ergibt. Mit einem string–matcher–Generator z.B. nach [AC75] wird zur Vorlaufzeit aus diesem Trie der Treepattern ein endlicher Automat erzeugt, der zur Laufzeit (Matchzeit) genau die Pfadstrings matcht, d.h. für jeden Pfadstring genau dann in einem akzeptierenden Zustand endet, falls dieser einen Knoten des Eingabebaumes matcht. Zur Match–Zeit wird bei Beginn eines Depth-First–Durchlaufs durch den Eingabebaum dieser Automat an der Wurzel des Eingabebaumes gestartet. Wird ein Pfadstring erkannt, so wird dies mittels je eines Zählers für jedes Treepattern an der Wurzel des Unterbaumes

[2] Ein *Trie*, auch *digitaler Suchbaum* [Meh86] genannt, stellt eine Menge von Zeichenketten als Pfade in einem Suchbaum dar. Dabei gibt jede Verzweigung in Tiefe k die Optionen für die Wahl des k–ten Zeichens (Kantenbeschriftung) bei festem Präfix an. Die Wurzel hat Tiefe 1. Seien z_1, z_2 zwei Zeichenketten, deren erste $k-1$ Zeichen gleich sind und die sich im k–ten Zeichen unterscheiden. Dann sind die z_1 und z_2 entsprechenden Pfade π_1 bzw. π_2 im Trie bis in Tiefe k identisch, wo sie dann unterschiedlich verzweigen.

vermerkt, der den Pfadstring matcht. Stellt man an einem Knoten n des Eingabebaumes fest, daß alle Pfadstrings eines Treepattern p akzeptiert wurden, wird (n, p) ausgegeben. Die Größe des Automaten und damit die Vorlaufzeit ist proportional zur Größe aller Treepattern. Die Matchzeit ist proportional zur Größe des Eingabebaumes plus der Anzahl der Zähleränderungen; im besten Fall ist sie also linear in der Größe des Eingabebaumes, im schlechtesten Fall wird aber die Komplexität des naiven Verfahrens erreicht.

4.1.3 Nichtlineare Treepattern

Es gibt prinzipiell drei Möglichkeiten für das Matchen nichtlinearer Treepattern. Zum einen kann man sie zunächst wie lineare Treepattern behandeln und dann nach jedem erfolgreichen Match Tests auf Gleichheit entsprechender Unterbäume durchführen. Dabei wird es im allgemeinen erforderlich werden, daß Teile des Eingabebaumes mehrfach besucht und damit Effizienzeinbußen in Kauf genommen werden müssen.

Als Alternative bietet sich an, die Ergebnisse aller möglichen Tests auf Gleichheit von Unterbäumen des Eingabebaumes vorab zu berechnen [Sny82]. Dazu werden vor dem Matchen in einem Durchlauf durch den Eingabebaum für alle Knoten der Höhen 1, 2, ... identische Unterbäume bestimmt und durch jeweils gleiche Ordnungsnummern gekennzeichnet, deren Gleichheit dann beim Matchen unmittelbar getestet werden kann. Der Erwartungswert der Laufzeit für diesen Vorab–Durchlauf ist bei Verwendung von bucket–sort linear in der Größe des Eingabebaumes.

Als dritte Möglichkeit modifiziert [PB85] den bottom–up–Algorithmus aus [HO82] für nichtlineare Treepattern. Die Zeit der Vorlaufphase ist linear in der Größe aller Treepattern. Alle Treepattern werden durch einen einzigen komprimierten gerichteten azyklischen Graphen (DAG) dargestellt; gleiche Unterausdrücke werden in jedem Treepatternknoten v durch eine geeignete Abbildung der Variablen der Unterpattern–Knoten auf die Variablen des Treepattern unter v vermerkt. Der Eingabebaum wird durch Zusammenfassung gemeinsamer Teilausdrücke als DAG dargestellt. Die Match–Sets enthalten neben den matchenden Unterpattern–Knoten auch die zum Matchen notwendigen Variablenbindungen. Durch spezielle Optimierungen werden in großen Eingabebäumen Laufzeitverbesserungen gegenüber der oben erwähnten ersten Möglichkeit erreicht.

4.1.4 Vergleich

Zusammenfassend kann man feststellen, daß bottom–up Treepatternmatcher schnellere Laufzeiten (Match–Zeiten) aufweisen, aber mehr Platz (und damit Vorlaufzeit) benötigen (schlimmstenfalls exponentiell in der Größe des Treepattern–Walds F), während top–down Treepatternmatcher schneller generiert werden können und weniger Platz erfordern.

Weil alle Treepatternmatcher nur auf syntaktischer Ebene arbeiten, haben sie keine Möglichkeiten, festzustellen, ob verschiedene Unterpattern beispielsweise die gleiche Semantik besitzen und somit in einem Match–Set zusammengefaßt werden können. Die Ausdrücke `2*(i+1)` und `2*i+2` beispielsweise sind strukturell verschieden; um sie zu matchen, sind zwei verschiedene Treepattern erforderlich. In der üblichen Interpretation als arithmetische Ausdrücke wäre ihre Semantik aber die gleiche.

Das Problem exponentiellen Platzbedarfs für die Match–Sets kann — wie von uns später angewandt — dadurch gelöst werden, daß den vergleichsweise „dummen“ Baumautomaten an geeigneter Stelle ausreichend lokaler random–access–Speicher (und damit Turing–Mächtigkeit) verliehen wird.

Treepatternmatcher werden zur Code–Erzeugung (insbesondere bei Codeerzeuger–Generatoren) und zum Transformieren von Bäumen eingesetzt.

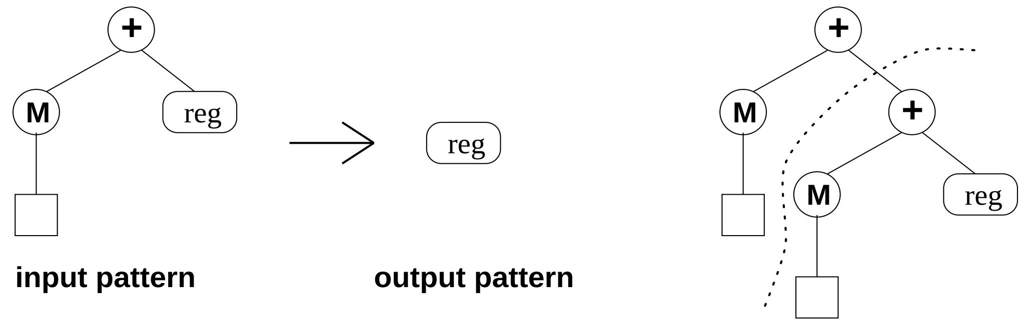


Abbildung 4.1 Links: Produktion einer Baumgrammatik mit input–pattern und output–pattern. Rechts: Eingabebaum. Das links stehende input–pattern matcht den Unterbaum rechts der gestrichelte Linie. Ersetzt man diesen Unterbaum durch das output–pattern (Reduktionsschritt), so matcht das input–pattern den reduzierten Baum an der Wurzel, sodaß noch einmal reduziert werden kann und nur ein Knoten *reg* übrigbleibt.

4.2 Baumgrammatiken, Treeparsing und Codeerzeugergeneratoren

Bei der Codeerzeugung entspricht der Eingabebaum dem abstrakten Syntaxbaum als Zwischendarstellung eines Ausdrucks im Quellprogramm und das Alphabet Σ der Signatur, mit der die Knoten dieses Syntaxbaumes beschriftet sind, nämlich Operatorsymbole, Konstanten– und Variablennamen.

Codeerzeuger–Generatoren ermöglichen ein bequemes nachträgliches Ändern der Zielmaschinenspezifikation (*retargetable code generation*). Die Zielmaschine wird dabei durch eine Maschinengrammatik beschrieben.

Jedes Treepattern in F, nennen wir es jetzt *input–pattern*, entspricht *einer* Zielmaschineninstruktion bei *einer* Adressierungsart, die Variablensymbole entsprechen z.B. den Registertypen, die diese Instruktion erwartet. Diese Instruktion, ausgeführt auf der Zielmaschine, produziert ein Ergebnis, dessen Registertyp durch einen entsprechend beschrifteten Knoten (*output–pattern*) dargestellt werden kann.

Man betrachte das Beispiel in Abb. 4.1. Dort wird durch zwei Reduktionsschritte der Eingabebaum auf der rechten Seite zu einem Knoten reduziert.

Eine (annotierte) *Reduktionsregel* besteht aus einem input–pattern, einem output–pattern, einer Maschineninstruktion und einer Kostenfunktion, welche den benötigten Speicherplatz und/oder die Anzahl der benötigten Maschinenzyklen dieser Instruktion angibt.

Durch die Hintereinanderausführung von Reduktionsregeln kann der Eingabebaum i.allg. sukzessive auf einen Knoten reduziert werden, woraus sich entsprechender Zielcode ableiten läßt. Aufgrund des Datenflusses von den Blättern des Eingabebaumes in Richtung Wurzel kommen nur bottom–up–Reduktionen in Frage.

Wir formalisieren dies wie folgt:

Definition 4.6 (**Baumgrammatik**) Eine (reguläre) Baumgrammatik $G = (N, \Sigma, \mathcal{R})$ besteht aus einer endlichen Menge N von Nichtterminalen (mit Stelligkeit 0), einem Alphabet Σ mit Stelligkeit von Terminalen mit $\Sigma \cap N = \emptyset$, und einer endlichen Menge $\mathcal{R}$ von Reduktionsregeln der Form $r \to X$, wobei $X \in N$ ein Nichtterminal und r ein $(\Sigma \cup N)$–Baum (input–pattern) ist. $\square$

Die Nichtterminale entsprechen somit gerade den Variablensymbolen (Registertypen).

Definition 4.7 (**Ableitung, Reduktion**) Seien t_1, t_2 zwei $(\Sigma \cup N)$–Bäume. t_1 leitet t_2 direkt ab bzw. t_2 kann zu t_1 reduziert werden ($t_1 \Rightarrow t_2$), falls es eine Reduktionsregel $R \in \mathcal{R}$ gibt, $R = \ r \to X$, sodaß t_2 durch Ersetzen eines mit X markierten Blattes durch r aus t_1 hervorgeht. Eine *Ableitung* eines $(\Sigma \cup N)$–Baumes t ist eine Folge $t_0, t_1, ..., t_n$ mit $t_0 = X$ für ein $X \in N$, $t_n = t$

und $t_i \Rightarrow t_{i+1}$ für $0 \leq i < n$. Eine *Reduktion* eines $(\Sigma \cup N)$–Baumes t ist eine Folge $t_n, t_{n-1}, ..., t_0$ mit $t_0 = X$ für ein $X \in N$, $t_n = t$ und $t_i \Rightarrow t_{i+1}$ für $0 \leq i < n$. □

Bis auf die Vertauschung der Seiten bei den Produktionen/Reduktionen sind die Baumgrammatiken den kontextfreien Grammatiken ähnlich. Die verschiedenen möglichen Reduktionen entsprechen den verschiedenen möglichen Ableitungen des Ausdrucksbaumes aus den Nichtterminalen der Baumgrammatik. Daher wird das Reduktionsproblem auch als *Treeparsing* bezeichnet.

Im Gegensatz zu „reinen" attributierten Grammatiken (vgl. [ASU86]), die Attributberechnungen nur lokal zu einer Grammatikregel erlauben, wird es mit Baumgrammatiken möglich, größere Baumstücke fester Größe auf einmal zu betrachten.

Codeerzeugung heißt ja, daß für jeden Ausdruck in der Zwischendarstellung eine Folge von Maschineninstruktionen gefunden werden muß, die den Ausdruck berechnet. In anderen Worten, es muß eine vollständige Überdeckung des Ausdrucksbaumes durch die input–pattern gefunden werden, womit das Codeerzeugungsproblem auf das Treepatternmatching–Problem zurückgeführt[3] wird.

Im allgemeinen gibt es mehrere Möglichkeiten, vollständige Überdeckungen zu finden. Natürlich ist man an einer Lösung interessiert, die sich durch minimale Kosten auszeichnet und damit optimalen Code repräsentiert.

Treepatternmatching wurde zur Codeerzeugung mit Baumgrammatiken erstmals im Twig–System [AG85, AGT89] eingesetzt. Dabei wurde das top–down–Verfahren von [HO82] angewandt und eine Überdeckung minimaler Kosten durch eine vereinfachte *dynamic programming*–Technik nach [AJ76] bestimmt.

[WW88] beschreibt ebenfalls die Anwendung von Standard–Treepatternmatchern nach [Kro75] und [HO82] (bottom–up und top–down) auf das Reduktionsproblem. Für das bottom–up–Verfahren nach [HO82] werden die Reduktionsregeln in die Definition der Match–Sets umgesetzt, indem jedesmal, wenn ein volles Treepattern (input pattern) in einem Match–Set vorkommt, alle Vorkommen des entsprechenden output patterns als Unterpattern von input–pattern anderer Reduktionsregeln auch zu diesem Match–Set hinzugefügt werden. [WW88] bemerkt, daß bei sogenannten Kettenregeln[4] eine Endlosreduktion verhindert werden kann durch eine geeignete Erweiterung der Definition der Match–Sets. Wie in [AG85, AGT89] werden die Reduktionen auf dem Eingabebaum nicht tatsächlich ausgeführt, sondern nur simuliert, sodaß alle möglichen Überdeckungen in einem einzigen Analysepaß berechnet werden können.

[PG88] benutzt bottom–up Treepatternmatching, verlagert die Kostenberechnung jedoch in die Generierungsphase des Treepatternmatchers, d.h. die Kostenmanipulation wird direkt in die Automatentabellen codiert. Da in [PG88] die output–pattern auch nichttriviale Bäume sein dürfen, gehört dieser Ansatz, auch *bottom–up rewrite system (BURS)* genannt, eigentlich zu den Baumtransformationssystemen (siehe nächster Abschnitt).

Dagegen empfiehlt [Gie92] in Verbindung mit bottom–up Treepatternmatching die explizite und effiziente Darstellung aller möglichen Überdeckungen und erst danach die Auswahl einer optimalen Lösung.

Burg [FHP92] ist ein Generator für vereinfachte bottom–up rewrite–Systeme nach [PG88]. Burg ist weniger flexibel als Twig, aber „viel schneller" [FHP92].

4.2.1 Treepatternmatching und endliche Baumautomaten

Endliche Baumautomaten sind eine Verallgemeinerung endlicher Automaten [TW68]. Während endliche Automaten auf Zeichenketten arbeiten, arbeiten endliche Baumautomaten auf Σ–Termen.

[3]Alternativen zu Treeparsing durch Treepatternmatching gibt es, z.B. den Ansatz von Graham/Glanville [GG78], der die Baumgrammatik als kontextfreie Grammatik uminterpretiert (die Reduktionsregeln werden zu Produktionen), dazu einen LR–Parser konstruiert und diesen auf den durch Unparsing in eine Zeichenkette konvertierten Syntaxbaum in Präfixdarstellung ansetzt. Mehrdeutigkeiten werden durch Backtracking behandelt. Das Unparsen und Neuparsen ist eine umständliche Lösung.

[4]Eine *Kettenregel* ist eine Reduktionsregel, deren input–pattern aus einem einzelnen Nichtterminal besteht.

[TW68] zeigt, daß die Klasse der von endlichen Baumautomaten akzeptierten (d.h., erkannten) Sprachen äquivalent zu der der regulären Sprachen ist, falls die Konkatenation bei regulären Sprachen durch die Einsetzung von Unterbäumen an Variablensymbolen ersetzt wird.

Definition 4.8 (Endlicher Baumautomat) [FSW92] Ein endlicher Baumautomat besteht aus einer endlichen Zustandsmenge Q, einem Baumalphabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup ...$, wobei Σ_j alle Symbole mit Stelligkeit j enthält, ferner einer Menge von Übergangsfunktionen $\delta \subseteq \cup_{j \geq 0} \left(Q \times \Sigma_j \times Q^j\right)$ und einer Menge $Q_F \subseteq Q$ von akzeptierenden Endzuständen.

Der Baumautomat heißt *nichtdeterministisch*, falls es mehr als eine mögliche Rechnung für einen Σ–Term gibt, d.h. falls es in mindestens einem Zustand q mehr als einen möglichen Übergang gibt; ansonsten heißt er *deterministisch*. Der Baumautomat heißt *top–down*, falls die Zustandsübergänge als Mengen partieller Funktionen $\delta_j : (Q \times \Sigma_j) \rightarrow Q^j$ für alle j interpretiert werden, und *bottom–up*, falls sie als $\delta_j : (Q^j \times \Sigma_j) \rightarrow Q$ für alle j interpretiert werden. □

Ein Zustandsübergang $(q, a, q_1, ..., q_j)$ in einem Baumautomat entspricht also je nach Interpretation der Ersetzung eines Variablensymbols durch einen Unterbaum mit Wurzelsymbol a (top–down) oder der Reduktion eines Unterbaumes mit Wurzelsymbol a auf ein Variablensymbol (bottom–up).

Mit Hilfe der Teilmengenkonstruktion aus [TW68] bzw. [FSW92] zeigt man die Äquivalenz deterministischer bottom–up–, nichtdeterministischer bottom–up– und nichtdeterministischer top–down–Baumautomaten. Deterministische top–down–Baumautomaten erweisen sich als schwächer[5].

[FSW92] stellt die Verbindung zwischen den oben beschriebenen Treepatternmatchingtechniken und den Berechnungen endlicher Baumautomaten her. Aus allgemeinen Methoden zur Implementierung deterministischer Baumautomaten durch Bildung von Teilmengen von Zuständen nichtdeterministischer, direkt aus den Treepattern abgeleiteter Baumautomaten ergeben sich auch die Analoga zu den Algorithmen aus [Kro75], [HO82] und [Cha87] als Spezialfälle.

4.3 Baumtransformationssysteme

Baumtransformationssysteme (auch Termersetzungssysteme genannt) sind allgemeiner als Codeerzeugungssysteme: sie können ebenfalls zur Codeerzeugung, aber auch für Programmtransformationen auf der Ebene der Zwischendarstellung eingesetzt werden.

Die auf Baumgrammatiken beruhenden Codeerzeugungssysteme verändern den Eingabebaum nicht bzw. reduzieren ihn sukzessive bottom–up auf ein einzelnes Nichtterminal. Im Gegensatz dazu können bei Baumtransformationssystemen als output–patterns auch nichttriviale Bäume auftreten; der Eingabebaum kann durch Anwendung einer derartigen Transformationsregel lokal verändert werden. Als Konsequenz verändern sich dabei auch die berechneten Match–Sets und, falls betrachtet, auch gewisse Attribute des Eingabebaums. Auch die Reihenfolge der Anwendung der einzelnen Transformationsregeln ist nun im allgemeinen nicht mehr eindeutig bestimmt, sodaß sie gesondert spezifiziert werden muß. Eine Angabe von Kosten für Transformationsregeln, ähnlich wie bei den meisten Baumgrammatiken für Codeerzeugersysteme, ist nicht vorgesehen.

Neben [PG88] wären hier insbesondere die Systeme Optran, Txl, Puma und TraFoLa zu nennen.

Optran [Wil79, LMW88] kann zu einer gegebenen Attributgrammatik und einer gegebenen Menge von Transformationsregeln ein Transformationsprogramm generieren, das einen auf der Attributgrammatik basierenden Eingabebaum entsprechend der Transformationsregeln modifiziert. Eine Transformationsregel darf immer dann ausgeführt werden, falls sowohl ihre syntaktische

[5] Dies kann man an folgendem Beispiel einsehen: Man kann leicht einen nichtdeterministischen top–down–Baumautomaten konstruieren, der die Sprache $L = \{a(b,b),\ a(c,c)\}$ akzeptiert. Es gibt jedoch keinen deterministischen top–down–Baumautomaten, der L akzeptiert, denn der würde zwangsläufig auch $a(b,c) \notin L$ und $a(c,b) \notin L$ akzeptieren.

(input–pattern) als auch semantische (ein Boolesches Prädikat auf Attributwerten) Vorbedingung erfüllt ist. Konflikte können auftreten, falls mehrere Transformationsregeln gleichzeitig ausgeführt werden können. Für diese Fälle können vom Benutzer Vorrangregelungen (z.B. bottom–up, top–down, left-to-right, right-to-left) spezifiziert werden. Sich durch die Strukturänderung des transformierten Baums ändernde Attribute müssen neu ausgewertet werden; hierbei kann OPTRAN durch statische Analysen feststellen, wo solche Änderungen anfallen. Semantische Prädikate müssen als PASCAL–Funktionen, Attribute als PASCAL–Datentypen programmiert werden. Eine in der Praxis einsatzfähige Version von OPTRAN war zum Zeitpunkt der Planung unserer Implementierung nicht verfügbar.

TXL [CC92, CC93] arbeitet im Gegensatz zu OPTRAN nur auf syntaktischer Ebene, d.h. ohne Attribute, liefert aber Parser und Unparser gleich mit, sodaß im Prinzip auf Quelltextebene transformiert wird.

PUMA (*p*attern *ma*tching and *u*nification, [Gro92]) ist ein Transformationssystemgenerator, der ähnlich wie OPTRAN als Eingabe eine attributierte Grammatik zur Typ–Beschreibung des abstrakten Syntaxbaumes sowie eine Folge von Transformationsregeln erhält. Die Attribute dürfen beliebige Typen besitzen. Die Spezifikation der Transformationsregeln folgt dem imperativem Programmierstil, erweitert um einen Typ–Konstruktor für Bäume. Mehrere Transformationsregeln können zu Sequenzen zusammengefaßt werden; die Reihenfolge der anwendbaren Transformationsregeln wird explizit vom Anwender kontrolliert. Der zu matchende Knoten des Syntaxbaumes muß in jedem Schritt explizit vorgegeben werden. Nichtlineare Pattern sind erlaubt; Unifikation (von links nach rechts) ist erwartungsgemäß nur möglich zwischen einem Treepattern auf der einen und einem (Σ–) Baum auf der anderen Seite. Backtracking gibt es nicht. PUMA benutzt das naive Treepatternmatching–Verfahren und versucht, durch Optimierungen Laufzeit zu sparen. Nach einer Transformation werden sich potentiell ändernde Attribute nicht neu ausgewertet. PUMA generiert ein Transformationssystem in C oder Modula-2. Zum Zeitpunkt der Planung unserer Implementierungen befand sich PUMA noch in einem rudimentären Zustand.

TRAFOLA [HS93, Fer90] ist eine funktionale Sprache zur Spezifikation von Programmtransformationen. Neben statischem Typkonzept gibt es mächtige Konstrukte zur Beschreibung von Treepattern. Nichtlineare Treepattern sind erlaubt. Das Treepatternmatching wird durch ein bottom–up–Verfahren oder durch einen Backtracking–Algorithmus implementiert [Fer90]. Die Ineffizienz der Implementierung schränkt jedoch die praktische Benutzbarkeit von TRAFOLA erheblich ein [Fer94].

4.4 Grenzen syntaktischer Mustererkennung

Das Generieren von Transformationssystemen aus formalen Spezifikationen der Transformationsregeln erscheint, da es die Implementierung der Treepatternmatching–Algorithmen vor den Augen des Anwenders verbirgt, auf den ersten Blick sehr elegant und komfortabel. Es stellt sich also die Frage, inwieweit dies auch für die Erkennung semantischer Muster verwendet werden kann.

Allerdings haben alle diese Verfahren, die ja auf rein syntaktischer Ebene arbeiten, einen entscheidenden Nachteil: eine (nichttriviale) Berechnung (semantische Operation, semantisches Muster) kann viele syntaktisch verschiedene Codierungen (Treepattern, Inkarnationen) haben.

Die Probleme beginnen z.B. mit der Kommutativität und Assoziativität gewisser Operatoren wie Addition oder Multiplikation, deren syntaktischen Symbolen im abstrakten Syntaxbaum der Treepatternmatcher diese Eigenschaft ja nicht ansieht. In der Tat ist Treepatternmatching für nichtlineare Treepattern bei Berücksichtigung kommutativer und assoziativer Operatoren NP–vollständig [BKN85]. Nichtlineare Treepattern kommen in Eingabebäumen zu in imperativen Sprachen formulierten Programmen häufig vor (vgl. nächstes Kapitel und Anhang B).

Das Hauptanwendungsgebiet für Treepatternmatcher scheint in Ausdrucksbäumen zu liegen. Sobald nämlich imperative Konstrukte wie Anweisungsfolgen, Bedingungen, Schleifen und indizierte Feldvariablen hinzukommen, treten Datenflußprobleme auf, deren Behandlung sich in Treepatternmatcher nicht ohne weiteres einbetten läßt. Darüberhinaus haben wir in Abschnitt 3.2

gesehen, wie sehr insbesondere die Schleifentransformationen oder das Einführen temporärer Variablen das Erscheinungsbild des Programmcodes variieren können, ohne dessen Semantik zu verändern.

Da wir Muster als semantische Funktionen definieren, kann ein Muster auf sehr viele syntaktisch verschiedene Weisen (Inkarnationen) implementiert sein. Beispielsweise kann eine Matrix–Vektor–Multiplikation als Schleife über einem Skalarprodukt

```
for (i=1; i<=n; i++)
  for (j=1; j<=m; j++)
    x[i] = x[i] + A[i][j] * b[j];
```

oder als Schleife über einer Vektortriade

```
for (j=1; j<=m; j++)
  for (i=1; i<=n; i++)
    x[i] = x[i] + A[i][j] * b[j];
```

implementiert sein; diese semantischen Untermuster wiederum könnten auch mit abgerollten Schleifen programmiert sein; die äußere Schleife der Matrix–Vektor–Multiplikation könnte ebenfalls abgerollt sein, die Initialisierung des Lösungsvektors kann auf verschiedenen Schleifenebenen (oder überhaupt nicht) erfolgen, Skalarprodukt und Vektortriade könnten mit Hilfsvektoren umgeschrieben sein, usw. Jede Kombination dieser Variationsmöglichkeiten entspricht einer möglichen Inkarnation der Matrix–Vektor–Multiplikation. Für jede Inkarnation müßten wir ein eigenes Treepattern (Inkarnationsschablone) bereitstellen, was für die beabsichtigte Anwendung aus Effizienzgründen nicht in Frage kommt.

4.5 Unsere Lösung

Wir stellen im nächsten und übernächsten Kapitel eine eigene Lösung vor, die speziell auf unser Mustererkennungsproblem, insbesondere auf das Problem der semantischen Äquivalenz syntaktisch verschiedener Syntaxbäume, zugeschnitten ist.

Im Prinzip gehen wir folgendermaßen vor: Zunächst werden, analog zum syntaktischen Fall, die möglichen (aber nicht explizit konstruierten!) Inkarnationsschablonen der Muster in (semantische) Untermuster zerlegt. Jedes Untermuster wird als eigenständiges Muster in die Musterbibliothek aufgenommen.

Folglich kann jede Inkarnationsschablone auch als *flaches* Treepattern der Höhe 2 (Operator und Operanden), bei uns *Schablone* genannt, formuliert werden, wenn die Söhne als Instanzen der Untermuster (nicht Untertreepattern!) beschrieben werden. Damit gibt es, wie wir noch sehen werden, nur noch etwa eine Handvoll wirklich syntaktisch verschiedener solcher Schablonen für jedes Muster. Für die Matrix–Vektor–Multiplikation selbst genügen drei Schablonen (vgl. Anhang B).

Eine Schablone[6] (semantisches Äquivalent zum input–pattern) reduziert denn auch auf das (semantische) Muster (Musterinstanz als semantisches Äquivalent zum output–pattern), d.h. jedes erkannte syntaktische Konstrukt wird *sofort* auf seine Semantik reduziert und unter diesem Namen weiterverarbeitet, und nicht erst, nachdem (vielleicht) eine gesamte Inkarnationsschablone syntaktisch erkannt wurde.

Damit entspricht der „Automat“ der *semantischen* Hierarchie der Muster; er arbeitet auf der semantischen Ebene. Dadurch wird er sehr kompakt: Mit einer linear großen Anzahl von Schablonen kann er exponentiell viele Inkarnationsschablonen erkennen.

[6] Da die Gesamtmusteranzahl (siehe Kapitel 5) und die Gesamtschablonenanzahl (durchschnittlich drei pro Muster, siehe Anhang B) relativ klein ist, kann der „Automat“, bestehend aus Schablonen, einigen Transformationen und semantischer Musterhierarchie, von uns problemlos direkt als C–Programm niedergeschrieben werden (vgl. Abschnitt 6.11.2).

Unterbäume, die für gleiche Variablensymbole in nichtlinearen Treepattern stehen sollen, werden der Einfachheit halber durch explizite Tests verglichen. Wir lassen Knoten mit variabler Operandenzahl zu und ersetzen, wo sinnvoll, die „Baumautomaten“ durch mächtigere Konstrukte.

Anweisungen werden genauso wie Ausdrücke behandelt; Datenabhängigkeits- und Datenflußanalyse werden explizit eingefügt, um Unifikation über Ausdruck–Grenzen hinweg auch in Gegenwart von Feldern zu ermöglichen. Dabei wird ein erweitertes Matchen auf einem modifizierten Datenflußgraphen durchgeführt, was auch das Zusammenziehen delokalisierter Codeteile ermöglicht.

Nach jedem erfolgreichen Matchen werden normalisierende Transformationen wie Schleifenaufgliederung (loop distribution), Bedingungsaufgliederung (`IF` distribution) und Schleifenaufrollen (loop rerolling) durchgeführt.

Ferner berechnen wir auch keine Gesamtüberdeckung des Eingabebaumes mit (semantischen) Mustern (das wird schwerlich möglich sein), sondern suchen nach einer größtmöglichen partiellen bottom–up–Überdeckung. Aufgrund der speziellen Zielsetzung unseres semantischen Mustererkennungsproblems ist diese Überdeckung eindeutig[7], d.h. an jedem Knoten des Eingabebaumes kann höchstens eine Schablone matchen; eine Angabe von Kosten kann somit entfallen.

[7]Insbesondere ist die resultierende Baumgrammatik *monoton*, d.h. es kann (bis auf triviale Zyklen) jede Reduktion (Schablone) auf jedem Pfad zur Wurzel höchstens einmal angewendet werden, weil jedes (semantische) Muster nur an wenigen Knoten des Eingabebaum überhaupt zutreffen kann und die Semantik der Muster hierarchisch geordnet ist.

5 Muster in numerischen Programmen

"The regularity of scientific codes is particularly useful here,
since it affords the potential for such understanding
to be extrapolated from very small problem sizes." [SH91]

5.1 Muster, Inkarnationen, Instanzen, Implementierungen

Ein *Muster* im Sinne dieser Arbeit ist die Abstraktion eines Programmfragments. Ein Programm besteht bekanntlich aus Algorithmus (d.h. zu berechnender semantischer Funktion) und Datenstruktur. Daher müssen für dieselbe semantische Operation bei verschiedene Datenstrukturen verschiedene Muster verwendet werden.

Definition 5.1 (Muster, Instanz eines Musters) Ein *Muster* m ist formal die Definition einer externen Prozedur, deren Semantik (*Semantikfunktion* des Musters) f_m bekannt ist, deren Implementierung dem Betrachter jedoch verborgen bleibt.

Die formalen Parameter dieser Prozedur, vereinigt mit etwa darin vorkommenden lokalen Variablen, nennen wir *Slots*. Die Anzahl der Slots ist für die meisten Muster genau festgelegt, für einige gibt es aber nur eine Mindestanzahl. Die Slots sind getypt mit einem reichhaltigen Typsystem, das neben den Standardtypen (wie etwa `ival` oder `dvar`, vgl. Abschnitt 2.2) auch weitere syntaktische Bedingungen an die aktualen Parameter wie `isscalar()` oder `issimpleidx()` (vgl. Abschnitt 2.3) enthält.

So wie die Parameter einer Prozedur als Dateneingang (lesend) oder Datenausgang (schreibend) dienen können, gibt es für jeden Slot einen *Status* $s_j \in \{R, W, RW, I\}$, der angibt, ob der j–te Parameter in der Prozedur nur gelesen (R), nur geschrieben (W) oder gelesen und überschrieben (RW) wird. Slots, die (z.B. als lokale Variablen) nur von interner Bedeutung sind, erhalten den Status I.

Wir listen in Abschnitt 5.3 alle unsere Muster auf. Wir geben dabei jeweils alle Slots mit zugehörigem Status und Typ an und definieren die jeweilige Semantikfunktion mit einer prozeduralen Notation.

Eine *Instanz* eines Musters entspricht dann formal einem Aufruf dieser Prozedur. Wir stellen sie auch syntaktisch ebenso dar wie einen Prozeduraufruf, nämlich durch den Namen des Musters, gefolgt von einer Liste von aktualen Parametern, den *Slot–Einträgen*, die in Anzahl und Typ den formalen Parametern des jeweiligen Musters entsprechen müssen. Slot–Einträge dürfen, mit wenigen Ausnahmen, nur Konstanten, Variablen, Feldzugriffe, Vektoren, Matrizen, höherdimensionale Tensoren und Bereichsausdrücke sein. □

Die Semantikfunktion eines Musters kann üblicherweise auf viele verschiedene Arten (*Inkarnationen*)[1] sequentiell in C, `FORTRAN` bzw. unserer vereinfachten Quellsprache LATINUS programmiert werden. Dies beruht zum einen auf der Möglichkeit des Programmierers, für das Muster verschiedene Algorithmen zu wählen, und zum anderen, gewisse Transformationen darauf auszuführen (z.B. Blocken von Schleifen, Vertauschen von Schleifen, temporäre Variablen), die das äußere Erscheinungsbild einer Inkarnation sehr verändern können.

[1] Eine formale Definition der Inkarnation eines Musters enthält Abschnitt 6.2.

Das Ziel der Mustererkennung wird es sein, Vorkommen von Mustern in Teil–Syntaxbäumen zu erkennen und sie durch die entsprechende Musterinstanz zu ersetzen. Erst die maschinenabhängige Code–Erzeugung wird dann entscheiden, welcher Algorithmus in der aktuellen Situation am geeignetsten ist, und welche Transformationen durchzuführen sind (*Implementierung* des Musters). Wir *abstrahieren* somit eine Inkarnation eines Musters auf eine Instanz dieses Musters und *konkretisieren* später die Instanz zu einer geeigneten Implementierung. Diese zweistufige Code–Erzeugung ist die wichtigste Schlüsselidee unseres Ansatzes.

Folglich bildet ein Programm, bei dessen sequentieller Codierung möglichst viele Musterinstanzen verwendet werden (seien diese nun vom Programmierer explizit so angegeben oder durch die Mustererkennung herausgefunden), eine maschinenunabhängige Formulierung, die gleichzeitig viele Optionen für automatische zielmaschinenspezifische Optimierungen bietet.

Wir haben mit dem Konglomerat aus unserer vereinfachten Quellsprache LATINUS und den darin nicht vorkommenden Musterinstanzen eine weitgehend maschinenunabhängige, dafür problemabhängige neue sequentielle Programmiersprache gebildet, und nennen sie PALATINUS (Pattern–Augmented LATINUS).

Der Rest dieses Kapitels beschreibt die Kriterien, nach welchen wir unsere Muster spezifiziert haben, gibt einen Überblick über alle bisher spezifizierten Muster und ihre Slot–Belegungen, zeigt auf, daß diese Muster für die Praxis relevant sind, und vergleicht diese Basis–Musterbibliothek mit gängigen mathematischen Software–Bibliotheken.

5.2 Prinzipien für das Muster–Design

Beim Aufstellen der Muster–Bibliothek und dem Design der Slot-Einträge für jedes Muster müssen verschiedene Gesichtspunkte berücksichtigt werden:

- Die Bibliothek muß übersichtlich bleiben. Eine zu große Anzahl von Mustern ist weder für einen PALATINUS–Programmierer wünschenswert, noch für das Parallelisierungssystem, das ja für jedes Muster Implementierungen und Laufzeittabellen bereithalten muß, wie wir in Kapitel 7 sehen werden.
- Die einzelnen Muster müssen möglichst orthogonal zueinander sein, d.h., weichen zwei semantische Funktionen nur in *unwesentlichen* Details voneinander ab, so sollten sie in einem Muster zusammengefaßt werden und jene Details über einen zusätzlichen Parameter unterschieden werden.

  So wurde zum Beispiel der ursprüngliche Plan (wie in [KP93] beschrieben), uninitialisierte und initialisierte Reduktionen ($\mathsf{VSUM}^{(1)}$, $\mathsf{VPROD}^{(1)}$, $\mathsf{SSP}^{(1)}$, $\mathsf{MV}^{(2)}$, $\mathsf{MM}^{(3)}$, $\mathsf{FOLR}^{(1)}$) jeweils als *verschiedene* Muster zu behandeln (d.h. $\mathsf{VSUMO}^{(1)}$, $\mathsf{VPRODO}^{(1)}$, $\mathsf{SSPO}^{(1)}$ usw. für die nichtinitialisierten Varianten), wieder aufgegeben. Stattdessen gibt es in all diesen Mustern einen Initialisierungsslot, der bei der uninitialisierten Variante einfach mit der reduzierenden Variable belegt wird.
- Soll ein Muster m in die Musterbibliothek aufgenommen werden, so müssen auch alle semantischen Untermuster von m, wie in Abschnitt 4.5 erwähnt, in die Musterbibliothek aufgenommen werden.
- Aus dem Musternamen und den Slots einer Musterinstanz muß ein semantisch korrektes Programm jederzeit (!) reproduzierbar sein, auch während des Mustererkennungsprozesses. Die Bedeutungen der einzelnen Slots müssen klar festgelegt sein. Damit wird es leicht, die Korrektheit der Mustererkennung zu beweisen.
- Eine Musterinstanz ist ein Basisbaustein für die Datenflußanalyse. Aus den Slot-Einträgen müssen alle nach rückwärts (*upwards exposed*) und vorwärts (*downwards exposed*) sichtbaren Variablen klar hervorgehen. Das “Innere” der Musterinstanz ist für die Datenflußanalyse eine “black box”. Der Datenfluß im Innern des Musters ist durch den Namen des Musters und ggf. durch einige Slot–Einträge eindeutig bestimmt.

- Ein Muster muß sich direkt in (mindestens) eine parallele Zielimplementierung übersetzen lassen. Es darf daher nicht zu viele Sonderfälle beinhalten. Trotzdem bleibt die genaue Implementierung eines Musters für den Programmierer verborgen — die Musterinstanz wird als "black box" behandelt.

- Je höherer Ordnung ein Muster ist, d.h. je umfangreichere und (laufzeitmäßig) teurere Operationen es beschreibt, desto größere Freiheiten bieten sich bei der Gestaltung der Implementierung an.

  Zum Beispiel ist es wesentlich günstiger, ein eigenes Muster $\mathsf{MM}^{(3)}$ für Matrix-Matrix-Multiplikation zur Verfügung zu haben, anstatt dieses immer mit einer Schleife über Matrix-Vektor-Multiplikation $\mathsf{MV}^{(2)}$ umschreiben zu müssen:

```
for (j=1; j<=n; j++)   MV(i,k, C[:][j], A[:][:], B[:][j], 0.0);
```

  Zum einen offeriert erstere Variante mehr Parallelität implizit, zum anderen gibt es für Matrix-Matrix-Multiplikation wesentlich mehr Spielraum für die Wahl einer guten Implementierung auf der Zielmaschine als für die zweite Variante, die auf die Implementierung der Matrix-Vektor-Multiplikation zurückgreifen muß.

  Generell sollte die Basis–Musterbibliothek die wichtigsten numerischen Operationen *mindestens* soweit unterstützen, daß alle indizierenden Schleifen gebunden werden können, d.h., daß möglichst alle Phasen erkannt werden. Da die gerade erwähnte zweite Variante der Matrix-Vektor-Multiplikation noch eine indizierende Schleife enthält, *muß* es also — dieser Regel zufolge — ein Muster für Matrix-Matrix-Multiplikation geben.

- Alle von den gängigen Parallelrechnerbauarten unterstützte Standardfunktionen, die von der Hardware oder dem Betriebssystem angeboten werden, sollten als Muster (bzw. als Spezialfall eines solchen) in der Musterbibliothek vorkommen. Beispiele sind $\mathsf{VSUM}^{(1)}$, $\mathsf{VPROD}^{(1)}$, $\mathsf{VROR}^{(1)}$. Gleiches gilt für die vordefinierten Standardfunktionen in `FORTRAN` bzw. LATINUS.

- Ein Muster ist ein Basisbaustein für Alignment und Datenaufteilung. Daher müssen für dieselbe semantische Operation bei verschiedenen Datenstrukturen auch verschiedene Muster angelegt werden.

- Ein Muster ist ein Basisbaustein für die Laufzeitvorhersage.

  So spielt es z.B. eine erhebliche Rolle, ob einer der Operandenvektoren einer Vektoraddition ($\mathsf{VADD}^{(1)}$) mit dem Resultatvektor identisch ist. In diesem Fall etwa könnte ein Vektorregister weniger benutzt werden, was deutliche Laufzeitgewinne verursachen kann (siehe [KPR92]). Daher wird diese *akkumulative* Vektoraddition als ein eigenes Muster $\mathsf{VAADD}^{(1)}$ behandelt.

- Die wichtigste Regel lautet aber: Ein Muster muß *signifikant* sein, d.h. es muß in den infrage kommenden Anwendungen hinreichend oft vorkommen. Es sollte möglich sein, viele Standardalgorithmen insbesondere aus der Linearen Algebra überwiegend unter Verwendung von Musterinstanzen zu codieren.

  Ferner halte man sich vor Augen: Der Autor des seqentiellen Programms war höchstwahrscheinlich kein Dummkopf, sondern hat sich bereits Gedanken gemacht um die effiziente Ausführung seines Programms auf sequentiellen oder Vektor-Architekturen. So wäre es beispielsweise unnütz, ein Muster $\mathsf{SVSUM}^{(1)}$ zu generieren[2], das der folgenden "Programmiersünde" entspricht:

```
s = 0.0;
for (i=1; i<=n; i++)
   s = s + c * v[i];
```

[2]Tatsächlich gibt es dieses Muster intern in der Mustererkennungsphase; allerdings ist es instabil, d.h. es ist für die Codeerzeugung nicht sichtbar, weil es unmittelbar nach der Mustererkennungsphase in seine Grundbestandteile $\mathsf{VSUM}^{(1)}$ und MUL zerlegt wird.

Ein vernünftiger Programmierer würde die Multiplikation mit dem schleifenunabhängigen Faktor c aus der Schleife herausziehen:

```
s = 0.0;
for (i=1; i<=n; i++)
   s = s + v[i];
s = c * s;
```

was als `VSUM(i,s,v,0.0); mult(s,c,s);` erkannt wird[3].

All diese Bedingungen müssen gleichzeitig erfüllt werden. Bisweilen ist die Entscheidung über das Design bestimmter Muster und ihre Aufnahme oder Nicht-Aufnahme in die Basis-Musterbibliothek Ermessenssache. Daher ist die im folgenden Abschnitt aufgeführte Basis–Musterbibliothek eher als eine von mehreren möglichen Lösungen anzusehen.

5.3 Basis–Muster–Bibliothek

Die Basis-Musterbibliothek beinhaltet diejenigen Muster, die wir auf jeden Fall für geeignet und bedeutsam halten. Weitere Muster, die hier nicht aufgeführt sind, können gleichwohl in späteren Zusatzmodulen eingefügt werden. Dieser modulare Aufbau empfiehlt sich vor allem bei speziellen Anwendungsgebieten, wie z.B. Operationen auf dünn besetzten Matrizen (vgl. Anhang D), trigonometrische Operationen und FFT oder seminumerische Algorithmen wie Sortieren und Suchen.

Wir geben im folgenden alle Muster der Basismusterbibliothek an. Sie sind geordnet nach aufsteigender Ordnungszahl *order*, die die Anzahl der Schleifen angibt, die zu einer naheliegenden sequentiellen Implementierung notwendig wären. Ferner sind die Muster gleicher Ordnungszahl in Untergruppen aufgegliedert, die etwa gleichen Zugriffsarten entsprechen.

Die Semantik eines Musters geben wir meist als ein kurzes Programmstück an, das die Semantik operationell beschreibt. Gleiche (formale) Variablen bedeuten dabei, daß hier auch gleiche Variablen stehen müssen, während für verschiedene formale Variablen durchaus gleiche Variablen eingesetzt werden können, falls dies nicht explizit anders angegeben ist. Addition und Multiplikation werden grundsätzlich als kommutativ und assoziativ angesehen (vgl. hierzu die Fußnote auf Seite 54).

`DO i ...` steht dabei für eine `for`– bzw. `do`–Schleife mit Schleifenvariable `i`. Sind die Schleifengrenzen bzw. die Schrittweite von Belang, so werden sie als Bereichsausdruck wie in `DO i=[l:u:s]` angegeben. Enthält eine solche Schleife eine weitere (unabhängige) Schleife und spielt die Reihenfolge der Schleifeniterationen keine Rolle, so wird diese Doppelschleife mit `DODO i,j` bezeichnet; analog steht `DODODO` für eine Dreifachschleife usw. Optionale Teile sind in einfache Anführungszeichen `‘’` eingeschlossen. Wo erforderlich, wird die Semantik komplexerer Muster durch begleitenden Text beschrieben.

5.3.1 Skalare Muster (Ordnung 0)

Standardausdrücke und Zuweisungen

```
SINIT ( x, c )
Sem.:  x = c;
Slot 0: W variable  x
Slot 1: R constant  c
```

[3]Das Muster SVSUM$^{(1)}$ wäre auch wegen anderer genannter Argumente nicht sinnvoll. Gegenüber der Vektoroperation VSUM$^{(1)}$ wäre bei hinreichend großer Vektorlänge die skalare Multiplikation eine unwesentliche Erweiterung.

```
SCOPY ( x, y )
Sem.:  x = y;
Slot 0: W variable  x
Slot 1: R variable  y


ADD, MUL, MOD, EQ, LT, GT, LE, GE, NE, AND, OR ( z, x, y )
Sem.:  'z =' x BinOp y
Slot 0: W variable  z          nur, falls Zuweisung erfolgt
Slot 1: R variable  x
Slot 2: R variable  y
Bem.: BinOp in {+, *, mod, ==,<,>,<=,>=,!=,&&,||}


AADD, AMUL, AMOD ( z, z, y )
Sem.:  z = z + y  bzw.  z = z * y bzw.  z = z mod y
Slot 0: W variable  z
Slot 1: R variable  z          eqex (Slot1, Slot0)
Slot 2: R variable  y


UnOp ( z, x )
Sem.:  'z =' UnOp(x)
Slot 0: W variable  z          nur, falls Zuweisung erfolgt
Slot 1: R variable  x
Bem.: UnOp in {sin, cos, tan, exp, log, sqrt, twoto (2^n)}


POWER ( x, a, b, init )
Sem.:  x = init * a^b
Slot 0: W variable  x          falls Zuweisung erfolgt
Slot 1: R variable  a
Slot 2: R integer   b
Slot 3: R expr      init


ADDMUL ( z, x, u, v )
Sem.:   {z =} x + u * v
Slot 0: W variable  z          falls Zuweisung erfolgt
Slot 1: R variable  x          not(eqex(z,x))
Slot 2: R variable  u
Slot 3: R variable  v


AADDMUL ( x, x, u, v )
Sem.:   x = x + u * v)
Slot 0: W variable  x
Slot 1: R variable  u
Slot 2: R variable  v
Slot 3: R variable  x


ADDMULTIMUL ( z, x, a, b, ... )
Sem.:   {z =} x + a * b * ...
Slot 0: W variable  z          falls Zuweisung erfolgt
Slot 1: R variable  x          not(eqex(z,x))
Slot 2: R variable  a
Slot 3: R variable  b
....
```

```
AADDMULTIMUL ( x, x, a, b, ... )
Sem.:   x = x + a * b * ...)
Slot 0: W variable  x
Slot 1: R variable  x
Slot 2: R variable  a
Slot 3: R variable  b
....


MULADDMUL ( z, x, y, u, v )
Sem.:   {z=} x * (y + u * v)
Slot 0: W variable  z           falls Zuweisung erfolgt
Slot 1: R variable  x
Slot 2: R variable  y
Slot 3: R variable  u
Slot 4: R variable  v


MULMUL ( z, b, c, d, e )
Sem.:  {z =} b * c + d * e
Slot 0: W variable  z           falls Zuweisung erfolgt
Slot 1: R variable  b
Slot 2: R variable  c
Slot 3: R variable  d
Slot 4: R variable  e
```

Sammelmuster: MULTIADD$^{(0)}$, MULTIMUL$^{(0)}$, STAR$^{(0)}$

MULTIMUL$^{(0)}$ bezeichnet ein Produkt von mehr als zwei (ggf. invertierten) skalaren Variablen oder Konstanten.

MULTIADD$^{(0)}$ bezeichnet die Addition oder Subtraktion von mehr als zwei (ggf. negierten) skalaren Variablen, Konstanten oder Produkten von Variablen oder Konstanten.

Wird das Ergebnis einer (skalaren) Variablen zugewiesen, so wird diese Variable mit Status `W` in Slot 0 geschrieben; ansonsten bleibt Slot 0 leer (_). Die übrigen Operanden folgen in den Slots 1, 2, ... Eventuell vorhanden konstante Teilausdrücke werden ausgerechnet und in Slot 1 gespeichert. Variablen–Operanden werden mit Status `R` aufgenommen. Die Operanden einer MULTIADD$^{(0)}$–Instanz, die Produkte sind, werden als Instanzen mit dem Status `R` in ihre Slots aufgenommen, sodaß alle Operanden mit ihrer Statusangabe verfügbar sind.

Beispiel 5.1 Der Ausdruck $-\frac{4a}{2bc}$ wird durch die Instanz `MULTIMUL(_,-2,a,1/b,1/c)` dargestellt.

`MULTIADD(_,12,a,mult(b,c),-1/d)` steht für den Ausdruck $a + bc - \frac{1}{d} + 12$. ◇

Die Reihenfolge der nichtkonstanten Operanden von MULTIADD$^{(0)}$ oder MULTIMUL$^{(0)}$ ist beliebig[4].

Differenzensterne Differenzensterne (*stencils*) begegnen uns z.B. als LAPLACE–Operator bei der Diskretisierung partieller Differentialgleichungen in Gitterrelaxationen (Glättungsoperationen) wie Jacobi–, Gauß–Seidel–, JOR– und SOR–Relaxation, aber auch bei Gitterinterpolationen und –restriktionen in Mehrgitterverfahren.

Beispiel 5.2 (Jacobi–Relaxation)

[4]Dies stellt in Sprachen wie `FORTRAN` manchmal ein Problem dar, weil dort die Fließkomma–Addition bzw. –Multiplikation im allgemeinen nicht kommutativ, ja nicht einmal assoziativ ist. Wir bemerken hierzu allerdings, daß (1) dies nur der leichteren Mustererkennung dient und die ursprüngliche Operandenfolge bei der Code–Erzeugung bei Bedarf wiederhergestellt werden kann, und (2) dieses Problem auch bei vielen Parallelisierungstransformationen ignoriert wird, wie beispielsweise beim baumartigen Aufsummieren der VSUM$^{(1)}$–Operanden.

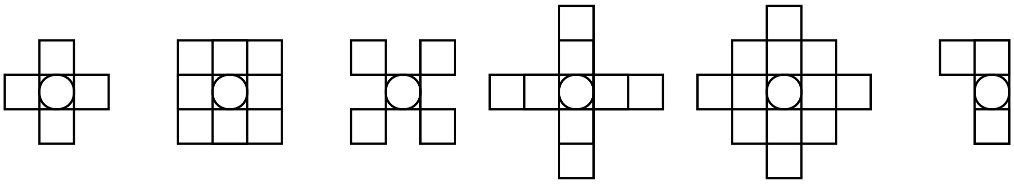


Abbildung 5.1 Beispiele für zweidimensionale Differenzensterne. Der erste entspricht dem aus Beispiel 5.2, wo die vier nächsten Gitternachbarn des zu berechnenden Gitterpunktes referenziert werden. Differenzensterne können auch unregelmäßig sein, wie der letzte.

```
   do 10 j=2,n-1
    do 10 i=2,n-1
     uhelp(i,j) = (1-omega)*u(i,j) + omega*0.25*( f(i,j)
  *        + u(i-1,j) + u(i+1,j) + u(i,j+1) + u(i,j-1) )
10 continue
```

◇

Einige Beispiele für Differenzensterne sind in Fig. 5.1 zu sehen.

$\mathsf{HSTAR}^{(0)}$ versammelt die Operanden eines eindimensionalen, $\mathsf{STAR}^{(0)}$ die eines zweidimensionalen Differenzensterns. Die wichtigsten Stern–Positionen für ein– und zweidimensionale Differenzensterne numerieren wir in lexikalischer Reihenfolge von `[i-1][j-1]` (= Position 1) bis `[i+1][j+1]` (= Position 9) wie folgt durch:

```
[i-1]       [i]        [i+1]          ---> HSTAR Pos. 1  2  3

[i-1][j-1]  [i-1][j]   [i-1][j+1]                     1  2  3
[i][j-1]    [i][j]     [i][j+1]       ---> STAR  Pos. 4  5  6
[i+1][j-1]  [i+1][j]   [i+1][j+1]                     7  8  9
```

Eine Erweiterung dieses Schemas auf größere Differenzensterne (wie z.B. den *diamond*, vorletztes Beispiel in Bild 5.1) ist kein Problem.

Die `i`'s und `j`'s in diesen Indexmustern können auch mit einem ganzzahligen konstanten Vorfaktor (Restriktionsfaktor) behaftet sein. In Beispiel 5.2 ist dieser Faktor 1, aber bei Gitterrestriktionen oder Gitterinterpolationen in Mehrgitteralgorithmen tritt typischerweise ein Skalierungsfaktor von 2 auf der linken oder rechten Seite auf.

Die Slots von $\mathsf{STAR}^{(0)}$ und $\mathsf{HSTAR}^{(0)}$ sind wie folgt belegt:

```
Slot 0: W vec/mat lhs     Linke Seite der Zuweisung, falls existent
Slot 3: R scalar  Cges    gemeinsamer Vorfaktor fuer alle C[.]
Slot 7: R vec/mat starvar das Feld, das den Stern aufspannt
Slot 8: F vec/mat F       weiterer in [i][j] indizierter Feldzugriff,
                          nicht identisch mit starvar
Slot 9: R scalar  CF      konstanter Koeffizient zu F, falls existent
Slot10: R leaf    C[1]    direkter Koeffizient bei Pos. 1
                           oder leer, falls nicht referenziert
    ..  . ....    ....    .....
Slot18: R leaf    C[9]    direkter Koeffizient bei Pos. 9 ...
Slot19: R ivar    i       erste den Stern aufspannende Indexvar. i
Slot20: I ival    idimnr  Dimensionsnummer von i in starvar
Slot21: I ival    Ri      Restriktionsfaktor der i-Indizes von starvar
Slot22: R ivar    j       zweite den Stern aufspannende Indexvar. j
Slot23: I ival    jdimnr  Dimensionsnummer von j in starvar
Slot24: I ival    Rj      Restriktionsfaktor der j-Indizes von starvar
```

Die Slots 22, 23 und 24 sind bei $\mathsf{HSTAR}^{(0)}$ unbesetzt.

Givens–Rotation

```
ROT ( a, b, t, c, s )
Sem.: t = c*a + s*b;
      b = c*b + s*a;
      a = t;
Slot 0: W dvar      a
Slot 1: W dvar      b
Slot 2: W dvar      t
Slot 3: R dvar      c
Slot 4: R dvar      s
```

Dreieckstausch

```
SWAP ( x, y, t )
Sem.:   t = x; x = y; y = t;
Slot 0: RW variable  x
Slot 1: RW variable  y
Slot 2: W  variable  t
```

Bedingte Zuweisungen

```
MAXVAL ( x, a, b )
Sem.:  x = max(a,b)
Slot 0: W variable  x         MaxVal
Slot 1: R variable  a
Slot 2: R variable  b
```

`MINVAL` analog

```
MAXLOC ( p, C, a, b )  // Index des Feldelem. mit groesserem Eintrag
Sem.:  if (‘abs’C[a]>=‘abs’C[b]) p = a; else p = b;
Slot 0: W ivar      p         MaxLoc
Slot 1: R arref     C
Slot 2: R ivar      a
Slot 3: R ivar      b
```

`MINLOC` analog

```
MAXLOCM ( p, q, C, a, b )  // Indizes des Feldel. mit groesserem Eintrag
Sem.:  if (‘abs’C[a][b]>=‘abs’C[c][d]) { p=a; q=b; } else { p=c; q=d; }
Slot 0: W ivar      p         MaxLocRow
Slot 1: W ivar      q         MaxLocCol
Slot 2: R arref     C
Slot 3: R ivar      a         precedes( C[a][b], a, b )
Slot 4: R ivar      b
Slot 5: R ivar      c         precedes( C[c][d], c, d )
Slot 6: R ivar      d
```

`MINLOCM` analog

```
MAXVL ( p, m, C, a )
Sem.:  if (m<=C[a]) { p=a; m=‘abs’(C[a]) }
Slot 0: W  ivar      p         MaxLoc
Slot 1: RW ivar      m         MaxVal
Slot 2: R  arref     C
Slot 3: R  ivar      a
```

`MINVL` analog

```
MAXVLM ( p, q, m, C, a, b )
Sem.:  if (m<=C[a][b]) { p=a; q=b; m=‘abs’(C[a][b]) }
Slot 0: W  ivar      p         MaxLocRow
Slot 1: W  ivar      q         MaxLocCol
Slot 2: RW ivar      m         MaxVal
Slot 3: R  arref     C
Slot 4: R  ivar      a         precedes( C[a][b], a, b )
Slot 5: R  ivar      b
```

`MINVLM` analog

Das folgende Muster CONDASS$^{(0)}$ ist instabil, d.h. nur von interner Bedeutung, wird aber hier wegen der Slotbelegung aufgeführt:

```
CONDASS ( y, cond, E1, E2, Slots 1,... von E1, ‘Slots 1,... von E2’)
Sem.:  if (cond) E1(y,...); ‘else E2(y,...);’
Slot 0: W variable  y           gleiche linke Seite von E1 und E2
Slot 1: R expr      cond        Bedingung
Slot 2: R expr      E1          erkannte Zuweisung an y, then-Teil
Slot 3: R expr      E2          erkannte Zuweisung an y, else-Teil
                                 sofern vorhanden
Slot 4,...                      Slots 1,... von E1 und E2
```

5.3.2 Muster der Ordnung 1

Skalare Iterationen

```
FSUM ( i, res, init, fn, v1, v2, ... )
Sem.: ‘res = init;’  DO i  res = res + fn( i, v1, v2, ... )
Slot 0: I range      i           Schleifenvariable, Schleifengrenzen
Slot 1: W dvar       res         notoccurin(res, i)
Slot 2: R dvar/dval  init        Init-Wert oder res
Slot 3: I expr       fn          Term von Skalaren in i
Slot 4: R dvar       v1          notoccuridx(v1,i), erste Var. in fn
Slot 5: R dvar       v2          notoccuridx(v2,i), zweite Var. in fn
.... .  . ....       ..          ...
```

Quasiskalare Vektoroperationen

```
VCOPY ( i, x, y )
Sem.:  DO i  x[i] = y[i]
Slot 0: I rng       i           Schleifenvariable, Schleifengrenzen
Slot 1: W vector    x           issimpleidx(x, i)
Slot 2: R vector    y           issimpleidx(y, i)
```

```
VASSIGN ( i, x, y )
Sem.:  DO i  x[i] = y[i]
Slot 0: I rng       i
Slot 1: W vector    x           issimpleidx(x, i)
Slot 2: R scalarvar v           notoccurin(v, i), isvar(v)
```

VASSIGNSP$^{(1)}$ initialisiert alle Elemente eines Vektors `x` mit dem Wert eines skalaren Ausdrucks `fn`. VASSIGNSP$^{(1)}$ ist instabil; eine VASSIGNSP$^{(1)}$–Instanz kann als Sequenz aus einer `fn`–Instanz und einer VASSIGN$^{(1)}$–Instanz ausgedrückt werden.

```
VASSIGNSP ( i, x, fn, v1, v2, ... )
Sem.:  DO i  x[i] = fn
Slot 0: I rng       i
Slot 1: W vector    x           issimpleidx(x, i)
Slot 2: I expr      fn          notoccuridx(fn,i), not(isleaf(fn))
Slot 3: R scalarvar v1          erste Variable in fn
...                 ..          ........
```

Bemerkung: `s` darf von `i` abhängen, darf aber keine Feldreferenz `A[...i...]` enthalten, in der ein `i` enthaltender Indexausdruck vorkommt.

```
VINIT ( i, x, c )
Sem.:   DO i  x[i] = c
Slot 0: I rng       i
Slot 1: W vector    x           issimpleidx(x, i)
Slot 2: R ival/dval c           isconstant(s, i)
```

```
VINITSP ( i, x, s )
Sem.:   DO i  x[i] = s
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: I expr      s         notoccuridx(s,i), not(isconst(s))
```

In `s` darf nur `i` als Variable vorkommen, aber nicht in einem Indexausdruck einer Feldreferenz.

```
VADD, VAADD, VMUL, VAMUL ( i, x, y, z )
Sem.:  DO i  x[i] = y[i] +-*/ z[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
Slot 3: R vector    z         issimpleidx(z,i)
```

Bei VAADD$^{(1)}$ bzw. VAMUL$^{(1)}$ müssen die Einträge der Slots 1 und 2 gleich sein.

Analog zu VADD$^{(1)}$ sind VAND$^{(1)}$ (elementweises AND$^{(0)}$) und VOR$^{(1)}$ (elementweises OR$^{(0)}$) definiert.

```
VSTDFN ( i, x, y )            (VSIN, VCOS, ...)
Sem.:  DO i  x[i] = stdfn( y[i] )
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
```

Standardfunktion `stdfn` $\in$ {`sin, cos, tan, exp, log, max, min`}

```
VMOD ( i, x, y, m )
Sem.:  DO i  x[i] = y[i] % m
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
Slot 3: R scalar    m         notoccuridx(m,i)


VAMOD ( i, x, x, m )
Sem.:  DO i  x[i] = x[i] % m
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    x         eqex(x auf lhs, x auf rhs)
Slot 3: R scalar    m         notoccuridx(m,i)


VADDMUL ( i, x, a, b, c )
Sem.:  DO i  x[i] = a[i] +- b[i] */ c[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    a         issimpleidx(a,i)
Slot 3: R vector    b         issimpleidx(b,i)
Slot 4: R vector    c         issimpleidx(c,i)


VINC ( i, x, y, c )
Sem.:  DO i  x[i] = y[i] +- c
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
Slot 3: R constant  +-c       notoccurin(c,i)


VAINC ( i, x, x, c )
Sem.:  DO i  x[i] = x[i] +- c
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    x         eqfex(x auf lhs, x auf rhs)
Slot 3: R constant  +-c       notoccurin(c,i)


VINV ( i, x, y, valifyis0 )
Sem.:  DO i  x[i] = 1.0 / y[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
Slot 3: R scalar    valifyis0 zugewiesener Wert f. x[i], wo y[i]==0


SV ( i, x, y, c )
Sem.:  DO i  x[i] = c * y[i]    (bzw.  c / y[i])
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    y         issimpleidx(y,i)
```

```
VAADDSS ( i, x, x, u, v )
Sem.:  DO i  x[i] = x[i] +- u * v
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    x         eqfex(x auf lhs, x auf rhs)
Slot 2: R scalar    u         notoccurin(u,i)
Slot 3: R scalar    v         notoccurin(v,i)


VAADDSV ( i, x, s, v, x )
Sem.:  DO i  x[i] = x[i] +- s * v[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R ivar/dvar s         notoccurin(s,i)
Slot 3: R vector    v         issimpleidx(v,i)
Slot 4: R vector    x         eqfex(x auf lhs, x auf rhs)


VADDSV ( i, x, s, v, y )
Sem.:  DO i  x[i] = y[i] +- s * v[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R ivar/dvar s         notoccurin(s,i)
Slot 3: R vector    v         issimpleidx(v,i)
Slot 4: R vector    y         not(eqsym(y,x))


VAADDMULTISV ( i, x, s, v, a, b, ... )
Sem.:  DO i  x[i] = x[i] +- v[i] * a * b * ...
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    x         eqfex(x auf lhs, x auf rhs)
Slot 3: R scalar    a         notoccurin(a,i)
Slot 4: R scalar    b         notoccurin(b,i)
...


VMULMUL ( i, x, a, b, c, d )
Sem.:  DO i  x[i] = a[i] * b[i] + c[i] * d[i]
Slot 0: I rng       i
Slot 1: W vector    x         issimpleidx(x,i)
Slot 2: R vector    a         issimpleidx(a,i)
Slot 3: R vector    b         issimpleidx(b,i)
Slot 4: R vector    c         issimpleidx(c,i)
Slot 5: R vector    d         issimpleidx(d,i)
```

GVOP$^{(1)}$ bezeichnet eine allgemeine Vektoroperation, auf die keiner der bisher beschriebenen Spezialfälle zutrifft.

```
GVOP ( i, x, fn, y1, y2, ... )
Sem.:  DO i  x[i] = fn( y1, y2, ... )
Slot 0: I rng       i
Slot 1: W vector    x       issimpleidx(x, i)
Slot 2: I expr      fn      issimpleidx(fn,i) (fn quasiskalar in i)
Slot 3: R vector    y1      if (occurin(y1,i))  issimpleidx(y1,i)
Slot 4: R vector    y2      if (occurin(y2,i))  issimpleidx(y2,i)
...                 ..      ...
```

Einige der Operanden `y1, y2,...` können Skalare sein.

GVOP$^{(1)}$ kann stets in elementare quasiskalare Vektoroperationen zerlegt werden. Sie ist daher instabil, d.h. nur als technisches Hilfsmittel bei der Mustererkennung konzipiert.

LR–Zerlegung (Ordnung 1)

```
SVDIAG ( i, x, y, c )
Sem.:  DO i  x[i] = c * y[i][i]    (bzw. c / y[i][i])
Slot 0: I rng        i
Slot 1: W vector     x        issimpleidx(x,i)
Slot 2: R matrix     y        isdoubleidx(y,i)
Slot 3: R scalar     c        notoccuridx(c,i)
```

```
VLUD (j,A[c][j],Q[c][k],A[c][j],A[c][k],A[k][k],Q[c][k],A[k][j],k,c)
Sem.:  Q[c][k] = A[c][k] */ A[k][k];
       DO j [k+1:n]  A[c][j] = A[c][j] +- Q[c][k] * A[k][j];
Slot 0: I rng    j
Slot 1: W vector A[c][j]     (lhs) issimpleidx(A,j)
Slot 2: W arref  Q[c][k]     (lhs)
Slot 3: R vector A[c][k:m]   (rhs)
Slot 4: R vector A[c][k]     (rhs)
Slot 5: R scalar ‘1/’A[k][k] notoccurin(A[k][k],c),
                             notoccurin(A[k][k],j)
Slot 6: I vector Q[c][k]     (rhs)
Slot 7: R vector A[k][j]     notoccurin(Akj,c), issimpleidx(Akj,j)
Slot 8: R scalar k
Slot 9: R scalar c
```

Givens–Rotation

```
VROT( i, a, b, t, c, s )
Sem.: DO i  { t=a[i]*c+b[i]*s;  b[i]=b[i]*c+a[i]*s; a[i]=t; }
Slot 0: I  rng           i
Slot 1: RW vector        a       issimpleidx(a,i)
Slot 2: RW vector        b       issimpleidx(b,i)
Slot 3: W  dvar/vector   t       if (occurin(t,i))  issimpleidx(t,i)
Slot 4: R  dvar          c
Slot 5: R  dvar          s
```

Vektoren–Dreieckstausch

```
VSWAP ( i, x, y, t )
Sem.:  DO i  SWAP(x[i],y[i],t{[i]})
Slot 0: I  rng       i
Slot 1: RW vector    x        issimpleidx(x,i)
Slot 2: RW vector    y        issimpleidx(y,i)
Slot 3: W  dvar/vec  t        if (occurin(t,i))  issimpleidx(z,i)
```

Vektor–Shift

```
VSHIFT ( i, a, a, k )
Sem.:  DO i  a[i] = a[i+-k] bzw. a[i-+k] = a[i]
Slot 0: I  rng       i        Schleifenvariable, Shift--Bereich
Slot 1: RW vector    a[i]     (lhs) issimpleidx(a,i)
Slot 2: RW vector    a[i+-k]  isoffsetidx(a,i,+-k)
Slot 3: R  scalar    k        Shift--Distanz
```

Maskierte Vektoranweisungen

Das Muster VCONDASS(1) ist instabil, d.h. nur von interner Bedeutung, wird aber hier wegen der Slotbelegung aufgeführt. Es steht für elementweise bedingte Vektoroperationen.

```
VCONDASS (i,y,cond,M1,M2, Slots von M1, Slots von M2, Slots von cond)
Sem.:  DO i  if (cond) M1(y,...); ‘else M2(y,...);’
Slot 0: I  rng      i
Slot 1: W scal/vect y    gleiche linke Seite von M1 und M2
Slot 2: R expr      cond erkannte Bedingung, issimpleidx(cond,i)
Slot 3: I expr      M1   erkannter then-Teil
Slot 4: I expr      M2   erkannter else-Teil, falls vorhanden
Slot 5: R scal/vect s11  Slot 1 von M1,
                          if (occuridx(s12,i)) issimpleidx(s12,i)
.... .  . ...       ...  .... . ... ..
```

1D–Reduktionen, Standardskalarprodukt und Euklidische Norm

```
VSUM ( i, s, y, c )
Sem.:  ‘s = c;’  DO i  s = s + ‘abs’y[i]
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s
Slot 2: R  vector    y      issimpleidx(y,i)
Slot 3: R  scalar    c      init-Wert; s falls nicht initialisiert
```

```
VPROD ( i, s, y, c )  similar
```

```
SSP ( i, x, y, z, c )
Sem.:  ‘s = c;’  DO i  s = s + y[i] * z[i]
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s
Slot 2: R  vector    y      issimpleidx(y,i)
Slot 3: R  vector    z      issimpleidx(z,i)
Slot 4: R  scalar    c      Init-Wert; s falls nicht initialisiert
```

```
VQSUM ( i, s, y, c )
Sem.:  ‘s = c;’  DO i  s = s + y[i] * y[i]
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s
Slot 2: R  vector    y      issimpleidx(y,i)
Slot 3: R  scalar    c      Init-Wert; s falls nicht initialisiert
```

ENORM$^{(1)}$ berechnet die Euklidische Norm. ENORM$^{(1)}$ ist instabil; es kann, wie hier, als Sequenz aus VQSUM$^{(1)}$ und SQRT$^{(0)}$ dargestellt werden.

```
ENORM ( i, s, z, x, c )
Sem.:  VQSUM(i,x,z,c); SQRT(s,x);
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s
Slot 2: W  vector    x      issimpleidx(x,i)
Slot 3: R  vector    z      issimpleidx(z,i)
Slot 3: R  scalar    c      Init-Wert; s falls nicht initialisiert
```

Die Muster GVSUM$^{(1)}$ und GVPROD$^{(1)}$ sind instabil, werden hier aber zur Angabe der Slotbelegungen aufgeführt:

```
GVSUM ( i, s, c, fy, y1, y2, ... )
Sem.:  'S = c;'  DO i  s = s + fy(y1[i],y2[i],....)
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s
Slot 2: R  scalar    c      Init-Wert; s falls nicht initialisiert
Slot 3: I  expr      fy     issimpleidx(fy,i) (fy quasiskalar)
Slot 4: R  scal/vect y1     erster Operand von fy, not(eqex(s,y1))
Slot 5: R  scal/vect y2     zweiter Operand von fy, not(eqex(s,y2))
.... .  .  ....      ..     ....
```

GVPROD (i, s, c, fy, y1, y2, ...) analog

Kernel zur 1D–Konvolution

```
S1CONV ( l, y, y, u, v, c )
Sem.:  DO l  y[l] = y[l] +- u[l] * v[c-l];
Slot 0: I  rng       l
Slot 1: W  vector    y          issimpleidx(y, l)
Slot 2: R  vector    y          eqfex Slot 1
Slot 3: R  vector    u          issimpleidx(u, l)
Slot 4: R  vector    v          issimpleidx(v, c-l)
Slot 5: R  scalar    c          notoccurin(c,l)
```

Lineare Rekurrenzen erster Ordnung

```
PREVSUM ( i, x, A, s )
Sem.:  'x[L] = s;'  DO i=[L+st:n:st]  x[i] = A[i]+-x[i-st];
Slot 0: I  rng       i=[L+st:n:st]      isconst(st)
Slot 1: W  vector    x     issimpleidx(x,i) (lhs), not eqsym(A,x),
                           isoffsetidx(x,i,-st) (rhs)
Slot 2: R  vector    A     issimpleidx(A,i) if occuridx(A,i)
Slot 3: R  scalar    s     s oder x[L], falls uninitialisiert
```

```
SUFVSUM ( i, x, A, s )
Sem.:  'x[n] = s;'  DO i=[n:L+st:st]  x[i] = A[i]+-x[i+st];
Slot 0: I  rng       i=[n:L+st:st]      isconst(st), st < 0
Slot 1: W  vector    x     issimpleidx(x,i) (lhs),
                           isoffsetidx(x,i,-st) (rhs)
Slot 2: R  vector    A     issimpleidx(A,i) if occuridx(A,i)
Slot 3: R  scalar    s     s oder x[L], falls uninitialisiert
```

```
FOLR ( i, x, A, B, C, s )
Sem.:  ‘x[L] = s;’  DO i=[L+st:n:st]  x[i] = C[i]*(A[i]+-B[i]*x[i-1]);
Slot 0: I  rng       i=[L+st:n:st]
Slot 1: W  vector    x     issimpleidx(x,i) (lhs),
                           isoffsetidx(x,i,-1) (rhs)
Slot 2: R  vec/leaf  A     issimpleidx(A,i) if occuridx(A,i)
Slot 3: R  vec/leaf  B     issimpleidx(B,i) if occuridx(B,i)
Slot 4: R  vec/leaf  C     issimpleidx(C,i) if occuridx(C,i)
Slot 5: R  scalar    s     s oder x[L], falls uninitialisiert
```

1D–Maximierungen/Minimierungen

```
VMAXVAL ( i, s, x, c )
Sem.:  ‘s = c;’  DO i  s = max( s, x[i] );
Slot 0: I  rng       i
Slot 1: W  ivar/dvar s     notoccurin(s,i)
Slot 2: R  vector    x     issimpleidx(x, i)
Slot 3: R  scalar    c     s falls nicht initialisiert
```

`VMINVAL` analog

```
VMAXLOC ( i, k, x, t )
Sem.:  ‘k = t;’  DO i  if (x[k] < x[i])  k = i;
Slot 0: I  rng       i
Slot 1: W  ivar      k     notoccurin(k,i)
Slot 2: R  vector    x     issimpleidx(x, i)
Slot 3: R  ivar/ival t     Init-Position; k falls uninitialisiert
```

`VMINLOC` analog

```
VMAXLOCM ( i, k, l, x, t, j )
Sem.:  ‘k = t;’  DO i  if (x[k][l] < x[i][j]) { k=i; l=j; }
Slot 0: I  rng       i
Slot 1: W  ivar      k     notoccurin(k,i)
Slot 2: W  ivar      l     notoccurin(l,i)
Slot 3: R  vector    x     issimpleidx(x,i)
Slot 4: R  ivar/ival t     Init-Position; k falls uninitialisiert
Slot 5: R  ivar      j     notoccurin(j,i)
```

`VMINLOCM` analog

```
VMAXVL ( i, k, x, s, c, t )
Sem.:  ‘k=t;’ ‘s=c;’  DO i  if (s < x[i]) { k = i; s = x[i] }
Slot 0: I  rng       i
Slot 1: W  ivar      k     notoccurin(k,i)
Slot 2: R  vector    x     issimpleidx(x, i)
Slot 3: W  ivar/dvar s     notoccurin(s,i)
Slot 4: R  scalar    c     s falls nicht initialisiert
Slot 5: R  ivar/ival t     k falls nicht initialisiert
```

`VMINVL` analog

```
VMAXVLM ( i, k, l, x, s, c, t, j )
Sem.:  'k=t;' 's=c;'  DO i  if (s<x[i][j]) { k=i; l=j; s = x[i][j]; }
Slot 0: I  rng       i
Slot 1: W  ivar      k     notoccurin(k,i)
Slot 2: W  ivar      l     notoccurin(k,i)
Slot 3: W  ivar/dvar s     notoccurin(s,i)
Slot 4: R  vector    x     issimpleidx(x, i)
Slot 5: R  scalar    c     s falls nicht initialisiert
Slot 6: R  ivar/ival t     k falls nicht initialisiert
Slot 7: W  ivar      j     notoccurin(j,i)
```

`VMINVLM` analog

Falls Absolutbeträge `abs(x[i])` minimiert werden (in Slot 2 ist das ABS–Flag gesetzt), so muß die Initialisierung bei VMAXVAL$^{(1)}$, VMINVAL$^{(1)}$, VMAXVL$^{(1)}$, VMINVL$^{(1)}$, VMAXVLM$^{(1)}$ und VMINVLM$^{(1)}$, sofern keine Konstante, ebenfalls mit dem Absolutbetrag erfolgen. Das gleiche gilt für die Vorkommen von `x` in der Bedingung.

1D–Relaxationen

```
VJACOBI (i, _, y[i], ... (other STAR slots))
Sem.:  DO i  (H)STAR( y[i], ..., starvar[i], ..., i, ...)
Slot 0:  I rng       i       in Slot 19/22 vorkommend
Slot 2:  W vector    y       issimpleidx(y,i)
Slot 7:  R vector    starvar issimpleidx(starvar,i);  starvar != y
Slot 8:  R vector    F       issimpleidx(F,i)  falls F vorhanden
Slot 10: R vect/scal C[1]    if occurin(C[1],i)  issimpleidx(C[1],i)
...
Slot 18: R vect/scal C[9]    if occurin(C[9],i)  issimpleidx(C[9],i)
...
```

`VGAUSSSEIDEL` analog (Bedingung: `starvar` = `y`)

5.3.3 Muster der Ordnung 2

Quasiskalare Matrixoperationen

Die Schleifenvariablen `i,j` sind in Slot 0 und 1 so geordnet, daß `i` in Slot 0 und `j` in Slot 1 steht g.d.w. `precedes(A,i,j)`.

```
MCOPY (i, j, A, B )
Sem.:  DODO i,j  A[i][j] = B[i][j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R matrix    B         issimpleidx(B,i), issimpleidx(B,j)
```

```
MINITSP ( i, j, A, s )
Sem.:  DODO i,j  A[i][j] = s
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R scalarvar s         notoccuridx(s,i), notoccuridx(s,j)
```

`s` kann gleich `i` oder gleich `j` sein.

```
MINIT ( i, j, A, c )
Sem.:  DODO i,j  A[i][j] = c
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R constant  c         notoccurin(c,i), notoccurin(c,j)
```

MASSIGN$^{(2)}$ initialisiert alle Elemente einer Matrix `A` mit einer skalaren Variablen `v`:

```
MASSIGN ( i, j, A, v )
Sem.:  DODO i,j  A[i][j] = v
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R variable  v         notoccurin(v,i), notoccurin(v,j)
```

MASSIGNV$^{(2)}$ initialisiert alle Elemente je eines Spalten– bzw. Zeilenvektors der Matrix `A` mit je einem Element eines Vektors `b`:

```
MASSIGNV ( i, j, A, v )
Sem.:  DODO i,j  A[i][j] = b[i]     bzw.  b[j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A  issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R vector    b  (notoccurin(v,i), issimpleidx(b,j))
                       oder (notoccurin(v,j), issimpleidx(b,i))
```

MASSIGNSP$^{(2)}$ initialisiert alle Elemente einer Matrix `x` mit demselben skalaren Wert eines Ausdrucks `fn`. MASSIGNSP$^{(2)}$ ist instabil; eine MASSIGNSP$^{(2)}$–Instanz kann mit Hilfe einer temporären skalaren Variable durch eine Sequenz aus einer `fn`–Instanz und einer MASSIGN$^{(2)}$–Instanz dargestellt werden.

```
MASSIGNSP ( i, j, x, fn, v1, v2, ... )
Sem.:  DODO i,j  x[i][j] = fn
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W vector    x         issimpleidx(x,i)
Slot 3: I expr      fn        notoccuridx(fn,i),notoccuridx(fn,j)
Slot 4: R scalarvar v1        erste (skalare) Variable in fn
....
```

In `fn` können `i` oder `j` oder beide vorkommen, aber nicht in einem Indexausdruck einer Feldreferenz.

```
MADD, MAADD, MMUL, MAMUL (i, j, A, B, C )
Sem.:  DODO i,j  A[i][j] = B[i][j] +-*/ C[i][j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R matrix    B         issimpleidx(B,i), issimpleidx(B,j)
Slot 4: R matrix    C         issimpleidx(C,i), issimpleidx(C,j)
```

Für MAADD$^{(2)}$ und MAMUL$^{(2)}$ müssen die Einträge von Slot 2 und 3 gleich sein.

```
MINC ( i, j, x, y, c )
Sem.:  DODO i,j  x[i][j] = y[i][j] +- c
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    x         issimpleidx(x,i), issimpleidx(x,j)
Slot 3: R matrix    y         issimpleidx(y,i), issimpleidx(y,j)
Slot 4: R constant  c         notoccurin(c,i), notoccurin(c,j)
```

```
MAINC
Sem.:  DODO i,j  x[i][j] = x[i][j] +- c
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    x         issimpleidx(x,i), issimpleidx(x,j)
Slot 3: R matrix    x         eqfex(x auf lhs, x auf rhs)
Slot 4: R constant  c         notoccurin(c,i), notoccurin(c,j)
```

```
SM
Sem.:  DODO i,j  x[i][j] = y[i][j] * c
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    x         issimpleidx(x,i), issimpleidx(x,j)
Slot 3: R matrix    y         issimpleidx(y,i), issimpleidx(y,j)
Slot 4: R constant  c         notoccurin(c,i), notoccurin(c,j)
```

```
MAADDSS ( i, j, A, A, u, v )
Sem.:  DODO i,j  A[i][j] = A[i][j] +- u * v
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(x, i), issimpleidx(x, j)
Slot 3: R matrix    A         eqfex(Slot 2, Slot 3)
Slot 4: R scalar    u         notoccurin(u,i), notoccurin(u,j)
Slot 5: R scalar    v         notoccurin(v,i), notoccurin(u,j)
```

```
MAADDSM ( i, j, A, A, U, v )
Sem.:  DODO i,j  A[i][j] = A[i][j] +- U[i][j] * v
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(x,i), issimpleidx(x,j)
Slot 3: R matrix    A         eqfex(Slot 2, Slot 3)
Slot 4: R matrix    U         issimpleidx(u,i), issimpleidx(u,j)
Slot 5: R scalar    v         notoccurin(v,i), notoccurin(u,j)
```

```
MINV ( i, j, A, B )
Sem.: DODO i,j  A[i][j] = 1/B[i][j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A         issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R matrix    B         issimpleidx(B,i), issimpleidx(B,j)
```

`GMOP ( i, j, A, fn, B, C, D, ...)` und

ist analog zu GVOP[(1)] definiert und ebenfalls instabil. Die Operanden `B, C, ...` dürfen auch Skalare oder Vektoren in `i` oder `j` sein.

Maskierte Matrixoperationen

Das Muster MCONDASS(2) ist instabil, d.h. nur von interner Bedeutung, wird aber hier wegen der Slotbelegung aufgeführt. Es steht für elementweise bedingte Matrixoperationen.

```
MCONDASS (i,j,y,cond,M1,M2, Slots von M1, Slots von M2, Slots von cond)
Sem.:  DO i  if (cond) M1(y,...); ‘else M2(y,...);’
Slot 0: I  rng   i
Slot 1: I  rng   j
Slot 2: W matrix y    gleiche linke Seite von M1 und M2
Slot 3: R expr   cond erkannte Bedingung;
                      issimpleidx(cond,i), issimpleidx(cond,j)
Slot 4: I expr   M1   erkannter then-Teil
Slot 5: I expr   M2   erkannter else-Teil, falls vorhanden
Slot 6: R        s11  Slot 1 von M1,
                       if (occuridx(s12,i)) issimpleidx(s12,i)
                       if (occuridx(s12,j)) issimpleidx(s12,j)
.... .  . ...    ...  .... . ... ..
```

Shiften von Matrizen

```
VVSHIFT ( i, j, a, a, k )
Sem.:  DODO i,j  a[i][j] = a[i+-k][j] bzw. a[i-+k][j] = a[i][j]
Slot 0: I rng    i          Schleifenvariable, Shift-Bereich
Slot 1: I rng    j          Schleifenvariable, Schleifengrenzen
Slot 2: W vector a[i][j]    issimpleidx(a,i), issimpleidx(a,j)
Slot 3: R vector a[i+-k][j] issimpleidx(a,j), isoffsetidx(a,i,+-k)
Slot 4: R scalar k          Shift-Distanz
```

```
MSHIFT ( i, j, a, a, k1, k2 )
Sem.:  DODO i,j  a[i][j]=a[i+-k1][j+-k2] bzw. a[i-+k1][j-+k2]=a[i][j]
Slot 0: I rng    i             Schleifenvariable, Shift-Bereich
Slot 1: I rng    j             Schleifenvariable, Shift-Bereich
Slot 2: W vector a[i][j] (lhs) issimpleidx(a,i), isoffsetidx(a,j),
Slot 3: R vector a[i][j] (rhs) isoffsetidx(a,i,+-k1) bzw. (a,j,+-k2)
Slot 4: R scalar k1            Shift-Distanz in i-Richtung
Slot 5: R scalar k2            Shift-Distanz in j-Richtung
```

Matrix–Vektor–Multiplikation und verwandte Muster

```
MAADDVV ( i, j, A, b, c, A )
Sem.:  DODO i,j  A[i][j] = A[i][j] +- b[i] */ c[j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A        (lhs) issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R vector    b        issimpleidx(b,i), notoccurin(b,j)
Slot 4: R vector    c        notoccurin(C,i), issimpleidx(C,j)
Slot 5: R matrix    A        (rhs) eqfex Slot 2
```

```
MAADDSVV ( i, j, A, A, s, b, c )
Sem.: ‘DODO i,j A[i][j]=init;’ DODO i,j A[i][j]=A[i][j]+-s*b[i]*c[j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W matrix    A        (lhs) issimpleidx(A,i), issimpleidx(A,j)
Slot 3: R matrix    A        (rhs) eqfex Slot 2
Slot 4: R scalar    s        notoccurin(s,i), notoccurin(s,j)
Slot 5: R vector    b        issimpleidx(b,i), notoccurin(b,j)
Slot 6: R vector    c        notoccurin(C,i), issimpleidx(C,j)
```

```
MV ( i, j, a, B, c, init )
Sem.:  ‘DO i  a[i] = init;’  DODO i,j  a[i] = a[i] +- B[i][j] * c[j];
Slot 0: I rng    i
Slot 1: I rng    j
Slot 2: W vector a    (lhs) issimpleidx(a,i), notoccurin(a,j)
Slot 3: R matrix B    issimpleidx(B,i), issimpleidx(B,j),
Slot 4: R vector c    issimpleidx(c,j),notoccurin(c,i),not(eqsym(a,c))
Slot 5: R expr   init (rhs) Init-Wert / Init-Vektor oder a
```

```
FSUBST ( i, j, a, B, a, a )
Sem.:  DO i  DO j=[lb:i-1]  a[i] = a[i] +- B[i][j] * a[j];
Slot 0: I rng       i
Slot 1: I rng       j=[lb:i]
Slot 2: W vector    a          (lhs) issimpleidx(a,i), notoccurin(a,j)
Slot 3: R matrix    B          issimpleidx(B,i), issimpleidx(B,j),
Slot 4: R vector    a          issimpleidx(a,i), notoccurin(a,j)
Slot 5: R vector    a          issimpleidx(a,j), notoccurin(a,i)
```

```
BSUBST ( i, j, a, B, a )
Sem.:  DO i  DO j=[lb:i-1]  a[i] = a[i] +- B[i][j] * a[j];
Slot 0: I rng       i
Slot 1: I rng       j=[i+1:ub]
Slot 2: W vector    a          (lhs) issimpleidx(a,i), notoccurin(a,j)
Slot 3: R matrix    B          issimpleidx(B,i), issimpleidx(B,j),
Slot 4: R vector    a          issimpleidx(a,j), notoccurin(a,i)
Slot 5: R vector    a          issimpleidx(a,i), notoccurin(a,j)
```

LR–Zerlegung (Ordnung 2)

```
MLUD ( i, j, Aij, Qik, Aij, Aik, Akk, Qik, Akj, k, diagval, colsign)
Sem.:  ‘A[k][k] = 1.0 ‘/ A[k][k]’ ’
       DO i[k{+1}:n] {
         Q[i][k] = A[i][k] */ A[k][k];
         DO j[k+1:n]
            A[i][j] = A[i][j] +- Q[i][k] * A[k][j];
       }
Bem.:  Q darf gleich A sein!
Slot 0: I rng    i
Slot 1: I rng    j
Slot 2: W matrix A[i][j] issimpleidx(A,i) und (A,j), notoccurin(A,k)
                         (‘+1’, falls A[k+1:n][k] zu Null wird)
Slot 3: W vector Q[i][k] (lhs) issimpleidx(Q,i), issimpleidx(Q,k)
```

```
Slot 4: R matrix A[i][j] (rhs) eqfex Slot 2
Slot 5: R vector A[i][k] issimpleidx(A,i) und (A,k); notoccurin(A,j)
Slot 6: R scalar A[k][k] notoccurin(A[k][k],i) und (A[k][k],j)
Slot 7: R vector Q[i][k] (rhs) issimpleidx(Q,i), issimpleidx(Q,k)
Slot 8: I vector A[k][j] notoccurin(A,i); issimpleidx(A,j) und (A,k)
Slot 9: R scalar k       lb(i)>k, lb(j)>k
Slot10: I scalar diagval Endwert von A[k][k]  (1 oder ‘-’1/A[k][k]alt)
Slot11: I ival   colsign -1 falls Q[i][k]=A[k][k+1:m]/A[k][k] negiert;
                          0  falls Q[i][k] zu Null gemacht wird;
                         +1 sonst (falls +-1, wird ein Vektor
                              Q[i]=‘-’A[i]/A[k][k] dahin geschrieben)
```

2D–Reduktionen

```
VVSUM ( i, j, v, A, c )
Sem.:  DO i  v[i] = c;  DODO i,j  v[i] = v[i] +- A[i][j];
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W vector    v       (lhs) issimpleidx(v,i), notoccurin(v,j)
Slot 3: R matrix    A       issimpleidx(A,i), issimpleidx(A,j),
Slot 4: R expr      c       (rhs) Init-Wert oder -Vektor oder v
```

```
VVPROD ( i, j, v, A, c )  similar
```

```
MSUM ( i, j, s, A, c )
Sem.:  {DO i  s = c;}  DODO i,j  s = s +- A[i][j];
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W scalar    s       (lhs) notoccurin(s,i), notoccurin(s,j)
Slot 3: R matrix    A       issimpleidx(A,i), issimpleidx(A,j),
Slot 4: R expr      c       (rhs) Init-Wert oder s
```

```
MPROD ( i, j, s, A, c )  similar
```

Konvolutionen (Ordnung 2)

```
S2CONV ( l, k, y, init, u, v, c, d )
Sem.:  {y=init;} DODO k,l  y = y +- u[l][k] * v[c-l][d-k];
Slot 0: I  rng       l
Slot 1: I  rng       k
Slot 2: W  scalar    y    notoccurin(y,l), notoccurin(y,k)
Slot 3: R  scalar    init Init-Wert oder y, falls uninitialisiert
Slot 4: R  matrix    u    issimpleidx(u, l), issimpleidx(u, k)
Slot 5: R  matrix    v    issimpleidx(v, c-l), issimpleidx(v, d-k)
Slot 6: R  ivar      c    notoccurin(c,l)
Slot 7: R  ivar      d    notoccurin(d,k)
```

```
VCONV ( j, l, y, s, u, v )
Sem.:  y = s;  DODO j,l  y[j] = y[j] +- u[l] * v[j-l];
Slot 0: I  rng       j
Slot 1: I  rng       l
Slot 2: W  vector    y    issimpleidx(y,j), notoccurin(y,k)
Slot 3: R  scal/vec  s    Init-Wert oder y, falls uninitialisiert
Slot 4: R  vector    u    issimpleidx(u,l), notoccurin(u,j)
Slot 5: R  vector    v    issimpleidx(v, j-l)
```

```
V1CONV ( i, l, y, s, u, v, c )
Sem.:  y = s;  DODO i,l  y[i] = y[i] +- u[l] * v[c-l];
Slot 0: I  rng       i
Slot 1: I  rng       l
Slot 2: W  vector    y    issimpleidx(y, j), notoccurin(y,k)
Slot 3: R  scal/vect s    Init-Wert/-Vektor; y, falls uninitialisiert
Slot 4: R  vector    u    issimpleidx(u, l), notoccurin(u,i)
Slot 5: R  vector    v    issimpleidx(v, c-l), notoccurin(c,i)
Slot 6: R  ivar      c    notoccurin(c,l)
```

2D–Maximierungen/Minimierungen

```
MMAXVAL ( i, j, s, A, c )
Sem.:  s = c;  DODO i,j  s = max( s, A[i][j] );
Slot 0: I  rng       i
Slot 1: I  rng       j
Slot 2: W  scalarvar s       notoccurin(s,i), notoccurin(s,j)
Slot 3: R  matrix    A       issimpleidx(A,i), issimpleidx(A,j)
Slot 4: R  scalar    c       Init-Wert oder s, falls uninitialisiert
```

`MMINVAL ( i, j, s, A, c )` analog

```
MMAXLOC ( i, j, ki, kj, A, ti, tj )}
Sem.:  ki=ti; kj=tj; (ki,kj) = Indizes von max( A[i][j] );
Slot 0: I  rng       i
Slot 1: I  rng       j
Slot 2: W  ivar      ki      not(eqsym(i,ki))
Slot 3: W  ivar      kj      not(eqsym(j,kj))
Slot 4: R  matrix    A[i][j] issimpleidx(A,i), issimpleidx(A,j)
Slot 5: R  scalar    ti      init-Wert oder ki, falls uninitialisiert
Slot 6: R  scalar    tj      init-Wert oder kj, falls uninitialisiert
```

Bei Existenz mehrerer Maxima wird das mit lexikalisch kleinerem Indexpaar `(i,j)` gewählt.

`MMINLOC` analog

`MMAXVL ( i, j, s, ki, kj, A, c, ti, tj )`

Kombination aus MMAXVAL und MMAXLOC. Slot–Typen siehe dort.

`MMINVL` analog

```
VVMAXVAL ( i, j, v, A, c )
Sem.:  s = c;  DODO i,j  v[i] = max( v[i], A[i][j] );
Slot 0: I  rng       i
Slot 1: I  rng       j
Slot 2: W  vector    v    notoccurin(s,i), notoccurin(s,j)
Slot 3: R  matrix    A    issimpleidx(A,i), issimpleidx(A,j)
Slot 4: R  scal/vect c    Init-Wert/Vektor; s, falls uninitialisiert
```

`VVMINVAL` analog

```
VVMAXLOC ( i, j, k, A, t )
Sem.:  DO i k[i]=t‘[i]’; DO i  k[i] = Index von max{j}( A[i][j] );
Slot 0: I  rng        i
Slot 1: I  rng        j
Slot 2: W  vector     k[i]     issimpleidx(k,i)
Slot 3: R  matrix     A[i][j] issimpleidx(A,i), issimpleidx(A,j)
Slot 4: R  scal/vect t        Init-Wert/Vektor; k, falls uninitialis.
```

`VVMINLOC` analog

```
VVMAXVL ( i, j, s, ki, kj, A, c, ti, tj )
```

Kombination aus `VVMAXVAL` und `VVMAXLOC`; Slot–Typen siehe dort.

`VVMINVL` analog

2D–Relaxationen

```
MJACOBI ( i, j, y[i][j], other STAR slots )
Sem.:  DODO i,j  STAR(-,-,y[i][j],...,starvar,...,i,...,j,...)
Slot 0: I rng        i       in Slot 19/22 vorkommend
Slot 1: I rng        j       in Slot 22/19 vorkommend
Slot 2: W matrix     y       issimpleidx(y,i), issimpleidx(y, j)
Slot 7: R matrix     starvar issimpleidx(starvar,i) und (starvar,j),
                             not(eqsym(starvar,y))
Slot 8: R matrix     F       issimpleidx(F,i) und (F,j) falls F vorkommt
Slot 10:R matr/scal C[1]    issimpleidx(C[1],i) falls occurin(C[1],i),
                             issimpleidx(C[1],j) falls occurin(C[1],j)
...
Slot 18:R matr/scal C[9]    issimpleidx(C[9],i) falls occurin(C[9],i),
                             issimpleidx(C[9],j) falls occurin(C[9],j)
...
```

`MGAUSSSEIDEL` analog, aber `eqsym(y,starvar)`

```
VVJACOBI ( i, j, y[i][j], .... )
Sem.:  DODO i,j  HSTAR(-,-,y[i][j],...,starvar[i][j],...,j,...)
Slot 0: I rng        i       nicht in Slot 19/22 vorkommend
Slot 1: I rng        j       in Slot 22/19 vorkommend
Slot 2: W matrix     y       issimpleidx(y,i), issimpleidx(y,j)
Slot 7: R matrix     starvar issimpleidx(starvar,i) und (starvar,j),
                             starvar != y
Slot 8: R matrix     F       issimpleidx(F,i), (F,j) falls F vorkommt
Slot 10:R matr/scal C[1]    issimpleidx(C[1],i) falls occuridx(C[1],i),
                              issimpleidx(C[1],j) falls occuridx(C[1],j)
...
Slot 18:R matr/scal C[9]    issimpleidx(C[9],i) falls occuridx(C[9],i),
                              issimpleidx(C[9],j) falls occuridx(C[9],j)
...
```

`j` spannt einen (eindimensionalen) Stern auf.

`VVGAUSSSEIDEL` analog, aber `eqsym(y,starvar)`

5.3.4 Muster der Ordnung 3

Matrix–Matrix–Multiplikation und verwandte Muster

```
MM ( i, j, k, C, A, B, c )
Sem.:  'DODO i,j  a[i][j] = d;'
       DODODO i,j,k  C[i][j] = C[i][j] +- A[i][k] * B[k][j];
Slot 0: I rng    i
Slot 1: I rng    j
Slot 2: I rng    k
Slot 3: W matrix C  (lhs) issimpleidx(C,i), issimpleidx(C,j)
Slot 4: R matrix A  issimpleidx(A, i), issimpleidx(A, k),
Slot 5: R matrix B  issimpleidx(B, k), issimpleidx(B, j)
Slot 6: R expr   c  Init-Wert/Matrix oder C, falls uninitialisiert
```

```
SMM ( i, j, k, C, init, s, A, B )
Sem.:  'DODO i,j a[i] = d;'
       DODODO i,j,k  C[i][j] = C[i][j] +- A[i][k] * B[k][j];
Slot 0: I rng    i
Slot 1: I rng    j
Slot 2: I rng    k
Slot 3: W matrix C    (lhs) issimpleidx(C,i), issimpleidx(C,j)
Slot 4: R expr   init Init-Wert/Matrix oder C, falls uninitialisiert
Slot 5: R scalar s    notoccurin(s,i), notoccurin(s.j)
Slot 6: R matrix A    issimpleidx(A,i), issimpleidx(A,k),
Slot 7: R matrix B    issimpleidx(B,k), issimpleidx(B,j)
```

Konvolutionen (Ordnung 3)

```
V2CONV ( i, k, l, y, s, u, v, c )
Sem.:  'DO i  y[i] = s;'
       DODODO i,k,l  y[i] = y[i] +- u[k][l] * v[i-k][c-l];
Slot 0: I rng       i
Slot 1: I rng       k
Slot 2: I rng       l
Slot 3: W vector    y  (lhs) issimpleidx(y,i), issimpleidx(y,j)
Slot 4: R vect/scal s  Init-Wert/Vektor oder y, falls uninitialisiert
Slot 5: R matrix    u  issimpleidx(u,k), issimpleidx(u,l)
Slot 6: R matrix    v  issimpleidx(v,i-k),issimpleidx(v,c-l)
Slot 7: R ivar      c  notoccurin(c,l)
```

```
M1CONV ( i, j, l, y, s, u, v )
Sem.:  'DODO i,j  y[i][j] = s;'
       DODODO i,j,l  y[i][j] = y[i][j] +- u[l] * v[i-c][j-l];
Slot 0: I rng       i
Slot 1: I rng       k
Slot 2: I rng       l
Slot 3: W matrix    y  (lhs) issimpleidx(y, i), issimpleidx(y, j)
Slot 4: R vect/scal s  Init-Wert/Vektor oder y, falls uninitialisiert
Slot 5: R vector    u  issimpleidx(u,l), notoccurin(u,k)
Slot 6: R matrix    v  issimpleidx(v,i-k),issimpleidx(v,c-l),
                       notoccurin(c,l)
```

LR–Zerlegung

```
LUD (k,i,j,A[i][j],Q[i][k],A[i][j],diagval,colsign,A[i][k],A[k][j])
Sem.:  DO k { A[k][k] = 1.0 ‘/A[k][k]’
              DO i[k+1:n]  Q[i][k] = ‘-’‘1/’A[k][k] * A[i][k];
              DODO i[k+1:n],j[k{+1}:n]
                   A[i][j] = A[i][j] +- Q[i][k] * A[k][j];
            }
Slot 0: I rng    k
Slot 1: I rng    i
Slot 2: I rng    j
Slot 3: W matrix A[i][j]  rechteckige Matrix
Slot 4: W matrix Q[i][k]  linke untere Dreiecksmatrix, falls Q!=A
Slot 5: R matrix A[i][j]  wie Slot 3
Slot 6: I scalar diagval  Endwert von A[k][k] (1 oder 1/A[k][k]alt)
Slot 7: I ival   colsign  -1 falls Q[i][k]=A[k][k+1:m]/A[k][k] negiert;
                          0  falls Q[i][k] zu Null gemacht wird;
                          +1 sonst (falls +-1, wird die L-Matrix
                                    L = ‘-’Q dorthin geschrieben)
Slot 8: I vector A[i][k]  issimpleidx(A,i) und (A,k); notoccurin(A,j)
Slot 9: I vector A[k][j]  notoccurin(A,i); issimpleidx(A,j) und (A,k)
```

Relaxationsalgorithmen

```
JACOBI (i, j, y[i][j], ..., k )
Sem.: DO k[lb,ncycles] {
        DODO i,j  STAR(_,_,y[i][j],...,starvar,...,i,...,j,...);
        DODO i,j  starvar[i][j] = y[i][j];
      }
Slot 0: I rng        i     in Slot 19 oder 22 vorkommend
Slot 1: I rng        j     in Slot 22 oder 19 vorkommend
Slot 2: W matrix     y     issimpleidx(y,i), issimpleidx(y,j)
Slot 7: R matrix   starvar issimpleidx(starvar,i) und (starvar,j),
                            not(eqsym(starvar,y))
Slot 8: R matrix     F     issimpleidx(F,i) und (F,j), falls F vorkommt
Slot 10:R mat/scalar C[1]  issimpleidx(C[1],i) falls occurin(C[1],i),
                            issimpleidx(C[1],j) falls occurin(C[1],j)
...
Slot 18:R mat/scalar C[9]  issimpleidx(C[9],i) falls occurin(C[9],i),
                            issimpleidx(C[9],j) falls occurin(C[9],j)
...
Slot 25:I rng        k     notoccurin(S,k) fuer alle Slots S in 0...24
```

`GAUSSSEIDEL` analog, mit `y` = `starvar`

5.3.5 Muster der Ordnung 4

```
MCONV ( i, j, l, y, s, u, v )
Sem.:  {DODO i,j  y[i][j]=s;}
       DODODODO i,j,k,l  y[i][j] = y[i][j]+-u[k][l]*v[i-k][j-l];
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: I rng       k
```

```
Slot 3: I rng        l
Slot 4: W matrix     y  (lhs) issimpleidx(y,i), issimpleidx(y,j)
Slot 5: R matr/scal  s  Init-Wert/-Matrix oder y, falls uninitialisiert
Slot 6: R matrix     u  issimpleidx(u,k), issimpleidx(u,l)
Slot 7: R matrix     v  issimpleidx(v,i-k),issimpleidx(v,j-l),
```

5.3.6 Instabile Muster

Unter instabilen Mustern verstehen wir weitere Muster, die sich unter Verwendung von temporären Feldern als Folge schon bekannter Muster schreiben lassen, also redundant sind.

Instabile Muster werden vom PARAMAT Mustererkennungswerkzeug erkannt, aber mit Hilfe temporärer Variablen noch vor Beginn der Codeerzeugungsphase in ihre Grundbestandteile zerlegt, sodaß sie für die Codeerzeugungsphase nicht sichtbar sind.

Dies betrifft die Muster

- SVSUM$^{(1)}$, SSSP$^{(1)}$, SMV$^{(2)}$, SMM$^{(3)}$, die aus VSUM$^{(1)}$, SSP$^{(1)}$, MV$^{(2)}$ bzw. MM$^{(3)}$ durch Multiplikation eines zusätzlichen skalaren Faktors hervorgehen;
- ENORM$^{(1)}$, das in VQSUM$^{(1)}$ und SQRT$^{(0)}$ zerlegt werden kann;
- VAADDMULTISV$^{(1)}$, dessen skalarer Anteil (MUL oder MULTIMUL$^{(0)}$) von der VAADDSV$^{(1)}$–Restoperation abgespalten werden kann;
- VASSIGNSP$^{(1)}$ und MASSIGNSP$^{(2)}$, deren skalarer Anteil von VASSIGN$^{(1)}$ bzw. MASSIGN$^{(2)}$ abgespalten werden kann;
- MAADDSVV$^{(2)}$, dessen skalarer Anteil (MUL) von der MAADDVV$^{(2)}$–Restoperation abgespalten werden kann;
- GVSUM$^{(1)}$, dessen quasiskalarer Anteil (GVOP$^{(1)}$ oder eine andere quasiskalare Vektoroperation wie VADD$^{(1)}$, VMUL$^{(1)}$,...) von der Reduktion VSUM$^{(1)}$ abgespalten werden kann (Analoges gilt für GVPROD$^{(1)}$);
- GVOP$^{(1)}$,die in elementare Vektoroperationen aufgespalten wird. Gleiches gilt für GMOP$^{(2)}$;
- CONDASS$^{(0)}$ und
- alle skalaren Muster, deren Instanzen einfach ignoriert werden.

Die betreffenden Schablonen sind im Anhang B aufgeführt.

5.3.7 Geplante Erweiterungen

Für zukünftige Versionen der Musterbibliothek planen wir u.a. folgende zusätzlichen Muster:

- VRAND$^{(1)}$ (globales UND) und VROR$^{(1)}$ (globales ODER) in Analogie zu VSUM$^{(1)}$, sowie MRAND$^{(2)}$ und MROR$^{(2)}$ in Analogie zu MSUM$^{(2)}$.
- PREVMAXVAL$^{(1)}$, PREVMINVAL$^{(1)}$, PREVRAND$^{(1)}$, PREVROR$^{(1)}$ in Analogie zu PREVSUM$^{(1)}$, sowie SUFVMAXVAL$^{(1)}$, SUFVMINVAL$^{(1)}$, SUFVRAND$^{(1)}$, SUFVROR$^{(1)}$ in Analogie zu SUFVSUM$^{(1)}$.
- APREVSUM$^{(1)}$, APREVMAXVAL$^{(1)}$ usw. analog zu PREVSUM$^{(1)}$, PREVMAXVAL$^{(1)}$ usf., für den Fall, daß Operanden– und Ergebnisvektor identisch sind,
- VSORT$^{(2)}$ zum Sortieren von Vektoren,
- FFT$^{(2)}$ (Fast–Fourier–Transformation),

- MULTISTAR$^{(0)}$ (Mehrere Differenzensterne in einem Ausdruck; instabil).

Ferner beabsichtigen wir, für fast alle Muster der Ordnungen 1,2,... zusätzliche *Masken–Slots* einzuführen, die die bisherige, ziemlich umständliche Doppelmuster–Lösung VCONDASS$^{(1)}$ bei der Darstellung maskierter Vektoranweisungen ersetzen soll. Derartige Maskierungsparameter gibt es auch in einigen `Fortran 90`–Standardfunktionen. Einen ähnlichen Maskierungsparameter hat [LC91] für höhere Kommunikationsroutinen vorgesehen (aber auch noch nicht implementiert).

5.4 Analyse von Quellprogrammen

In diesem Abschnitt wollen wir verdeutlichen, warum wir gerade die im vorigen Abschnitt aufgelisteten Muster für bedeutsam halten. Wir wollen demonstrieren, daß wir mit unseren Mustern große Teile numerischer Programme, die auf dichtbesetzten Matrizen arbeiten und die geeignete Kandidaten für die Portierung auf DMS sind, abdecken können.

5.4.1 Typische Indizierungen in numerischen Programmen

Mit diesem Unterabschnitt wollen wir einige statistische Ergebnisse aus [SLY90] über numerische `FORTRAN`–Programme vorausschicken, die unsere These zusätzlich stützen.

[SLY90] beobachtete an einer aus über hunderttausend `FORTRAN`–Zeilen wissenschaftlicher Anwendungen und Bibliotheksroutinen bestehenden Benchmark–Suite typische Eigenschaften der Indexausdrücke von Feldvorkommen. Daraus ergab sich folgendes Bild:

- Felder mit mehr als zwei Dimensionen sind selten (8%).
- 53% aller Indizierungen sind linear (es gilt `issimpleidx(e)` für alle Indexausdrücke `e`), und weitere 13% sind teilweise linear (`issimpleidx` gilt für mindestens einen Indexausdruck).
- Nichtlinearität wurde (da in die Datenabhängigkeitsanalyse integriert) oft (> 83%) durch symbolische Variablen wie etwa Schleifenobergrenzen (`N`) induziert. In unseren Musterinstanzen dürfen symbolische Variablen jedoch vorkommen, wo sie nicht ausdrücklich untersagt sind. Echte Nichtlinearität von Indexausdrücken im mathematischen Sinne kommt also fast nie vor[5].
- Die Koeffizienten linearer Indexausdrücke sind fast immer konstant und sehr häufig `+1` oder `-1`. Die Indexausdrücke selbst sind konstant in 15% (eindimensional) bis 46% (vierdimensional) aller Fälle.
- gekoppelte Indexausdrücke (eine gebundene Schleifenvariable kommt in mehr als einem Indexausdruck vor) betreffen fast ausschließlich nur zwei Dimensionen, und kommen nie in mehr als dreidimensionalen Feldreferenzen vor.

5.4.2 Benchmark Suites: Livermore Loops und Purdue Set

Mit den vorliegenden Mustern aus der Musterbibliothek können über die Hälfte der 24 Livermore Loops [McM86] und alle Routinen des Purdue Set aus der HPF Benchmark Suite [MFL+92] zu meist großen Teilen abgedeckt werden (siehe Tabellen 5.1 und 5.2).

[5] [Pet93] beobachtete ferner an den Perfect Club Benchmarks, daß dort der Hauptverursacher 'echter' nichtlinearer Indexausdrücke die Operationen der Fast–Fourier–Transformation sind.

Loop	Name	erkannte Muster	erkannte Schleifen
1	Hydrofragment	`GVOP`	1 von 1
3	Inner Product	`SSP`	1 von 1
5	tri-diag. elim., below diagonal	`FOLR`	1 von 1
7	equation of state fragment	`GVOP`	1 von 1
8	A.D.I Integration	`VJACOBI` (3), `GVOP` (3)	6 von 6
9	Numerical Integration	`GVOP`	1 von 1
10	Numerical Differentiation	`VCOPY` (10), `VADD` (9)	19 von 19
11	First Sum	`PREVSUM`	1 von 1
12	First Difference	`VJACOBI`	1 von 1
13	2D particle in a cell	`VCOPY` (4), `VAMOD` (4),`VAINC` (2), `VAADD` (2)	12 von 17
14	1D particle in a cell	`GVOP` (3), `VCOPY`, `VADD` (2)	6 von 12
18	2D explicit hydrodyn. fragment	`GMOP` (4), `MAADDSM` (2)	6 von 6
21	Matrix Product	`MM`	1 von 1
22	Planckian Distribution	`GVOP` (2)	2 von 2
23	2D implicit hydrodyn. fragment	`MGAUSSSEIDEL`	1 von 1
24	1D Minimization	`VMINLOC`	1 von 1

Tabelle 5.1 In den Livermore Loops mit der vorliegenden Version der Musterbibliothek erkennbare Muster. Die rechte Spalte gibt an, wieviele Schleifen (nach der Anwendung von Schleifenaufgliederung) durch unsere Muster abgedeckt werden können. In einigen Kernels wurde außerdem Feldexpansion angewendet. GVOP$^{(1)}$ steht stellvertretend für alle darin enthaltenen Elementar–Vektoroperationen. Quellcodes und Details entnehme man Abschnitt C.2. In den übrigen Livermore Loops konnten bislang nur skalare Muster gefunden werden.

No.	Name	erkannte Muster	erk. Schleifen
1	Trapezoidal rule	`FSUM`	1 von 1
2	reduction function 1	`MINITSP, VVPROD, VSUM`	3 von 3
3	reduction function 2	`MINIT, VVPROD, VSUM`	3 von 3
4	reduction function 3	`VINIT, VINV, VSUM`	3 von 3
5	simple search	`MINITSP, MSUM, -`	2 von 3
6	tridiag. set of lin. eqns.	`VINIT` (8), `VMUL` (4), `GVOP` (8), `VCOPY` (5), `VSUM`	26 von 26
7	Lagrange interpolation	`VINITSP`(2), `VINC` (2), `VINV`, `VPROD` (2), –	7 von 8
8	divided differences	`VINITSP, VSIN, -, MSUM`	3 von 4
9	finite differences	`MINITSP, MINIT, MJACOBI, -, MCOPY, MSUM`	5 von 6
11	Fourier's moments	`VINITSP, GVOP, VSUM`	3 von 3
12	array construction	`VINITSP` (2), `MINITSP, MCOPY, VCOPY` (2)	6 von 6
13	floating point arithmetic	`VINITSP, GVOP` (4) `VCONDASS(VADD), VQSUM`	7 von 7
14	Simpson's and Gauss' integration	`FSUM` (5)	5 von 5
15	Chebyshev interpolation	`VINITSP` (2), `GVOP` (3), `VCOPY, -`	6 von 7

Tabelle 5.2 Im Purdue Set–Benchmark mit der vorliegenden Version der Musterbibliothek erkennbare Muster. Die rechte Spalte gibt wiederum an, wieviele Schleifen (nach der Anwendung von Schleifenaufgliederung) durch unsere Muster abgedeckt werden können. GVOP$^{(1)}$ steht stellvertretend für alle darin enthaltenen Elementar–Vektoroperationen. Ein Strich '–' steht für Teilberechnungen, in denen keines unserer Muster erkannt werden konnte.

5.4.3 Array–Primitive in Fortran 90 und HPF

Tabelle 5.3 gibt eine Übersicht über Standardfunktionen auf Vektoren und dichtbesetzten Matrizen in `Fortran 90`.

HPF [HPF93] bietet als zusätzliche Basisroutinen `COPY` (VCOPY$^{(1)}$) und `PARITY` (Vektor-XOR-Reduktion, —) sowie bitweise Reduktionsoperationen an. Darüberhinaus können alle 12 Reduktionsroutinen R mit den Bezeichnern R`_PREFIX` (*parallel prefix*) und R`_SUFFIX` (*parallel suffix*)

Fortran 90–Standardfunktion	entsprechendes Muster
DOT_PRODUCT	SSP(1)
SUM	VSUM(1)
PRODUCT	VPROD(1)
MAXVAL	VMAXVAL(1)
MINVAL	VMINVAL(1)
MAXLOC	VMAXLOC(1)
MINLOC	VMINLOC(1)
MATMUL	MM(3)
ANY	VROR(1)
AND	VRAND(1)
COUNT	—
TRANSPOSE	Spezialfall von VCOPY(1)
EOSHIFT	VSHIFT(1) (nichtzyklischer SHIFT in einer Dimension
CSHIFT	— (zyklischer SHIFT in einer Dimension

Tabelle 5.3 Standardfunktionen in Fortran 90 auf DOUBLE–Vektoren und dichtbesetzten Matrizen, und die ihnen entsprechenden Muster. Die Funktionen MAX/MINVAL/LOC, ANY, AND und COUNT haben einen zusätzlichen Maskierungsparameter (boole'scher Vektor), der die Ausführung der Funktion auf die Indizes mit wahrem Maskenwert einschränkt.

sowie *R*_SCATTER (Zugriff über Indexvektoren) kombiniert werden.Ferner gibt es Sortierroutinen GRADE_UP und GRADE_DOWN (VSORT(2)).

5.4.4 Numerische Standardsoftware–Pakete

BLAS–Routinen

Die BLAS–Routinen (Basic Linear Algebra Subroutines, [LHKK79, DDHH88]) sind in numerischen Programmen mit Vektoren und dichtbesetzten Matrizen häufig vorkommende Grundoperationen und über netlib als FORTRAN–Prozeduren frei erhältlich. Sie sind, genau der *order*–Nummer unserer Muster entsprechend, in (bisher) drei Pakete BLAS1 (einfache Vektoroperationen und –Reduktionen), BLAS2 (Matrix–Vektor–Multiplikationen) und BLAS3 (Matrix–Matrix–Multiplikation und LR–Zerlegung) aufgeteilt. Ferner sind sie in je einer Ausfertigung für die vier Datentypen SINGLE REAL, DOUBLE REAL, SINGLE COMPLEX und DOUBLE COMPLEX verfügbar. Die Tabellen 5.4, 5.5 und 5.6 enthalten Aufstellungen über die BLAS–Routinen und über unsere Muster, die ihnen entsprechen. Da die BLAS–Routinen vielfach als Basisbausteine numerischer Implementierungen verwendet werden, ist es besonders wichtig, daß sie möglichst vollständig von unseren Mustern abgedeckt und auch erkannt werden können.

Sonstige

Numerische Funktionsbibliotheken wie die kommerziellen Softwarepakete NAG oder IMSL sowie solche akademischer Herkunft wie LINPACK enthalten eine größere Anzahl von Basisroutinen, meistens in FORTRAN 77. Sie sind für viele verschiedene Rechnertypen erhältlich und werden mit nicht unbeträchtlichem personellen Aufwand ständig gepflegt.

NAG [NAG] beispielsweise ist ein Softwarepaket von über 1400 Unterprogrammen, das den gesamten Bereich der numerischen Mathematik abdeckt. Eine Übersicht über die wichtigsten Routinen enthält [Köc90]. Die meisten Routinen, die im Rahmen dieser Arbeit von Interesse sind, sind Analoga zu den oben genannten BLAS–Routinen. IMSL [IMS] bietet einen ähnlichen Bestand an numerischen Routinen. Für PC–Systeme gibt es die Unterprogrammbibliothek MATLAB [Mol].

BLAS1–Routine	entsprechendes Muster
`dasum`	$\mathsf{VSUM}^{(1)}$
`daxpy`	$\mathsf{VADDMUL}^{(1)}$
`dcopy`	$\mathsf{VCOPY}^{(1)}$
`ddot`	$\mathsf{SSP}^{(1)}$
`dnrm2`	$\mathsf{ENORM}^{(1)}$
`drot`	$\mathsf{VROT}^{(1)}$
`drotg`	(skalare Rechnung)
`dscal`	$\mathsf{SV}^{(1)}$
`dswap`	$\mathsf{VSWAP}^{(1)}$
`idamax`	$\mathsf{VMAXLOC}^{(1)}$

Tabelle 5.4 Die BLAS1–Routinen auf DOUBLE–Vektoren und die ihnen entsprechenden Muster.

BLAS2–Routine	entsprechendes Muster
`dgemv`	$\mathsf{MV}^{(2)}+\mathsf{SV}^{(1)}$, Spezialfälle: $\mathsf{MV}^{(2)}$,$\mathsf{SV}^{(1)}$, $\mathsf{SINIT}^{(0)}$
`dtrmv`	$\mathsf{FSUBST}^{(2)}$
`dtrsv`	$\mathsf{BSUBST}^{(2)}$
`dger`	$\mathsf{MAADDVV}^{(2)}+\mathsf{SM}^{(2)}$

Tabelle 5.5 BLAS2–Routinen, die auf dichtbesetzten, nichtsymmetrischen, ungepackten DOUBLE–Matrizen operieren, und die ihnen entsprechenden Muster.

LINPACK [Don79] enthält Routinen zum Lösen linearer Gleichungssysteme und baut auf den BLAS–Routinen auf. Zur Zeit werden ein Nachfolgepaket LAPACK mit äquivalenten SMS–parallelen Algorithmen sowie eine DMS–Variante ScaLAPACK [CDPW92] entwickelt. Für spezielle Aufgabengebiete gibt es speziellere Pakete, wie z.B. EISPACK [Smi76] für Eigenwertprobleme, (//)ELLPACK [HRC⁺90] zur Lösung elliptischer partieller Differentialgleichungen, oder FFT-PACK für Fast–Fourier–Transformationen. Viele solcher Pakete kann man als Public–Domain–Software via `netlib` beziehen.

Eine Routine aus EISPACK mit den darin vorkommenden Mustern ist in Anhang C.4 abgedruckt.

Auch im Rahmen numerischer Formelsammlungen wurden Unterprogrammsammlungen entwickelt, wie z.B. die „Numerical Recipes“ [PTVF92] oder [ER88].

5.4.5 Numerikpakete der Supercomputer–Hersteller

Aber auch fast jeder Hersteller von Rechnern, die sich zur Lösung wissenschaftlich–numerischer Probleme eignen, bietet eine hauseigene numerische Funktionenbibliothek an.

Beispielsweise enthält die CMSSL [Thi93, Thi92] für die CM-5 in der Version 3.1 etwa 250 vom Anwender benutzbare Routinen aus dem numerischen Bereich, darunter die BLAS–Analoga sowie BLAS–Routinen für Bandmatrizen und dünnbesetzte Matrizen, ferner direkte und iterative Löser für lineare Gleichungssysteme, Routinen für Eigenwertprobleme, für gewöhnliche Differentialgleichungen, für Fast Fourier Transformationen, den Simplex–Algorithmus, Konvolutionen, Sortieren sowie Kommunikationsroutinen für regelmäßige und unregelmäßige Kommunikation auf der CM-5.

Wir finden fast alle unsere Muster in der CMSSL wieder. Unter den CMSSL–Routinen für dichtbesetzte Matrizen finden wir etwa die Analoga zu $\mathsf{SSP}^{(1)}$, $\mathsf{ENORM}^{(1)}$, $\mathsf{MAADDVV}^{(2)}$, $\mathsf{MV}^{(2)}$, $\mathsf{VVMAXVAL}^{(2)}$, $\mathsf{MM}^{(3)}$, $\mathsf{LUD}^{(3)}$.

BLAS3–Routine	entsprechendes Muster
`dgemm`	MM$^{(3)}$; Spezialfälle: SM$^{(2)}$, MINIT$^{(2)}$
`dtrsm`	— (nur bis Höhe 2);

Tabelle 5.6 BLAS3–Routinen, die auf dichtbesetzten, nichtsymmetrischen, ungepackten DOUBLE–Matrizen operieren, und die ihnen entsprechenden Muster.

5.4.6 Sonstige

[Ble90] beschreibt eine maschinenunabhängige Sammlung von Basis–Operationen auf Feldern, mit denen viele, auch nichtnumerische Algorithmen neu formuliert werden können. Einige dieser Primitive finden wir in unserer Musterbibliothek wieder: Quasiskalare Vektoroperationen wie VCOPY$^{(1)}$, VADD$^{(1)}$, VMUL$^{(1)}$, ..., VINIT$^{(1)}$, VINITSP$^{(1)}$, VASSIGN$^{(1)}$; Permutationen; „scan“–Instruktionen (prefix) für + (PREVSUM$^{(1)}$, FOLR$^{(1)}$), max (PREVMAXVAL$^{(1)}$), min (PREVMINVAL$^{(1)}$), or (PREVROR$^{(1)}$), and (PREVRAND$^{(1)}$); Reduktionen wie VSUM$^{(1)}$, VMAXVAL$^{(1)}$, VMINVAL$^{(1)}$, VROR$^{(1)}$, VRAND$^{(1)}$, VMAXLOC$^{(1)}$, VMINLOC$^{(1)}$; ferner etliche Operationen mit indirektem Zugriff sowie einige weitere Operationen für den eher nichtnumerischen Anwendungsbereich.

[BBC$^+$93] enthält eine Sammlung von state-of-the-art–Algorithmen zum Lösen linearer Gleichungssysteme (siehe Tabelle 5.7). Die Programme sind als MATLAB–Programme oder mit Aufrufen der BLAS–Routinen erhältlich. Betrachtet man die Pseudocodes zu den vorgestellten Algorithmen, so stellt man fest, daß diese größtenteils aus einfachen Vektor– und Matrixoperationen zusammengesetzt sind, wie sie in unserer Musterbibliothek enthalten sind (rechte Spalte in Tabelle 5.7). Dieses Ergebnis ist besonders wichtig, weil (1) diese Algorithmen als Kernbestandteile in vielen numerischen Anwendungsprogrammen vorkommen, und (2) weil solche Übersichten auch von Anwendungsprogrammierern gelesen und als Grundlage zur Programmierung benutzt werden, was sich im Erscheinungsbild des Quellprogramms niederschlägt.

Das gleiche Argument benutzt [SCSS92] bei der Motivation eines parallelen Softwarepakets zur Lösung linearer dünnbesetzter Gleichungssysteme. Es enthält die sparse–Varianten von SSP$^{(1)}$, MV$^{(2)}$, VAADDSV$^{(1)}$, ... (siehe Anhang D).

Eine Implementierung des CG–Verfahrens [Lou92] ist im Anhang C.1 aufgeführt, nebst den darin erkannten Mustern. Dieses CG–Programm läßt sich *vollständig* in Musterinstanzen zerlegen, d.h. alle Phasen des Programms werden als Musterinstanzen erkannt.

Algorithmus	Muster zu den Grundoperationen
Jacobi solver	SSP$^{(1)}$, VAADD$^{(1)}$, SV$^{(1)}$
Gauß–Seidel solver	SSP$^{(1)}$, VINC$^{(1)}$, SV$^{(1)}$
successive overrelaxation (SOR)	SSP$^{(1)}$, VINC$^{(1)}$, VAADDSV$^{(1)}$
symmetric SOR (SSOR)	”
Conjugate Gradient (CG), siehe Anhang	MV$^{(2)}$, SSP$^{(1)}$, VADDMUL$^{(1)}$, ENORM$^{(1)}$
Generalized Min. Residual (GMRES)	SSP$^{(1)}$, VAADDSV$^{(1)}$, MV$^{(2)}$
Biconjugate Gradient (BiCG)	SSP$^{(1)}$, VAADDSV$^{(1)}$, MV$^{(2)}$
Quasi-Min. Residual (QMR)	SSP$^{(1)}$, VAADDSV$^{(1)}$, SV$^{(1)}$, MV$^{(2)}$
Conj. Gradient Squared (CGS)	SSP$^{(1)}$, VAADDSV$^{(1)}$, MV$^{(2)}$
Biconj. Gradient Stabilized (BiCGSTAB)	SSP$^{(1)}$, VAADDSV$^{(1)}$, MV$^{(2)}$
Chebyshev Iteration	VAADDSV$^{(1)}$, MV$^{(2)}$

Tabelle 5.7 (vgl. [BBC$^+$93] S. 31) Die wichtigsten Algorithmen aus [BBC$^+$93] zur Lösung linearer Gleichungssysteme, und die (stabilen) Muster, die darin vorkommen.

Ähnliche Ergebnisse erhalten wir für die in [Mül89] vorgestellten Algorithmen.

5.5 Diskussion

Dünnbesetzte Matrizen Interessant ist auf jeden Fall die Erweiterung der Musterbibliothek für Operationen auf dünnbesetzten Matrizen. Eine erste Liste von Mustern für dieses Gebiet auf der Basis indirekter Feldzugriffe existiert bereits und ist im Anhang D aufgeführt. Probleme bereitet hier die Tatsache, daß es, im Gegensatz zum „natürlichen“ Format dichtbesetzter Matrizen, kein einheitliches Format zur Speicherung dünnbesetzter Matrizen gibt. Da für jede Speicherungsvariante jeweils ein eigenes Muster bereitgestellt werden muß, kann dies die Anzahl der Muster unter Umständen stark erhöhen. Die diesbezügliche Weiterentwicklung von PARAMAT ist Gegenstand aktueller und zukünftiger Forschung [Zap94].

6 Erkennung von Mustern in numerischen Programmen

"The HOW question cannot be answered with a purely syntactic approach (e.g., by the use of grammar rules). This is because abstract concepts may not be localized; they may not occupy consecutive sections of code. More commonly, they interleave with each other and sometimes even share components." [KNE93]

Nachdem wir im vorangegangenen Kapitel die wichtigsten Muster, die häufig in numerischen Programmen vorkommen, betrachtet und in der Basis–Musterbibliothek gesammelt haben, wollen wir in diesem Kapitel erläutern, wie diese Muster in einem Programm der vereinfachten sequentiellen Quellsprache LATINUS identifiziert werden können.

Das Prinzip unserer Mustererkennungstechnik ist einfach. Es wird zum einen durch die Zwischendarstellung des Quellprogramms als abstraktem Syntaxbaum unterstützt, zum anderen durch die Ausnutzung der natürlichen semantischen Hierarchie der Muster in der Musterbibliothek.

Grob vereinfacht, durchmustert unser Algorithmus den abstrakten Syntaxbaum in Postorder (von links nach rechts und bottom–up). In jedem inneren Knoten v des Syntaxbaumes untersucht er auf der Basis der schon erkannten Muster für die Kinder von v, ob es für diesen Knoten ein passendes Muster m in der Bibliothek gibt. Dies geschieht durch den Aufruf eines kurzen Programmstücks, der Realisierung einer sogenannten *Schablone (template)*. Im allgemeinen gibt es für jedes Muster mehrere verschiedene Schablonen. Eine *Schablonenrealisierung* umfaßt eine Folge von Bedingungen (constraints), denen v und die Musterinstanzen seiner Kinder genügen müssen, um als Inkarnation von m erkannt werden zu können. Falls alle Bedingungen erfüllt werden, so wird durch die Realisierung der Schablone eine entsprechende Musterinstanz erzeugt, die aktualen Parameter werden in die vorgeschriebenen Slots eingetragen, und zuletzt wird die Instanz in den Knoten als Attribut $v.matched$ in v eingetragen. Falls nicht alle Bedingungen erfüllt werden konnten, bleibt dieses Attribut unbesetzt — der Knoten v bleibt unerkannt.

Die schon erkannten Muster der Kinder des Knotens beschränken die Anzahl der Alternativen für mögliche Muster von v erheblich. Meist genügt schon ein einziges charakteristisches Muster eines Kindes (*Triggermuster*) in Verbindung mit dem Operator von v, um die richtige Schablone eindeutig auszuwählen. Bei mehreren Möglichkeiten werden diese eine nach der anderen durchprobiert. Unzutreffende Schablonen brechen frühstmöglich ab, sodaß die nächste Alternative ausprobiert werden kann. Es kann niemals vorkommen, daß zwei Muster gleichzeitig auf dasselbe Stück Code passen (ansonsten wäre uns nämlich ein Fehler beim Entwurf der Musterbibliothek unterlaufen) — es sei denn, daß das eine ein Spezialfall des anderen ist (z.B. ist $\mathsf{VADD}^{(1)}$, die Vektoraddition, ein Spezialfall von $\mathsf{GVOP}^{(1)}$, einer allgemeinen Vektoroperation, und $\mathsf{VAADD}^{(1)}$ ist wiederum ein Spezialfall von $\mathsf{VADD}^{(1)}$). In diesem Fall müssen daher die Schablonen speziellerer Muster *vor* denen allgemeinerer Muster getestet werden. (vgl. Anhang B.6).

Wir stellen die Verbindung von Triggermuster zu möglichem Muster für den Elternknoten als gerichtete Kante in einem Graphen, dem *Musterhierarchiegraphen (pattern hierarchy graph, PHG)*, dar.

Das bislang Gesagte betrifft das Erkennen von Mustern „entlang" vertikaler Kanten im Syntaxbaum und entspricht einem speziellen deterministischen bottom-up–Baumautomaten. Dieses Schema kann jedoch erweitert werden um das Erkennen entlang „horizontaler" Kanten, d.h. das Zusammenziehen von zusammengehörigen Anweisungen, die innerhalb desselben Hierarchieblockes stehen (also Kinder desselben Elternknotens sind).

Zusammengehören können solche Anweisungen höchstens, wenn sie durch Datenflußkanten (Querkanten, *cross edges*) miteinander verbunden sind. Die Berechnung dieser Querkanten (d.h. die Bestimmung exakten Datenflusses von Feldern) ist im allgemeinen kein einfaches Problem. In unserem Falle kommen uns jedoch die einfachen Feldzugriffsstrukturen unserer Muster zugute. Darauf werden wir in Abschnitt 6.7 zurückkommen. Im Abschnitt 6.2 beschränken wir uns zunächst auf die „vertikale“ Mustererkennung.

6.1 Vorabtransformationen

Bevor wir die Mustererkennung starten, werden auf den abstrakten Syntaxbaum des LATINUS–Quellprogramms einige wichtige Vorabtransformationen angewendet. Diese Transformationen haben das Ziel, die Zwischendarstellung des Quellprogramms so explizit wie möglich zu machen:

- Prozedur–Expansion (vgl. Abschnitt 3.2.1),
- Konstantenpropagation (vgl. Abschnitt 3.2.2),
- Erkennung und Ersetzung von Induktionsvariablen (vgl. Abschnitt 3.2.3),
- Vereinfachung von Feldern (vgl. Abschnitt 3.2.5),
- Entfernung von totem oder nutzlosem Code (vgl. Abschnitt 3.2.6).

Die Transformationen werden in dieser Reihenfolge genau einmal ausgeführt (vgl. z.B. [WS90]).

6.2 Inkarnationen, Schablonen und Musterhierarchiegraph

Jedes nichttriviale Muster m kann für gewöhnlich auf viele Arten sequentiell implementiert werden. Wir definieren daher:

Definition 6.1 (Inkarnationen eines Musters) Ein Codefragment C heißt *Inkarnation* eines Musters m, falls die durch C berechnete Funktion[1] gleich der in der Musterbibliothek für m definierten Semantik ist. □

Ein Codefragment C kann, wie in Abschnitt 2.2 beschrieben, als ein abstrakter Syntaxbaum dargestellt werden. Ist der C entsprechende Syntaxbaum als Inkarnation eines Musters m erkannt worden, so darf die Wurzel w dieses Syntaxbaumes mit einer C entsprechenden Instanz von m als Attribut $w.matched$ annotiert werden, d.h. C kann durch den der Instanz I entsprechenden Prozeduraufruf von m ersetzt werden, ohne daß sich dadurch die Semantik des Quellprogramms ändert.

Für *triviale Syntaxbäume* mit nur einem Knoten (eine Variable oder Konstante) ist es trivial, festzustellen, ob er eine Inkarnation eines Musters ist. Nehmen wir nun an, der Syntaxbaum habe mehrere Knoten. Sei w die Wurzel dieses Syntaxbaumes. Dann ist der Operator op von w gemäß Abschnitt 2.2 entweder ein `for`–Schleifenkopf, eine bedingte Anweisung, eine Zuweisung oder ein unärer oder binärer Operator in einem Ausdruck. Der bzw. die Söhne $v_1, ..., v_k$ von w stellen entsprechend den Schleifenrumpf, den `then`– bzw. `else`–Teil, die linke bzw. rechte Seite der Zuweisung oder die Operanden dar.

Wir notieren die Semantikfunktion eines Musters m mit f_m.

[1] Die durch ein Programm berechnete Funktion ist gemäß den üblichen semantischen Konventionen für die Ausführung von `FORTRAN`– oder C–Programmen definiert, betrachtet als Abbildung des Speicherinhalts vor der Ausführung auf den Speicherinhalt nach der Ausführung des Programms.

Definition 6.2 (Schablone, Triggermuster, Rumpfmuster, Zielmuster, Untermuster) Sei w die Wurzel des abstrakten Syntaxbaumes T zu einem nichttrivialen Codefragment C. Sei h die von T bzw. C berechnete Funktion. Seien die Söhne $v_1, v_2, ..., v_k$ der Wurzel w allesamt mit Instanzen $I_1, I_2, ..., I_k$ von (ggf. trivialen) Mustern $m_1, m_2, ..., m_k$ aus der Musterbibliothek annotiert. Sei g eine Funktion. Sei $i \in \{1, ..., m\}$.

Das $k+2$–Tupel $S = (g, m_1, ..., m_k, i)$ heißt *Schablone* zu m, falls $g(f_{m_1}, ..., f_{m_k}) = f_m = h$ ist.

m_i heißt *Triggermuster*. i ist gemäß untenstehender Tabelle in Abhängigkeit von *op* eindeutig festgelegt.

Die Muster $m_1, ..., m_k$ nennen wir *Rumpfmuster* von S, m das *Zielmuster* von S.

Ferner nennen wir $m_1, ..., m_k$ (potentielle) *Untermuster* (*subpattern*) von m. □

Schablonen können technisch *realisiert* werden als eine kleine Prozedur, die auf der Basis der vorgegebenen Rumpfmusterinstanzen und den Attributen des zu erkennenden Wurzelknotens untersucht, ob eine Inkarnation des Zielmusters vorliegt, und die zu erfüllenden Bedingungen abtestet. Bei erfolgreichem Test wird eine Instanz des Zielmusters generiert, in die die aus den Rumpfmusterinstanzen und den Wurzelattributen gewonnenen aktualen Parameter eingetragen werden; die Instanz wird zurückgegeben. Sobald die Realisierung der Schablone feststellt, daß eine Bedingung verletzt ist, bricht sie unverzüglich ab und liefert ein negatives Signal (FAIL) zurück.

Ein Muster m hat gewöhnlich mehrere (realisierte) Schablonen. Oft genügt neben dem Operator *op* bereits eines der Rumpfmuster einer Schablone S, um S eindeutig unter allen Schablonen zu m zu identifizieren. Dieses charakteristische Rumpfmuster, das Trigger–Muster m_i, wird wie folgt gewählt:

Knoten w	Knoten mit Trigger–Muster
`for`–Schleifenkopf	Schleifenrumpf (erste Anweisung)
bedingte Anweisung	then–Teil
Zuweisung	rechte Seite
Ausdruck (Operator)	linker oder rechter Unterausdruck

Die für die in Abschnitt 5.3 vorgestellten Muster realisierten Schablonen geben wir im Anhang B an.

Definition 6.3 (Musterhierarchiegraph) Ein *Musterhierarchiegraph* (pattern hierarchy graph, PHG) für eine Menge $\mathcal{M}$ von Mustern m ist ein gerichteter Graph $G = (V, E)$. Die Knotenmenge V umfaßt alle Muster $m \in \mathcal{M}$. Für jede (realisierte) Schablone $S = (g, m_1, ..., m_i, ..., m_k, i)$ zu einem Muster m mit Triggermuster m_i wird eine Kante (m_i, m) in E gezogen. Da mitunter $m_i = m$ vorkommen kann, d.h. ein Muster als Untermuster in einer seiner eigenen Schablonen erscheinen kann, können triviale Zyklen von einem Muster zu sich selbst auftreten. Abgesehen von diesen Selbstzyklen ist ein Musterhierarchiegraph jedoch immer *azyklisch*. □

Wir assoziieren mit jedem Muster m eine Ordnungsnummer *order*, die angibt, wieviel verschachtelte Schleifen eine Inkarnation von m bei einer naheliegenden Implementierung normalerweise (d.h. ohne Blockung von Schleifen) hat. So hat z.B. die Matrix–Vektor–Multiplikation die Ordnung $order(\mathsf{MV}^{(2)}) = 2$ und Matrix–Matrix–Multiplikation die Ordnung $order(\mathsf{MM}^{(3)}) = 3$. Eine Kante (m_i, m) im PHG impliziert $order(m_i) \leq order(m)$.

Ein Musterhierarchiegraph heißt *vollständig* für ein Muster m, falls seine Knotenmenge m und alle dessen (realisierte) Untermuster $m_1, ..., m_k$ enthält und für alle m_j, $1 \leq j \leq k$, vollständig ist.

Folglich gilt: Ist m_i ein Untermuster zu m, so ist der vollständige PHG zu m_i ein Teilgraph des vollständigen PHG zu m.

Ist m_i ein Triggermuster zu m, so nennen wir m ein *Obermuster* zu m_i. Wir bezeichnen mit $OM(m_i)$ die Menge aller Obermuster zu m_i. Gewöhnlich hat ein Muster nur einige wenige Obermuster (vgl. Anhang 6.2).

Sei w die Wurzel des abstrakten Syntaxbaumes T, und seien die Söhne $v_1, v_2, ..., v_k$ der Wurzel w mit Instanzen $I_1, I_2, ..., I_k$ von Mustern $m_1, m_2, ..., m_k$ annotiert. Dann ist die Menge $K(w)$ der Kandidaten für das Muster zum Syntaxbaum mit Wurzel w gerade die Schnittmenge aller Obermuster zu allen m_j, $1 \leq j \leq k$, zu denen überhaupt ein Obermuster im PHG existiert:

$$K(w) = \bigcap_{\substack{1 \leq j \leq k \\ OM(m_j) \neq \emptyset}} \{m : \ (m_j, m) \text{ Kante im } PHG\}$$

Der Mustererkennungsalgorithmus braucht also nur die Schablonen zu den Mustern in $K(w)$ durchzutesten[2].

6.2.1 Komplexitätsbetrachtungen (vertikale Mustererkennung)

An dieser Stelle betrachten wir noch keine Querkanten. Da wir Schablonen bisher immer entlang „vertikaler" Kanten im Syntaxbaum anlegen, sprechen wir hier auch von vertikalen Schablonen bzw. vertikaler Mustererkennung.

Satz 6.1 *Sei $G = (V, E)$ der kleinste vollständige Musterhierarchiegraph für ein Muster $\tilde{m}$. Für alle Muster $m \in V$ bezeichne n_m die Anzahl (vertikaler) Schablonen von m (für triviale Muster wie Variablen oder Konstanten setzen wir $n_m = 1$). Dann ist die Anzahl der durch G verkörperten verschiedenen sequentiellen Implementierungsmöglichkeiten für die Berechnung der Semantikfunktion von $\tilde{m}$ gleich der Anzahl aller Pfade in G, also gleich $\prod_{m \in V} n_m$.*

Der vollständige Musterhierarchiegraph G verkörpert also bei linearer Größe ($\sum_{m \in V} n_m$) exponentiell viele Implementierungsmöglichkeiten für die Semantikfunktion von $\tilde{m}$.

6.2.2 Der Grundalgorithmus zur Mustererkennung (vertikal)

Die Mustererkennung wird mit *stmtdescend*(*main*) aufgerufen und durchmustert den Syntaxbaum wie folgt:

---

function *stmtdescend*(*node*)
if *node* ist keine Zuweisung
then forall Söhne s von *node* (in textueller Reihenfolge) **do** *stmtdescend*(s) **od fi**
forall Ausdrücke e, die in *node* vorkommen **do** *exprdescend*(e) **od**
/⋆ Nun sind alle Unterbäume von *node* besucht und ggf. erkannt ⋆/
forall Obermuster m für *node* im PHG
do teste mit Schablone *match*(m,*node*),
    ob es eine Instanz I von m gibt, die auf *node* paßt **od**
falls ja, annotiere *node* mit I;
end *stmtdescend*()

Die Funktion *exprdescend*() durchmustert die Ausdrucksbäume in analoger Weise.

Jeder Knoten des Syntaxbaumes wird genau einmal besucht. Für jeden erkannten Knoten w gibt es nur eine konstante Anzahl möglicher Obermuster für den Vaterknoten v, die sich aus dem Musterhierarchiegraphen ablesen lassen, und die zugehörigen Schablonen werden — prinzipiell nebenläufig — ausgetestet. Die Mustererkennung ist deterministisch: Entweder liefert genau eine Schablone eine Musterinstanz zurück, d.h. der Knoten wird erkannt, oder alle liefern FAIL, d.h. der Knoten wird nicht erkannt. Jede erfolgreiche Mustererkennung reduziert die Anzahl der inneren Knoten des Syntaxbaumes um mindestens einen Knoten. Da alle Bedingungen einer Schablone in konstanter Zeit überprüft werden können, benötigt der gesamte Algorithmus lineare Laufzeit.

[2] In der Praxis spielt diese Formel kaum eine Rolle, da wir häufig den Fall $k = 1$ und kleine Obermustermengen vorfinden (vgl. Anhang 6.2). Aus Gründen der Vereinfachung vernachlässigen wir das Bilden des Durchschnitts und suchen sequentiell alle Obermustermengen $OM(m_i)$ insgesamt ab.

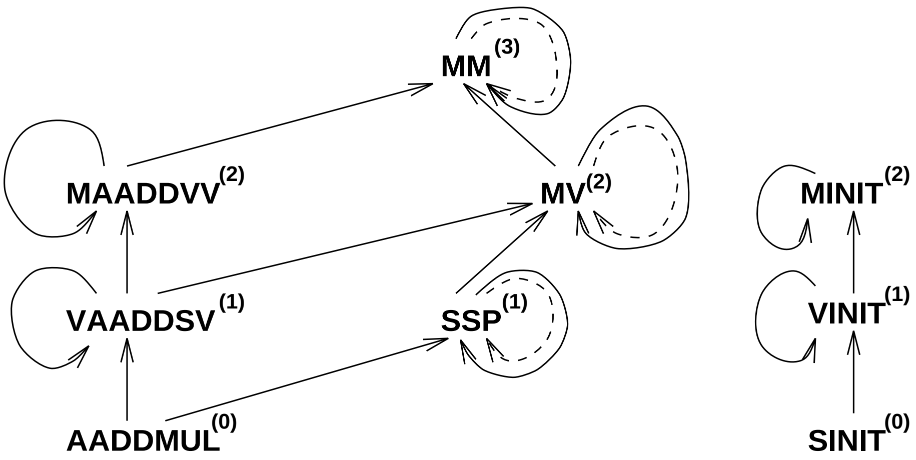


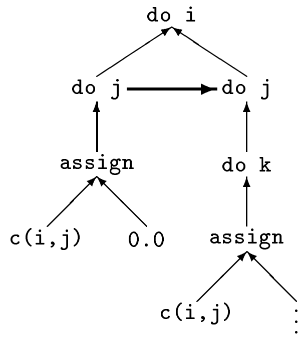
Abbildung 6.1 Der Musterhierarchiegraph von Matrix–Matrix–Multiplikation. Durchgezogene Kanten stehen für Schablonen der vertikalen Mustererkennung, gestrichelte Kanten stehen für Schablonen der horizontalen Mustererkennung (siehe nächste Abschnitte). Durchgezogene Zyklen entsprechen Entblockungsschablonen oder der Eliminierung semantisch invarianter Bedingungen. Der linear große Musterhierarchiegraph verkörpert exponentiell viele mögliche Inkarnationen der Matrix–Matrix–Multiplikation.

6.2.3 Beispiel: Matrix–Matrix–Multiplikation

Wir wollen ein einfaches Beispiel geben. Matrix–Matrix–Multiplikation ist dazu gut geeignet, weil sie nicht ganz trivial ist, ihr Musterhierarchiegraph (Abb. 6.1) aber noch sehr übersichtlich ist und die Semantikfunktionen aller darin vorkommenden Muster allgemein geläufig sind.

Angenommen, der Programmierer habe eine Matrix–Matrix–Multiplikation wie folgt programmiert:

```
do i=1,n
  do j=1,m
S1: c(i,j)=0.0
  enddo
  do j=1,m
    do k=1,r
S2:  c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

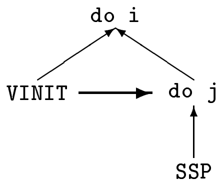


Der Mustererkennungsalgorithmus durchmustert den Syntaxbaum wie beschrieben in *postorder*. Dabei wird zuerst die `S1` entsprechende Zuweisung durch die Instanz `SINIT (c,i,j, 0.0)` ersetzt.

Beim „Wiederaufsteigen" in Richtung Wurzel wird die `do j`–Schleife um `S1` als Inkarnation von VINIT$^{(1)}$ (Vektor–Initialisierung) erkannt. Das Resultat ist die neue Musterinstanz `VINIT( j=[1:m], c[i][1:m], 0.0)`.

Dann durchmustert der Algorithmus die Schleifenschachtelung um `S2` und ersetzt die Zuweisung `S2` durch eine Musterinstanz `AADDMUL(c[i][j],a[i][k],b[k][j],c[i][j])` (akkumulative Addition von Produkten). Diese ergibt, der entsprechenden Kante im PHG folgend, zusammen mit der `do k`–Schleife eine Instanz des Standardskalarproduktes SSP$^{(1)}$. Da der akkumulierende Skalar `c[i][j]` (noch) nicht initialisiert wurde, wird in den entsprechenden Slot (lesend) die akkumulierende Variable eingetragen: `SSP(k=[1:r],c[i][j],a[i][1:r],b[1:r][j], c[i][j])`. `a` und b werden zu Vektoren, da sie in einer Dimension durch die Schleifenvariable k gebunden werden. Das teilweise erkannte Quellprogramm und der entsprechende Syntaxbaum sehen nun so aus (Codeteile „unter" erkannten Knoten werden nicht mehr dargestellt):

```
do i=1,n
  VINIT(j=[1:m], c[i][1:m], 0.0)
  do j=1,m
   SSP(k=[1:r],c[i][j],a[i][1:r],b[1:r][j],c[i][j])
  enddo
enddo
```



Die `do j`–Schleife um die SSP$^{(1)}$-Instanz wird erkannt als Instanz `MV(k=[1:r],j=[1:m],c[i][1:m], b[1:r][1:m], a[i][1:r], c[i][1:m])` (Matrix–Vektor–Multiplikation); auch hier wird in den Initialisierungs–Slot die akkumulierende Variable, der Vektor `c[i][1:r]`, eingetragen.

An dieser Stelle kämen wir nun nicht weiter, würden wir nur vertikale Mustererkennung durchführen. Wir möchten die Initialisierung und die Matrix–Vektor–Multiplikation gerne zusammenfassen, also „horizontale" Mustererkennung betreiben. Der Vektor `c[i][1:r]` wird in der VINIT$^{(1)}$–Instanz geschrieben und in der MV$^{(2)}$–Instanz gelesen und überschrieben. Es fließen also Daten von der VINIT$^{(1)}$–Instanz zur MV$^{(2)}$–Instanz. Wir kennzeichnen dies durch eine sogenannte *Querkante*, die den Datenfluß zwischen verschiedenen Instanzen desselben Blocks angibt. Wir beschreiben in den nächsten Abschnitten, wie solche Querkanten allgemein bestimmt werden.

Entlang dieser Querkante können wir nun — horizontal — die beiden Instanzen zusammenziehen. Dabei wird der Initialisierungs–Slot der MV$^{(2)}$–Instanz mit dem Wert `0.0` aus der VINIT$^{(1)}$–Instanz gefüllt: `MV (k=[1:r], j=[1:m], c[i][1:m], b[1:r][1:m], a[i][1:r], 0.0)`. Diese wiederum kann mit der sie umgebenden `do i`–Schleife in die Instanz `MM(k=[1:r],i=[1:n],j=1:m], c[1:n][1:m],a[1:n][1:r],b[1:r][1:m],0.0)` (Matrix–Matrix–Multiplikation) verwandelt werden, die das gesamte Code–Stück repräsentiert.

Wir waren beim Erkennen den Pfaden SINIT$^{(0)}$...VINIT$^{(1)}$ bzw. AADDMUL$^{(0)}$...SSP$^{(1)}$...MV$^{(2)}$... ...MV$^{(2)}$...MM$^{(3)}$ im Musterhierarchiegraphen gefolgt. Ein Vertauschen der Schleifen oder eine mehr oder weniger intensive Aufgliederung (loop distribution) der Schleifen hätte dazu geführt, daß bei der Mustererkennung andere Pfade im Musterhierarchiegraphen zu MM$^{(3)}$ gewählt worden wären.

6.2.4 Beispiel: Eliminieren semantisch invarianter Bedingungen

Selbstschleifen im Musterhierarchiegraphen können, außer durch horizontales Zusammenziehen von Mustern entlang Querkanten wie im vorangegangenen Beispiel, auch z.B. durch das Eliminieren semantisch invarianter Bedingungen verursacht werden.

Das folgende Beispiel stammt aus der `MATMUL`–Routine des Programms DYFESM aus der Perfect Club Benchmark Suite (vgl. [Ber92]).

```
      DO 400 K = 1, N
        DO 100 I = 1, L
          C(I,K) = 0.0
 100    CONTINUE
        DO 300 J = 1, M
          IF (B(J,K) .NE. 0.) THEN
          DO 200 I = 1, L
            C(I,K) = C(I,K) + A(I,J)*B(J,K)
 200      CONTINUE
 300    CONTINUE
 400  CONTINUE
```

Der Autor dieser Programmzeilen hat die Bedingung `IF (B(J,K).NE.0.0)` eingefügt, um Multiplikationen und Additionen mit 0 zu verhindern. Diese Programmoptimierung ist bezüglich der Semantik des Programms redundant. Daher sollte sie bei der Mustererkennung erstens kein Hindernis darstellen und zweitens eliminiert werden. Dies wird sehr einfach dadurch gelöst, daß eine neue Schablone für das Muster $\mathsf{VAADDSV}^{(1)}$, einer Selbstschleife im Musterhierarchiegraphen entsprechend, eingeführt wird, die eine solche Abfrage erkennt und eliminiert, indem die an die `DO 200`–Schleife annotierte $\mathsf{VAADDSV}^{(1)}$–Instanz einfach an die übergeordnete `IF`–Abfrage unverändert weitergereicht wird. Danach können wir, wie bisher, mit der Mustererkennung fortfahren.

Ein weiteres Beispiel hierzu ist die Eliminierung der Abfrage vor einem $\mathsf{SWAP}^{(0)}$ oder $\mathsf{VSWAP}^{(1)}$, ob die zu vertauschenden Elemente bzw. Vektoren überhaupt verschieden sind (Quellprogramm: in `ludcmp()` aus [PTVF92], bereits teilweise erkannt):

```
if (j != imax)                /* Do we need to interchange rows? */
  VSWAP(k=[1:n], a[imax][:], a[j][:], dum);    /* yes, do so ... */
```

Die Bedingung kann entfallen, da sie semantisch redundant ist. Der Erkennungsprozeß wird danach nicht mehr durch einen unerkannten `if`–Knoten blockiert.

6.2.5 Beispiel: Ausschluß einzelner Werte von Schleifenvariablen

In Vektor– und Matrixoperationen werden des öfteren Bedingungen eingefügt, die die Ausführung dieser Operation für bestimmte Werte der Schleifenvariablen verhindern. Im folgenden Beispiel etwa wird die `ix`–te Iteration der Vektoraddition nicht ausgeführt:

```
for (i=1; i<=n; i++)
  if (i != ix)
    a[i] = b[i] + c[i];
```

Die Bedingung wird zunächst bei der Instanz `ADD(a[i],b[i],c[i])` als zusätzlicher Eintrag vermerkt. In der Prototyp–Implementierung merken wir uns die Bedingung vorläufig noch mit Hilfe eines speziellen Musters $\mathsf{CONDASS}^{(0)}$, das die Bedingung und das bedingte Muster als Argumente hat:

```
for (i=1; i<=n; i++)
  CONDASS( a[i], (i!=xi), ADD(a[i],b[i],c[i]), b[i], c[i] );
```

Dies wird nun mit der `for`–Schleife als eine $\mathsf{VADD}^{(1)}$–Instanz „mit Lücke“ erkannt, wobei der ausgeschlossene Index `ix` im range–Slot vermerkt wird:

```
VADD( i=[1:n]!ix, a[i], b[i], c[i] );
```

6.2.6 Beispiel: Entblocken von Schleifen

Geblockte Schleifen (siehe Abschnitt 3.2) findet man oft in alten Codes, die bereits für andere Zielmaschinen mit Caches (vgl. [WL91]) optimiert wurden.

Im folgenden Programmstück wurde die `do i`–Schleife mit dem Faktor `k` geblockt:

```
      do 50 i = 1,n,k
       do 40 j = i,i+k-1
        dy(j) = dy(j) + da*dx(j)
   40  continue
   50 continue
```

Die Mustererkennung erkennt die innere Schleife als Instanz von VAADDSV$^{(1)}$:

```
      do 50 i = 1,n,k
       VAADDSV(j=[i:i+k-1], dy[i:i+k], da, dx[i:i+k], dy[i:i+k])
   50 continue
```

Um die äußere Schleife zu erkennen, ist ein Selbstzyklus im Musterhierarchiegraphen erforderlich, d.h. der Algorithmus wählt eine Schablone für VAADDSV$^{(1)}$ aus, die aus obiger VAADDSV$^{(1)}$–Instanz und der `do i`–Schleife eine neue Instanz

```
      VAADDSV(i=[1:n], dy[1:n], da, dx[1:n], dy[1:n])
```

generiert[3].

Wir stellen somit fest, daß sich die Schleifentransformation „Entblockung“ einfach als Schablone formulieren läßt, weil sie die Struktur des Syntaxbaumes nicht verändert. Dies ist nicht für alle Schleifentransformationen der Fall; beispielsweise verändert die Schleifenaufgliederung (vgl. Abschnitt 3.2.13) die Struktur des Syntaxbaumes, weil sie neue Knoten generiert.

6.2.7 Beispiel: MULTIADD$^{(0)}$ und MULTIMUL$^{(0)}$

MULTIADD$^{(0)}$ steht für Summen skalarer Werte und Summen von Produkten skalarer Werte mit mehr als zwei Summanden, MULTIMUL$^{(0)}$ steht für Produkte von mehr als zwei skalaren Werten (vgl. Abschnitt 5.3.1).

Triggermuster für MULTIADD$^{(0)}$ sind ADD, ADDMUL$^{(0)}$, MULMUL$^{(0)}$, MULTIMUL$^{(0)}$ und MULTIADD$^{(0)}$, d.h. es gibt einen Selbstzyklus von MULTIADD$^{(0)}$ im Musterhierarchiegraphen. Triggermuster für MULTIMUL$^{(0)}$ sind MUL und MULTIMUL$^{(0)}$, also auch hier ein Selbstzyklus. Das Distributivgesetz wird nicht angewendet. Doppelte Negationen (`-(-a))`) oder Inversionen (`1/1/a`) werden eliminiert.

6.2.8 Beispiel: Differenzensterne (stencils)

Differenzensterne (siehe Abschnitt 5.3.1 wie HSTAR$^{(0)}$ (eindimensional) und STAR$^{(0)}$ (zweidimensional) besitzen als Grundstruktur immer ein ADD bzw. AADD (nur bei HSTAR$^{(0)}$) oder ein MULTIADD$^{(0)}$ (bei HSTAR$^{(0)}$ und STAR$^{(0)}$). Betrachten wir das folgende Beispiel einer Gauß–Seidel–Relaxation (vgl. Livermore Loop 23):

```
  for (j=2; j<=6; j++)
    for (i=2; i<=N; i++)
      ZA[i][j] = ZA[i][j]
               + 0.175 * ( ZA[i][j+1]*ZR[i][j]
                         + ZA[i][j-1]*ZB[i][j]
                         + ZA[i+1][j]*ZU[i][j]
                         + ZA[i-1][j]*ZZ[i][j]
                         - ZA[i][j] );
```

Nach der Erkennung des obersten MULTIADD$^{(0)}$ sieht der Code so aus:

[3]Die Routine `blockreroll()` als Realisierung dieser Schablone kann übrigens auch für alle anderen Vektor– und Matrixoperationen (einschließlich der Reduktionen) benutzt werden, die sich auf diese Weise blocken lassen. Mit anderen Worten, diese Realisierung `blockreroll()` teilen sich mehrere Schablonen.

```
for (j=2; j<=6; j++)
  for (i=2; i<=N; i++)
    ZA[i][j] = ZA[i][j] + 0.175 *
               MULTIADD ( ZA[i][j+1]*ZR[i][j],  ZA[i][j-1]*ZB[i][j],
                          ZA[i+1][j]*ZU[i][j],  ZA[i-1][j]*ZZ[i][j],
                          - ZA[i][j] );
```

Der hier auftretende Differenzenstern ist als *Fünfpunktstern* (five point stencil) bekannt (erster Stern in Abbildung 5.1): Der neue Gitterwert `ZA[i][j]` ergibt sich als Funktion seines alten Wertes und derer seiner direkten Nachbarn. Diese können, wie hier, individuell durch zusätzliche Feldelemente gewichtet werden.

Die Routine `stargazer()` — die Realisierung der Schablone zu $\mathsf{HSTAR}^{(0)}$ und $\mathsf{STAR}^{(0)}$ — verfeinert (konkretisiert) eine gerade erkannte $\mathsf{MULTIADD}^{(0)}$–Instanz bei Vorliegen einer Stern–Struktur zu einer $\mathsf{HSTAR}^{(0)}/\mathsf{STAR}^{(0)}$–Instanz und trägt die anfallenden Parameter ein. `stargazer` wird anschließend noch so oft aufgerufen, wie es weitere, in die optionalen $\mathsf{HSTAR}^{(0)}/\mathsf{STAR}^{(0)}$–Slots passende Operanden in die Instanz eintragen kann (Selbstzyklus bei $\mathsf{HSTAR}^{(0)}/\mathsf{STAR}^{(0)}$ im Musterhierarchiegraphen). Für obiges Beispiel ergibt sich schließlich

```
for (j=2; j<=6; j++)
  for (k=2; k<=N; k++)
     STAR ( ZA[i][j], _ , _ , 0.175000, _ , _ , _ ,
            ZA[i][j], _ , _ ,
            _,             ZB[i][j],      _,
            ZZ[i][j],      4.714286,      ZU[i][j],
            _,             ZR[i][j],      _,
            i,1,1, j,2,1);
```

Nun stellt die Erkennung von Mustern wie $\mathsf{VJACOBI}^{(1)}$, $\mathsf{MJACOBI}^{(2)}$, $\mathsf{VGAUSSSEIDEL}^{(1)}$ oder $\mathsf{MGAUSSSEIDEL}^{(2)}$ beim Betrachten der die $\mathsf{HSTAR}^{(0)}$– bzw. $\mathsf{STAR}^{(0)}$–Instanz umgebenden Schleifen kein Problem mehr dar (Schablonen siehe Anhang B). Da die Variable auf der linken Seite der Zuweisung mit der Stern–Variable `ZA[i][j]` identisch ist, erhalten wir für obiges Beispiel die $\mathsf{MGAUSSSEIDEL}^{(2)}$–Instanz

```
MGAUSSSEIDEL (k=[2:n], j=[2:6], ZA[2:n][2:6], 0.175000, _, _, _,
              ZA[2:n][2:6], _, _,
              _,            ZB[2:n][2:6], _,
              ZZ[2:n][2:6], 4.714286,     ZU[2:n][2:6],
              _,            ZR[2:n][2:6], _,
              k,1,1, j,2,1);
```

6.3 Ein kompakter Zugriffsdeskriptor für Felder

In diesem Abschnitt zeigen wir, wie wir Zugriffe auf Felder kompakt und symbolisch beschreiben. Die beiden darauffolgenden Abschnitte geben an, wie wir damit Datenabhängigkeits- und Datenflußanalyse für Felder ausführen können, und wie man daraus die Querkanten generiert, die für die „horizontale“ Mustererkennung wichtig sind.

6.3.1 Motivation

Wir stellen die Feldelemente, die durch einen von einer oder mehreren verschachtelten Schleifen umgebenen Feldzugriff adressiert werden, dar als einen (diskreten) rechtwinkligen Polyeder im Indexbereich der deklarierten Grenzen dieses Feldes, der alle adressierten Elemente einschließt, erweitert um Schrittweitenangaben, Gebietsanzeiger (area bits) zur genaueren Spezifizierung bei trapezoidalen Zugriffsformen, und Angaben über ausgeschlossene Indexwerte.

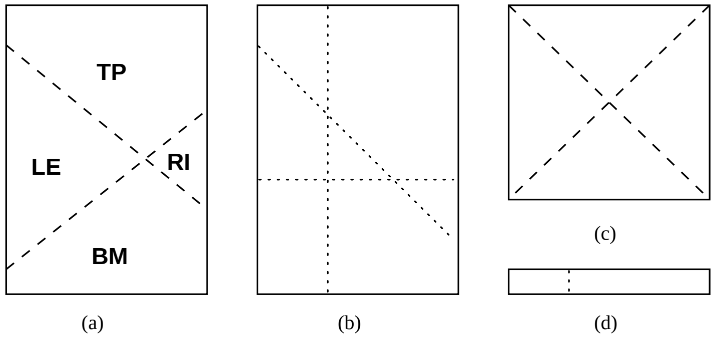


Abbildung 6.2 (a-c): Mögliche Variationen des zweidimensionalen Deskriptors: (a) die vier Gebiete für dreieckige bzw. trapezoidale Matrizen, (b) ausgeschlossene Zeilen, Spalten oder Diagonalen einer Vollmatrix, (c) Diagonalvektor oder Gegendiagonalvektor in einer Diagonalmatrix. (d) zeigt ein ausgeschlossenes Element in einem Vektor (eindimensionaler Deskriptor).

Dieser Deskriptor wird zum einen in unserem zentraler Datenabhängigkeitstest benutzt, zum anderen verwenden wir ihn zur Berechnung von Datenflußkanten (Querkanten).

Unser Deskriptor bietet verschiedene Vorteile: Erstens deckt er fast alle direkten Feldzugriffe in realen Programmen ab; so können z.B. rechteckige, trapezoidale, dreieckige oder diagonale Feldzugriffe meist exakt beschrieben werden[4]. Zweitens sind symbolische Berechnungen und Vergleiche voll integriert, drittens erfordert er minimalen Platz und bietet trotzdem viel Flexibilität zur exakten Beschreibung von Feldzugriffen, und viertens können wichtige Anfrageoperationen wie Gleichheit, Einschluß, Disjunktheit und Nachbarschaft von Deskriptoren in sehr kurzer Zeit berechnet werden.

6.3.2 Definition des Deskriptors

Betrachten wir einen Teilbaum der Syntaxbaum–Darstellung des sequentiellen Quellprogramms. Erinnern wir uns: Eine ganzzahlige Variable heißt *gebunden* (ranging), wenn sie die Schleifenvariable[5] einer in diesem betrachteten Teilbaum vorkommenden Schleife ist. Für ein Vorkommen eines (möglicherweise mehrdimensionalen) Feldes A nennen wir die Dimension d von A *gebunden*, wenn mindestens eine gebundene Variable in A's d-tem Indexausdruck vorkommt. Ein *Skalar* ist entweder eine gewöhnliche skalare Variable oder eine Feldreferenz ohne gebundene Dimensionen. Ein *Vektor* ist eine Feldreferenz mit genau einer gebundenen Dimension. Eine *Matrix* ist eine Feldreferenz mit genau zwei gebundenen Dimensionen. Damit wird ein Diagonalvektor einer Matrix (eigentlich eine eindimensionale Struktur) ebenfalls als (spezielle) Matrix behandelt. Skalare Feldreferenzen, Vektoren, Matrizen und ihre höherdimensionalen Äquivalente (k-dimensionaler Tensor für $k > 2$) sind „Sichtfenster“auf ein Feld.

Sei a eine Referenz des Feldes A. Wir führen nun die verschiedenen Parameter ihres Deskriptors d_a ein.

Der *einhüllende Indexbereich* eines Vektors, einer Matrix usw. ist definiert als der kleinste rechteckige Polyeder (z.B. Rechteck, Quader), der alle adressierten Indexpositionen gebundener Dimensionen in a einschließt (d.h. *nicht* die Werte der Schleifenvariablen, sondern die Werte, die die Indexausdrücke annehmen können). Dieser Polyeder wird dargestellt durch die Grenzen in jeder gebundenen Dimension: $d_a.lb[d]$ bezeichnet die untere Grenze, $d_a.ub[d]$ die obere Grenze und $d_a.st[d]$ die Schrittweite in Dimension d. Untere und obere Grenze werden so gesetzt, daß

[4]Falls die Feldzugriffsform durch unseren Deskriptor nicht exakt dargestellt werden kann, bestimmen wir als Überschätzung nur den (kleinsten) einhüllenden Polyeder; nur im schlimmsten Fall muß ein Zugriff des gesamten deklarierten Indexbereiches angenommen werden.

[5]Andere Induktionsvariablen wurden, wie vorne erwähnt, bereits eliminiert

```
for i from 1 to 4
  for j from i+1 downto 1
    ... A[2*i][k-j] ...
```

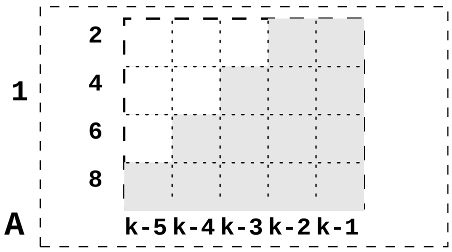


Abbildung 6.3 Deskriptor für eine Feldreferenz. Die vertikale Achse (aufsteigende Werte nach unten) stellt die möglichen Werte dar, die der erste Indexausdruck (`i`) annehmen kann, die horizontale Achse die Werte für den zweiten Indexausdruck (`k-j`). `k` ist eine ungebundene ganzzahlige Variable und wird als symbolische Konstante mitverrechnet. Die schattierten Feldelemente werden adressiert.

die Schrittweite immer positiv ist[6] Die Achsen des Polyeders werden somit in aufsteigender Reihenfolge adressiert; der Deskriptor soll beschreiben, *welche* Feldelemente adressiert werden, nicht aber, in welcher Reihenfolge oder wie oft.

Für den interessantesten Fall zweier gebundener Dimensionen (Matrix) ist der einhüllende Indexbereich ein Rechteck. Abb. 6.3 zeigt ein Beispiel mit einer trapezoidalen Feldzugriffsform.

Für eine trapezoidale Feldzugriffsform ist entweder die *Diagonale* (gerichtet von links oben nach rechts unten) oder die *Gegendiagonale* (von links unten nach rechts oben) des einhüllenden Rechtecks auf natürliche Weise festgelegt, sodaß (1) mindestens eine Zeile und mindestens eine Spalte vollständig adressiert werden, und (2) es genau eine Spalte oder genau eine Zeile gibt, die vollständig adressiert wird.

Diagonale bzw. Gegendiagonale unterteilen das einhüllende Rechteck in zwei trapezoidale Bereiche (siehe Abb. 6.2 (a)), die als Kombination von je zwei benachbarten der vier *Gebiete* TP (top), LE (left), RI (right) und BM (bottom) beschrieben werden können. Um den adressierten trapezoidalen Teil einer Matrix zu beschreiben, braucht man nun nur noch die entsprechenden zwei *Gebietsanzeiger (area bits)* einzuschalten. Im Beispiel von Abb. 6.3, sind die Gebietsanzeiger RI und BM beide 1, und die beiden anderen sind 0.

Sind alle vier Bits gesetzt, so wird die Matrix als ganzes Rechteck adressiert (*voller Zugriff*). Allerdings dürfen Ausnahmen spezifiziert werden: Wenn ein ausgenommener Index $excidx[1]$ angegeben ist, dann wird dieser Zeilenvektor nicht adressiert (Fig. 6.2 (b)). Im folgenden Beispiel

```
do i=1,m
  if i.ne.r
    do j = 1,m
      b(i,j) = b(i,j)-d(i)*b(r,j)
```

aus der Routine "revised simplex" aus [MFL$^+$92], wird der `r`te Zeilenvektor von `b` nicht geschrieben. Daher wird im Deskriptor für das erste Vorkommen von `b` der Ausnahmeindex $excidx[1] = \mathtt{r}$ gesetzt. — Analog konstituiert $excidx[2]$ einen ausgenommenen Spaltenvektor. $excdiag[1]$ bestimmt den Offset[7] für eine ausgeschlossene Diagonale, und entsprechend $excdiag[2]$ for eine ausgeschlossene Gegendiagonale. Im folgenden Beispiel

[6] Ist die Schrittweite eine vorzeichenlose (symbolische) Variable, so wird diese als positiv betrachtet, falls die Untergrenze offensichtlich kleiner als die Obergrenze ist, und als negativ, falls dies offensichtlich umgekehrt ist. Falls hier keine Klarheit geschaffen werden kann, muß ein Zugriff auf den gesamten Indexbereich angenommen werden. Symbolische Schrittweiten unklaren Vorzeichens treten in der Realität jedoch sehr selten auf und kommen in unseren Mustern auch nicht vor, sodaß dieser Fall uns nicht weiter interessiert.

[7] Der Offset ist absolut in bezug auf die vertikalen Achseneinträge.

```
do i=1,m
  do j = 1,m
    if i.ne.j
      b(i,j) = 1.0/a(i,m-j)
```

wird die Diagonale im Deskriptor für die `b`–Referenz ausgeschlossen, und desgleichen die Gegendiagonale für den Deskriptor der `a`–Referenz.

Schließlich gibt es noch den Fall, daß alle vier Gebietsanzeiger 0 sind. Dann gibt das *diag*–Bit an, ob die Diagonale oder die Gegendiagonale adressiert werden.

Alle Deskriptorparameter (mit Ausnahme der Bits natürlich) können auch *symbolische* Ausdrücke sein.

Die Deskriptor–Darstellung ist eindeutig, d.h. sind zwei Deskriptoren verschieden, dann sind auch die von ihnen beschriebenen Zugriffsformen verschieden.

6.3.3 Berechnung des Deskriptors

Seien k verschachtelte Schleifen um eine Feldreferenz wie folgt gegeben:

for i_1 **from** left_1 **to** right_1 **by** step_1 (**except** ix_1)
  for i_2 **from** left_2 **to** right_2 **by** step_2 (**except** ix_2)
    $\ddots$
      for i_k **from** left_k **to** right_k **by** step_k (**except** ix_k)
        ... $A[e_1][e_2]...[e_{dim(A)}]$ ...

wobei alle Schleifengrenzen left_l, right_l, Schleifenschrittweiten step_l, ausgeschlossene Werte ix_l von Schleifenvariablen und Indexausdrücke e_j symbolische Variablen enthalten dürfen. Die Indexausdrücke e_j seien konstante Terme oder mögen linear (aber nicht indirekt[8]) von den Schleifenvariablen i_l abhängen. Für jede Schleife l kann entweder left_l or right_l (aber niemals step_l oder ix_l) eine Schleifenvariable aus $\{i_1, ..., i_{l-1}\}$ enthalten. Wir fassen eine Schleife l zusammen durch den Bereichsausdruck $\mathit{looprange}_l \;=\; i_l = [\mathit{left}_l : \mathit{right}_l : \mathit{step}_l]!xi_l$.

Zur geeigneten Behandlung symbolischer Ausdrücke benutzen wir *symbolische arithmetische Operationen* $+_{sym}$, $-_{sym}$, $*_{sym}$, $\max_{sym}$, $\min_{sym}$ und $|.|_{sym}$ zum symbolischen Addieren, Subtrahieren, Multiplizieren, Maximieren, Minimieren und Bilden des Absolutbetrags, und *symbolische Vergleichsoperationen* $=_{sym}$ (`eqfex`), $\neq_{sym}$, $\geq_{sym}$ (`geqex`), $\leq_{sym}$ (`leqex`), $>_{sym}$, $<_{sym}$, die TRUE liefern, falls die entsprechende Relation durch symbolischen Vergleich bestätigt werden kann, und FALSE sonst. Zur Implementierung dieser symbolischen Operationen benutzen wir einfach die Mustererkennung, die wir auf die Ausdrucksbäume von Schleifengrenzen und Indexausdrücken anwenden.

Mit diesen Basisoperationen können wir die folgenden Operationen definieren: $\mathit{eval}(e, v, e')$ ersetzt alle Vorkommen der Variablen v im Ausdruck e durch eine Kopie des Ausdrucks e' und wertet konstante Teilausdrücke des resultierenden Ausdrucks soweit wie möglich aus. $\mathit{maxeval}_k(e, \mathit{looprange}_1, \ldots, \mathit{looprange}_k)$ liefert den maximalen symbolischen Wert für e, wenn die Schleifenvariablen $i_1, ..., i_k$, die in e vorkommen, entsprechend den Bereichsausdrücken $\mathit{looprange}_1$, ...,$\mathit{looprange}_k$ gebunden sind. $\mathit{mineval}_k(e, \mathit{looprange}_1, \ldots, \mathit{looprange}_k)$ funktioniert analog für symbolische Minimierung. *mineval* und *maxeval* liefern exakte untere bzw. obere Grenzen für den Wert von e, falls (1) die Absolutbeträge der entsprechenden Schleifenschrittweiten symbolisch gleich sind, und falls (2) alle unbekannten symbolischen Variablen oder indirekte Feldreferenzen als nicht–negativ angenommen werden können.

[8] Bei indirekten Feldzugriffen kann in der Regel nicht statisch festgestellt werden, welche Feldelemente referenziert werden, sodaß immer der schlimmstmögliche Fall (Zugriff auf den gesamten deklarierten Indexbereich) angenommen werden muß.

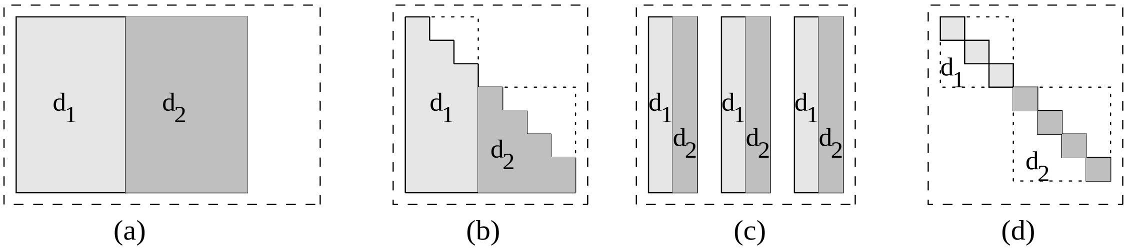


Abbildung 6.4 Beispiele für benachbarte Deskriptoren.

Um das einhüllende Polyeder für eine Feldreferenz zu berechnen, wenden wir *eval* auf Indexausdrücke an, die genau eine gebundene Variable enthalten, und *mineval* bzw. *maxeval* auf Indexausdrücke, die mehr als eine gebundene Variable enthalten.

Die Kombinationen unterer und oberere Indexgrenzen, die von *mineval* bzw. *maxeval* berechnet werden, sind die nicht–linearisierten Äquivalente der Grenzen von Banerjee's Ungleichung (vgl. Abschnitt 3.1.2 sowie [Ban88], S. 50–65 für rechteckige und trapezoidale Zugriffsformen) für linearisierte Indizierungen.

Im ungünstigsten Fall liefern *mineval* und *maxeval* die entsprechenden deklarierten Feldgrenzen, um eine konservative Abschätzung zu erhalten.

Sind Ausnahme–Werte ix_l für einige der Schleifenvariablen l gegeben (technisch als Eintrag im entsprechenden Bereichsausdruck eines range–Slots in der Musterinstanz realisiert, vgl. Abschnitt 6.2.5), so können die entsprechenden „leeren" Zeilen, Spalten usw. ebenfalls mit *eval* berechnet werden.

Für eine Matrix (genau zwei gebundene Variablen), bei der in jedem gebundenen Indexausdruck genau eine gebundene Variable vorkommt, können die Bereichsanzeiger gesetzt werden durch Inspizieren der Schleifengrenzen, der Vorzeichen der Schleifenschrittweiten, der Anordnung der Schleifenvariablen und gebundenen Indexausdrücke und der Vorzeichen der gebundenen Variablen in den gebundenen Indexausdrücken. Falls dieselbe (und einzige) gebundene Variable in zwei Indexausdrücken vorkommt, wird ein „leerer" Matrixdeskriptor generiert, der nur die Diagonale bzw. die Gegendiagonale enthält.

All diese Parameter werden von der Funktion $exprtodescr(a,\ looprange_1, \ldots\ looprange_k)$ gesetzt, die den kompletten Deskriptor für eine Feldreferenz a generiert, gegeben die Schleifenbereichsausdrücke einschließlich auszuschließender Werte von Schleifenvariablen.

6.3.4 Operationen auf Deskriptoren

Das PARAMAT Mustererkennungsverfahren benötigt vier wichtige Operationen auf Deskriptoren:

- $eqdescr(d_1, d_2)$ liefert TRUE falls die Gleichheit der Deskriptoren d_1 und d_2 nachgewiesen werden kann, und FALSE sonst.

  Da die Deskriptordarstellung eindeutig ist, sind zwei Deskriptoren gleich genau dann, wenn alle ihre Parameter (symbolisch) gleich sind. Da die Gebietsanzeiger (area bits) bitweise verglichen werden, läuft *eqdescr* sehr schnell.

- $geqdescr(d_1, d_2)$ liefert TRUE, falls nachgewiesen werden kann, daß alle durch d_2 repräsentierten Feldelemente auch in d_1 enthalten sind, und FALSE sonst.

  Die Implementierung von *geqdescr* ist etwas komplizierter als die von *eqdescr*, aber läuft ebenfalls sehr schnell.Anhang A enthält die Details.

- $disjdescr(d_1, d_2)$ liefert TRUE, falls nachgewiesen werden kann, daß d_1 und d_2 disjunkte Bereiche desselben Feldes adressieren, und FALSE sonst.

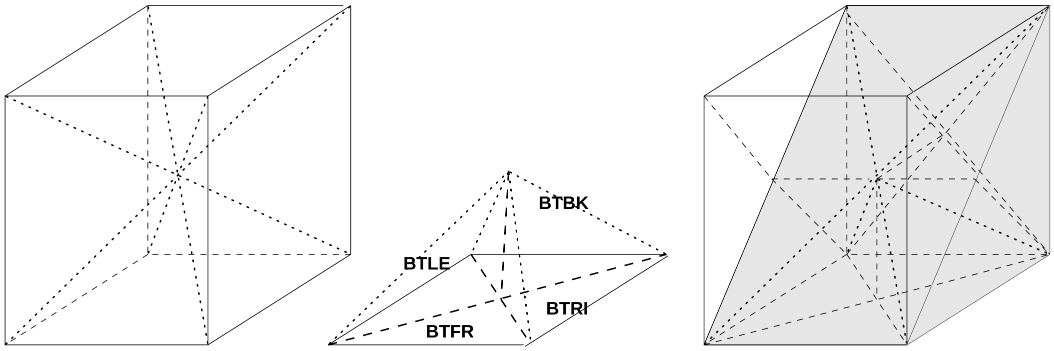


Abbildung 6.5 Für drei gebundene Dimensionen verwandeln sich die vom Zweidimensionalen her bekannten Gebiete in ein Viertelstück von einer der sechs Pyramiden, die durch die inneren Diagonalen getrennt werden. Es werden nun 24 Bits benötigt, wie z.B. BTFR für "bottom pyramid, front part". Für das Prisma rechts sind alle Bits der unteren und der hinteren Pyramide eingeschaltet, und ebenso die Bits LEBM, LEBK, RIBM, RIBK.

Die Details der Berechnung von *disjdescr* befinden sich ebenfalls im Anhang A.

Wir benutzen *disjdescr* als Haupt–Datenabhängigkeitstest in PARAMAT. Die Operation ist konservativ, da Deskriptoren immer eine Obermenge der tatsächlich adressierten Feldelemente beschreiben. Wir weisen ausdrücklich darauf hin, daß wir nur an der Antwort auf die Frage $d_1 \cap d_2 = \emptyset$? interessiert sind, nicht aber an einem Deskriptor zur Schnittmenge $d_1 \cap d_2$ selbst.

- $neighbdescr(d_1, d_2)$ liefert TRUE, falls nachgewiesen werden kann, daß d_1 und d_2 benachbarte Feldbereiche beschreiben, und FALSE sonst.

  Zwei disjunkte Deskriptoren d_1 und d_2 desselben d-dimensionalen Feldes sind *benachbart* falls (1) ihre ungebundenen Indexausdrücke gleich sind, (2) die Achsenabschnitte von d_1 und d_2 benachbart sind für genau eine gebundene Dimension (bei einer Diagonalmatrix: für beide gebundenen Dimensionen), und (3) für alle anderen gebundenen Dimensionen die Achsenabschnitte zusammenpassen wie in Abb. 6.4 dargestellt, d.h. die Schrittweiten stimmen überein, sie haben mindestens eine Grenze gemeinsam, und alle anderen Grenzen passen je nach Situation zusammen. Der Anhang A liefert weitere Details.

  Nachbarschaft von Deskriptoren ist z.B. nützlich für das Erkennen abgerollter Schleifen.

6.3.5 Anwendung des Gebietsanzeigerkonzeptes auf Tensoren höherer Ordnung

Nicht–rechteckige trapezoidale Feldzugriffsformen mit mehr als zwei gebundenen Dimensionen erfordern eine Erweiterung des Gebietsanzeiger–Konzeptes auf höhere Dimensionen. Dies ist, zumindest im Prinzip, möglich, wenn auch ab vier Dimensionen die menschliche Vorstellungskraft versagt. Allerdings sind Tensoren vierter oder höherer Ordnung in realen Programmen extrem selten, insbesondere nachdem die Transformation „Vereinfachung von Feldern" (siehe Abschnitt 3.2.5) ausgeführt wurde.

Für den dreidimensionalen Fall können wir uns noch graphisch veranschaulichen (siehe Fig. 6.5), wie die vier inneren Diagonalen (ähnlich wie im Zweidimensionalen definiert) den Quader in sechs Pyramiden (ggf. mit Sockel) teilen, von denen jede wiederum aus vier Teilen besteht, die durch die Diagonalen der Grundfläche abgeteilt werden, wie schon vom Zweidimensionalen her bekannt. Somit genügen 24 Bit, um diese Gebiete zu codieren. Wir können sie zusammensetzen, um damit Prismen, Pyramiden oder Tetraeder zu beschreiben. Desweiteren können wir auch Diagonalflächen oder Innendiagonalvektoren selektieren.

Da der gegenwärtige Stand der Musterbibliothek (noch) keine Tensoren von höherer Ordnung als zwei vorsieht, haben wir bislang nur den zweidimensionalen Fall implementiert.

6.3.6 Andere Arbeiten über Deskriptoren

Deskriptoren für Feldzugriffe wurden bereits in verschiedenen früheren Arbeiten entwickelt.

[TIF86] gibt eine allgemeine, aber kaum praktikable Methode, die beliebige konvexe Zugriffsformen zuläßt, die durch eine Menge linearer Ungleichungen begrenzt werden können.

[CK87] definiert den *Regular Section Descriptor (RSD)*, der einzelne Elemente, komplette Zeilen, Spalten und Diagonalen oder deren höherdimensionale Äquivalente beschreiben kann. Der RSD kann keine dreieckigen oder trapezoidalen Formen beschreiben und kann auch keine ausgeschlossenen Elemente oder Vektoren betrachten. [HK91] erweitert den RSD durch Zulassung von Schrittweiten größer als 1. Ein RSD für ein d-dimensionales Feld kann in Platz $O(d)$ gespeichert werden.

[GS90] beschränkt Feldzugriffe ebenfalls auf rechteckige Polyeder, um effiziente Operationen für Vereinigung, Durchschnitt und Differenz von Deskriptoren zu erhalten. Außerdem werden die zulässigen Indexausdrücke stark eingeschränkt und symbolische Ausdrücke nicht zugelassen.

[BK89] erweitert den RSD aus [CK87] zum *Data Access Descriptor (DAD)*, indem beliebige konvexe Polyeder zugelassen werden, die entweder begrenzt werden durch eine Linie (im Zweidimensionalen) oder Ebene (im Dreidimensionalen), die orthogonal zu einer Feldachse ist, oder aber durch eine Linie oder Ebene in einem 45-Grad-Winkel zu einem Achsenpaar. Für ein d-dimensionales Feld wird Platz $O(2d^2)$ benötigt, um diese Grenzen zu speichern. Ferner wird eine Beschreibung der inneren (sequentiellen) Struktur des Feldzugriffs eingefügt. Schleifenschrittweiten müssen gleich 1 sein. Symbolische Ausdrücke werden nur in begrenztem Maße unterstützt. [HHL90] liefert einen Algorithmus, der herausfindet, ob die Vereinigung zweier DAD's exakt ist oder nicht, und erweitert den DAD, um schleifengetragene Abhängigkeiten, Richtungsvektoren und Feld-Flußinformation (array kills) zu berechnen.

Unser Deskriptor ist dahingehend einfacher, daß nur *eine* 45-Grad-Grenze zugelassen ist, während die übrigen orthogonal sein müssen. Wir sehen allerdings, daß „Parallelogramme" oder „Achtecke" in realen Programmen sehr selten auftreten[9]. In unserer Musterbibliothek treten sie nicht auf.

Auf der anderen Seite kann unser Deskriptor ausgenommene Zeilen, Spalten und Diagonalen in Matrizen spezifizieren, was in [BK89] — wie in allen anderen Arbeiten — nicht unterstützt wird.

Ferner erlaubt unser Deskriptor (fast) beliebige, sogar symbolische Schrittweitenausdrücke, Schleifenschrittweiten und Indexausdrücke in jeder Dimension. Die teuren Operationen für Durchschnitt und Vereinigung aus [BK89] werden für unsere Zwecke nicht benötigt: Vereinigung und Differenz von Deskriptoren werden explizit durch die Mustererkennung behandelt, und betreffs des Durchschnitts sind wir nur daran interessiert, ob er (beweisbar) leer ist oder nicht, aber nicht, wie er aussieht. Ferner brauchen wir auch keine Information darüber, in welcher Reihenfolge oder wie oft die Feldelemente adressiert werden. Das Gebietsanzeigerkonzept liefert uns eine kompakte Darstellung von Zugriffsformen mit einem Platzbedarf von $O(d)$ und ermöglicht schnelle Vergleiche von Deskriptoren.

[HN93] beschränkt den DAD wiederum auf rechteckige Polyeder, benötigt jedoch Platz $O(kn)$ für den "augmented DAD" (k war die Tiefe der Schleifenschachtelung), beschränkt Schleifenschrittweiten auf 1 und verbietet symbolische Ausdrücke; dies alles nur mit dem Ziel, den Deskriptor mit Alignment–Information zu konnotieren.

Allgemeinere Arbeiten ([Fea91], [MAL93]) über exakte Feld–Datenflußanalyse konzentrieren sich ebenfalls mehr auf die sequentiell–zeitliche Struktur der Feldzugriffe, um den letzten schreibenden Zugriff für jedes Feldelement in einer Schleife zu bestimmen. Diese Information ist für unser System nicht erforderlich, da sie implizit in den Musternamen enthalten ist.

[9] Da wir Durchschnitt und Vereinigung nicht berechnen müssen, entfällt auch diese Quelle nicht–trapezoidaler Zugriffsformen.

6.4 Berechnung von Datenfluß–Querkanten (cross edges)

Querkanten im Syntaxbaum zeigen mögliche Datenabhängigkeiten bzw. Datenfluß an. Sie verbinden im Syntaxbaum Codeteile, die zum selben Berechnungsstrang (thread) gehören, auch wenn sie textuell durch andere, von diesem Strang unabhängige Codeteile getrennt sind. Querkanten sind von existentieller Bedeutung für die Mustererkennung.

Um Querkanten berechnen zu können, benötigen wir eine Datenflußanalyse, die so exakt wie möglich arbeitet. Obwohl das allgemeine Problem der exakten Feld–Datenflußanalyse schwierig ist, (vgl. [Fea91, MAL93]), wird es hier doch sehr vereinfacht, weil unsere Muster die Feldelemente in einer sehr regelmäßigen Struktur adressieren.

Wir berechnen Deskriptoren und Datenfluß erst unmittelbar, bevor sie für die Berechnung der Querkanten benötigt werden, weil erst dann die maximal mögliche Information durch die Musterinstanzen verfügbar ist, und weil so immer nur eine Schleifenebene auf einmal betrachtet werden muß.

Für unerkannte Anweisungen bzw. Codeteile müssen worst–case–Annahmen gemacht werden (d.h. der gesamte Extent einer gebundenen Dimension könnte adressiert werden); für erkannte Codeteile ist die durch den Deskriptor gelieferte Zugriffsinformation aufgrund der regelmäßigen Struktur der Muster fast immer exakt.

Jeder Slot einer Musterinstanz hat standardmäßig einen von vier möglichen Zugriffsmodi: I (ignore), R (read), W (write), RW (read and overwrite).

Eine *Querkante* verbindet zwei Instanzen I_1, I_2, die zum selben Block gehören. Nehmen wir an, I_1 stehe textuell vor I_2. Wir unterscheiden fünf verschiedene Arten von Querkanten:

1. FLOW: Falls I_1 den Deskriptor d_1 schreibt und I_2 den Deskriptor d_2 liest, ferner $geqdescr(d_1, d_2)$ gilt und alle Deskriptoren d_3 desselben Feldes, die von Instanzen zwischen I_1 und I_2 geschrieben werden, $disjdescr(d_3, d_2)$, erfüllen, dann wird eine Querkante vom Typ FLOW von I_1 nach I_2 gezogen. Dies entspricht einer schleifenunabhängigen echten Datenabhängigkeit von I_1 nach I_2.

2. ANTI: Falls I_1 den Deskriptor d_1 liest und I_2 den Deskriptor d_2 schreibt, ferner $geqdescr(d_1, d_2)$ gilt und alle Deskriptoren d_3 desselben Feldes, die von Instanzen zwischen I_1 und I_2 geschrieben werden, $disjdescr(d_3, d_2)$, erfüllen, dann wird eine Querkante vom Typ ANTI von I_1 nach I_2 gezogen. Dies entspricht einer schleifenunabhängigen Anti–Datenabhängigkeit von I_1 nach I_2.

3. INPUT: Falls I_1 den Deskriptor d_1 liest und I_2 den Deskriptor d_2 liest, ferner $geqdescr(d_1, d_2)$ oder $geqdescr(d_2, d_1)$ gilt und alle Deskriptoren d_3 desselben Feldes, die von Instanzen zwischen I_1 und I_2 geschrieben werden, $disjdescr(d_3, d_1)$ und $disjdescr(d_3, d_2)$ erfüllen, dann wird eine Querkante vom Typ INPUT von I_1 nach I_2 gezogen.

4. DISJOUT: Falls I_1 den Deskriptor d_1 schreibt und I_2 den Deskriptor d_2 schreibt, ferner $neighbdescr(d_1, d_2)$ gilt und alle Deskriptoren d_3 desselben Feldes, die von Instanzen zwischen I_1 und I_2 geschrieben werden, $disjdescr(d_3, d_1)$ und $disjdescr(d_3, d_2)$ erfüllen, dann wird eine Querkante vom Typ DISJOUT von I_1 nach I_2 gezogen.

5. DISJIN: Falls I_1 den Deskriptor d_1 liest und I_2 den Deskriptor d_2 liest, ferner $neighbdescr(d_1, d_2)$ gilt und alle Deskriptoren d_3 desselben Feldes, die von Instanzen zwischen I_1 und I_2 geschrieben werden, $disjdescr(d_3, d_1)$ und $disjdescr(d_3, d_2)$ erfüllen, dann wird eine Querkante vom Typ DISJIN von I_1 nach I_2 gezogen.

Im allgemeinen bilden die Querkanten eines Blocks einen gerichteten azyklischen Graphen.

Nur Musterinstanzen, die durch Querkanten verbunden sind, kommen für eine etwaige Verschmelzung in Betracht. Die Auswahl der infrage kommenden Schablonen für die Mustererkennung entlang von Querkanten wird durch den Typ der Querkante und die Namen der beteiligten Musterinstanzen gesteuert. Triggermuster ist immer das Muster von I_2. Kommen mehrere Schablonen in Frage, so ist — wie beim vertikalen Fall — gewährleistet, daß höchstens eine zu einer erfolgreichen Mustererkennung führen kann, weil auch hier die Mustererkennung deterministisch ist (siehe Musterhierarchiegraph für horizontale Mustererkennung, Anhang B.6.2). Bei mehreren in eine Musterinstanz eingehenden Querkanten werden alle Quer–Vorgänger in umgekehrter textueller Reihenfolge getestet[10].

Paßt eine Schablone für die Querkante, so werden die Instanzen I_1 und I_2 entfernt und eine Instanz des neuerkannten Musters generiert, die — außer in wenigen Ausnahmefällen — an der Stelle von I_2 eingefügt wird. Die Behandlung des vorher mit I_1 annotierten Knotens ist abhängig von der Schablone; verschiedene Möglichkeiten dazu werden in Abschnitt 6.6 erläutert. Gegebenenfalls muß der zu I_1 gehörende Knoten deaktiviert oder I_1 leicht modifiziert werden. Unmittelbar danach müssen auch die Querkantenrelationen zu und von I_1 bzw. I_2 aktualisiert werden. Trotz dieser (leichten) Modifikationen im Syntaxbaum können wir Schablonen entlang Querkanten in den bisherigen Mustererkennungsalgorithmus integrieren, da sie nicht in Konflikt mit den Schablonen zur vertikalen Mustererkennung stehen können.

Sind die die Querkante verursachenden Deskriptoren d_1 (zu I_1) und d_2 (zu I_2) nicht vollständig gleich, so muß eventuell eine Rest–Instanz von I_1 erhalten bleiben, um die Semantik des Programms nicht zu verändern. Dazu das folgende Beispiel:

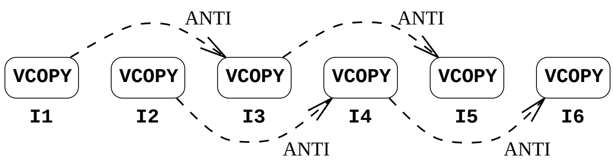
```
I1:  VINIT(j=[1:u2], y[i][1:u2], 0.0);
I2:  V2CONV(j=[1:u2-1],l,k, y[i][1:u2-1], y[i][1:u2-1],
                            -a[:][:], x[i-:][:-:]);
```

kann entlang der Querkante (Typ FLOW), die durch `y[i][1:u2-1]` verursacht wird, zusammengezogen werden[11] Da `y[i][u2]` nicht in `I2` überschrieben wird, muß eine Rest–Initialisierung eingefügt werden:

```
I1':  SINIT(y[i][u2], 0.0);
I2':  V2CONV(j=[1:u2-1], l, k, y[i][:], 0.0, -a[:][:], x[i-:][:-:]);
```

Querkanten vom Typ ANTI sind z.B. nützlich beim Erkennen von $\mathsf{VSWAP}^{(1)}$ aus einzelnen $\mathsf{VCOPY}^{(1)}$ (vector copy) – Instanzen:

```
I1: VCOPY(i=[1:n],t1[:], a[:]);
I2: VCOPY(i=[1:n],t2[:], b[:]);
I3: VCOPY(i=[1:n], a[:], c[:]);
I4: VCOPY(i=[1:n], b[:], d[:]);
I5: VCOPY(i=[1:n], c[:],t1[:]);
I6: VCOPY(i=[1:n], d[:],t2[:]);
```



Instanzen, die zum gleichen $\mathsf{VSWAP}^{(1)}$–Strang gehören könnten, sind durch ANTI–Querkanten verkettet.

Eine $\mathsf{VSWAP}^{(1)}$–Instanz kann aus drei $\mathsf{VCOPY}^{(1)}$–Instanzen generiert werden. Die Vermischung der einzelnen Instanzen stört in keiner Weise den Erkennungsprozeß, der ja durch die Querkanten gesteuert wird. Es ergibt sich

```
I1': VSWAP(i=[1:n], a[:], c[:], t1[:]);
I2': VSWAP(i=[1:n], b[:], d[:], t2[:]);
```

Querkanten vom Typ INPUT treten z.B. auf beim Erkennen von Gauß–Elimination mit einem Lösungsvektor. Man betrachte folgendes teilweise erkannte Code–Fragment:

[10] Dies könnte theoretisch auch parallel erfolgen.

[11] Hierzu brauchen wir jedoch die Bestimmung der Differenz von Deskriptoren. In Spezialfällen wie diesem (eindimensionale oder rechteckige Zugriffsform) kann der Differenzdeskriptor leicht bestimmt werden: `y[i][u2]`. Die Berechnung des Differenzdeskriptors für kompliziertere Zugriffsformen ist bisher nicht implementiert und Gegenstand zukünftiger Entwicklung.

```
I1: VAADDSV(j=[k+1:n], a[i][:], a[i][:], -a[i][k], a[k][:]);
I2: AADDMUL(b[i], b[i], -a[i][k], b[k]);
```

Da sowohl `I1` als auch `I2` das Feldelement `a[i][k]` lesen, werden sie durch eine Querkante vom Typ INPUT verbunden.

Querkanten vom Typ DISJIN und DISJOUT sind nützlich beim Erkennen abgerollter Schleifen. Dies wird im Abschnitt 6.6.1 genauer besprochen.

6.5 Algorithmus (vertikale und horizontale Mustererkennung)

Mit Querkanten formuliert sich die Funktion *stmtdescend*() nun wie folgt:

---

function *stmtdescend*(*node*)
if *node* ist keine Zuweisung
then forall Söhne s von *node* (in textueller Reihenfolge) **do** *stmtdescend*(s) **od fi**
forall Ausdrücke e, die in *node* vorkommen **do** *exprdescend*(e) **od**
/⋆ Nun sind alle Unterbäume von *node* besucht und ggf. erkannt ⋆/
forall Obermuster m für *node* im PHG
do teste mit Schablone *match*(m,*node*),
    ob es eine Instanz I von m gibt, die auf *node* paßt **od**
falls ja, annotiere *node* mit I;
berechne Deskriptoren und Querkanten zu *node*
repeat
    forall direkten Quer–Vorgänger x von *node* im selben Block
    do /⋆ x war bereits vor *node* besucht worden ⋆/
        teste mit Schablone *merge*(x,*node*), ob die Sequenz von x und *node*
         eine Inkarnation eines Musters m' ist
        falls ja, kontrahiere x und *node* zu *node*
         und annotiere *node* mit einer Instanz I' von m'; **break**;
    od
until es keine Quervorgänger von *node* mehr gibt, die sich über Querkanten mit *node*
        verschmelzen ließen.
end *stmtdescend*()

Die Funktion *exprdescend*() durchmustert die Ausdrucksbäume in analoger Weise, wobei bei einzelnen Ausdrücken jedoch keine Querkanten auftreten können.

6.5.1 Komplexitätsbetrachtungen (vertikale und horizontale Mustererkennung)

Satz 6.2 *Sei $G = (V, E)$ der kleinste vollständige Musterhierarchiegraph für ein Muster $\tilde{m}$ einschließlich der Kanten zu den Schablonen für die Mustererkennung entlang von Querkanten. Für alle Muster $m \in V$ bezeichne n_m die Anzahl vertikaler Schablonen von m (für triviale Muster wie Variablen oder Konstanten setzen wir $n_m = 1$), und c_m die Anzahl der horizontalen Schablonen plus eins. Dann ist die Anzahl der durch G verkörperten verschiedenen sequentiellen Implementierungsmöglichkeiten für die Berechnung der Semantikfunktion von $\tilde{m}$ gleich der Anzahl aller Pfade in G, also gleich $\prod_{m \in V}(n_m \cdot c_m)$.*

Der Musterhierarchiegraph verkörpert also noch immer bei linearer Größe ($\sum_m n_m + \sum_m c_m$) exponentiell viele Implementierungsmöglichkeiten für die Semantikfunktion.

6.6 Spezielle Schablonen bei Querkanten

Mustererkennung entlang von Querkanten kann verschiedene Formen ausbilden:

- **Absorption/Verschmelzung**: Sie tritt auf, wenn eine Variable von einer Instanz geschrieben und von einer nachfolgenden Instanz gelesen und überschrieben wird, z.B. bei Initialisierungen:

```
SINIT( s, 0.0 );
...        /* kein Schreib- oder Lesezugriff auf s */
VSUM( i, s, a[:], s );
```

  Schreiben wir den Wert `0.0` statt der gelesenen Variablen `s` in den Initialisierungsslot der $\mathsf{VSUM}^{(1)}$–Instanz, so kann die $\mathsf{SINIT}^{(0)}$–Instanz entfallen:

```
VSUM( i, s, a[:], 0.0 );
```

  Dies wird von einer Schablone durchgeführt, die einem Selbstzyklus von $\mathsf{VSUM}^{(1)}$ im Musterhierarchiegraphen entspricht.

  Im Falle eines Lesezugriffs auf `s` an der Stelle `...` müssen wir hier allerdings die $\mathsf{SINIT}^{(0)}$–Instanz (vorläufig) stehenlassen. Dies entspricht dem nächsten Verwendungszweck von Querkanten:

- **Inferenz**: Sie tritt auf, wenn eine Variable von einer Instanz geschrieben und von einer nachfolgenden Instanz gelesen, aber nicht überschrieben wird. Insbesondere beim Vorkommen temporärer Variablen im Quellprogramm kann so die nachfolgende Musterinstanz präzisiert werden, wie etwa im folgenden Beispiel:

```
MUL( temp, a, b );
ADD( d, temp, c );
```

  kann entlang der durch `temp` verursachten Querkante — durch eine weitere Schablone zu $\mathsf{ADDMUL}^{(0)}$ — transformiert werden zu

```
MUL( temp, a, b );
ADDMUL( d, c, a, b );
```

  sofern `a` und `b` dazwischen nicht neu geschrieben werden. Später kann gegebenenfalls die MUL–Instanz als nutzloser Code entfernt werden, wenn `temp` nachfolgend nicht mehr gelesen wird.

Weitere spezielle Schablonen zu Querkanten werden im folgenden Abschnitt vorgestellt.

6.6.1 Aufrollen von Schleifen

In Altprogrammen für skalare oder Vektorarchitekturen wurden oft Schleifen abgerollt, um eine bessere Leistung zu erzielen (vgl. Abschnitt 3.2.11). Beispielsweise enthalten fast alle BLAS1–Routinen (siehe Abschnitt 5.4.4) abgerollte Schleifen.

Abgerollte Schleifen erschweren nicht nur dem Benutzer die Lesbarkeit des Programms, sondern erfordern auch besondere Maßnahmen bei der Mustererkennung: Das Wieder–Aufrollen der Schleifen.

Das Abrollen von Schleifen kann auf drei Ebenen auftreten: (1) als Replikation auf Ausdrucksebene (im selben Ausdruck), (2) als Replikation auf Anweisungsebene (verschiedene Anweisungen im selben Block), oder (3) als Blocken einer Schleife (auf Schleifenebene), das bereits in Abschnitt 6.2.6 besprochen wurde.

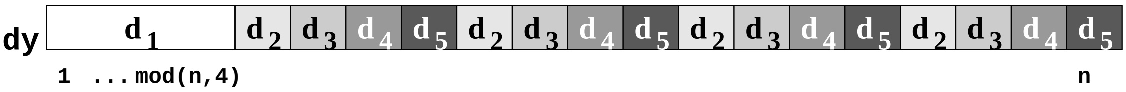


Abbildung 6.6 Deskriptoren des Feldes `dy`, geschrieben durch `I1`, `I2`, `I3`, `I4`, `I5`

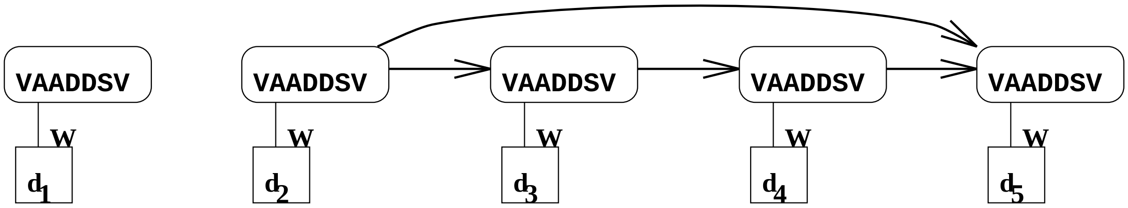


Abbildung 6.7 Querkanten des Typs DISJOUT, bedingt durch `dy`. Die durch `da` verursachten INPUT–Querkanten wurden nicht eingezeichnet.

Aufrollen von Schleifen auf Anweisungsebene

Als Beispiel betrachte man die folgende, 4-fach abgerollte `daxpy`–Routine (`mod(n,4)` wird als symbolischer Ausdruck behandelt):

```
      do 30 j = 1,mod(n,4)
        dy(j) = dy(j) + da*dx(j)
   30 continue
      do 50 i = mod(n,4)+1,n,4
        dy(i)   = dy(i)   + da*dx(i)
        dy(i+1) = dy(i+1) + da*dx(i+1)
        dy(i+2) = dy(i+2) + da*dx(i+2)
        dy(i+3) = dy(i+3) + da*dx(i+3)
   50 continue
```

Um dies als Vorkommen von VAADDSV$^{(1)}$ zu erkennen, muß die Schleife aufgerollt werden. Schleifenverteilung und „vertikale" Mustererkennung erzeugen die Instanzen

```
I1:   VAADDSV(j=[1:mod(n,4)], dy,da,dx)
I2:   VAADDSV(i=[mod(n,4)+1:n:4], dy,da,dx)
I3:   VAADDSV(i=[mod(n,4)+2:n:4], dy,da,dx)
I4:   VAADDSV(i=[mod(n,4)+3:n:4], dy,da,dx)
I5:   VAADDSV(i=[mod(n,4)+4:n:4], dy,da,dx)
```

Für jeden Slot der Musterinstanz wird ein Deskriptor berechnet (siehe Abb. 6.6 für die Slots Nr. 1), und die Querkanten werden erzeugt. Es gibt nun vier Querkanten des Typs DISJOUT: von `I2` nach `I3` und `I5`, von `I3` nach `I4` und von `I4` nach `I5` (siehe Abb. 6.7). Diese Querkanten geben die Nachbarschaftsbeziehungen zwischen den Instanzen wieder und veranlassen die Mustererkennung, die Realisierung `stmtreroll` einer Aufroll–Schablone von VAADDSV$^{(1)}$ aufzurufen. Diese Routine ist die gemeinsame Realisierung aller Abroll–Schablonen von Mustern der Ordnung 1, die auf der Anweisungsebene abgerollt werden können. Sie sammelt alle Instanzen, die in dieser DISJOUT–Zusammenhangskomponente (`I2,I3,I4,I5`) liegen und verschmilzt diese[12] zu

```
I1:   VAADDSV(j=[1:mod(n,4)], dy,da,dx)
I2':  VAADDSV(i'=[mod(n,4)+1:n], dy,da,dx)
```

[12]Das Verschmelzen wird nicht verhindert durch etwaige textuell dazwischenliegende Instanzen, die nicht `dy[mod(n,4)+1:n]` beschreiben können.

Für die neue Instanz `I2'` werden wiederum Deskriptoren berechnet, und nun erhalten wir eine neue Querkante des Typs DISJOUT von `I1` nach `I2'`. Deren Verschmelzung liefert schließlich die gewünschte Instanz `VAADDSV(j'=[1:n],dy,da,dx)`.

Falls Reduktionen auf Vektoren (z.B. die BLAS–Routinen `ddot`, `dasum`, `idamax`) oder Matrizen auf der Anweisungsebene abgerollt wurden, sind die teilweise erkannten Instanzen durch Querkanten des Typs DISJIN verbunden:

```
I1:  VSUM( i=[1:n:2], s, s, b[i]);
I2:  VSUM( i=[2:n:2], s, s, b[i]);
```

Hier wird der Verschmelzungsprozeß `redstmtreroll` durch die DISJIN–Querkante von `I1` nach `I2` gesteuert.

Aufrollen von Schleifen auf Ausdrucksebene

Als Beispiel betrachte man folgenden Ausschnitt aus der BLAS1–Routine `dasum`, die um den Faktor 6 auf Ausdrucksebene abgerollt worden ist (`mod(n,6)` wird wiederum als symbolischer Ausdruck behandelt):

```
    dtemp = 0.0d0
    do 30 i = 1,mod(n,6)
      dtemp = dtemp + dabs(dx(i))
 30 continue
    do 50 i = mod(n,6) + 1, n, 6
      dtemp = dtemp + dabs(dx(i))   + dabs(dx(i+1))
   *                + dabs(dx(i+2)) + dabs(dx(i+3))
   *                + dabs(dx(i+4)) + dabs(dx(i+5))
 50 continue
```

Der Rumpf der `do 50`–Schleife läßt sich als Instanz von $\mathsf{MULTIADD}^{(0)}$wie folgt schreiben:

```
    do 50 i = mod(n,6) + 1, n, 6
      MULTIADD ( dtemp, dtemp, abs(dx[i]), abs(dx[i+1]), abs(dx[i+2]),
                 abs(dx[i+3]), abs(dx[i+4]), abs(dx[i+5]))
 50 continue
```

Nun wollen wir die `do 50`–Schleife als Inkarnation von $\mathsf{VSUM}^{(1)}$ erkennen. Dazu muß zunächst die innere Schleife, durch Replikation in der $\mathsf{MULTIADD}^{(0)}$–Instanz versteckt, rekonstruiert werden. Dies erledigt die Routine `exprcollect()`, eine Realisierung, die sich die Aufrollschablonen fast aller Reduktionsmuster ($\mathsf{VSUM}^{(1)}$, $\mathsf{VPROD}^{(1)}$, $\mathsf{SSP}^{(1)}$) teilen.

`exprcollect` sortiert die Feldoperanden nach lexikalisch aufsteigenden Indizierungen. Damit läßt sich einfach testen, ob eine versteckte, abgerollte Schleife vorliegt. In unserem Beispiel produziert `exprcollect` folgendes:

```
      do 50 i = mod(n,6) + 1, n, 6
       do 51 t = i,i+5
        VSUM (t=[i:i+5], dtemp, abs(dx[i:i+5]), dtemp);
   51 continue
   50 continue
```

was anschließend, wie in Abschnitt 6.2.6 gezeigt, entblockt wird. Zusammen mit der Erkennung der `do 30`–Schleife ergibt sich schließlich

```
      VSUM(i=[1:mod(n,6)], dtemp,abs(dx[1:mod(n,6)]), 0.0)
      VSUM(i=[mod(n,6)+1,n], dtemp, abs(dx[mod(n,6)+1:n]), dtemp)
```

wobei von der ersten $\mathsf{VSUM}^{(1)}$–Instanz zur zweiten eine Querkante vom Typ FLOW, durch `dtemp` verursacht, und eine Querkante vom Typ DISJIN, durch `dx[]` verursacht, verläuft. Entlang letzterer Querkante können nun, mit Hilfe von `redstmtreroll()` aus dem vorigen Teilabschnitt (gemeinsame Realisierung der Aufroll–Schablonen, gemäß Selbstzyklen von $\mathsf{VSUM}^{(1)}$, $\mathsf{VPROD}^{(1)}$, $\mathsf{SSP}^{(1)}$ usw. im Musterhierarchiegraphen) die beiden Instanzen zu einer einzigen zusammengezogen werden, was einer Schleifenverschmelzung entspricht.

6.6.2 Umbenennen von Ergebnissen

Häufig werden in sequentiellen Implementierungen bei Reduktionen (z.B. $\mathsf{VSUM}^{(1)}$, $\mathsf{SSP}^{(1)}$, $\mathsf{MV}^{(2)}$, ...) temporäre Variablen, Vektoren oder Matrizen als akkumulierende Variablen eingeführt, um dadurch zahlreiche indizierte (bzw. „indiziertere") Feldzugriffe einzusparen, was natürlich der Laufzeit zugutekommt. Für $\mathsf{MV}^{(2)}$ etwa sieht das typischerweise so aus:

```
DO J=1,M
  S = 0.0
  DO I=1,N
    S = S + A[I][J] * B[I]
  ENDDO
  X[J] = S
ENDDO
```

Die innere Schleife wird als Inkarnation von $\mathsf{SSP}^{(1)}$ erkannt und die resultierende Instanz anschließend mit der davorstehenden Initialisierung verschmolzen:

```
DO J=1,M
  SSP( I=[1:N], S, A[:][J], B[:], 0.0 );
  SCOPY( X[J], S );
ENDDO
```

Über eine weitere Schablone, die wiederum einem Selbstzyklus von $\mathsf{SSP}^{(1)}$ im Musterhierarchiegraphen entspricht, können wir entlang der durch `S` verursachten Querkante von der $\mathsf{SSP}^{(1)}$– zur $\mathsf{SCOPY}^{(0)}$–Instanz vom Typ FLOW beide Instanzen vereinigen, indem wir die „zweite" Ausgangsvariable `X[J]` ebenfalls in den Slot Nr. 1 der $\mathsf{SSP}^{(1)}$–Instanz eintragen[13]:

```
DO J=1,M
  SSP( I=[1:N], X[J] | S, A[:][J], B[:], 0.0 );
ENDDO
```

Beim weiteren Erkennen muß nun der zusätzliche Sloteintrag mitverarbeitet werden; dies ist von den vertikalen Schablonen zu berücksichtigen. In unserem Beispiel resultiert dies, da die Berechnung von `X` für die letzte Iteration der `DO J`–Schleife nicht überschrieben wird, in den folgenden beiden Instanzen[14]

```
MV( J=[1:M], I=[1:N], X[:], A[:][:], B[:], 0.0 );
SSP( I=[1:N], S, A[:][M], B[:], 0.0 );
```

[13]In der Prototyp–Implementierung werden mehrere Ausgangsvariablen in einer linearen, einfach verketteten Liste in den Slot eingehängt, und zwar in der Reihenfolge absteigender Dimensionalität. Nur Slots mit Status 'W' dürfen mehrfach besetzt werden.

[14]Dieser Schritt ist zulässig, falls `X` nicht mit `A` oder `B` identisch ist.

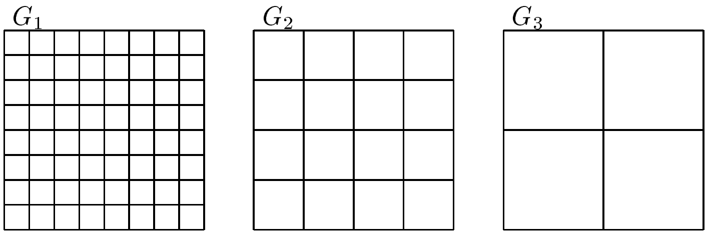
Nur auf dem Papier wird hier etwas doppelt berechnet. Da `S` ja in der Regel nur als temporäre Variable benutzt wurde, wird der Wert von `S` nun nicht mehr gebraucht; damit kann die $\mathsf{SSP}^{(1)}$–Instanz später als nutzloser Code ersatzlos gestrichen werden (vgl. Abschnitt 6.9.1). Einstweilen belassen wir die $\mathsf{SSP}^{(1)}$–Instanz aus Konsistenzgründen, da ja die Informationen über etwaige weitere Benutzungen von `S` im noch unbesuchten Teil des Syntaxbaumes zu diesem Zeitpunkt noch nicht vorliegen.

Es bleibt der maschinenspezifischen Codeerzeugung (siehe nächstes Kapitel) überlassen, ob eine temporäre Variable bei der Berechnung der Vektor–Matrix–Multiplikation benutzt wird oder nicht.

6.7 Erkennung von Datenstrukturkonzepten

Über den beschriebenen Algorithmus zur Mustererkennung hinaus ist es möglich, während der Mustererkennung Buch zu führen über statische Beziehungen zwischen einzelnen Datenstrukturen. Ein einleuchtendes Beispiel dazu ist die Identifizierung statisch bekannter Gitterhierarchien in Mehrgitterprogrammen.

Mehrgitterprogramme (multigrid, vgl. [ST82, RS93]) operieren auf einer Hierarchie von mehreren Gittern unterschiedlicher Größe. Im untenstehenden einfachen Beispiel ist G_1 das feinste Gitter (64 Elemente) und G_3 das gröbste Gitter (4 Elemente):



Die wichtigsten Komponenten von Mehrgitterprogrammen sind

- ein Relaxationsalgorithmus (Glätter) auf dem feinsten Gitter,
- eine Restriktionsoperation als Übergang zum nächstgröberen Gitter,
- ein Lösungsverfahren auf dem gröbsten Gitter, und
- eine Interpolationsoperation als Übergang zum nächstfeineren Gitter.

Diese vier Grundkomponenten können individuell zu beliebig komplexen Verfahren zusammengestellt werden. Beispiele hierzu nennt [Mül89].
Man betrachte folgendes Codefragment:

```
do i=1,3
 do j=1,3
  G2(i,j)= F1(2*i,2*j) - 4.0*G1(2*i,2*j) + G1(2*i,2*j-1)
         + G1(2*i,2*j+1) + G1(2*i-1,2*j) + G1(2*i+1,2*j)
```

Dieser Code wird durch `stargazer()`, wie oben beschrieben, als Inkarnation eines $\mathsf{STAR}^{(0)}$ mit Skalierungsfaktor 2 in beiden Dimensionen auf der rechten Seite erkannt. Darüberhinaus liefert uns diese Tatsache die Erkenntnis, daß das Gitter `G2` der linken Seite das nächstgröbere Gitter zu `G1` auf der rechten Seite ist. Für die Parallelisierung und automatische Datenaufteilung ist eine solche Information ungemein wertvoll:

Aufgrund des Fehlens dynamischer Felder in `FORTRAN 77` und wegen Speicherplatzoptimierungen sind reale Mehrgitterprogramme sehr oft durch das *workspace*–Konzept realisiert, d.h. alle Gitter, vom größten (feinsten) bis zum kleinsten (gröbsten) werden linearisiert in einem einzigen großen linearen Arbeitsfeld abgespeichert:

G_1	G_2	G_3	$\cdots$

Dies bringt drei Nachteile mit sich: Erstens werden dadurch alle Feldzugriffe eindimensional und damit komplizierter, aber bleiben doch noch erkennbar mit Hilfe eines zweiten Schablonen–Satzes für $\mathsf{STAR}^{(0)}$. Zweitens wird das Arbeitsfeld meistens durch indirekte Zugriffe (ein weiteres lineares Feld enthält die Startadressen der einzelnen Gitter im Arbeitsfeld) adressiert. Sind diese Startadressen jedoch, wenn auch ihr Wert nicht zur Übersetzungszeit vorliegt, so doch als *konstant* bekannt, kann man mit den Startadressen symbolisch weiterrechnen und später daraus die Einzelgitter rekonstruieren; wir werden hierauf in Abschnitt 7.1.1 zurückkommen. Drittens ist die Kenntnis der verborgenen Gitterhierarchie unabdingbar für die Bestimmung effizienter Datenaufteilungen. Im SUPERB–System löst Gerndt [Ger90] dieses Problem dadurch, daß der Benutzer mittels Direktiven dem Compiler die wirkliche Gitterstruktur erklärt. Andernfalls würde nämlich das lineare Arbeitsfeld in gleich große, vermutlich sogar zusammenhängende Teile aufgeteilt, was zu einer suboptimalen Lastverteilung und zu unnötiger Kommunikation führen würde.

6.8 Spezielle Transformationen bei der Mustererkennung

Manche Transformationen, wie das Aufgliedern von bedingten Anweisungen oder Schleifenanweisungen, deren Rumpf aus jeweils mehr als einer Anweisung besteht, erleichtern die Mustererkennung erheblich, bringen jedoch im Normalfall keine negativen Auswirkungen auf deren Laufzeit mit sich.

6.8.1 Aufgliederung bedingter Anweisungen (IF–distribution)

Die Replikation einer Bedingung über einen Block von Anweisungen im `then`– bzw. `else`–Teil wurde in Abschnitt 3.2.7 beschrieben.

Nützlich ist diese Transformation im Rahmen der Mustererkennung vor allem, weil sich Bedingungen über einzelnen Anweisungen leichter erkennen lassen als über einem Block von Anweisungen, und weil eine nicht erkannte Bedingung den weiteren Erkennungsprozeß bei Knoten „über" der Bedingung blockiert.

6.8.2 Skalar–Expansion

Das Expandieren von Skalaren zu Vektoren haben wir in Abschnitt 3.2.16 erläutert. Im Rahmen der Mustererkennung ist diese Transformation wichtig, da sie oft nachfolgend das Aufgliedern von Schleifen ermöglicht, was vorher durch eine (durch den Skalar bedingte) schleifengetragene Anti–Abhängigkeit verhindert wurde.

Für den häufiger vorkommenden Spezialfall

```
for (k ... ) {
S1:  SOME_OPERATION( s, op11[k], op12[k], ...);
S2:  SOME_OPERATION( t, op21[k], ..., s, ... );
}
```

(`S1` und `S2` wurden erkannt; `S1` schreibt einen Skalar `s`; `S2` liest ihn, konnte aber nicht entlang der von `s` verursachten FLOW–Querkante von `S1` nach `S2` mit `S1` zusammengezogen werden) führt der Mustererkenner vor einer etwaigen Aufgliederung der `for`–Schleife, sofern legal und sinnvoll, eine Skalar–Expansion für `s` durch, geleitet durch die Querkante. Indiziert die Schleife Operanden in `S1`, wäre eine Anwendung der Skalar–Expansion auf `s` sinnvoll. Legal ist sie, wenn `s` und `t` nicht als (gelesene) Operanden von `S1` vorkommen, d.h. kein Abhängigkeitszyklus besteht, der durch die Skalar–Expansion gebrochen würde. Dann erhalten wir

```
for (k ... ) {
S1:  SOME_OPERATION( s[k], op11[k], op12[k], ...);
S2:  SOME_OPERATION( t, op21[k], ..., s[k], ... );
}
```

Da `k` ungebunden ist, ändert sich an den Musternamen zu `S1` und `S2` nichts.

Eine entsprechende Transformation gibt es auch für das Expandieren von Vektoren zu Matrizen.

Eine nützliche Anwendung dieses Prinzips sieht man am folgenden Beispiel aus der EISPACK[15]–Routine `tred2` (von uns von `FORTRAN` nach C umformuliert):

```
for (j=1; j<=L; j++) {
  G = 0.0
  for (k=1; k<=L; k++)  G = G + Z[k][i] * Z[k][j];
  for (k=1; k<=L; k++)  Z[k][j] = Z[k][j] - G * D[k];
}
```

Die inneren Schleifen werden als initialisiertes $\mathsf{SSP}^{(1)}$ bzw. als $\mathsf{VAADDSV}^{(1)}$ erkannt. Da es keinen Abhängigkeitszyklus zwischen diesen Instanzen gibt, darf die Skalar–Expansion auf `G` angewendet werden:

```
for (j=1; j<=L; j++) {
  SSP ( k, G[j], Z[k][i], Z[k][j], 0.0);
  VAADDSV ( k, Z[k][j], Z[k][j], -G[j], D[k]);
}
```

Nach Aufgliederung der `j`–Schleife (siehe nächster Abschnitt) kann die Mustererkennung fortgesetzt werden und generiert schließlich

```
MV (j, k, G[j], Z[k][j], Z[k][i], 0.0);
MAADDVV ( k, Z[k][j], Z[k][j], -G[j], D[k]);
```

6.8.3 Aufgliederung von Schleifen (loop distribution)

Das Aufgliedern von Schleifen wurde in Abschnitt 3.2.13 vorgestellt.

Diese Transformation ist erlaubt, wenn im Datenabhängigkeitsgraphen keine Zyklen von Datenabhängigkeitskanten auftreten. Zu diesem Zweck berechnen wir den Datenabhängigkeitsgraphen[16]. Zur Feststellung von Zyklen im Datenabhängigkeitsgraphen verwenden wir Tarjan's Algorithmus zur Konstruktion der starken Zusammenhangskomponenten bei gerichteten Graphen (vgl. [Tar72]; eine Implementierung findet man in [ZC90]).

Die Umkehrtransformation (Schleifenverschmelzung) ist in der Mustererkennungsphase nicht erlaubt, um deren Terminierung sicherzustellen.

Die Aufgliederung von Schleifen und Bedingungen kann die Anzahl der (inneren) Knoten im Syntaxbaum vorübergehend erhöhen: bei Blockgröße n bis zu $2n-2$ Knoten pro Schleifenebene. Da die Schachtelungstiefe von Schleifen für erkennbare Muster durch eine kleine Konstante nach oben abgeschätzt werden kann, wird die asymptotische Gesamtlaufzeit der Mustererkennung durch diesen Effekt nicht beeinträchtigt.

[15] Vgl. Abschnitt 5.4.4; die Routine `tred2` ist im Anhang C.4 abgedruckt.

[16] Wir benutzen in der Implementierung die suboptimale Adjazenzmatrixdarstellung des Datenabhängigkeitsgraphen, die bei der Konstruktion und der nachfolgenden SZK–Bestimmung asymptotisch Platz und Zeit in quadratischer Größenordnung der Blockgröße erfordert. Diese bietet sich dennoch an, da die Knotenanzahl des Graphen (die Anzahl der Anweisungen im Block) meistens sehr klein gegenüber der Gesamtgröße des Quellprogramms ist. Ist sie es nicht, so kann die Laufzeitabschätzung (linear) für den Gesamtalgorithmus nicht aufrechterhalten werden. Wir nehmen dies in Kauf, weil diese Transformation für die Mustererkennung ungemein wichtig ist.

Das Aufgliedern von Schleifen ist für die Mustererkennung besonders wichtig, weil sie Schleifenrümpfe von kleiner Größe (im günstigsten Fall aus nur einer (erkannten) Anweisung bestehend) erzeugen kann, wodurch oft weitere Mustererkennungsschritte erst ermöglicht werden (vgl. Tabelle 5.1, z.B. in Kernel 18 der Livermore Loops hätte ohne Schleifenaufgliederung gar kein Muster von höherer Ordnung als 0 erkannt werden können).

6.8.4 Einordnung der Transformationen

Da diese Transformationen im allgemeinen die Struktur des abstrakten Syntaxbaumes modifizieren, müssen sie im Mustererkennungsalgorithmus *stmtdescend*(v) bzw. *exprdescend*(v) an geeigneter Stelle *vor* dem eigentlichen Erkennungsversuch eines Knotens v plaziert werden, um nicht mit den Schablonen zur vertikalen bzw. horizontalen Mustererkennung in Konflikt zu geraten. Insbesondere sind diese Transformationen, gleichwohl sie die Semantik des Programms erhalten, *keine* Schablonen, d.h., sie sind nicht in die globale Steuerung des Mustererkennungsalgorithmus durch den PHG integriert.

Mit den drei vorgenannten Transformationen ändert sich der Grundalgorithmus *stmtdescend* zur vertikalen Mustererkennung aus Abschnitt 6.2.2 bzw. aus Abschnitt 6.5 wie folgt:

---

function *stmtdescend*(*node*)
if *node* ist keine Zuweisung
then forall Söhne s von *node* (in textueller Reihenfolge) **do** *stmtdescend*(s) **od fi**
forall Ausdrücke e, die in *node* vorkommen **do** *exprdescend*(e) **od**
/⋆ Nun sind alle Unterbäume von *node* besucht und ggf. erkannt ⋆/
if *node* ist eine Bedingung
then *try_IF_distribution*(*node*)
fi
if *node* ist ein `for`–Schleifenkopf
then *try_loop_distribution*(*node*)
fi
forall Obermuster m zu *node* im PHG
do teste mit Schablone *match*(m,*node*),
    ob es eine Instanz I von m gibt, die auf *node* paßt **od**
falls ja, annotiere *node* mit I;

.... horizontale Mustererkennung ....

Die Funktion *try_IF_distribution* versucht, Bedingungsaufgliederung anzuwenden.

try_loop_distribution versucht zunächst, Skalarexpansion durchzuführen, und danach Schleifenaufgliederung.

Die Aufrufe *try_IF_distribution* bzw. *try_loop_distribution* für *node* erfolgen, nachdem die Söhne von *node* besucht wurden. Für die erkannten Söhne kann die zur Berechnung der Datenabhängigkeiten erforderliche Information direkt aus den Instanzen entnommen werden.

Ist die jeweilige Voraussetzung zur Anwendung erfüllt, werden die Transformationen angewendet, wodurch sich die Struktur des Syntaxbaumes entsprechend verändert, d.h. *node* bekommt dann einige „rechte“ Brüder und verliert an diese einige Anweisungen aus seinem Rumpf. Anschließend fährt die Mustererkennung mit dem Knoten *node* fort. Die neu entstandenen Brüder von *node* werden anschließend besucht, als wären sie schon immer da gewesen. Ein erneutes Besuchen von deren Kindern (die vorher ja Kinder von *node* waren) ist nicht erforderlich, da deren Musterinstanzen unverändert vorliegen.

6.9 Transformationen nach der Mustererkennung

6.9.1 Entfernen von nutzlosem Code

Bei der horizontalen Mustererkennung (Inferenz) und den beschriebenen Transformationen kann in Verbindung mit Hilfsvariablen manchmal nutzloser Code (vgl. Abschnitt 3.2.6) entstehen. Nutzloser Code berechnet „nutzlose" Variablen, die nie oder nicht vor dem nächsten Überschreiben gelesen werden.

Das Erkennen von nutzlosen Variablen und die Eliminierung aller nur sie berechnenden Anweisungen erfolgt *nach* der Mustererkennungsphase, weil dann die maximale Datenflußinformation in den Deskriptoren der Musterinstanzen für das gesamte Quellprogramm vorliegt, und weil während der Mustererkennung ja die Informationen über weitere Benutzungen von Variablen im noch unbesuchten Teil des Syntaxbaumes noch nicht gegeben sind.

6.9.2 Zerfall instabiler Muster

Instanzen instabiler Muster (vgl. Abschnitt 5.3.6) werden nach Abschluß der Mustererkennung gemäß Abschnitt 5.3.6 in ihre Grundbestandteile zerlegt.

Zum einen stellt dies bereits eine (zielmaschinenunabhängige) Optimierung dar: Die Multiplikation mit einem schleifeninvarianten Wert beispielsweise ($\mathsf{SVSUM}^{(1)}$, $\mathsf{SSSP}^{(1)}$, $\mathsf{SMV}^{(2)}$, $\mathsf{SMM}^{(3)}$) wird so aus den Schleifen herausgezogen und auf der obersten Schleifenebene als skalare Multiplikation MUL hinter die Restoperation ($\mathsf{SSP}^{(1)}$, $\mathsf{MV}^{(2)}$ bzw. $\mathsf{MM}^{(3)}$) gesetzt.

Zum anderen wird dadurch die Anzahl der für die Codeerzeugungsphase sichtbaren Muster auf ein notwendiges Maß begrenzt.

6.9.3 Konstruktion von Vektor–DAGs

Blöcke von quasiskalaren Vektor– oder Matrixanweisungen können als Vektor–DAG dargestellt werden, wenn man die vorkommenden (Feld–)Variablen als Knoten und die Datenabhängigkeiten zwischen diesen als Kanten im Vektor–DAG interpretiert (vgl. [Keß90]). Je größer solch ein Vektor–DAG wird, desto mehr Spielraum ergibt sich für Optimierungen während der Codeerzeugungsphase.

6.10 Korrektheit

Satz 6.3 *Das vorgestellte Mustererkennungsverfahren arbeitet korrekt, d.h. die Semantik des* LATINUS*–Quellprogramms und die des daraus produzierten* PALATINUS*–Programms gemäß der Definitionen in Abschnitt 5.3 sind gleich.*

Der Korrektheitsbeweis für das Mustererkennungsverfahren umfaßt drei Teilbereiche:

1. induktiver Korrektheitsbeweis für jede Schablone. Zu zeigen ist für jede Schablone jedes Musters: Nach dem erfolgreichen Erkennen des Musters durch die betrachtete Schablone beschreibt die generierte Instanz genau die Semantik des Teil–Syntaxbaumes, den sie annotiert, unter der (Induktions–) Annahme, daß dies unmittelbar vor dem Aufruf der Schablone bereits galt.

   Da die Arbeitsweise jeder Schablone leicht überschaut werden kann (siehe Anhang B), halten wir einen formalen Beweis nicht für erforderlich.

2. Korrektheit für die Transformationen Schleifenaufgliederung, Schleifenaufrollen, usw. ist bereits an anderer Stelle bewiesen. Da wir uns an die Standardverfahren halten, kann dieser Teil abgehakt werden.

3. Korrektheit von Datenfluß– und Datenabhängigkeitsanalyse: Auch hier halten wir uns an konservative Standardverfahren, deren Korrektheit an anderer Stelle bewiesen wurde.

6.11 Implementierung und Ergebnisse

6.11.1 Warum kein Treepatternmatcher–Generator von der Stange?

Zusätzlich zu den in Abschnitt 4.5 genannten prinzipiellen Gründen wollen wir nicht unerwähnt lassen, daß wir auch mit einem Treepatternmatcher–Generator (z.B. mit einem praktisch einsatzfähigen OPTRAN–System) kaum weniger Platz zur Spezifizierung der Schablonentests (syntaktische und semantische Bedingungen) benötigt hätten. Die Nebenroutinen z.B. zur Datenflußanalyse oder zum symbolischen Vergleich von Ausdrücken hätten sowieso explizit codiert werden müssen. Einzig die Formulierung der Hauptfunktionen *stmtdescend* und *exprdescend*, die nur wenige Zeilen umfassen und nicht weiter schwierig sind, hätten wir uns erspart, dabei aber Flexibilität, Geschwindigkeit und die unmittelbare Kontrolle über den Erkennungsprozeß verloren, die wir für die semantische Mustererkennung auf höheren Anweisungsebenen imperativer Sprachen unbedingt brauchen.

Zudem war zum Zeitpunkt der Planung unserer Implementierung kein verläßlicher Treepatternmatchergenerator verfügbar, und einen Mißerfolg aufgrund externer Programmfehler oder –schwächen konnten wir uns aufgrund knapper Zeitvorgaben nicht leisten.

6.11.2 Stand der Implementierung

Als Frontend dient uns ein selbstentwickelter Recursive–Descend–Parser[17], der gerade den LATINUS–Sprachumfang umfaßt und die in Abschnitt 2.2 beschriebene Zwischendarstellung erzeugt. Die Vorabtransformationen (wie Prozedur–Inlining, Konstantenpropagation oder Erkennung von Induktionsvariablen, siehe Abschnitt 6.1) wurden aus Zeitgründen nicht implementiert, sondern manuell in den Quellprogrammen durchgeführt. Für eine zukünftige Weiterentwicklung von PARAMAT muß dieser Bereich noch wesentlich aufgebohrt werden. Auch ein `FORTRAN`–Frontend steht auf der Liste der geplanten Erweiterungen, um größere `FORTRAN`–Anwendungen (z.B. die Perfect Club Benchmarks [Ber92]) angehen zu können.

Ein Prototyp des Mustererkenners wurde implementiert und getestet. Die aktuelle Implementierung besteht aus etwa 12000 Zeilen C–Code und erkennt zuverlässig z.Zt. 91 implementierte Muster mit etwa 150 realisierten Schablonen. Jede realisierte Schablone ist als eine C–Routine von ca. 20–50 Zeilen implementiert, die syntaktische und semantische Bedingungen abtestet und, falls erfolgreich, die Musterinstanz generiert und die Slot–Einträge ausfüllt. Da viele nützliche syntaktische und semantische Prädikate vordefiniert worden sind (vgl. Abschnitt 2.3), ist das Schreiben von Schablonenrealisierungen recht handlich und einfach und kaum umständlicher als eine entsprechende Spezifikation in einem Treepatternmatcher–Generator. Mehrere Schablonen zu Mustern mit ähnlichen Strukturen können auch eine gemeinsame Realisierung haben, zum Beispiel die Schablonen für das Aufrollen von Schleifen. Weitere Muster und Schablonen können leicht hinzugefügt werden. Ein großes Maß an Robustheit gegenüber Schleifenvertauschung, Schleifenverteilung, Schleifenabrollen und Umstellung von Anweisungen wurde in der Praxis bestätigt.

Ein Blockdiagramm des Mustererkennungsverfahrens zeigt Abb. 6.8.

[17] Dieses Frontend wurde nach einem Beispiel aus [ASU86] in einem früheren Projekt entwickelt und für den LATINUS–Sprachumfang angepaßt.

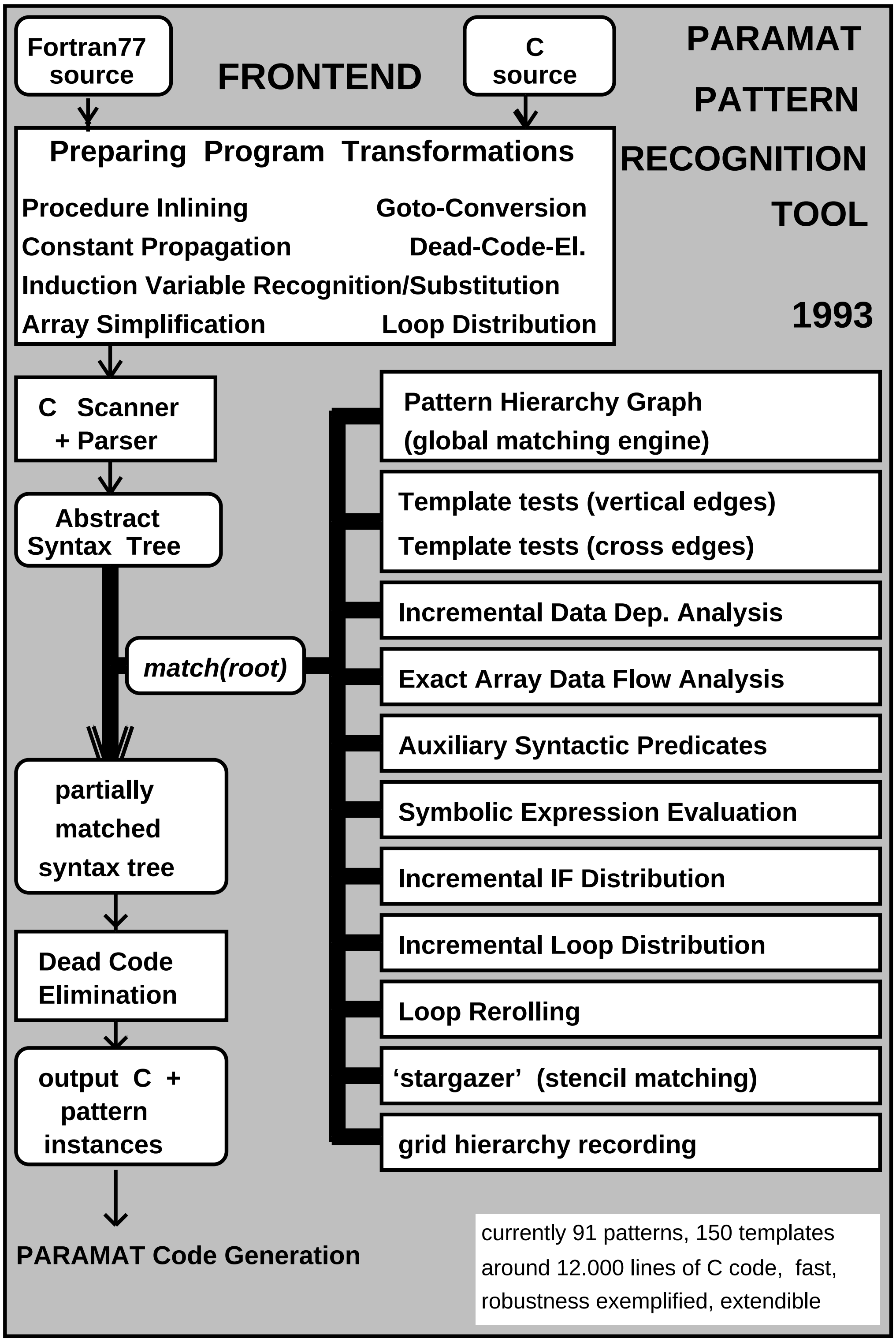


Abbildung 6.8 Das PARAMAT–Mustererkennungswerkzeug im Überblick.

6.11.3 Ergebnisse

Die zu erwartenden Mustererkennungen zu den Benchmarksuiten Livermore Loops (Tab. 5.1) und Purdue Set (Tab. 5.2) wurden experimentell bestätigt. Einzelne Beispiele befinden sich im Anhang C.

6.12 Diskussion

Eine mögliche Alternative zur beschriebenen Mustererkennungstechnik böte wohl das Erkennen von Mustern im *Kontrollflußgraphen* anstelle des abstrakten Syntaxbaum als Zwischendarstellung des Quellprogramms. Hierzu ist folgendes zu bemerken:

- Die Darstellung des Quellprogramms als Syntaxbaum ist uns durch das verwendete Frontend bereits gegeben.
- Der Kontrollflußgraph besitzt weniger Struktur als der abstrakte Syntaxbaum. Bei der Umwandlung des Syntaxbaumes in einen Kontrollflußgraphen würde zwangsläufig Information verloren gehen, z.B. Information über die Schleifenstruktur (Schleifenvariablen). Die Mustererkennung würde schwieriger, undurchsichtiger und langsamer.
- Keines der in Abschnitt 4.3 genannten Programmtransformationssysteme, wollte man denn eines benutzen, arbeitet auf dem Kontrollflußgraphen.
- Der Kontrollflußgraph bietet sich an, falls das Quellprogramm viele Sprunganweisungen enthält („Spaghetti–Code"). Die von uns gesuchten Muster enthalten jedoch Sprunganweisungen nur in Ausnahmefällen, und in eben diesen Ausnahmefällen könnte man die Sprunganweisungen durch strukturierende Anweisungen wie `IF-THEN-ELSE` oder `WHILE` (sogar automatisch) ersetzen.

6.13 Verwandte Arbeiten

Von Ansätzen zur syntaktischen Mustererkennung haben wir unsere Arbeit bereits in den Abschnitten 4.4, 4.5 und 6.11.1 abgegrenzt.

In diesem Abschnitt wollen wir die wichtigsten Gemeinsamkeiten und Unterschiede der PARAMAT–Mustererkennung gegenüber anderen *semantischen* Mustererkennungsansätzen herausstellen.

PAT („Program Analysis Tool") [HN90a] und darauf aufbauende Arbeiten [KNS92, KNE93] versuchen wirklich, Programme zu „verstehen". Sie verwenden ein unserer Mustererkennung dem Prinzip nach ähnliches Verfahren zur Konzepterkennung bei sehr einfachen, nicht–numerischen COBOL–Programmen. Dem nichtnumerischen Anwendungsbereich entsprechend wird größeres Gewicht auf das Erkennen von Datenstrukturen gelegt. Eine brauchbare Musterbibliothek ist selbst für den nichtnumerischen Bereich kaum vorhanden; einfache Beispiele beschreiben etwa Bubble–Sort oder Binary–Search in eindimensionalen Feldern. Feld–Datenfluß wird nicht berechnet, da er für das vorgesehene Anwendungsspektrum nicht benötigt wird. Die „plan library" von PAT entspricht in etwa unserem Musterhierarchiegraphen. Eine explizite Unterscheidung von Mustern und Schablonen wie bei uns gibt es in PAT nicht. Während der laufenden Erkennungsphase werden in [KNE93] ebenfalls gewisse „Transformationen" durchgeführt, so wie bei uns etwa Schleifenaufgliederung oder –aufrollung.

Ähnlich wie PAT — und auch auf dem gleichen Anwendungsgebiet — arbeitet das Mustererkennungssystem aus [RW90] im Rahmen des *Programmer's Apprentice*–Projekts, das vornehmlich für die Anwendung zur automatischen Dokumentierung beim Software–Reengineering von CommonLisp–Programmen konzipiert ist. Es gibt darin ein — allerdings nur skalares — Analogon zu unserem Mustererkennen entlang von Querkanten auf der Basis von Graphgrammatiken.

Durch Abstrakte Interpretation [CC77, CC79] des sequentiellen Programms berechnet [AH90] eine sequentielle Speicherzugriffsabbildung (*abstract store*), die zu jedem Feldelement, das in einer Schleife referenziert wird, eine symbolische Darstellung seines Inhalts angibt. Danach werden Schleifen, wo moeglich, durch ihre explizite Darstellung (*closed form*) ersetzt, was etwa unseren Musterinstanzen entspricht. Erkannt werden nur Muster der Ordnung ≤ 1, und zwar Äquivalente zu POWER, VSUM$^{(1)}$, VPROD$^{(1)}$, PREVSUM$^{(1)}$, SSP$^{(1)}$. Auf diesen expliziten Darstellungen baut die Erkennung von Induktionsvariablen auf. Das Verfahren versagt offenbar bei abgerollten oder geblockten Schleifen. Über die benötigte Laufzeit wurden keine Angaben gemacht.

Einen weiteren Ansatz zur Erkennung von Rekurrenzen stellt [RF93] vor. Während dieser Vorschlag bei nicht unbeträchtlichem Aufwand die Erkennung recht allgemeiner und mehrdimensionaler Rekurrenzen zuläßt, werden doch eine Reihe von Voraussetzungen gemacht, die in realen Programmen schwerlich immer erfüllt sind. Da komplizierte Rekurrenzen in realen Programmen selten sind, erscheint der Aufwand dieses Ansatzes nicht gerechtfertigt.

Eine eingeschränkte Erkennung von einfachen Reduktionen ist ferner im parallelisierenden SUIF–Compiler [Sta94] sowie im kommerziellen Parallelisierungssystem KAP [Kuc] implementiert.

7 Muster–gesteuerte Code–Erzeugung für DMS

7.1 Algorithmen–Ersetzung und Transformationen

Für jedes Muster gibt es gute parallele Implementierungen auf der gewünschten parallelen Zielmaschine. Die Auswahl einer Implementierung für eine gegebene Musterinstanz (Code–Erzeugung) ist abhängig von den Problemgrößen (z.B. Feld–Extents) und von den zu wählenden Datenaufteilungen der beteiligten Felder. Die parallelen Implementierungen für eine gegebene Zielmaschine werden daher in parametrisierter Form in einer Implementierungsbibliothek angeordnet. Ferner steht für jedes Muster eine in Problemgrößen und Datenaufteilungen parametrisierte Tabelle von auf der realen Zielmaschine gemessenen Laufzeiten zur Verfügung, die schnelle Rückschlüsse auf die Qualität der gewählten Datenaufteilung zuläßt, womit ein Optimierungsverfahren eine gute Datenaufteilung bestimmen kann. Fig. 7.1 gibt einen Überblick über die Codeerzeugungsphase von PARAMAT.

7.1.1 Code–Erzeugung durch Algorithmenersetzung

Der Hauptvorteil der im vorangehenden Abschnitt beschriebenen Erkennung semantischer Muster im Quellprogramm ist, daß, nachdem erst einmal ein Codestück als Inkarnation eines Musters erkannt worden ist, die beste verfügbare parallele Implementierung ausgewählt werden kann, ohne daß der Benutzer sich hierum Gedanken machen muß.

Sei T der (teilweise) mit Instanzen erkannter Muster annotierte abstrakte Syntaxbaum nach Beendigung der Mustererkennungsphase. Sei T_v der Unterbaum von T mit Wurzel v. Falls T_v als Inkarnation eines Musters m erkannt wurde, ist seine Wurzel v mit einer Instanz $v.matched$ von m annotiert. Die Instanz enthält, wie beschrieben, neben dem Musternamen m auch die Liste der Slot–Einträge (Variablen, Konstanten oder einfache Terme). Man beachte, daß $w.matched$ für einen Knoten w auch dann nicht gelöscht wurde, falls der Vater von w erkannt wurde, sodaß für *alle* Knoten $w \in T_v$ mit $v.matched \neq \emptyset$ die Instanzen $w.matched$ noch vorhanden sind.

Unter einer *Standardparallelisierung* eines sequentiellen Programms C (dargestellt als abstrakter Syntaxbaum) verstehen wir den durch die in Abschnitt 3.4 beschriebene halbautomatische Parallelisierung erzeugten Code für C. Eine Standardparallelisierung für eine einzelne Schleife l über einem Rumpf R besteht aus einer Spezialisierung dieses Konzepts, wobei dimensionsspezifische Masken $owned_d(A[...])$ und dimensionsspezifische Kommunikationsanweisungen `EXCH_d(A[...])` benötigt werden, falls l die d–te Dimension eines Feldvorkommens $A[...]$ in R indiziert. Im Gegensatz zu einem explizit parallelen Algorithmus orientiert sich die Standardparallelisierung an der Struktur des sequentiellen Quellprogramms.

Für jeden Knoten $v \in T$ mit $v.matched \neq \emptyset$, der einen `for`–Schleifenkopf darstellt, gibt es mehrere Möglichkeiten, Code für T_v zu generieren, aus denen der PARAMAT–Benutzer durch die Codeerzeugungs–Direktiven SeqDebug$[m]$, ReplSeq$[m]$ und NoReplace$[m]$ für jedes Muster individuell auswählen kann (wir setzen $m = v.matched.name$):

1. eine *sequentielle Implementierung* Δ_m für m (Berechnung auf einem Knotenprozessor oder dem Host nebst dabei anfallender Kommunikation) als Kontrollimplementierung, falls das Debugging–bit SeqDebug$[m]$ gesetzt ist;

2. eine *replizierte sequentielle Implementierung* Ξ_m für m (sequentielle Berechnung auf allen Knotenprozessoren gemäß den vorgegebenen Datenaufteilungen), falls das Sequentialisierungsbit ReplSeq$[m]$ gesetzt ist;

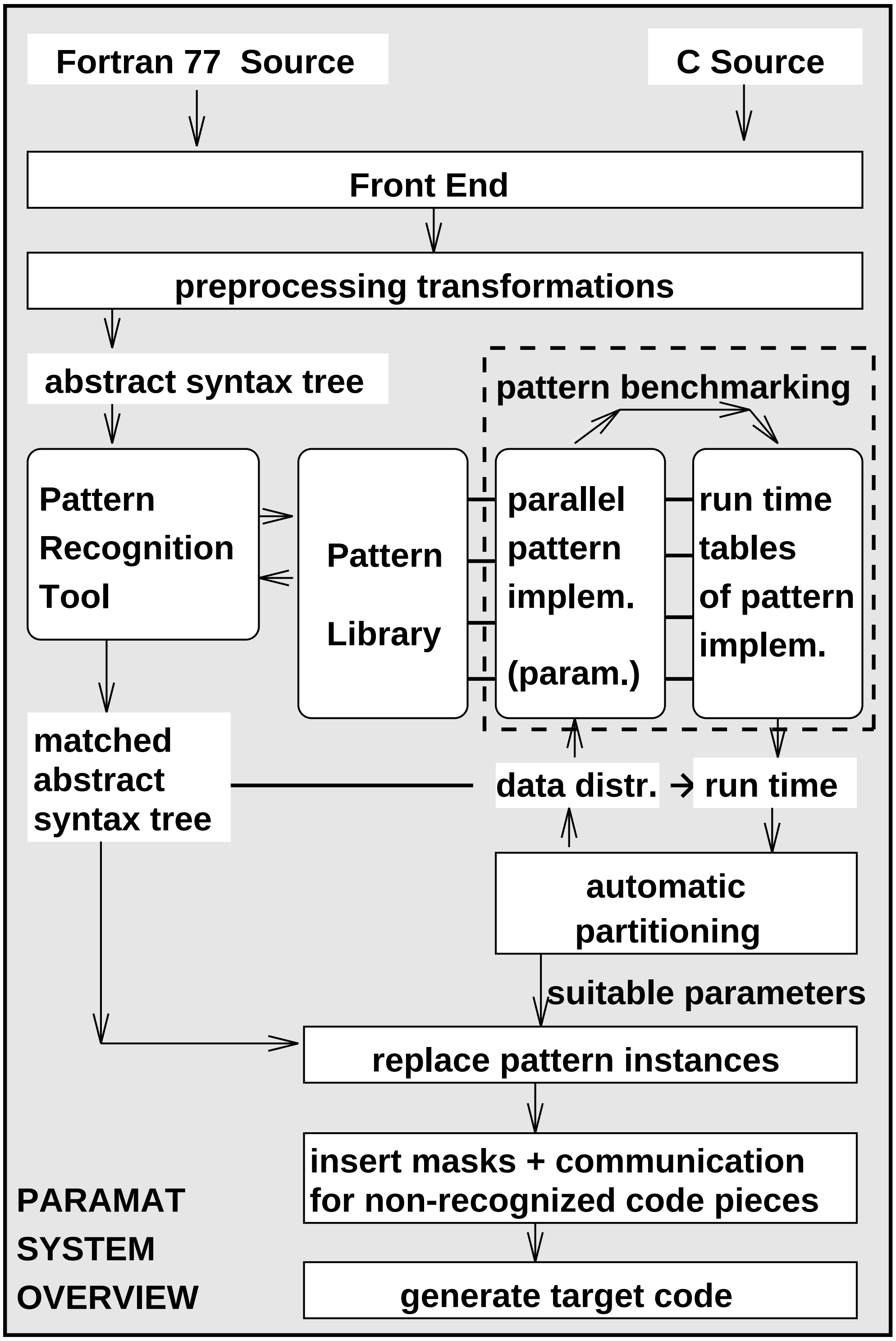


Abbildung 7.1 Überblick über die mustergesteuerte Codeerzeugung in PARAMAT. Nur die Komponenten innerhalb des gestrichelten Rechtecks sowie die Generierung von Zielcode für nicht erkannte Codeteile sind zielmaschinenabhängig. Die Berechnung der Laufzeittabellen zu den Musterimplementierungen (pattern benchmarking) erfolgt (offline) zu einem früheren Zeitpunkt (Compilergenerierungszeit), sodaß die Tabellen zur Übersetzungszeit vollständig vorliegen.

3. eine *Standardparallelisierung* Ψ_m für die oberste Schleife, falls das Bit NoReplace$[m]$ gesetzt ist, und eine ggf. andere Implementierung für den Rest.

   Sofern dabei blockende Schleifen auftreten, müssen wir diese wie folgt behandeln: Sei L die Menge von Schleifenköpfen l in T_v, für deren Instanzen $l.matched.name = m$ gilt. Sei R der Rumpf der innersten Schleife $l_{inn} \in L$. Sei $L' \subset L$ die Menge von Schleifen in L, die eine Schleife in R blocken. Technisch wird $L' \cup R$ zusammenhängend gemacht, indem alle Schleifen $l' \in L'$ durch Schleifenvertauschung mit einer nächstinneren Schleife $l \in L - L'$ nach „unten", also hin zum "Rumpf", getauscht[1] werden, sodaß T_v nunmehr aus den zusammenhängenden äußeren Schleifen aus $L - L'$ um einen neuen Rumpf R', bestehend aus den Schleifen aus L' um R, zusammengesetzt ist. Falls $R' - R \neq \emptyset$, muß für $R' - R$ die Mustererkennung nochmals aufgerufen werden, um die Instanzen der Schleifenköpfe in $R' - R$ zu aktualisieren. Gleiches gilt für $L - L'$, falls eine Vertauschung vorgenommen wurde. Die Struktur von R bleibt unverändert. Für alle $l \in L - L'$ wird nun eine Standardimplementierung erzeugt. Sei r' die Wurzel von R'. Auf welche Weise für $T_{r'}$ Code erzeugt wird, ist abhängig von den Codeerzeugungsdirektiven für das (aktualisierte) Muster $r'.matched.name$.

   Die Wirkung von NoReplace$[m]$ ist somit genau die, als wären die Schleifen in $L - L'$ niemals erkannt worden (aber R').

   Beispiel 7.1 Nehmen wir an, der Mustererkenner habe folgendes Codefragment

```
for (i=1; i<=n; i+=x)
  for (j=1; j<=m; j++)
    for (k=i; i<=min(i+x-1,n); k++)
      for (l=1; l<=r; l++)
         a[j][k] = a[j][k] + b[j][l] * c[l][k];
```

   als Matrix–Matrix–Multiplikation erkannt und den dem i–Schleifenkopf entsprechenden Knoten $v = l_i$ mit der Instanz

```
MM(j,i,l, a[:][:], b[:][:], c[:][:], a[:][:])
```

   annotiert. Auch der j–Schleifenkopf (nennen wir ihn l_j) ist mit einer $\mathsf{MM}^{(3)}$–Instanz annotiert, weil die i–Schleife lediglich die k–Schleife blockt. Folglich haben wir $L = \{l_i, l_j\}$ und $L' = \{l_i\}$. Nehmen wir ferner an, der PARAMAT–Benutzer habe NoReplace$[\mathsf{MM}^{(3)}]$ gesetzt. Da l_i eine andere Schleife (l_k) blockt, tauschen wir sie zum "Rumpf" hin (also mit l_j) und erhalten

```
for (j=1; j<=m; j++)
  for (i=1; i<=n; i+=x)
    for (k=i; i<=min(i+x-1,n); k++)
      for (l=1; l<=r; l++)
         a[j][k] = a[j][k] + b[j][l] * c[l][k];
```

   Durch Nachrechnen sehen wir, daß sich bei einem neuerlichen Lauf der Mustererkennung nur die Instanz von l_i ändern würde (nämlich zu einer $\mathsf{MV}^{(2)}$–Instanz). Daher wird die Mustererkennung für $R' - R = \{l_i\}$ nochmals aufgerufen, wobei die $\mathsf{MV}^{(2)}$–Instanz zu l_k bereits gegeben ist.

   Die Standardparallelisierung liefert dann

```
for (j=1; j<=m; j++)
  EXCH_1( a[j][:] );        /* Kommunikation in Dimension 1 */
  if ( owned_1( a[j][:] ) ) {
    Code fuer MV(i,l, a[j][:], b[j][:], c[:][:], a[j][:]);
  }
```

[1] Diese Schleifenvertauschung (vgl. Abschnitt 3.2.12) ist i.allg. möglich, weil für das Blocken vertauschbarer Schleifen ähnliche Vorbedingungen gelten wie für die Schleifenvertauschung. Andernfalls wäre l' wohl kaum erkannt worden. Alternativ könnte man hier auch die Blockung explizit aufheben.

◇

4. eine *parallele Implementierung* Π_m, eine Kombination aus ggf. mehreren geeigneten parallelen Algorithmen, von denen sich mindestens einer von der Standardparallelisierung unterscheidet, falls ein solcher existiert. Dieser Algorithmus ist ebenfalls parametrisiert in Datenaufteilungen und Problemgrößen.

Jede solche Implementierung ist ein vorgegebenes Code–Gerippe, in das die Slot–Einträge an den entsprechenden Stellen eingefügt werden können. Sie enthält bereits alle erforderlichen Kommunikationsanweisungen und Empfehlungen für die Registervergabe. Falls einige wichtige Parameter (z.B. die Problemgröße) zur Übersetzungszeit nicht bekannt sind, so wird Code eingefügt, der die Auswahl zur Laufzeit trifft.

Alle Implementierungen sind maschinenabhängig. Sie müssen in einer Sprache geschrieben sein, die dem Benutzer ungehinderten Zugriff auf die Zielmaschine ermöglicht und die direkt in Maschinencode übersetzt werden kann, sodaß zumindest für die Musterinstanzen keine anspruchsvolle und damit für die Laufzeitabschätzung schwer überschaubare Übersetzungsstufe dazwischengeschaltet werden muß. Daher werden sie in *C mit inline–Assembler* geschrieben, um die Optimierungsmöglichkeiten der Knotenprozessoren voll ausschöpfen zu können, was mit Hochsprachen allein im allgemeinen nicht möglich ist, weil die verfügbaren Übersetzer die notwendigen Optimierungen nicht beherrschen (vgl. [Fri91] für den i860).

Man beachte, daß auch für Standardparallelisierungen und parallele Implementierungen die Wahl der Datenaufteilungen zu impliziten Sequentialisierungen führen kann (vgl. Abschnitt 3.7).

Für manche Muster wird man eventuell keine geeignete parallele Implementierung finden, die nicht mit der Standardparallelisierung identisch ist. Auch bei deren Vorhandensein kann der Benutzer mittels der Direktive NoReplace[m] PARAMAT zwingen, eine Standardparallelisierung für die Schleife(n) mit m–Instanzen zu wählen. Um eine Standardparallelisierung für ganz T_v zu erzwingen, müssen für alle verschiedenen in Unterbäumen von T_v erkannten Muster $m_1, m_2, \ldots$ die Direktiven NoReplace[m_1], NoReplace[m_2], ... gesetzt sein.

Für jede Instanz I eines Musters m bezeichne NoParallel[m](I) ein boolesches Prädikat, das sich genau dann zu True auswerten läßt, wenn es bei den gegebenen Datenaufteilungen und Problemgrößen unklug wäre, für I parallelen[2] Code zu erzeugen. NoParallel[m] hängt natürlich ab von NoReplace[m]. Evaluiert NoParallel[m](I) zu True, so ist die Wirkung diegleiche, als wenn ReplSeq[p] gesetzt wäre.

Damit läßt sich allgemein die Routine zur Codeerzeugung (*implementation driver*) für ein Muster m wie folgt formulieren:

---

generate_code[m](I, T_v):
if SeqDebug[m] **then** generiere Δ_m für I; **exit fi**;
if NoParallel[m](I) zur Übersetzungszeit vollständig auswertbar
then if NoParallel[m](I)
    then generiere Ξ_m für I
    else if NoReplace[m] oder Π_m nicht vorhanden
        then generiere Ψ_m für $L - L'$ (s.o.)
            um *generate_code*[r'.*matched.name*](r'.*matched*, $T_{r'}$)
        else generiere Π_m **fi**
    fi
else (irgendeine Problemgröße ist zur Übersetzungszeit unbekannt:)

[2] Für den Fall, daß sich eine massiv–parallele Implementierung aufgrund geringer Problemgrößen nicht lohnt, könnte man alternativ auch eine parallele Implementierung mit einer geringeren Anzahl von Prozessoren wählen. Abgesehen davon, daß man hierzu einen skalierbaren Parallelrechner benötigte, hätte dies gegenüber der strikten Sequentialisierung zwei Nachteile: Erstens müßte eine zusätzliche Parametrisierung der gesamten Codeerzeugung in gängigen Prozessorzahlen (z.B. Zweierpotenzen) eingeführt werden; zweitens würden damit neue Konstellationen von Datenaufteilungen eingeführt, die die Komplexität der automatischen Datenaufteilung verschlechtern würden (vgl. Abschnitt 7.2). Der bei geringen Problemgrößen zu erwartende mäßige Geschwindigkeitsgewinn gegenüber der Sequentialisierung kann diesen Aufwand nicht rechtfertigen.

generiere "**if** (NoParallel$[m](I)$) **then**";
generiere Ξ_m für I;
generiere "**else**";
if NoReplace$[m]$ oder Π_m nicht vorhanden
then generiere Ψ_m für $L - L'$ (s.o.)
um *generate_code*$[r'.matched.name](r'.matched,\ T_{r'})$
else generiere Π_m **fi**
fi
fi

Damit wird eine aufgrund zu kleiner Problemgrößen unrentable Parallelisierung und damit ein Speed–down verhindert. Falls die Problemgrößen zur Übersetzungszeit nicht bekannt sind, wird ein Laufzeittest eingefügt.

Ähnliche Laufzeittests können eingefügt werden, wenn PARAMAT über die Werte einiger wichtiger symbolischer Variablen im unklaren ist. Ein Beispiel ist folgende Situation, die häufig in Mehrgitter–Anwendungen auftritt: Der Programmierer benutzt dort ein großes lineares *workspace*–Feld, um darin alle (meist zweidimensionalen) Gitter abzulegen, und indiziert jedes einzelne Gitter über einen Offset, der meist wiederum eine Feldreferenz ist. Da somit ein indirekter Feldzugriff vorliegt, sind die Datenabhängigkeiten zur Übersetzungszeit allenfalls sehr grob berechenbar, sodaß PARAMAT nicht erkennen kann, ob hier überhaupt eine solche workspace–Gitterfolge vorliegt. Als Konsequenz würde eine Standard–Datenaufteilung (sei sie blockweise oder zyklisch) zu einer schlechten Lastverteilung und viel unnötiger Kommunikation führen (vgl. Abschnitt 6.7). Die Gitterhierarchie kann, wie in Abschnitt 6.7 beschrieben, festgestellt werden. PARAMAT behandelt die Offsets als symbolische Variablen. Um herauszufinden, ob die durch die Offsets adressierten Gitter wirklich disjunkt sind, kann PARAMAT einen entsprechenden *Laufzeittest* durchführen. Da die Anzahl der verschiedenen Gitter (und damit der Offsets) recht klein ist, nimmt dieser Laufzeittest nur wenig Laufzeit in Anspruch. Der potentielle Gewinn bei positivem Ausgang ist jedoch beträchtlich, sodaß sich diese Optimierung (als Bestandteil der Nichtstandard–parallelen Implementierung) in jedem Fall lohnt. Wird die Vermutung einer workspace–Gitterhierarchie zur Laufzeit bestätigt, wird das workspace–Feld implizit in die einzelnen Gitter zerlegt und statt der Implementierung mit Standardaufteilung eine zusätzlich generierte Alternativ–Implementierung mit separaten Datenaufteilungen für jedes Gitter angesprungen.

7.1.2 Beispiele für parallele Implementierungen

Die Zahl der derzeit (mehr oder weniger) bekannten parallelen Algorithmen auf mehr oder weniger speziellen Parallelrechnertypen ist in der Tat Legion, und ihre Beschreibung würde ein vielbändiges Werk füllen. Ebenso kennen wir eine Fülle von Optimierungstechniken für diverse Parallelrechnertypen.

Um einen Eindruck von der Vielfalt an Möglichkeiten und der Mächtigkeit der Algorithmenersetzung in PARAMAT zu vermitteln, stellen wir in diesem Abschnitt einige ausgewählte Beispiele vor.

Weitere parallele Algorithmen für Operationen auf dicht besetzten Matrizen entnehme man beispielsweise [CRT87], [FJL+88], [CS88], [BT89], [Mül89], [GHN+90], [FP92] und anderen.

Skalare Muster

Für skalare Muster benötigen wir allenfalls eine sequentielle, jedoch keine parallele Implementierung[3]. Daher sind alle skalaren Muster instabil, d.h. für die Codeerzeugung nicht sichtbar.

[3] Fine–Grain–Parallelisierung wäre nichtdestoweniger hier möglich, ist jedoch nicht Ziel dieser Arbeit.

Für die sequentialisierten Implementierungen Ξ_m von Mustern m höherer Ordnung können natürlich die bekannten Standardtechniken (vgl. [ASU88]) zur Codeerzeugung eingesetzt werden, z.B. das Zusammenfassen gemeinsamer Unterausdrücke, Registerbelegung [KPR91] usw.

Reduktionen

Für Instanzen der (stabilen) Reduktionsoperationen $\mathsf{VSUM}^{(1)}$, $\mathsf{MSUM}^{(2)}$, $\mathsf{VPROD}^{(1)}$, $\mathsf{MPROD}^{(2)}$, $\mathsf{VROR}^{(1)}$, $\mathsf{MROR}^{(2)}$, $\mathsf{VMINVAL}^{(1)}$, $\mathsf{VMINLOC}^{(1)}$ usw. können wir nun entsprechende, schon existierende Routinen des Laufzeit– bzw. Betriebssystems der Zielmaschine optimal ausnutzen. So kann z.B. beim INTEL *i*PSC/860 die Routine `gssum()` bei der Implementierung von $\mathsf{VSUM}^{(1)}$ verwendet werden.

Gitterrelaxationen

Eine einzelne Gitterrelaxation (grid update), z.B. $\mathsf{MJACOBI}^{(2)}$ und $\mathsf{MGAUSSSEIDEL}^{(2)}$ im zweidimensionalen Fall, besteht aus einem update–Schritt für jedes Gitterelement. Eine Folge solcher Relaxationsschritte, z.B. $\mathsf{JACOBI}^{(3)}$ oder $\mathsf{GAUSSSEIDEL}^{(3)}$, bietet ein zusätzliches Parallelisierungspotential.

Ein einzelner Relaxationsschritt Algorithmenersetzung muß stets konservativ mit der numerischen Stabilität und ggf. Konvergenzeigenschaften der erkannten Muster umgehen. Jede Algorithmenersetzung muß garantieren, daß die numerische Stabilität durch sie nicht beeinträchtigt wird. Gewisse Zugeständnisse z.B. bezüglich der Assoziativität und Kommutativität von arithmetischen Grundoperationen sollten aber vereinbart werden, da sonst auch andere Optimierungen oder gar die Parallelisierung ungültig werden können.

Da die Musternamen uns den Zugriff auf numerisches Hintergrundwissen eröffnen, dürfen wir dies bei der parallelen Implementierung des Musters diese Information verwenden. So kann PARAMAT z.B. für $\mathsf{MGAUSSSEIDEL}^{(2)}$–Instanzen eine parallele Implementierung auswählen, die für die gegebene Zielmaschine geeigneter ist. Anbieten würde sich hier zum Beispiel ein Rot–Schwarz–Iterationsschema (vgl. [Mül89]), das bei gleicher Konvergenzeigenschaft besser parallelisiert werden kann, oder die doppelte Anzahl von Iterationsschritten der Jacobi–Relaxation oder des Gauß–Seidel–Jacobi–Hybridverfahrens (wie in [Bab91]). Man kann dabei davon ausgehen, daß der Programmierer der Anwendung nicht Gauß–Seidel mit Jacobi vergleichen möchte (das Ergebnis ist bekannt), sondern die tatsächlich schnellste parallele Implementierung des Relaxationsverfahrens wünscht — und die kann je nach Zielmaschine und Problemgröße verschieden sein.

Feste Anzahl von Relaxationsschritten [Mül89] beschreibt eine Methode für $\mathsf{JACOBI}^{(3)}$, die durch Propagieren der Lokalitätsrelation *Ref* (vgl. Abschnitt 3.4.2) über mehrere Iterationen hinweg (Zeitachse) Startupzeiten bei der Interprozessorkommunikation einspart. Das Updaten von Gitterelementen in aufeinanderfolgenden Relaxationsschritten wird solange durchgeführt, wie aufgrund der vom Prozessor selbst berechneten Zwischenergebnisse noch lokale Rechnungen ohne Kommunikation möglich sind. Geht es nicht mehr weiter, so werden die Elemente am Rand der lokalen Raum–Zeit–Abschnitte in *einem* Schritt ausgetauscht.

Lineare Rekurrenzen

Für eine lokale Algorithmenersetzung sind einfache lineare Rekurrenzen ein Paradebeispiel. Gewöhnlich als sequentielle Schleife der Art

```
for (i=2; i<=n; i++)    X[i] = (A[i] * X[i-1]) + B[i];
```

implementiert, die eine Datenabhängigkeit trägt, ist die Standardimplementierung $\Psi[\mathsf{FOLR}^{(1)}]$, unabhängig von den Datenaufteilungen von `X`, `A` und `B`, zu sequentieller Ausführung verurteilt — für größere Problemgrößen `n` eine teure Unterlassung von Optimierungsmöglichkeiten! Als Instanz des $\mathsf{FOLR}^{(1)}$–Musters erkannt, haben wir die Möglichkeit, diesen Code durch eine parallele Implementierung zu ersetzen, die wir durch Anwendung einer geeigneten Anzahl von rekursiven Verdopplungsschritten (*recursive doubling*, [KS73]) gewinnen. Die optimale Anzahl von Verdopplungsschritten (bis zu $\min(P, \log \mathtt{n})$ viele sind möglich) ist abhängig von der Problemgröße und der Startup–Zeit der Interprozessorkommunikation auf der Zielmaschine. Für kleine Problemgrößen wird die sequentielle Variante $\Xi[\mathsf{FOLR}^{(1)}]$ schneller sein, die, bis auf die Replizierung, null Verdopplungsschritten entspricht. Damit kann ein größerer Speed–down zuverlässig unterbunden werden.

Quasiskalare Vektor– und Matrixoperationen

Für quasiskalare Vektoroperationen ($\mathsf{VADD}^{(1)}$, $\mathsf{VMUL}^{(1)}$, ...) und Matrixoperationen ($\mathsf{MADD}^{(2)}$, $\mathsf{MMUL}^{(2)}$, ...) ist die Codeerzeugung recht einfach, weil sie keine interne Kommunikation induzieren. Wichtig ist jedoch, daß bei diesen recht häufig vorkommenden Operationen insbesondere die Optimierungsmöglichkeiten zur lokalen Vektorisierung und zur Cache–Nutzung ausgeschöpft werden.

Bietet die Zielmaschine auf den Knotenprozessoren gepipelinete Vektoreinheiten an, wie beispielsweise die TMC CM-5, so sollten die parallelen Implementierungen bereits vektorisiert sein. Ferner sollten auch bereits Optimierungen für die Belegung etwa vorhandener Vektorregister durchgeführt werden. Hierzu werden bei Blöcken von Vektoranweisungen ($\mathsf{VADD}^{(1)}$, $\mathsf{VMUL}^{(1)}$, ...) etwaige gemeinsame Unterausdrücke zusammengefaßt und die zu berechnende vektorielle Funktion als Vektor–DAG [Keß90] dargestellt. Für diesen Vektor–DAG wird eine Auswertungsreihenfolge mit möglichst geringem (Vektor–)Registerbedarf berechnet [Keß90, KPR91, KR93, KPR92]. Gerade bei Vektorerweiterungen mit kleinen Register–Files (CM-5, iPSC860) können die Techniken zur Vektorregisterbelegung aus [Keß90, KPR92] die Ausführung von Blöcken von Vektoranweisungen durch Vermeidung unnötiger Auslagerungen von Teilvektoren in den Hauptspeicher erheblich beschleunigen, indem ein Tradeoff zwischen Vektorregisterlänge (dem Schleifen–Blockungsfaktor entsprechend) und den durch Auslagerungen von Vektorregister–Inhalten in den Hauptspeicher verursachten Verzögerungen ausgenutzt wird.

Für maskierte Vektoroperationen wird Code erzeugt, indem zuerst Code zur Auswertung der Maskierungsbedingung erzeugt wird. Das Ergebnis wird in einem temporären Booleschen Maskenvektor zwischengespeichert. Die eigentlichen Vektoroperationen können dann in Abhängigkeit der Einträge in diesem Maskenvektor von gewissen Vektorprozessoren sehr effizient ausgeführt werden (single-instruction-loop oder butterfly-jump im SPARK 2.0, Saarbrücken [FOP+92]).

Caches können (nicht nur) bei parallelen Programmen seltsame Effekte hervorrufen. Ist das Programm schlecht auf die Größe des Caches abgestimmt, so kann dadurch viel Laufzeit mit unnötigem Umkopieren (thrashing) verlorengehen. Eine optimale Abstimmung hingegen kann unter gewissen Umständen sogar zu *superlinearem Speedup* führen, wenn nämlich die Granularität bei Parallelverarbeitung dann gerade klein genug wird, sodaß die Operanden noch in den Cache hineinpassen, nicht aber bei sequentieller Abarbeitung. Die Abstimmung auf die Cache–Größe kann durch Blocken von Schleifen erreicht werden (strip mining, tiling). Wir verweisen hierzu auf [WL91].

Treten in Blöcken von Matrixanweisungen auch Vektoren auf, sollten die in [KD94] beschriebenen Optimierungen (*subspace optimizations*) durchgeführt werden, um unnötige Berechnungen durch zu früh erfolgte Expansion dieser Vektoren zu Matrizen zu vermeiden.

Matrix–Vektor–Multiplikation

Bei Matrix–Vektor–Multiplikation ($\mathsf{MV}^{(2)}$) kann man die Schulmethode entweder als `ij`–Variante (die innere Schleife ist ein Skalarprodukt $\mathsf{SSP}^{(1)}$) oder als `ji`–Variante (die innere Schleife ist ein

VAADDSV$^{(1)}$) implementieren. Die Wahl ist abhängig von der Speicherungskonvention: So ist die erste Variante nach [FP92] vorzuziehen, wenn die Matrix transponiert ist. Die letztere Variante ist für Vektorprozessoren meist besser geeignet, weil VAADDSV$^{(1)}$ eine quasiskalare Vektoroperation ist.

Alternativ kann man MV$^{(2)}$ durch ein systolisches Verfahren [Ull84] implementieren; dies erscheint uns am geeignetsten für Transputer–Arrays mit vergleichsweise niedrigem Verhältnis aus Kommunikationszeit zu Rechenzeit.

Matrix–Matrix–Multiplikation

Für die Matrixmultiplikation MM$^{(3)}$ expandiert die Schulmethode zu sechs verschiedenen Möglichkeiten, da alle drei Schleifen beliebig vertauschbar sind. Hierbei gilt ähnliches wie bei der Matrix–Vektor–Multiplikation.

Als Alternative gibt es zum Beispiel einen systolischen Algorithmus für die Multiplikation quadratischer Matrizen (siehe [Ull84],[FP92] oder [CS88]), der aber nur dann effizienter als die Schulmethode werden könnte, falls `SHIFT`–Kommunikation (also unidirektionale vektorisierbare Kommunikation zum nächsten Torus–Nachbarn in je einer Dimension) auf der Zielmaschine schnell ausführbar ist:

Beispiel 7.2 Systolische Multiplikation quadratischer Matrizen, Prinzip

forall $i, j \in [1:n][1:n]$ **do in parallel**
    verschiebe $A[i][(i+j) \bmod n]$ zum Prozessor von $C[i][j]$;
    verschiebe $B[(i+j) \bmod n][j]$ zum Prozessor von $C[i][j]$;
od
for $(k=1;\ k<=n;\ k++)$ **do in sequential**
    `SHIFT`$(B, [+1, 0])$;
    `SHIFT`$(A, [0, +1])$;
    forall $i, j \in [1:n][1:n]$ **do in parallel**
        $C[i][j] \ = \ C[i][j] + A[i][j] * B[i][j]$;
    od
od
forall $i, j \in [1:n][1:n]$ **do in parallel**
    verschiebe A wieder in seine Ausgangsposition;
    verschiebe B wieder in seine Ausgangsposition;
od

Die sehr gute Vektorisierbarkeit macht dieses Verfahren vor allem für SIMD–Maschinen erwägenswert. Man sieht, daß zwar im allgemeinen das doppelte Kommunikationsvolumen anfällt wie bei der Schulmethode, daß die Kommunikation aber einfacher strukturiert ist und jeder Prozessor nur wenig Speicherplatz für seine lokalen Blöcke von A, B und C benötigt, während er bei der Schulmethode und Totalreplikation eines Feldes bis zu n^2 Daten auf einmal empfangen und speichern muß. Welches Verfahren auf der Zielmaschine für welche Problemgrößen besser ist, kann nur durch Experimente herausgefunden werden.

Man sieht außerdem, daß die Datenaufteilung (genauer gesagt, die Prozessorabbildung) während des systolischen Programms variiert, aber am Ende wieder die gleiche ist wie am Anfang[4] und daher nach außen hin invariant ist. ◇

Ähnliche systolische Verfahren gibt es auch für die LR–Zerlegung (LUD$^{(3)}$) und für VCONV$^{(2)}$ (Referenzen siehe [Ull84]).

[4] Die Rückschiebung von A bzw. B in die Ausgangsposition ist natürlich nicht erforderlich, falls A bzw. B nach der Multiplikation (vor einer eventuellen Überschreibung) nicht mehr benutzt werden.

7.1.3 Diskussion

Algorithmenersetzung als Programmtransformation Das lokale Ersetzen von Implementierungen (*algorithm replacement*) ist die komplexeste und stärkste Programmtransformation überhaupt und wird erst durch das Vorhandensein von Musterinstanzen ermöglicht.

1. Sie schließt alle anderen maschinenspezifischen Codeoptimierungen mit ein, da die Routinen der Implementierungsbibliothek von Spezialisten in Ruhe so lange getrimmt werden können, bis sie ein Maximum an Leistung auf der Zielmaschine erbringen. Insbesondere können dabei Optimierungen wieder eingeführt werden, die bei der Mustererkennung zunächst eliminiert wurden (z.B. das Abrollen oder Blocken von Schleifen).
2. Auch die passenden Kommunikationsroutinen (vgl. Abschnitt 3.6) werden von vornherein in die parallele Implementierung eingefügt.
3. Sie bildet ein Rahmenwerk zum Einbau aller bislang bekannten guten parallelen (!) Algorithmen, die für die jeweilige Klasse der Zielmaschine (Topologie, Granularität, Kommunikationseigenschaften) entwickelt wurden. All dieses Expertenwissen wird so für den Anwender verfügbar, ohne daß der diese Algorithmen oder die Hardware der Zielmaschine überhaupt kennen muß.
4. Sie ermöglicht endlich die — lokale — Abkehr von der vielgeschmähten, aber bisher in Übersetzern imperativer Programmiersprachen für Parallelrechner immer praktizierten owner-computes–Regel (vgl. Abschnitt 3.4).

Algorithmenersetzung vs. zyklische Codetransformation Eine weitere Transformation der ersetzten Implementierungen, wie etwa in [HACZ93] vorgeschlagen, ist hier nicht erforderlich. Insbesondere entfällt die dort skizzierte interaktive zyklische Optimierung aus Sequenzen von Transformation, Feldaufteilung, halbautomatischer Parallelisierung und Auswertung der Laufzeitabschätzung, die solange iteriert werden, bis der Benutzer zufrieden ist.

Der einzige bei uns noch verbleibende Freiheitsgrad ist die Bestimmung der Aufteilungen der beteiligten Felder; diese wird im nächsten Abschnitt behandelt.

Algorithmenersetzung und numerische Stabilität Nur in Ausnahmefällen darf die numerische Stabilität der parallelen Implementierung (etwas) schlechter sein als die des sequentiellen Äquivalents im Quellprogramm. Wo PARAMAT aus Effizienzgründen doch gegen diese Richtlinie verstoßen muß, wird eine Warnung an den Benutzer ausgegeben, sodaß dieser gegebenenfalls mit Hilfe der NoReplace[m]–Direktive intervenieren kann.

Alternative Codegenerierung mit HPF Als Alternative zu C + Inline–Assembler könnten wir auch HPF oder eine andere maschinennahe datenparallele „Hochsprache" als Ausgabesprache generieren. Damit würden wir allerdings von der Qualität des nachgeschalteten HPF– bzw. sonstigen Compilers abhängig. Außerdem verlören wir dadurch Information, weil manche unserer Muster nicht als HPF–Intrinsics vorkommen und damit nicht in der maximal möglichen Abstraktionsebene dargestellt werden könnten. Immerhin eröffnet uns dies die kommerziell nicht uninteressante Perspektive, mit relativ geringem Aufwand (Herausnehmen der Muster, die nicht als HPF–Intrinsics vorkommen; Aufnahme aller HPF–Intrinsics als Muster in die Musterbibliothek) einen automatischen Konverter von `FORTRAN 77` bzw. C nach HPF zu erhalten.

Andere Zielmaschinen Prinzipiell ist das genannte Verfahren auch auf SIMD–Zielmaschinen anwendbar. Bei Vektorrechnern entfallen die Datenaufteilungen; stattdessen werden Speicherablageschemata (z.B. [Mac87]) interessant, die einen optimalen Zugriff auf die Feldelemente ermöglichen. Eine analytische Laufzeitvorhersage dürfte im allgemeinen ausreichen.

Auch für Shared–Memory–Systeme könnte PARAMATs musterbasierte Code–Erzeugung nützlich sein. Die Komponenten zur Datenaufteilung und Laufzeitvorhersage müßten dann entfallen.

Muster	Konformpräferenz	Aufteilungsempfehlung
MCOPY(..., A, B)	$A \equiv B$	aufteilen
VCOPY(..., V, W)	$V \equiv W$	aufteilen
MJACOBI(A, B)	$A \equiv B$	in quadratische Blöcke aufteilen
MM(..., C, A, B)	$A^1 \equiv C^1 \;\vee\; B^2 \equiv C^2$	A zeilenweise, B repliziert oder A repliziert, B spaltenweise
VSUM(s, V)	keine	aufteilen
SSP(s, V, W)	$V \equiv W$	aufteilen

Tabelle 7.1 Konform–Präferenzen und Aufteilungsempfehlungen für die Standardparallelisierungen einiger Muster. Das Symbol $\equiv$ steht für das in Abschnitt 3.5.1 beschriebene Alignen aller gebundenen, ähnlich indizierten Felddimensionen. Die Aufteilungsempfehlungen (und damit auch die Begriffe „zeilenweise" und „spaltenweise") beziehen sich auf die entsprechenden gebundenen Dimensionen der Matrizen. „aufteilen" bedeutet, daß jede Aufteilung der gebundenen Dimension(en) sinnvoll ist.

Eine noch weiterführendere Möglichkeit wäre die Erzeugung von Special-Purpose–Chip–Spezifikationen aus den Musterinstanzen, um die parallele Implementierung direkt in Hardware zu „gießen“.

7.2 Mustergesteuerte Datenaufteilung

Da wir die sequentiellen Varianten, die Standardparallelisierungen und die parallelen Implementierungen aller Muster a priori genau kennen, können wir — durch Nachdenken oder durch Ausprobieren — zu jeder Implementierung jedes Musters Empfehlungen für lokal optimales Alignment (Konform–Präferenzen) und für lokal optimale Feldaufteilungen angeben. Die Aufteilungsempfehlungen sind abhängig von der Zielmaschine und von den Problemgrößen[5], die Alignmentpräferenzen sind jedoch zielmaschinenunabhängig. Beispielsweise sind die in Tabelle 7.1 angegebenen Präferenzen für die Standardparallelisierungen einiger Muster bei den meisten DMS und bei hinreichend großen Problemgrößen sinnvoll.

Verwendet man ein Branch–and–Bound–Suchverfahren nach [DHR94], so eignen sich die lokalen Aufteilungsempfehlungen hervorragend als Startkonfiguration für die globale Suche. Allerdings sollte [DHR94] um die Ausnutzung von Alignment–Präferenzen erweitert werden. Außerdem muß die Anzahl der wählbaren Datenaufteilungen erheblich eingeschränkt werden. Wir empfehlen hierzu bei Matrizen die zusammenhängenden Aufteilungen ‘spaltenweise’, ‘zeilenweise’ und ‘blockweise’ (Blockgrößen etwa im Verhältnis der Matrixextents), die zyklischen Aufteilungen ‘spaltenweise’ und ‘zeilenweise’, und die Totalreplikation. Bei Vektoren hat man analog nur die Wahl zwischen zusammenhängender und zyklischer Aufteilung sowie der Totalreplikation. Mit den genannten Aufteilungsmöglichkeiten kann man für alle Standardimplementierungen unserer Muster lokal optimale Aufteilungspräferenzen erreichen.

Allerdings erscheint uns das Verfahren aus [CKKM94, BKK93] wesentlich geeigneter zur Bestimmung guter Datenaufteilungen. Zum einen liefern nämlich die Musterinstanzen genau die in diesen Arbeiten vorausgesetzte Phasendarstellung des Quellprogramms; zum anderen stellt PARAMAT auch die dort benötigten Laufzeittabellen für jede Implementierung und jede Kombination von Datenaufteilungen zur Verfügung, sodaß die günstigen Optimierungszeiten von [BKK93] nicht durch die online–Berechnung von Laufzeitschätzungen beeinträchtigt werden.

[5] Wir nehmen hier ohne Einschränkung an, daß die NoParallel[]–Prädikate nicht zutreffen, d.h. tatsächlich die Standardparallelisierung oder eine parallele Implementierung ausgeführt werden.

7.3 Mustergesteuerte synthetische Laufzeitvorhersage

Zu jeder parallelen Implementierung aus der Implementierungsbibliothek gibt es eine zugehörige *Kostenfunktion*, die deren approximierte Laufzeit, in Abhängigkeit von Feldaufteilungen und Problemgrößen bei festen Maschinendaten (z.B. Prozessorzahl) angibt.

Die Approximation beruht auf Tabellen von gemessenen Laufzeitwerten für jede Implementierung bei jeder Kombination von Feldaufteilungen und bei jeder Kombination von Problemgrößen, die Zweierpotenzen sind. Ist eine Problemgröße nicht bekannt[6], so wird ein mittlerer Wert angenommen, der vom PARAMAT–Benutzer zu spezifizieren ist (z.B. 256); ansonsten werden, sofern keine Zweierpotenz, die entsprechenden Tabelleneinträge der benachbarten Problemgrößen interpoliert.

Zusätzlich benötigen wir, ebenso wie [CKKM94, BKK93], noch Tabelleneinträge für alle Kommunikationsroutinen, die z.B. beim Umverteilen von Feldern (Transponierung von verteilten Feldern) gebraucht werden.

Die Tabellen können von einem automatischen Benchmarking–Werkzeug generiert werden. Die Generierung der Tabellen wird sehr viel Rechenzeit kosten[7], muß aber nur einmal für jede Hardware–Konfiguration durchgeführt werden. Dieses Werkzeug ist ebenfalls Gegenstand zukünftiger Entwicklung von PARAMAT.

Offene Probleme Probleme mit der Laufzeitvorhersage stellen sich ein, wenn die Knotenprozessoren der Zielmaschine — und das trifft leider in den meisten Fällen zu — einen (Daten–)*Cache* haben. Zwar geht prinzipiell die vom Cache ausgenutzte Lokalität einer Musterimplementierung in die gemessenen Laufzeit–Zahlen ein; allerding hängt die Laufzeit einer Musterimplementierung aber auch davon ab, ob gewisse Operanden (Felder oder Teile von Feldern) bei Ausführungsbeginn bereits im Cache vorliegen und damit ein schnellerer Zugriff möglich ist. Dieses Szenario kann unter Umständen durch etliche vorangehende Operationen beeinflußt werden. Der Cache wirkt sich offenbar umso stärker auf die Gesamtlaufzeit aus, je kleiner die Granularität (lokale Problemgrößen) wird. Für kleine (lokale) Problemgrößen könnte man die Laufzeitvorhersage um einen Korrekturterm erweitern, der diesen Cache–Effekt, soweit zur Übersetzungszeit bekannt, modelliert.

Diskussion Wir betrachten eine parallele Implementierung eines Musters als eine *black box*. Wir beschreiben nicht, wie deren Laufzeit sich eigentlich verhalten müßte, sondern wie sie sich in der Realität verhält, was von ersterem erheblich abweichen kann.

Durch die synthetische Laufzeitvorhersage auf der Ebene größerer Codeteile (Musterinstanzen) erwarten wir einen Zuwachs an Genauigkeit gegenüber den analytischen Methoden (vgl. Abschnitt 3.7). Wir berücksichtigen damit (zumindest teilweise) das Cache–Verhalten aufgrund der der Implementierung innewohnenden Lokalitätseigenschaften, ferner die Überlappung von Kommunikation und Rechnung, sowie den durch die Zugriffsstrukturen der Implementierungen induzierten charakteristischen Netzwerkverkehr.

Bei den parallelen Implementierungen versagen z.T. die analytischen Leistungsvorhersagetechniken [Gup92, DHR94, Fah93], die für Standardparallelisierungen im Rahmen von halbautomatischen Parallelisierungssystemen und Compilern für datenparallele Programmiersprachen entwickelt wurden. Bei der synthetischen Laufzeitvorhersage für alle Musterimplementierungen haben wir aus dieser Not eine Tugend gemacht.

Als Nebenprodukt liefern die Laufzeittabellen ein umfassendes Leistungsspektrum der Zielmaschine, das auch für andere Zwecke, z.B. zur Optimierung der Hardware, herangezogen werden kann.

[6]In [Who91] tritt dieses Problem nicht auf, da dort die Auswahl der Datenaufteilung erst zur Laufzeit erfolgt, wenn alle notwendigen Parameter bekannt sind.

[7]Um Zeit zu sparen, kann man eine langwierige Rechnung nach einer bestimmten, hinreichend langen Zeit abbrechen und den Tabelleneintrag durch einen großen Wert HUGE bezeichnen.

7.4 Codeerzeugung für nicht erkannte Programmteile

Für einen nicht erkannten Teil des sequentiellen Quellprogramms erzeugt PARAMAT eine Standardparallelisierung gemäß Abschnitt 3.4.

Damit arbeitet die Codeerzeugung in diesem Fall genauso, als wenn das Programmstück als Inkarnation eines Musters m erkannt und die entsprechende NoReplace$[m]$–Direktive gesetzt wäre, mit der Ausnahme, daß keine Einträge in den Präferenzenlisten für Alignment und Feldaufteilungen sowie keine Laufzeitwerte vorliegen.

Entweder übernimmt man dann die Empfehlungen für Alignment und Datenaufteilungen von den anderen Vorkommen der beteiligten Felder in erkannten Programmteilen, oder man wendet zur Bestimmung von lokalem Alignment und lokalen Datenaufteilungen die in Abschnitt 3.5 beschriebenen Standardverfahren an.

7.5 Implementierung

Die Codeerzeugungsphase von PARAMAT ist Gegenstand aktueller und zukünftiger Forschung und Entwicklung. Daher können wir zu den genannten Themen außer den aufgeführten grundlegenden Eigenschaften noch keine Details beschreiben und auch noch nicht mit konkreten Laufzeitergebnissen aufwarten.

Eine volle Implementierung der PARAMAT–Codeerzeugung ist zwar zeitaufwendig, aber hochgradig modular und parallel ausführbar.

Abgesehen von der Parametrisierung in Datenaufteilungen und Problemgrößen entspricht der Implementierungsaufwand im wesentlichen dem einer parallelen Funktionenbibliothek mittlerer Größe wie etwa LAPACK (vgl. Abschnitt 5.4.4).

7.6 Verwandte Arbeiten

Die Idee, Mustererkennungstechniken im Bereich numerischer Anwendungen zum Zwecke der vollautomatischen Parallelisierung einzusetzen, ist nicht neu, wurde jedoch nie konsequent ausgeführt.

[BS87] enthält einen sehr vagen Vorschlag, Mustererkennungstechniken zur automatischen Parallelisierung im Rahmen eines formalen SMS–Parallelisierungssystems [Bra88] einzusetzen. Als Kandidaten für zu erkennende Muster werden Rekurrenzen und Reduktionen (Skalarprodukt) genannt.

Eave (Expert Adviser for VEctorization) [Bos88b, Bos88a] ist ein Expertensystem zur interaktiven Vektorisierung von `FORTRAN`–Programmen für den Vektorprozessor IBM 3090VF. Es enthält ein einfaches Mustererkennungswerkzeug, das Muster der Ordnung 1 (Vektoroperationen, Reduktionen) erkennt. Erkannte Muster leiten Optimierungstransformationen auf Quelltextebene an, wie z.B. Skalarexpansion, Schleifenvertauschung, Vektorisierung, Ausnutzung von Vektorregistern oder die Ersetzung teurer Operationen (Fließkomma–Exponentiation mit evtl. konstantem ganzzahligem Exponent) durch billigere Varianten (Sequenz von Multiplikationen). Bei der Datenabhängigkeitsanalyse werden einfache symbolische Vergleiche unterstützt; Datenflußanalyse und horizontale Mustererkennung sind hingegen nicht vorgesehen. Eine potentielle Umarbeitung von Eave für automatische Parallelisierung wird in [Bos88b] als nicht realisierbar abgetan.

Ähnlich arbeitet das Expertensystem von [WG89, Wan92] für SMS. Als Zwischendarstellung wurde dort der Programmabhängigkeitsgraph (PDG) nach [FOW87] gewählt. Das System wählt einen Teilbereich des Abhängigkeitsgraphen aus, analysiert ihn und führt Transformationen darauf aus, geleitet von einer Ansammlung von Heuristiken in einer hierarchisch organisierten Wissensbasis. Diese Sequenz wird solange wiederholt, bis der Benutzer zufrieden ist oder die Geduld verloren hat.

[LC91] wendet Mustererkennungstechniken nur auf die Kommunikationsanweisungen eines bereits (für SMS) parallelisierten Programms an (vgl. Abschnitt 3.6). Dasselbe gilt für Aspar [IFKF90] (ebenfalls in Abschnitt 3.6 angeführt). Entsprechend begrenzt ist der jeweilige Mustervorrat. Im Gegensatz zu diesen Ansätzen wenden wir die Mustererkennung auch dazu an, die *Semantik* des Quellprogramms auf lokaler Ebene zu erfassen und geeignete *parallele Algorithmen* auszuwählen. Die geeigneten Kommunikationsroutinen sind als natürlicher Bestandteil bereits in der parallelen Implementierung enthalten.

Echte Algorithmenersetzung bietet dagegen der Vorschlag [PP91], der semantische Mustererkennung auf einem erweiterten und normalisierten Datenabhängigkeitsgraphen nach [FOW87] durchführt. Als erkennungswürdige Muster werden die BLAS–Routinen und Reduktionen sowie Rekurrenzen, Matrix–Transposition und FFT genannt. Der PDG jeder innersten Schleife wird dreimal repliziert, für eine initiale, eine „mittlere“ und eine abschließende Iteration. Wir sehen den Sinn dieser Konstruktion nicht recht ein, weil sie nur interessant ist bei ziemlich komplexen, Datenabhängigkeiten tragenden Schleifen mit Spezialfällen für die erste und/oder letzte Iteration (konstruierte Beispiele). Unser Mustererkennungsalgorithmus erkennt die von [PP91] anvisierten Muster mit wesentlich weniger Aufwand, und die exotischen Beispiele aus [PP91] würde PARAMAT auch erkennen, gäbe man ihm das entsprechende Muster mit den zugehörigen Schablonen. Der Aufwand bei der Erkennung von Mustern im PDG (die Ersetzungsregeln bilden eine Graphgrammatik) ist größer als bei Mustererkennung in Bäumen (Baumgrammatik, vgl. Abschnitt 4.2). Ferner muß die Normalisierung des PDG durch klassische Transformationen wie Schleifenabrollen, Schleifenvertauschung, Ersetzung temporärer Variablen und Skalarexpansion vom Benutzer geleistet werden, womit dies kein automatisches Verfahren mehr ist. — Sinnvoll erscheint uns hingegen der Vorschlag, Filter–Slots für (fast) jedes Muster aufzunehmen.

[BHMS91] enthält einen speziellen Übersetzer für Relaxationsalgorithmen in `Fortran 90`. Eine Reihe von zweidimensionalen Differenzensternen (vgl. Abbildung 5.1) können aus den `CSHIFT`–Primitiven im `Fortran 90`–Quellprogramm erkannt werden. Anschließend wird hochgradig optimierter Code für die TMC CM-2 generiert, der auch die Interprozessorkommunikation und die Cache–Nachladezeiten durch Ausnutzung der natürlichen Lokalität (Blockung/Abrollen von Schleifen) in diesen Anwendungen minimiert.

CMAX [SW93] übersetzt interaktiv FORTRAN77–Programme nach CM–Fortran [Sab92a, Sab92b], eine Fortran 90 ähnliche Erweiterung von FORTRAN77 um Array–Syntax und Standardfunktionen. CMAX erkennt (syntaktisch!) einige gängige Schleifenkonstruktionen (Äquivalente zu Vektoroperationen, VCONDASS$^{(1)}$, PREVSUM$^{(1)}$, VSUM$^{(1)}$, VMAXVAL$^{(1)}$, VMAXLOC$^{(1)}$, SSP$^{(1)}$, MM$^{(3)}$), d.h. eine Untermenge der entsprechenden PARAMAT–Schablonen. Da nicht zwischen Mustern und Schablonen unterschieden wird, ist die CMAX–Mustererkennung schwächer als die von PARAMAT, z.B. erkennt CMAX nur 14 Livermore Loops, kann beispielsweise Kernel 24 (VMINLOC$^{(1)}$) nicht erkennen. Die Stärken von CMAX liegen zum einen in der interaktiven Fensterarbeit, zum anderen ist die speziell auf Fortran–Codes zugeschnittene CMAX–Mustererkennung in der Lage, Fortran–spezifische Speicherungskonventionen maschinenunabhängig und damit aufteilbar umzugestalten. Die Datenaufteilungen der Felder muß der Benutzer interaktiv durchführen. CM–Fortran läßt dabei keine statische Umverteilung von Feldern zu.

[CHZ91b, CHZ91a, CFZ93] skizziert ein System, das mittels Mustererkennung auf den Feldindizierungen für spezielle Fälle — BLAS–Routinen, Reduktionen oder einfache Gitterrelaxationen — Alignment– und Aufteilungsempfehlungen ableitet. Die Ableitung semantischer Eigenschaften oder gar eine Algorithmenersetzung wie bei PARAMAT ist in diesem Ansatz nicht vorgesehen. Das geplante System basiert auf der Leistungsvorhersage von [Fah94, Fah93] im Rahmen des Vienna Fortran Compilation Systems zur Einschätzung der Qualität von Datenaufteilungen. Eine detaillierte Ausarbeitung oder Implementierung existiert nicht.

Ein im Dezember 1993 erschienener Entwurf eines weitergehenden interaktiven Expertensystems (eXPert Adviser) [HACZ93] lehnt sich an die Techniken aus [Bos88b] und [WG89] an. Hier will man eine benutzergeführte, iterative Optimierung von `FORTRAN`–Programmen im Rahmen des Vienna Fortran Compilation System (VFCS) erreichen, die durch Mustererkennungstechniken unterstützt werden soll. Auch Algorithmenersetzung ist geplant. Letztlich ist dieser Ansatz jedoch auf das VFCS beschränkt.

Ähnlich arbeitet der regelbasierte Übersetzer Partool [BPS92, Bre93] für die datenparallele Programmiersprache Booster [Paa92]. Eine Reihe von zielmaschinenspezifischen Optimierungstransformationen in der Form von Termersetzungsregeln kann auf die Zwischendarstellung V-cal von Booster angesetzt werden. Deren Effekt wird mit Hilfe eines Laufzeitvorhersagesystems beurteilt. Gegenwärtig wird versucht, die PARAMAT–Musterbibliothek für dieses System zu übernehmen und anzupassen [Tre94].

Auch an der Universität Kaiserslautern ist man darum bemüht, das Mustererkennungswerkzeug von PARAMAT vor einen wissensbasierten Codeerzeuger zu spannen [HKS94]. Die Zielmaschine ist der dort entwickelte rekonfigurierbare Feldrechner Xputer.

8 Zusammenfassung

Wir haben in dieser Arbeit das PARAMAT–System zur vollautomatischen Parallelisierung einer breiten Klasse von numerischen Anwendungsprogrammen beschrieben, die auf dichtbesetzten Matrizen arbeiten.

Die wichtigsten Komponenten von PARAMAT sind eine leistungsfähige Mustererkennung und die automatische Codeerzeugung mit lokaler Algorithmenersetzung, automatischer Datenaufteilung und synthetischer Laufzeitvorhersage.

PARAMAT nutzt die in Kapitel 5 festgestellte Tatsache aus, daß viele numerische Anwendungen, die auf dichtbesetzten Matrizen arbeiten, aus einer relativ kleinen Menge typischer Bausteine, semantischen Mustern, zusammengesetzt sind. Diese Muster haben wir in Kapitel 5 vorgestellt.

Wir haben ferner Semantik–basierte Schablonen zum Erkennen dieser Muster formuliert (Anhang B), sodaß unter Ausnutzung der natürlichen semantischen Hierarchie der Muster mit einer linearen Anzahl von Schablonen exponentiell viele syntaktische Variationen eines Musters erkannt werden können.

Wir haben in Kapitel 6 ein System zur Mustererkennung in numerischen C–Programmen im Detail vorgestellt. Das System arbeitet schnell, flexibel und ist robust gegenüber typischen Programmtransformationen. Es erkennt die in Kapitel 5 beschriebenen semantischen Muster, indem es zusätzlich zum üblichen 'vertikalen' deterministischen bottom–up–Treepatternmatching auch auf Feld–Datenfluß basierende Querkanten verschiedener Typen berechnet und 'horizontale' Mustererkennung in gerichteten azyklischen Graphen aus diesen Datenflußkanten betreibt. Die Querkanten werden mit Hilfe eines von uns weiterentwickelten Feldzugriffs–Deskriptors berechnet.

Wir haben sodann ein System zur Codeerzeugung skizziert, das aus der von Instanzen erkannter Muster gelieferten Information optimalen Nutzen bei der Bestimmung guter Feldaufteilungen zieht. Kernstück ist das lokale Einsetzen als effizient bekannter paralleler Algorithmen für die Musterinstanzen. Alternativ kann der PARAMAT–Benutzer auch Standardparallelisierungen für manche Codeteile oder sogar Sequentialisierung durch globale Codeerzeugungsdirektiven erzwingen.

Die Sammlung paralleler Algorithmen (die man in der Künstlichen Intelligenz als 'Wissensbasis' bezeichnen würde), bietet ein umfassendes Rahmenwerk zur Einarbeitung beliebiger problemspezifischer Optimierungstransformationen. Die Algorithmenersetzung eröffnet den Zugang zu Expertenwissen über numerische Eigenschaften der Muster und ihrer Implementierungen. Sie erweist sich als tragendes und leistungsstarkes Instrument effizienter automatischer Parallelisierung ohne jedwede Hilfen seitens des Programmierers oder des PARAMAT–Benutzers. Daher ist PARAMAT ein absichtlich nicht–interaktives Parallelisierungssystem. Weil der PARAMAT–Benutzer die parallelisierte Version seines Programms nicht wiedererkennen muß, kann PARAMAT wesentlich aggressivere Codeoptimierungen durchführen, als dies in interaktiven „WYSIWYG"–Parallelisierern möglich ist. Da der Benutzer nicht mehr von Hand parallelisiert, ist der erzeugte parallele Code (sofern man von eventuellen Programmierfehlern in PARAMAT absieht) korrekt; ein zeitaufwendiges Debuggen paralleler Programme entfällt.

Eine Tabelle (die man ebenfalls als 'Wissensbasis' bezeichnen könnte), gibt zu jeder Musterimplementierung Präferenzen zu Alignment und Aufteilung der beteiligten Felder. Das auf der Phasenstruktur von Programmen aufsetzende Verfahren von [BKK93] zur automatischen Feldaufteilung wird von PARAMAT optimal ergänzt. Die erforderlichen Laufzeitschätzungen werden durch vorab gemessene Einträge in Laufzeittabellen für jede Implementierung sowohl zur Übersetzungszeit schneller als auch exakter als bei ausschließlicher Anwendung analytischer Verfahren zur Laufzeitvorhersage.

PARAMAT macht automatisch die langjährigen Erfahrungen von Experten der Bereiche Parallele Algorithmen, Programmierung von Parallelrechnern sowie Code–Optimierung für alle Autoren wissenschaftlicher Programme verfügbar, kann so die Wiedererfindung des Rades in zahllosen Parallelisierungsprojekten vermeiden, und bietet damit den Programmierern aus vielen Teilbereichen der Wissenschaft die Möglichkeit, sich auf ihre eigene, fachbezogene Forschungsarbeit zu konzentrieren, anstatt ihre geistigen Kräfte an die zeitraubende Programmierung schnell veraltender Parallelrechner zu verschwenden.

A Details zu *geqdescr*, *disjdescr*, und *neighbdescr*

In diesem Anhang beschreiben wir einige Details zur symbolischen Berechnung der Prädikate *geqdescr*, *disjdescr*, und *neighbdescr* (vgl. Abschnitt 6.3.4). Wir benutzen eine geometrische Veranschaulichung dieser Operationen.

A.1 Einschluß von Deskriptoren: *geqdescr*

Wir zeigen, wie das Inklusions–Prädikat $geqdescr(d_1, d_2)$ zweier Deskriptoren d_1 und d_2 desselben d–dimensionalen Feldes in Zeit $O(d)$ berechnet werden kann. Aus schon in Abschnitt 6.3.5 genannten Gründen (sowie wegen besserer Veranschaulichung) beschreiben wir hier den wichtigsten, den zweidimensionalen Fall.

Ist einer der beiden Deskriptoren von niedrigerer Dimensionalität als der andere, dann „pumpen" wir ihn auf die höhere Dimensionalität auf, indem wir geeignete dummy–ranges mit Extent 1 hinzufügen, um den Vergleich zu vereinfachen. Falls danach nicht die selben Dimensionen in d_1 und d_2 gebunden sind, können wir FALSE zurückgeben.

Eine notwendige Bedingung für $geqdescr(d_1, d_2)$ ist in jedem Fall, daß die Wertebereiche der Indexausdrücke von d_1 die entsprechenden von d_2 in *jeder* Dimension enthalten (bezeichnet durch $geqranges(d_1, d_2)$). Man beachte, daß dazu nicht nur die Unter– und Obergrenzen von d_1 diejenigen von d_2 einschließen müssen, sondern auch die Schrittweiten von d_2 ein ganzzahliges (≥ 1) Vielfaches derer von d_1 sein müssen und die Bereiche jeweils mindestens einen gemeinsamen Indexwert (z.B. eine Unter– oder Obergrenze von d_2) haben. Kann dies nicht nachgewiesen werden, weil entweder d_1 wirklich nicht d_2 einschließt, oder weil an irgendeiner Stelle ein symbolischer Vergleich nicht bewiesen werden konnte, so müssen wir FALSE zurückgeben, um die Konsistenz zu wahren. Im folgenden wollen wir annehmen, daß $geqranges(d_1, d_2)$ erfüllt ist.

Falls sowohl d_1 als auch d_2 Deskriptoren von Vollmatrizen sind, ist die Berechnung von *geqdescr* einfach: Es genügt dann, nachzuprüfen, daß im Falle des Vorliegens ausgeschlossener Indexwerte $d_1.excidx[]$ oder Diagonalen $d_1.excdiag[]$ bei d_1 diese auch bei d_2 ausgeschlossen sind.

Der Fall, daß sowohl d_1 als auch d_2 leer sind (also beide Diagonalmatrizen sind), ist ebenfalls leicht: Die *diag*–Bits müssen gleich sein, die Schrittweiten von d_2 müssen ganzzahlige Vielfache derer von d_1 sein, und mindestens ein adressiertes Feldelement muß beiden gemeinsam sein (z.B. eine Ecke von d_2).

Ist d_1 Deskriptor einer Vollmatrix, so brauchen wir nur die *excidx*– und *excdiag*–Einträge zu inspizieren.

Ist d_2 leer, testen wir, ob das *diag*–Bit von d_2 mit den Gebietsanzeigern von d_1 kompatibel ist, und ob beide Endpunkte von d_2 in d_1 enthalten sind. Ferner müssen wir in diesem Fall ausschließen, daß d_2 gerade mit einer eventuell vorhandenen ausgeschlossenen Diagonale von d_1 übereinstimmt.

Es bleibt der Fall, daß sowohl d_1 als auch d_2 trapezoidal sind. Um TRUE zurückgeben zu dürfen, müssen wir sicherstellen, daß kein Teil von d_2 in dem freien Dreieck von d_1 liegt.

Weil $geqranges(d_1, d_2)$ gilt, und weil wir die Gebietsanzeiger kennen, können wir die „gefährlichste Ecke" $P = (x_p, y_p)$ von d_2 bezüglich d_1 berechnen, die die Inklusionsrelation verletzen könnte (wie in der folgenden Abbildung). Falls wir $P \in_{sym} d_1$ zeigen können, dürfen wir TRUE zurückgeben. Dies wird durch Austesten für alle vier möglichen Kombinationen von Gebietsanzeigern erledigt.

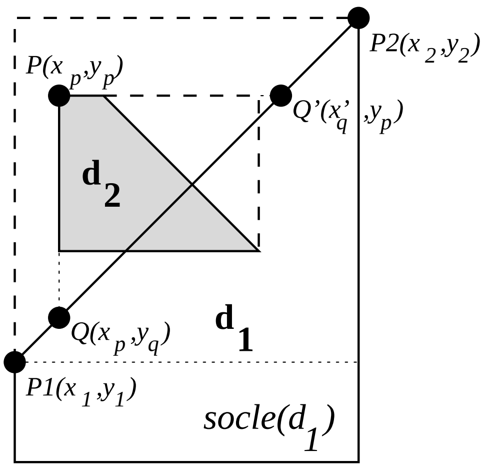
Als Beispiel geben wir hier die Berechnung für den Fall $d_1.areabits = \{BM, RI\}$ (siehe Abbildung rechts); die Rechnung für die übrigen Fälle ist analog.
Wir berechnen symbolisch *socle*(d_1) mit

$$y_1 = y_2 +_{sym} d_1.st[2] *_{sym} (x_2 -_{sym} x_1) /_{sym} d_1.st[1]$$

Dann ist es einfach, zu testen, ob $P \in socle(d_1)$. Falls dies erfüllt ist, sind wir fertig (TRUE). Andernfalls berechnen wir symbolisch

$$y_q = (x_p -_{sym} x_1) /_{sym} d_1.st[1] *_{sym} d_1.st[2] +_{sym} y_1.$$

Nun genügt es, zu prüfen, ob $y_p \geq_{sym} y_q$.
Alternativ könnten wir x'_q berechnen und die Bedingung als $x_p \geq_{sym} x'_q$ formulieren. Dies kann dann interessant sein, wenn die vorstehende Vergleich nicht geführt werden kann wegen zu komplexer symbolischer Ausdrücke.



Die Gesamtlaufzeit von *geqranges* ist $O(d)$, also konstant für alle in der Realität vorkommenden Dimensionalitäten (≤ 5; bei uns bisher ≤ 2). Selbiges gilt für *geqdescr*.

Für den dreidimensionalen Fall müssen nur ein paar Fälle mehr untersucht werden.

A.2 Disjunktheit von Deskriptoren: *disjdescr*

Nun zeigen wir die Berechnung von *disjdescr*(d_1, d_2) für zwei Deskriptoren d_1, d_2 eines d-dimensionalen Feldes. Der Einfachheit der Darstellung zuliebe beschreiben wir hier wiederum den zweidimensionalen Fall.

Eine *hinreichende* Bedingung für *disjdescr*(d_1, d_2) ist in jedem Fall, daß die Indexbereiche der einzelnen Dimensionen in d_1 and d_2 in mindestens einer Dimension verschieden sind[1], bezeichnet durch *disjranges*(d_1, d_2). Kann dies gezeigt werden, so sind wir fertig und geben TRUE zurück. Also nehmen wir für die folgenden Betrachtungen an, daß *disjranges*(d_1, d_2) nicht gezeigt werden konnte.

Beschreiben sowohl d_1 als auch d_2 Vollmatrizen, können wir FALSE zurückgeben, weil d_1 und d_2 entweder wirklich überlappen oder *disjranges* wegen eines nichtauflösbaren symbolischen Ausdrucks versagt hat.

Falls sowohl d_1 als auch d_2 leer sind (Diagonalmatrizen), so gilt *disjdescr*(d_1, d_2), wenn für gleiche *diag*–Bits die Bereiche sich in mindestens einer Grenze unterscheiden, oder wenn wir für komplementäre *diag*–Bits einen gemeinsamen Punkt der beiden Diagonalen ausschließen können.

Ist einer der Deskriptoren voll und der andere leer, so testen wir, ob der letztere in die *excdiag*[]–Einträge des ersteren paßt. Ist ein Deskriptor voll und der andere von (ursprünglich) niedrigerer Dimensionalität, so testen wir, ob der letztere in die entsprechenden *excidx*[]–Einträge des ersteren paßt.

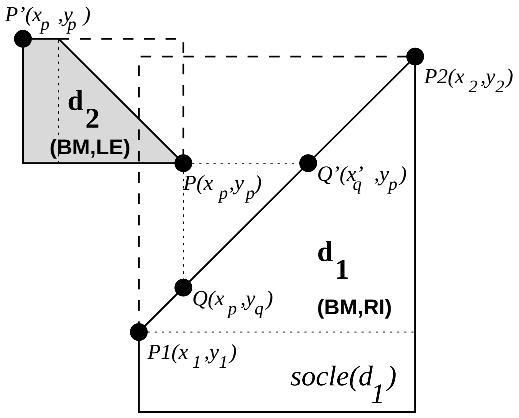
Es verbleibt der Fall, daß einer der Deskriptoren d_1, d_2 trapezoidal und der andere trapezoidal oder leer ist. Obwohl *disjranges* versagt hat, bleibt dennoch die Möglichkeit, daß ein Deskriptor zum Teil im freien Dreieck des anderen liegt, aber nicht mit ihm überlappt.

Der Einfachheit wegen machen wir hier die konservative Annahme[2], daß alle Schrittweiten gleich 1 sind. Damit erhalten wir eine rein geometrische Betrachtung des Schnittproblems.

[1] Dies deckt auch den Fall ab, daß die Bereiche in mindestens einer Dimension gerade verschränkt sind, sodaß z.B. in mindestens einer Dimension die ungeraden Indexwerte von d_1 und die geraden von d_2 adressiert werden.

[2] Wir verlieren durch diese Vereinfachung kaum an Genauigkeit: Falls die Deskriptoren beweisbar verschränkt gewesen wären, wäre *disjranges* erfolgreich gewesen. In der Praxis treten trapezoidale Deskriptoren mit Schrittweiten ungleich 1 selten auf.

Zunächst testen wir (mit *disjranges*), ob die Sockel (sofern vorhanden) disjunkt sind. Falls nicht, kehren wir mit FALSE zurück. Ansonsten gehen wir wie im vorangehenden Abschnitt bei *geqdescr* vor: Wir inspizieren die Gebietsanzeiger, und aus diesen ermitteln wir die d „gefährlichsten Ecken“ $P, P', \ldots$ von d_2 bezüglich d_1, die die Disjunktheitsrelation verletzen könnten (siehe Abbildung rechts).
Falls wir $P \notin_{sym} d_1$, $P' \notin_{sym} d_1$, ... zeigen können, dürfen wir TRUE zurückgeben; ansonsten müssen wir FALSE zurückgeben, um die Konsistenz zu wahren.
Die *point-location*–Aufgabe wird wiederum durch eine Fallunterscheidung aller vier möglichen Kombinationen von Gebietsanzeigern, analog zum Vorgehen von *geqdescr*, durchgeführt.



A.3 Nachbarschaft von Deskriptoren: *neighbdescr*

neighbdescr ist eigentlich ein Spezialfall von *disjdescr*. Weil nur disjunkte Deskriptoren echt benachbart sein können, ist *disjdescr* eine notwendige Vorbedingung für *neighbdescr*.

Die nächste notwendige Bedingung ist $d_1.areabits =_{bitw} d_2.areabits$, d.h. d_1 und d_2 müssen dieselbe Form haben. Weitergehend müssen die gebundenen Dimensionen von d_1 und d_2 identisch sein, und die Schrittweiten in jeder gebundenen Dimension müssen dimensionsweise gleich sein. Außerdem sollte keiner der Deskriptoren ausgeschlossene Indexwerte oder Diagonalen besitzen[3].

Als nächstes müssen wir die verschiedenen Möglichkeiten für die Gebietsanzeiger durchgehen:

Für Deskriptoren zu Vollmatrizen (siehe Abbildung 6.4(a)) testen wir, ob
(1) die Bereiche in $d-1$ Dimensionen gleich sind, und
(2) für die verbleibende gebundene Dimension $\tilde{d}$ entweder

$$\begin{aligned} d_1.ub[\tilde{d}] +_{sym} d_1.st[\tilde{d}] &=_{sym} d_2.lb[\tilde{d}] \quad \text{oder} \\ d_2.ub[\tilde{d}] +_{sym} d_2.st[\tilde{d}] &=_{sym} d_1.lb[\tilde{d}] \end{aligned}$$

gilt. Für leere Deskriptoren (vgl. Abbildung 6.4(b)) erhalten wir ähnliche Bedingungen, wobei in (1) die Deskriptoren in $d-2$ Dimensionen gleich sein müssen, und (2) für die zwei gebundenen Dimensionen $\tilde{d}, \tilde{d}'$, welche die Diagonale aufspannen, gilt.

Für trapezoidale Deskriptoren muß (1) für $d-2$ Dimensionen gelten, (2) muß für genau eine gebundene Dimension $\tilde{d}$ erfüllt sein, und für die übrige gebundene dimension $\tilde{d}'$ müssen die unteren und oberen Grenzen entsprechend der Gebietsanzeiger zusammenpassen (Fallunterscheidung). Sind beispielsweise die Gebietsanzeiger {BM,LE}, wie in Abbildung 6.4(b), so müssen wir zusätzlich prüfen, ob

$$d_1.lb[\tilde{d}'] +_{sym} (d_1.ub[\tilde{d}] -_{sym} d_1.lb[\tilde{d}]) /_{sym} d_1.st[\tilde{d}] *_{sym} d_1.st[\tilde{d}'] =_{sym} d_2.lb[\tilde{d}'] +_{sym} d_2.st[\tilde{d}']$$

oder

$$d_2.lb[\tilde{d}'] +_{sym} (d_2.ub[\tilde{d}] -_{sym} d_2.lb[\tilde{d}]) /_{sym} d_2.st[\tilde{d}] *_{sym} d_2.st[\tilde{d}'] =_{sym} d_1.lb[\tilde{d}'] +_{sym} d_1.st[\tilde{d}'].$$

[3]Dies ist eine Vereinfachung. Nachbarschaft kann auch bei ausgeschlossenen Indexwerten möglich sein. Eine entsprechende Untersuchung würde hier zuweit führen.

B Schablonen–Übersicht

Dieser Anhang enthält das Verzeichnis aller bereits realisierten Schablonen zu den in Kapitel 5 vorgestellten Mustern.

Soweit nicht explizit erwähnt, kann im Rumpf einer Schleife bzw. einer bedingten Anweisung jeweils keine weitere Anweisung mehr stehen als die angegebene. Variationsmöglichkeiten und semantische Bedingungen werden als Kommentarzeilen nach `--` angegeben.

Es gilt Unifikationssemantik, d.h. gleiche formale Variablen bedeuten, daß im Quellprogramm an diesen Stellen auch gleiche (aktuale) Variablen stehen müssen. Dies gilt auch für Feldreferenzen, die dann in allen Indexausdrücken (und in deren Reihenfolge!) übereinstimmen müssen. Allerdings dürfen gleiche aktuale Variablen auch an verschiedene formale gebunden werden, sofern dies nicht explizit verboten wurde. Die Flags ABS, NEG und INV dürfen bei allen Variablen beliebig gesetzt sein, sofern nicht anders erwähnt.

Indizierungen der Form `x[i]` sind, sofern nicht anders erwähnt, wie folgt zu interpretieren: In genau einem Indexausdruck dieser Feldreferenz darf die Variable `i` vorkommen. Dieser Indexausdruck darf eine lineare Funktion in `i` sein (`issimpleidx(x,i)`), d.h. `x[i]` steht abkürzend für `x[..]..[a*i+b]..[..]`, wo `a`, `b` von `i` unabhängige Variablen oder Konstanten sind.

Ferner gelten jeweils auch die schon in Kapitel 5 angegebenen Nebenbedingungen.

B.1 Schablonen zu skalaren Mustern

Das Wesentliche zum Erkennen von MULTIADD$^{(0)}$ und MULTIMUL$^{(0)}$ wurde bereits in Abschnitt 6.2.7 gesagt; das Erkennen von Differenzensternen (HSTAR$^{(0)}$, STAR$^{(0)}$) wurde in Abschnitt 6.2.8 beschrieben.

Standardausdrücke und Zuweisungen

SINIT$^{(0)}$

Schablone: `SINIT(x,c)`

```
x = c;    -- c ist eine numerische Konstante
```

SCOPY$^{(0)}$

Schablone: `SCOPY(x,y)`

```
x = y;    -- y ist eine Variable
```

Binäre Operationen:

ADD | AADD | MUL | AMUL | MOD$^{(0)}$ | AMOD$^{(0)}$

AND$^{(0)}$ | OR$^{(0)}$ | EQ$^{(0)}$ | LT$^{(0)}$ | GT$^{(0)}$ | LE$^{(0)}$ | GE$^{(0)}$ | NE$^{(0)}$

Schablone 1: `BINOP(_,x,y)` aus

```
x binop y;  -- oder y binop x, falls binop kommutativ
            -- bei AADD, AMUL, AMOD: eqex(z,x).
            -- bei ADD, MUL, MOD: not eqex(x,y), not eqex(x,z)
            -- binop in { +, *, %, &&, ||, ==, <, >, <=, >=, != }
```

Schablone 2: `BINOP(z,x,y)` aus

```
z = BINOP(_,x,y); -- BINOP in { ADD, AADD, MUL, AMUL, MOD, AMOD,
                                AND, OR, EQ, LT, GT, LE, GE, NE }
```

Unäre Operationen:

$\boxed{\mathsf{SIN}^{(0)}}$ $\boxed{\mathsf{COS}^{(0)}}$ $\boxed{\mathsf{TAN}^{(0)}}$ $\boxed{\mathsf{EXP}^{(0)}}$ $\boxed{\mathsf{LOG}^{(0)}}$ $\boxed{\mathsf{SQRT}^{(0)}}$ $\boxed{\mathsf{TWOTO}^{(0)}}$

Schablone 1: `Unop(_,x)` aus

```
unop( x );  -- unop in { sin, cos, tan, exp, log, sqrt, twoto }
```

Schablone 2: `Unop(z,x)` aus

```
z = Unop( x );  -- Unop in { SIN, COS, TAN, EXP, LOG, SQRT, TWOTO }
```

$\boxed{\mathsf{POWER}}$

Schablone 1: `POWER(_,a,b,1.0)` aus

```
pow(a,b);                -- pow() math. Standardfunktion
```

Schablone 2: `POWER(_,a,b,init)` aus

```
init * POWER(_,a,b,1.0);   -- Multiplikation ist kommutativ
```

Schablone 3: `POWER(x,a,b,init)` aus

```
x = POWER(_,a,b,init);
```

Schablone 4: `POWER(x,a,b,x)` aus

```
for (i=1; i<=b; i++)  -- Grenzen beliebig, sofern Differenz = b-1
  AMUL(x,x,a);
```

Schablone 5 (cross, FLOW): `POWER(x,a,b,init)` aus

```
x = init;
POWER(x,a,b,x);
```

$\boxed{\mathsf{ADDMUL}^{(0)}}$

Schablone 1: `ADDMUL(_,a,b,c)` aus

```
a + MUL(b,c)         -- Addition ist kommutativ
```

Schablone 2: `ADDMUL(x,a,b,c)` aus

```
x = ADDMUL(_,a,b,c);
```

$\boxed{\mathsf{AADDMUL}^{(0)}}$

Schablone: `AADDMUL(a,a,b,c)` aus

```
a = ADDMUL(_,a,b,c);
```

$\boxed{\mathsf{ADDMULTIMUL}^{(0)}}$

Schablone 1: `ADDMULTIMUL(_,x,a,b,...)` aus

```
x + MULTIMUL(_,a,b,...);      -- Addition ist kommutativ
```

Schablone 2: `ADDMULTIMUL(z,x,a,b,...)` aus

```
z = ADDMULTIMUL(_,x,a,b,...);
```

$\boxed{\mathsf{AADDMULTIMUL}^{(0)}}$

Schablone: `AADDMULTIMUL(x,x,a,b,...)` aus

```
x = ADDMULTIMUL(_,x,a,b,...);
```

$\boxed{\mathsf{MULADDMUL}^{(0)}}$

Schablone 1: `MULADDMUL(_,x,y,u,v)` aus

```
x * ADDMUL(_,y,u,v)      -- Multiplikation ist kommutativ
```

Schablone 2: `MULADDMUL( z,x,y,u,v)` aus

```
z = MULADDMUL(_,x,y,u,v);
```

$\boxed{\mathsf{MULMUL}^{(0)}}$

Schablone 1: `MULMUL(_,b,c,d,e)` aus

```
MUL(_,b,c) + MUL(_,d,e)  -- Addition ist kommutativ
```

Schablone 2: `MULMUL( z,b,c,d,e)` aus

```
z = MULMUL(_,b,c,d,e);
```

Swapping

$\boxed{\mathsf{SWAP}^{(0)}}$

`SWAP(x,y,t)`

```
SCOPY(t,x);
...         -- keine Redefinition von t,x,y; kein Lesen von y,t
SCOPY(x,y);
...         -- keine Redefinition von t,x,y; kein Lesen von x,t
SCOPY(y,t);
```

Bemerkung: Die $\mathsf{SWAP}^{(0)}$–Instanz wird an die Stelle einer der drei $\mathsf{SCOPY}^{(0)}$–Instanzen gesetzt, sodaß die Semantik der übrigen Anweisungen dazwischen nicht beeinflußt wird (Vorsicht bei Lesen von x,y,t); die beiden anderen Instanzen werden ersatzlos entfernt.

$\boxed{\mathsf{ROT}}$

Schablone (cross, ANTI/ANTI): `ROT(a,b,t,c,s)` aus

```
MULMUL(t,c,a,s,b);
...            -- keine Redefinition, kein Lesen von t,c,a,s,b
MULMUL(b,c,b,s,a);
...            -- keine Redefinition, kein Lesen von t,c,a,s,b
SCOPY(a,t);
```

Bemerkung: Die ROT–Instanz wird an die Stelle der $\mathsf{SCOPY}^{(0)}$–Instanz gesetzt, die beiden anderen Instanzen werden ersatzlos entfernt.

Bedingte Zuweisungen

$\boxed{\mathsf{MAXVAL}^{(0)}}$

Schablone 1: `MAXVAL(m,a,b)` aus

```
if (a>b)           -- Bedingung auch umstellbar, auch ">="
      SCOPY(m,a);
else  SCOPY(m,b);
```

Schablone 2: `MAXVAL(_,a,b)` aus

```
max(a,b);        -- max() math. Standardfunktion, kommutativ
```

Schablone 3: `MAXVAL(m,a,b)` aus

```
m = MAXVAL(_,a,b);
```

Schablone 4: `MAXVAL(a,a,b)` aus

```
if (a<b)        -- Bedingung auch umstellbar (<=)
   SCOPY(a,b);
```

$\boxed{\mathsf{MINVAL}^{(0)}}$ analog (4 Schablonen, mit denen für $\mathsf{MAXVAL}^{(0)}$ vereinigt)

$\boxed{\mathsf{MAXLOC}^{(0)}}$

Schablone: `MAXLOC(k,x,i,k)` aus

```
if ‘abs’(x[i])> ‘abs’(x[k])   -- Bedingung auch umstellbar
   SCOPY(k,i);
```

$\boxed{\mathsf{MINLOC}^{(0)}}$ analog

$\boxed{\mathsf{MAXLOCM}^{(0)}}$ analog

$\boxed{\mathsf{MINLOCM}^{(0)}}$ analog

$\boxed{\mathsf{MAXVL}^{(0)}}$

Schablone 1: `MAXVL ( p, m, x, a )` aus

```
if (‘abs’(x[i])>m) {       -- Bedingung auch umstellbar
   SCOPY(m,‘abs’(x[i]));   -- Reihenfolge auch umgekehrt
   SCOPY(p,i);
}
```

Schablone 2: `MAXVL ( p, m, x, a )` aus

```
SCOPY(temp,‘abs’(x[i]));
if (temp>m) {              -- Bedingung auch umstellbar
   SCOPY(m,temp);          -- Reihenfolge auch umgekehrt
   SCOPY(p,i);
}
```

$\boxed{\mathsf{MINVL}^{(0)}}$ analog (2 Schablonen, mit denen von $\mathsf{MAXVL}^{(0)}$ gemeinsam realisiert)

$\boxed{\mathsf{MAXVLM}^{(0)}}$ analog (2 Schablonen)

$\boxed{\mathsf{MINVLM}^{(0)}}$ analog (2 Schablonen, mit denen von $\mathsf{MAXVLM}^{(0)}$ gemeinsam realisiert)

$\boxed{\mathsf{CONDASS}^{(0)}}$

`CONDASS(y,cond,E1,E2,v11,v12,...,v21,v22,...)` aus

```
if  (cond)
      E1PATTERN(y,v11,v12,...);  -- erkannt
else  E2PATTERN(y,v21,v22,...);  -- erkannt
```

Der `else`–Teil kann entfallen.

Sonstiges

$\boxed{\mathsf{STRANGE}^{(0)}}$ ist ein instabiles Muster, das jeden (!) skalaren Ausdruck matcht, für den kein anderes Muster gefunden werden konnte.

B.2 Schablonen zu Mustern der Ordnung 1

Skalare Iterationen

$\boxed{\mathsf{FSUM}^{(1)}}$

Schablone: `FSUM(i,res,init,function,v1,v2,...)` aus

```
res = init;     -- kann entfallen
for (i=LB; i<=UB; i++)
   AADD(res,res,function(v1,v2,...)); -- function ist skalarer Term
```

`i` kann in `v1, v2,...` vorkommen, aber nicht in `res`.

Quasiskalare Vektoroperationen

In all diesen Schablonen kann die Schleifenrichtung auch umgekehrt (abwärts laufend) sein, da sie keine Datenabhängigkeiten trägt. Ebenso können auch Schrittweiten, deren Betrag größer als 1 ist, auftreten.

$\boxed{\mathsf{VCOPY}^{(1)}}$

`VCOPY(i,x,y)` aus

```
for (i=LB; i<=UB; i++)
   SCOPY(x[i],y[i]);
```

$\boxed{\mathsf{VASSIGN}^{(1)}}$

`VASSIGN(i,x,v)` aus

```
for (i=LB; i<=UB; i++)
   SCOPY(x[i],v);      -- notoccurin(v,i), isvar(v)
```

$\boxed{\mathsf{VASSIGNSP}^{(1)}}$

`VASSIGNSP(i,x,s,v1,v2,...)` aus

```
for (i=LB; i<=UB; i++)
s:  FN(x[i],v1,v2,...); -- notoccurin(v1,i), notoccurin(v2,i), ...
```

`FN` steht für eine beliebige (skalare) Musterinstanz, die an den Knoten `s` annotiert ist.

$\boxed{\mathsf{VINIT}^{(1)}}$

`VINIT(i,x,c)` aus

```
for (i=LB; i<=UB; i++)
   SINIT(x[i],c);
```

$\boxed{\mathsf{VINITSP}^{(1)}}$

`VINITSP(i,x,s)` aus

```
for (i=LB; i<=UB; i++)
s:   FN(x[i],v1,v2,...); -- notoccuridx(v1,i), (v2,i), ...
```

`FN` steht für eine beliebige (skalare) Musterinstanz mit Slot–Einträgen `v1,v2,...`, die an den Knoten `s` annotiert ist. Die `v1, v2, ...` sind entweder konstant oder gleich `i`.

$\boxed{\mathsf{VADD}^{(1)}}$

Schablone: `VADD(i,x,y,z)` aus

```
for (i=LB; i<=UB; i++)
   ADD(x[i],y[i],z[i]);   -- not(eqfex(x,y)), not(eqfex(x,z))
```

$\boxed{\mathsf{VMUL}^{(1)}}$ $\boxed{\mathsf{VAND}^{(1)}}$ $\boxed{\mathsf{VOR}^{(1)}}$ analog

$\boxed{\mathsf{VSIN}^{(1)}}$ $\boxed{\mathsf{VCOS}^{(1)}}$ $\boxed{\mathrm{VTAN}}$ $\boxed{\mathrm{VEXP}}$ $\boxed{\mathrm{VLOG}}$ $\boxed{\mathsf{VMIN}^{(1)}}$ $\boxed{\mathsf{VMAX}^{(1)}}$

analog aus $\mathsf{SIN}^{(0)}$, $\mathsf{COS}^{(0)}$, $\mathsf{TAN}^{(0)}$, $\mathsf{EXP}^{(0)}$, $\mathsf{LOG}^{(0)}$, $\mathsf{MINVAL}^{(0)}$, $\mathsf{MAXVAL}^{(0)}$

$\boxed{\mathsf{VAADD}^{(1)}}$

Schablone: `VAADD(i,x,x,z)` aus

```
for (i=LB; i<=UB; i++)
   AADD(x[i],x[i],z[i]);
```

$\boxed{\mathsf{VAMUL}^{(1)}}$ analog

$\boxed{\mathsf{VINC}^{(1)}}$

Schablone: `VINC(i,x,y,c)` aus

```
for (i=LB; i<=UB; i++)
   ADD(x[i],y[i],s);        -- notoccurin(c,i)
```

$\boxed{\mathsf{VMOD}^{(1)}}$ analog

$\boxed{\mathsf{VAINC}^{(1)}}$

Schablone: `VAINC(i,x,x,c)` aus

```
for (i=LB; i<=UB; i++)
   AADD(x[i],x[i],c);        -- notoccurin(c,i)
```

$\boxed{\mathsf{VAMOD}^{(1)}}$ analog

$\boxed{\mathsf{VINV}^{(1)}}$

Schablone 1: `VINV(i,x,y,--)` aus

```
for (i=LB; i<=UB; i++)
   SCOPY(x[i],y[i]);        -- INV(y) gesetzt
```

Schablone 2: `VINV(i,x,y,c)` aus

```
for (i=LB; i<=UB; i++)      -- INV(y) gesetzt
  CONDASS(x[i], y[i]!=0.0, SCOPY(x[i],y[i]),SINIT(x[i],c),y[i],c);
```

VADDMUL$^{(1)}$

`VADDMUL(i,x,y,u,v)` aus

```
for (i=LB; i<=UB; i++)
   ADDMUL(x[i],y[i],u[i],v[i]);
```

SV$^{(1)}$

`SV(i,x,y,c)` aus

```
for (i=LB; i<=UB; i++)
   MUL(x[i],y[i],c);     -- Faktoren y[i],c vertauschbar
```

SVDIAG$^{(1)}$

`SV(i,x,y,c)` aus

```
for (i=LB; i<=UB; i++)
   MUL(x[i],y[i][i],c);  -- Faktoren y[i][i],c vertauschbar
```

VAADDSS$^{(1)}$

Schablone 1: `VAADDSS(i,x,x,u,v)`

```
for (i=LB; i<=UB; i++)
   AADDMUL(x[i],x[i],u,v);  -- Faktoren u,v sowie x[i] vertauschbar
```

Schablone 2 (cross, FLOW): `VAADDSS(i,x,x,u,v)` aus

```
MUL(temp,u,v);
...                  -- kein Lese- oder Schreibzugriff auf temp
VAADD(i,x,x,temp);  -- eqfex(temp auf lhs, temp auf rhs)
```

Bemerkung: Die VAADDSS$^{(1)}$–Instanz überschreibt die VAADD$^{(1)}$–Instanz. Die MUL–Instanz bleibt vorerst stehen. Durch dead–code–Eliminierung kann sie später beseitigt werden, falls `temp` nicht mehr benötigt wird.

VAADDSV$^{(1)}$

Schablone 1: `VAADDSV(i,x,x,y,v)` aus

```
for (i=LB; i<=UB; i++)
   AADDMUL(x[i],x[i],y[i],v);     -- Faktoren y[i],v vertauschbar
```

Schablone 2: `VAADDSV(i,x,x,y,v)` aus

```
if (v!=0)                  -- oder Bed. if (v>EPS), EPS vorgegeben
   VAADDSV(i,x[i],x[i],y[i],v);
```

Schablone 3 (cross, FLOW): `VAADDSV(i,x,x,y,v)` aus

```
SV(i,temp,y,v);
...                 -- kein Lese- oder Schreibzugriff auf temp
VAADD(i',x,x,temp); -- eqdescr(temp auf lhs, temp auf rhs)
```

Bemerkung: Die VAADDSV$^{(1)}$–Instanz überschreibt die VAADD$^{(1)}$–Instanz. Die SV$^{(1)}$–Instanz bleibt vorerst stehen. Durch dead–code–Eliminierung kann sie später beseitigt werden, falls `temp` nicht mehr benötigt wird.

$\boxed{\text{VADDSV}^{(1)}}$ analog (3 Schablonen)

$\boxed{\text{VAADDMULTISV}^{(1)}}$

Schablone 1: `VAADDMULTISV(i,x,x,y,v1,v2,...)` aus

```
for (i=LB; i<=UB; i++)
   AADDMULTIMUL(x[i],x[i],y[i],v1,v2,...);
    -- Faktoren y[i], v1, ... vertauschbar
```

Schablone 2 (cross, FLOW): `VAADDMULTISV(i,x,x,y,v1,v2,...)` aus

```
MULTIMUL(temp,v1,v2,...);
...                     -- kein Lese- oder Schreibzugriff auf temp
VAADDSV(i,x,x,y,temp); -- eqdescr(temp auf lhs, temp auf rhs)
```

Bemerkung: Die VAADDMULTISV$^{(1)}$–Instanz überschreibt die VAADDSV$^{(1)}$–Instanz. Die MULTIMUL$^{(0)}$–Instanz bleibt vorerst stehen. Durch dead–code–Eliminierung kann sie später beseitigt werden, falls `temp` nicht mehr benötigt wird.

$\boxed{\text{VMULMUL}^{(1)}}$

Schablone: `VMULMUL(i,x,a,b,c,d)` aus

```
for (i=LB; i<=UB; i++)
   MULMUL(x[i],a[i],b[i],c[i],d[i]);   -- issimpleidx(x,i), ...
```

$\boxed{\text{VLUD}^{(1)}}$

Schablone (cross, FLOW):

`VLUD(j,A[c][j],Q[c][k],A[c][j],A[c][k],A[k][k],Q[c][k],A[k][j],k,c)` aus

```
MUL(Q[c][k],A[c][k],A[k][k]);
...          -- kein Lese- oder Schreibzugriff auf Q[c][k]
VAADDSV(i=[k:n], A[c][j], A[c][j], Q[c][k], A[k][j]);
```

$\boxed{\text{GVOP}^{(1)}}$ Untermuster zu GVOP$^{(1)}$ bei vertikaler Mustererkennung sind STRANGE$^{(0)}$, MULADDMUL$^{(0)}$ und HSTAR$^{(0)}$/STAR$^{(0)}$ (3 Schablonen). Kontraktionen aus einzelnen elementweisen Vektoranweisungen entlang von FLOW–Querkanten per Inferenz sind möglich, aber im Prototypen noch nicht implementiert.

$\boxed{\text{VCONDASS}^{(1)}}$

Schablone: `VCONDASS(i,y,cond,M1,M2,s11,s12,..., s21,s22,...)` aus

```
for (i=LB; i<=UB; i+=ST)
  CONDASS(y[i], cond, M1, M2, s11,s12,..., s21,s22,...);
  -- issimpleidx(y,i), issimpleidx(cond,i), issimpleidx(s11,i),...
  -- fuer alle Operanden sxy: falls eqsym(y,sxy), dann eqfex(y,sxy)
```

Vektor–Dreieckstausch

VSWAP$^{(1)}$

Schablone 1: `VSWAP(i,x,y,t)` aus

```
for (i=LB; i<=UB; i++)
   SWAP(x[i],y[i],t{[i]});  -- t optional in i indiziert
```

Schablone 2 (cross, ANTI): `VSWAP(i,x,y,t)` aus

```
VCOPY(i,t,x);
...         -- keine Redefinition von t,x,y; kein Lesen von y,t
VCOPY(i',x,y);
...         -- keine Redefinition von t,x,y; kein Lesen von x,t
VCOPY(i'',y,t);
```

Bemerkung: Die VSWAP$^{(1)}$–Instanz wird an die Stelle einer der drei VCOPY$^{(1)}$–Instanzen gesetzt, sodaß die Semantik der übrigen Anweisungen dazwischen nicht beeinflußt wird (Vorsicht bei Lesen von x,y,t); die beiden anderen Instanzen werden ersatzlos entfernt. Zum Test auf Redefinition benutzen wir *disjdescr*(). Schablone 3: `VSWAP(i,x[i][k],x[i][l],t)` aus

```
if (l!=k)
   VSWAP(i,x[i][k],x[i][l],t{[i]});  -- t optional in i indiziert
```

VROT$^{(1)}$

Schablone 1: `VROT(i,a,b,t,c,s)` aus

```
for (i=LB; i<=UB; i++)
   ROT(a[i],b[i],t‘[i]’,c,s);   -- t optional in i indiziert
```

Schablone 2 (cross, ANTI/FLOW): `VROT(i,a,b,t,c,s)` aus

```
VMULMUL(i,t[i],c[i],a[i],s[i],b[i]);
...         -- keine Redefinition von t,x,y; kein Lesen von y,t
VMULMUL(j,b[j],c[j],b[j],s[j],a[j]);
...         -- keine Redefinition von t,x,y; kein Lesen von x,t
VCOPY(k,y[k],t[k]);
```

Bemerkung: Die VSWAP$^{(1)}$–Instanz wird an die Stelle der VCOPY$^{(1)}$–Instanz gesetzt, die beiden anderen Instanzen werden ersatzlos entfernt. Zum Test auf Redefinition benutzen wir *disjdescr*().

Vektor–Shift

VSHIFT$^{(1)}$

Schablone: `VSHIFT(i,x,x,k)` aus

```
for (i=LB; i<=UB; i++)
   SCOPY(x[i],x[i+-k]);  -- oder [i+-k] auf lhs, [i] auf rhs
```

1D–Reduktionen

$\boxed{\mathsf{VSUM}^{(1)}}$

Schablone 1: `VSUM(i,x,y,x)` aus

```
for (i=LB; i<=UB; i++)
   AADD(x,x,y[i]);
```

Schablone 2 (cross, FLOW): `VSUM(i,x,y,c)` aus

```
SINIT(x,c);
...        -- kein schreibender Zugriff auf x
VSUM(i,x,y,x)
```

Bemerkung: Die neue $\mathsf{VSUM}^{(1)}$–Instanz überschreibt die alte $\mathsf{VSUM}^{(1)}$–Instanz. Falls an der Stelle `...` noch andere Anweisungen stehen, die `x` lesen, muß die $\mathsf{SINIT}^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\mathsf{SVSUM}^{(1)}}$ analog aus `AADDMUL` (2 Schablonen)

$\boxed{\mathsf{VPROD}^{(1)}}$ analog (2 Schablonen)

$\boxed{\mathsf{VROR}^{(1)}}$ analog (2 Schablonen)

$\boxed{\mathsf{VRAND}^{(1)}}$ analog (2 Schablonen)

$\boxed{\mathsf{SSP}^{(1)}}$

Schablone 1: `SSP(i,x,u,v,x)` aus

```
for (i=LB; i<=UB; i++)
   AADDMUL(x,x,u[i],v[i]);
```

Schablone 2 (cross, FLOW): `SSP(i,x,u,v,c)` aus

```
VMUL(i',temp,u,v);
...        -- kein schreibender Zugriff auf temp
VSUM(i,x,temp,c)
```

Bemerkung: Die $\mathsf{SSP}^{(1)}$–Instanz überschreibt die $\mathsf{VSUM}^{(1)}$–Instanz. Falls an der Stelle `...` noch andere Anweisungen stehen, die `temp` lesen, muß die $\mathsf{VMUL}^{(1)}$–Instanz (vorerst) stehenbleiben.

Schablone 3 (cross, FLOW): `SSP(i,x,u,v,c)` aus

```
SINIT(x,c);
...              -- kein schreibender Zugriff auf x
SSP(i,x,u,v,x)   -- not(eqex(u,v))
```

Bemerkung: Die neue $\mathsf{SSP}^{(1)}$–Instanz überschreibt die alte $\mathsf{SSP}^{(1)}$–Instanz. Falls an der Stelle `...` noch andere Anweisungen stehen, die `x` lesen, muß die $\mathsf{SINIT}^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\mathsf{SSSP}^{(1)}}$ analog aus `AADDMULTIMUL` bzw. aus einer Sequenz aus einer $\mathsf{GVOP}^{(1)}$–Instanz und einer $\mathsf{VSUM}^{(1)}$–Instanz, wobei die $\mathsf{GVOP}^{(1)}$–Instanz die Vektormultiplikation zweier Vektoren mit einem zusätzlichen skalaren Faktor beschreibt (3 Schablonen). Die $\mathsf{SSSP}^{(1)}$–Instanz wird vor der Codeerzeugung mittels eines temporären Vektors `temp` in die Bestandteile `SV(i,temp,u,...)` und `SSP(i,j,x,temp,v,...` zerlegt.

$\boxed{\mathsf{VQSUM}^{(1)}}$ analog (`eqex(u,v)`) (3 Schablonen)

$\boxed{\mathsf{ENORM}^{(1)}}$

Schablone (cross, FLOW): `ENORM(i,s,z,x,c)` aus

```
VQSUM(i,x,z,c);
...        -- kein schreibender Zugriff auf x
SQRT(s,x);
```

Bemerkung: Die ENORM$^{(1)}$–Instanz ersetzt die SQRT$^{(0)}$–Instanz; die VQSUM$^{(1)}$–Instanz muß vorerst stehenbleiben (Inferenz).

$\boxed{\mathsf{GVSUM}^{(1)}}$

Schablone 1: `GVSUM(i,x,x,fy,y1,y2,...)` aus

```
for (i=LB; i<=UB; i++)
 AADDMULTIMUL(x, x, y1‘[i]’, y2‘[i]’, ...);
  -- Indizierung der yj in i optional; zwingend fuer mind. ein yj
```

Schablone 2 (cross, FLOW): `GVSUM(i,x,c,fy,y1,y2,...)` aus

```
FN(i’,temp,y1,y2,...); -- quasiskalare Operation, eventuell GVOP
...                    -- kein Schreibzugriff auf temp
VSUM(i,x,temp,c);
```

Bemerkung: Die GVSUM$^{(1)}$–Instanz überschreibt die VSUM$^{(1)}$–Instanz. Falls an der Stelle `...` noch andere Anweisungen stehen, die `temp` lesen, muß `FN(...)` (vorerst) stehenbleiben.

Schablone 3 (cross, FLOW): `GVSUM(i,x,c,...)` aus

```
SINIT(x,c);
...        -- kein schreibender Zugriff auf x
GVSUM(i,x,x,...)
```

Bemerkung: Die neue GVSUM$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `x` lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\mathsf{GVPROD}^{(1)}}$ analog

Kernel zur 1D–Konvolution

$\boxed{\mathsf{S1CONV}^{(1)}}$ analog zu VAADDSV$^{(1)}$

Lineare Rekurrenzen erster Ordnung

$\boxed{\mathsf{PREVSUM}^{(1)}}$

Schablone 1: `PREVSUM(i,x,y,x)` aus

```
for (i=LB; i<=UB; i+=st)
   ADD(x[i],x[i-st],y[i]);  -- y[i], x[i-st] vertauschbar
```

Schablone 2 (cross, FLOW): `PREVSUM(i,x,y,c)` aus

```
SINIT(x,c);
...        -- kein schreibender Zugriff auf x
PREVSUM(i,x,y,x)
```

Bemerkung: Die neue PREVSUM$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die x lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\text{SUFVSUM}^{(1)}}$ analog

$\boxed{\text{FOLR}^{(1)}}$

Schablone 1: `FOLR(i,x,A,B,1.0,x)` aus

```
for (i=LB; i<=UB; i+=ST)
   ADDMUL(x[i],A[i],B[i],x[i-ST]);  -- B[i], x[i-ST] vertauschbar
```

Schablone 2: `FOLR(i,x,A,B,C,x)` aus

```
for (i=LB; i<=UB; i+=ST)
   MULADDMUL(x[i],C[i],A[i],B[i],x[i-ST]);
    -- B[i],x[i-ST] vertauschbar
```

Schablone 3 (cross, FLOW): `FOLR(i,x,A,B,C,c)` aus

```
SINIT(x[LB], c);
...         -- kein Schreibzugriff auf x[LB]
FOLR(i=[LB+ST:UB:ST], x,A,B,C,x)
```

Bemerkung: Die neue FOLR$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die `x[LB]` lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

1D–Maximierungen/Minimierungen

$\boxed{\text{VMAXVAL}^{(1)}}$

Schablone 1: `VMAXVAL(i,s,x,s)` aus

```
for (i=LB; i<=UB; i+=ST)
   MAXVAL(s,s,x[i]);  -- Faktoren s,x[i] vertauschbar
```

Schablone 2: `VMAXVAL(i,s,x,c)` aus

```
SINIT(s,c);
...         -- kein Schreibzugriff auf s
VMAXVAL(i,s,x,s)
```

Bemerkung: Die neue VMAXVAL$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die s lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\text{VMINVAL}^{(1)}}$ analog (2 Schablonen)

$\boxed{\text{VMAXLOC}^{(1)}}$

Schablone 1: `VMAXLOC(i,k,x,k)` aus

```
for (i=LB; i<=UB; i+=ST)
   MAXLOC(k,x,i,k);  -- i,k vertauschbar
```

Schablone 2: `VMAXLOC(i,k,x,t)` aus

```
SINIT(k,t);
...         -- kein Schreibzugriff auf k
VMAXLOC(i,k,x,k)
```

Bemerkung: Die neue VMAXVAL$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die k lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\text{VMINLOC}^{(1)}}$ analog (2 Schablonen, mit denen von VMAXLOC$^{(1)}$ gemeinsam realisiert)

$\boxed{\text{VMAXLOCM}^{(1)}}$ analog (2 Schablonen)

$\boxed{\text{VMINLOCM}^{(1)}}$ analog (2 Schablonen, mit denen von VMAXLOCM$^{(1)}$ gemeinsam realisiert)

$\boxed{\text{VMAXVL}^{(1)}}$

Schablone 1: `VMAXVL(i,k,x,s,s,k)` aus

```
for (i=LB; i<=UB; i+=ST)
   MAXVL(k,s,x,i);
```

Schablone 2: `VMAXVL(i,k,x,s,c,t)` aus

```
SINIT(s,c);
...         -- kein Schreibzugriff auf s
VMAXVL(i,k,x,s,s,t);
```

Bemerkung: Die neue VMAXVL$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die s lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

Schablone 3: `VMAXVL(i,k,x,s,c,t)` aus

```
SINIT(k,t);
...         -- kein Schreibzugriff auf k
VMAXVL(i,k,x,s,c,k)
```

Bemerkung: Die neue VMAXVL$^{(1)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die k lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\text{VMINVL}^{(1)}}$ analog (3 Schablonen, mit denen von VMAXVL$^{(1)}$ gemeinsam realisiert)

$\boxed{\text{VMAXVLM}^{(1)}}$ analog (3 Schablonen)

$\boxed{\text{VMINVLM}^{(1)}}$ analog (3 Schablonen, mit denen von VMAXVLM$^{(1)}$ gemeinsam realisiert)

1D–Relaxationen

$\boxed{\text{VJACOBI}^{(1)}}$

Schablone: `VJACOBI(i, y[i],...,starvar[i],F[i], ... )` aus

```
for (i=LB; i<=UB; i+=ST)
   (H)STAR(y[i],...,starvar[i],F[i],...,i,...);   -- y!=starvar
```

$\boxed{\text{VGAUSSSEIDEL}^{(1)}}$ analog (Bedingung: `y` = `starvar`)

B.3 Schablonen zu Mustern der Ordnung 2

Die beiden Schleifenvariablen werden entsprechend ihrer Reihenfolge im geschriebenen Feld (Slot 2) in die Slots 0 und 1 eingetragen.

$\boxed{\text{MCOPY}^{(2)}}$

Schablone: `MCOPY(i,j,A,B)` aus

```
for (j=LB; j<=UB; j+=ST)
   VCOPY(i,A[i][j],B[i][j]);
```

$\boxed{\mathsf{MINIT}^{(2)}}$

Schablone: `MINIT(i,j,A,c)` aus

```
for (j=LB; j<=UB; j+=ST)
   VINIT(i,A[i][j],c);
```

$\boxed{\mathsf{MINITSP}^{(2)}}$

Schablone 1: `MINITSP(i,j,A,i)` aus

```
for (j=LB; j<=UB; j+=ST)
   VINITSP(i,A[i][j],i);
```

Schablone 2: `MINITSP(i,j,A,j)` aus

```
for (j=LB; j<=UB; j+=ST)
   VASSIGN(i,A[i][j],j);
```

$\boxed{\mathsf{MASSIGN}^{(2)}}$

Schablone: `MASSIGN(i,j,A,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VASSIGN(i,A[i][j],v);
```

$\boxed{\mathsf{MASSIGNV}^{(2)}}$

Schablone 1: `MASSIGNV(i,j,A,y)` aus

```
for (j=LB; j<=UB; j+=ST)
   VASSIGN(i,A[i][j],y[j]);
```

Schablone 2: `MASSIGNV(i,j,A,y)` aus

```
for (j=LB; j<=UB; j+=ST)
   VCOPY(i,A[i][j],y[i]);
```

$\boxed{\mathsf{MASSIGNSP}^{(2)}}$

Schablone: `MASSIGNSP(i,j,A,fn,v1,v2,...)` aus

```
for (j=LB; j<=UB; j+=ST)
  VASSIGNSP(i,A[i][j],fn,v1,v2,...); -- notoccuridx(v1,j), ...
```

$\boxed{\mathsf{MADD}^{(2)}}$

Schablone: `MADD(i,j,A,B,C)` aus

```
for (j=LB; j<=UB; j+=ST)
   VADD(i,A[i][j],B[i][j],C[i][j]);
```

$\boxed{\mathsf{MMUL}^{(2)}}$ analog aus `VMUL`

$\boxed{\mathsf{MAADD}^{(2)}}$ analog aus `VAADD`

$\boxed{\mathsf{MAMUL}^{(2)}}$ analog aus `VAMUL`

$\boxed{\mathsf{MINC}^{(2)}}$ analog aus `VINC`

$\boxed{\mathsf{MAINC}^{(2)}}$ analog aus `VAINC`

$\boxed{\mathsf{SM}^{(2)}}$

Schablone: `SM(i,j,A,B,s)` aus

```
for (j=LB; j<=UB; j+=ST)
   SV(i,A[i][j],B[i][j],s);  -- notoccuridx(s,j)
```

$\boxed{\mathsf{MAADDSS}^{(2)}}$

Schablone: `MAADDSS(i,j,A,A,u,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VAADDSS(i,A[i][j],init[i][j],u,v);  -- notoccuridx(u,j), (v,j)
```

$\boxed{\mathsf{MAADDSM}^{(2)}}$

Schablone 1: `MAADDSM(i,j,A,A,U,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VAADDSV(i,A[i][j],v,U[i][j], A[i][j]);  -- notoccurin(v,j)
```

Schablone 2: `MAADDSM(i,j,A,A,U,v)` aus

```
if (v!=0)                -- oder Bed. if (v>EPS), EPS vorgegeben
   MAADDSM(i,j,A[i][j],A[i][j],U[i][j],v);
```

Schablone 3 (cross, FLOW): `MAADDSM(i,j,A,A,U,v)` aus

```
SM(i',j',temp,U,v);
...                    -- kein Schreibzugriff auf temp, U, v
MAADD(i,j,A,A,temp);
```

Bemerkung: Die neue $\mathsf{MAADDSM}^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `temp` lesen, muß die $\mathsf{SM}^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\mathsf{MINV}^{(2)}}$

Schablone: `MINV(i,j,A,B)` aus

```
for (j=LB; j<=UB; j+=ST)
   SV(i,A[i][j],B[i][j],s);  -- notoccuridx(s,j)
```

wird als Spezialfall vor $\mathsf{MCOPY}^{(2)}$ getestet.

$\boxed{\mathsf{GMOP}^{(2)}}$ aus $\mathsf{GVOP}^{(1)}$ oder sonstigen Vektoroperationen, die kein Äquivalent der Ordnung 2 besitzen (noch nicht implementiert).

$\boxed{\mathsf{MCONDASS}^{(2)}}$

Schablone: `MCONDASS(i,j,y,cond,M1,M2,s11,s12,...,s21,s22,...)` aus

```
for (j=LB; j<=UB; j+=ST)
  VCONDASS(i,y[i][j],cond,M1,M2,s11[i][j],s12[i][j],...);
  -- issimpleidx(y,j), issimpleidx(cond,j), issimpleidx(s11,j), ...
```

Shiften von Matrizen

$\boxed{\mathsf{VVSHIFT}^{(2)}}$

Schablone 1: `VVSHIFT(i,j,A,A,k)` aus

```
for (j=LB; j<=UB; j+=ST)
   VSHIFT(i,A[i][j],A[i+-k][j],k);
```

Schablone 2: `VVSHIFT(i,j,A,A,k)` aus

```
for (j=LB; j<=UB; j+=ST)
   VCOPY(i,A[i][j],A[i][j+-k]);
```

$\boxed{\mathsf{MSHIFT}^{(2)}}$

Schablone: `VVSHIFT(i,j,A,A,k1,k2)` aus

```
for (j=LB; j<=UB; j+=ST)
   VSHIFT(i,A[i][j],A[i+-k1][j+-k2],k1);
    -- oder Indizierung j+-k2 auf lhs, j auf rhs
```

Matrix–Vektor–Multiplikation und verwandte Muster

$\boxed{\mathsf{MAADDVV}^{(2)}}$

Schablone 1: `MAADDVV(i,j,A,u,v,A)` aus

```
for (j=LB; j<=UB; j+=ST)
   VAADDSV(i, A[i][j], v[j], u[i], A[i][j]); -- notoccuridx(u,j)
```

Schablone 2 (cross, FLOW): `MAADDVV(i,j,A,u,v,c)` aus

```
MINIT(i',j',A, c);  -- auch MASSIGN oder MINITSP moeglich
...                 -- kein Schreibzugriff auf A[][]
MAADDVV(i,j,A[i][j],u[i],v[j],A[i][j]);
```

Bemerkung: Die neue $\mathsf{MAADDVV}^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `A` lesen, muß die $\mathsf{MINIT}^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\mathsf{MAADDSVV}^{(2)}}$

Schablone 1: `MAADDSVV(i,j,A,A,s,u,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VAADDMULTISV(i,A[i][j],A[i][j],u[i],v[j],s);
   -- notoccuridx(u,j), notoccuridx(s,j)
```

Schablone 2 (cross, FLOW): `MAADDVV(i,j,A,u,v,c)` aus

```
MINIT(i',j',A, c); -- auch MASSIGN, MASSIGNV, MCOPY oder MINITSP
...                -- kein Schreibzugriff auf A[][]
MAADDSVV(i,j,A[i][j],u[i],v[j],A[i][j]);
```

Bemerkung: Die neue $\mathsf{MAADDSVV}^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `A` lesen, muß die $\mathsf{MINIT}^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\mathsf{MLUD}^{(2)}}$

Schablone 1: `MLUD(i,j,A[i][j],A[i][j],A[k][k],A[i][k],A[k][j], k)` aus

```
for (i=k+1; i<=n; i++)
  VLUD( j=[k+1:n], A[i][j], Q[i][k], A[i][j],
              A[i][k], A[k][k], Q[i][k], A[k][j], k, i);
```

Schablone 2 (cross, FLOW):

`MLUD(i,j,A[i][j],A[i][j],1/A[k][k],A[i][k],A[k][j],k)` aus

```
SCOPY(A[k][k], 1/A[k][k]); -- INV bei A[k][k] auf rhs gesetzt
...                        -- kein Schreib-/Lesezugriff auf A[k][k]
MLUD(i,j,A[i][j],A[i][j],A[k][k],A[i][k],A[k][j], k);
```

$\boxed{\mathsf{MV}^{(2)}}$

Schablone 1: `MV(i,j,a,C,b,init)` aus

```
for (j=LB; j<=UB; j+=ST)
  VAADDSV(i, a[i],b[j],C[i][j],init); -- notoccuridx(A,j), (init,j)
```

Schablone 2: `MV(i,j,a,C,b,init)` aus

```
for (i=LB; i<=UB; i+=ST)
  SSP(j,a[i],C[i][j],b[j],init); -- notoccuridx(b,i);
                                    Faktoren vertauschbar
```

Schablone 3 (cross, FLOW): `MV(i,j,a,C,b,init)` aus

```
VINIT(i',a, c);  -- auch VASSIGN, VCOPY oder VINITSP moeglich
...              -- kein Schreibzugriff auf A[]
MV(i,j,a[i],C[i][j],b[j],a[i]);
```

Bemerkung: Die neue $\mathsf{MV}^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `A` lesen, muß die $\mathsf{VINIT}^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\mathsf{SMV}^{(2)}}$ analog aus `VAADDMULTISV` bzw. `SSSP` (3 Schablonen).

Eine $\mathsf{SMV}^{(2)}$–Instanz wird vor der Codeerzeugungsphase mittels eines temporären Vektors `temp` in die Bestandteile `SV(i,temp,...)` und `MV(i,j,a,temp,...` zerlegt.

$\boxed{\mathsf{FSUBST}^{(2)}}$ und $\boxed{\mathsf{BSUBST}^{(2)}}$ analog zu $\mathsf{MV}^{(2)}$ mit `eqsym(a,b)` (2 Schablonen)

2D–Reduktionen

$\boxed{\mathsf{VVSUM}^{(2)}}$

Schablone 1: `VVSUM(i,j,v,A,c)` aus

```
for (i=LB; i<=UB; i+=ST)
   VSUM(j, v[i], A[i][j], c );  -- c optional indiziert: c[i]
```

Schablone 2: `VVSUM(i,j,v,A,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VAADD(i, v[i], v[i], A[i][j] );
```

Schablone 3 (cross, FLOW): `VVSUM(i,j,v,A,c)` aus

```
VINIT(i',j',A, c);  -- auch VASSIGN, VCOPY oder VINITSP moeglich
...                 -- kein Schreibzugriff auf A[][]
VVSUM(i,j,v[i],A[i][j],v[j]);
```

Bemerkung: Die neue VVSUM$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `A` lesen, muß die VINIT$^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\text{VVPROD}^{(2)}}$ analog aus VPROD$^{(1)}$ bzw. VAMUL$^{(1)}$ (3 Schablonen)

$\boxed{\text{MSUM}^{(2)}}$

Schablone 1: `MSUM(i,j,s,A,c)` aus

```
for (i=LB; i<=UB; i+=ST)
   VSUM( j, s, A[i][j], c);
```

Schablone 2 (cross, FLOW): `MSUM(i,j,s,A,c)` aus

```
MINIT(i',j',A, c);  -- auch MASSIGN, MASSIGNV, MCOPY oder MINITSP
...                 -- kein Schreibzugriff auf A[][]
MSUM(i,j,A[i][j],u[i],v[j],A[i][j]);
```

Bemerkung: Die neue MSUM$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `A` lesen, muß die MINIT$^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\text{MPROD}^{(2)}}$ analog aus `VPROD` (2 Schablonen)

$\boxed{\text{MROR}^{(2)}}$ analog aus `VROR` (2 Schablonen)

$\boxed{\text{MRAND}^{(2)}}$ analog aus `VRAND` (2 Schablonen)

Konvolutionen (Ordnung 2)

$\boxed{\text{S2CONV}^{(2)}}$

Schablone 1: `S2CONV(l,k,y,init,u,v,c,d)` aus

```
for (k=LB; k<=UB; k++)
   S1CONV(l,y,init,u[l][k],v[c-l][d-k],c);
```

Schablone 2 (cross, FLOW): `S2CONV(l,k,y,init,u,v,c,d)` aus

```
SINIT(y,init);  -- auch SCOPY moeglich
...             -- kein Schreibzugriff auf y
S2CONV(l,k,y,y,u[l][k],v[c-l][d-k],c,d);
```

Bemerkung: Die neue S2CONV$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `y` lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

$\boxed{\text{VCONV}^{(2)}}$

Schablone 1: `VCONV(j,l,y,init,u,v)` aus

```
for (j=LB; j<=UB; j++)
   S1CONV(l,y[j],init,u[l],v[j-l],j);  -- optional init[j]
```

Schablone 2: `VCONV(j,l,y,init,u,v)` aus

```
for (l=LB; l<=UB; l++)
   VAADDSV(j,y[j],init,v[j-l],u[l]);   -- optional init[j]
```

Schablone 3 (cross, FLOW): `VCONV(j,l,y,init,u,v)` aus

```
VINIT(j',y,init);   -- auch VCOPY, VASSIGN moeglich
...                 -- kein Schreibzugriff auf y
VCONV(j,l,y[j],y[j],u[l],v[j-l]);
```

Bemerkung: Die neue VCONV$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die y lesen, muß die VINIT$^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

V1CONV$^{(2)}$

Schablone 1: `V1CONV(i,l,y,init,u,v,c)` aus

```
for (i=LB; i<=UB; i++)
  S1CONV(l,y[i],init,u[l][i],v[c-l][i(-..)],c); -- optional init[i]
```

Schablone 2: `V1CONV(i,l,y,init,u,v,c)` aus

```
for (l=LB; l<=UB; l++)
  VADDMUL(i,y[i],init[i],v[c-l][i(-..)],u[l][i]);
```

Schablone 3 (cross, FLOW): `V1CONV(i,l,y,init,u,v,c)` aus

```
VINIT(i',y,init);   -- auch VCOPY, VASSIGN moeglich
...                 -- kein Schreibzugriff auf y
V1CONV(i,l,y[i],y[i],u[l][i],v[c-l][i],c);
```

Bemerkung: Die neue V1CONV$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die y lesen, muß die VINIT$^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

2D–Maximierungen/Minimierungen

MMAXVAL$^{(2)}$

Schablone 1: `MMAXVAL(i,j,s,A,c)` aus

```
for (i=LB; i<=UB; i+=ST)
   VMAXVAL( j, s, A[i][j], c);
```

Schablone 2 (cross, FLOW): `MMAXVAL(i,j,s,A,c)` aus

```
SINIT(s,c);  -- auch SCOPY moeglich
...          -- kein Schreibzugriff auf s, c;
MMAXVAL(i,j,s,A[i][j],s);
```

Bemerkung: Die neue MMAXVAL$^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die s lesen, muß die SINIT$^{(0)}$–Instanz (vorerst) stehenbleiben.

MMINVAL$^{(2)}$ analog aus `VMINVAL` (2 Schablonen, mit denen zu MMAXVAL$^{(2)}$ gemeinsam realisiert)

MMAXLOC$^{(2)}$ analog aus `VMAXLOCM` (2 Schablonen)

$\boxed{\mathsf{MMINLOC}^{(2)}}$ analog aus `VMINLOCM` (2 Schablonen, mit denen zu $\mathsf{MMAXLOC}^{(2)}$ gemeinsam realisiert)

$\boxed{\mathsf{MMAXVL}^{(2)}}$ analog aus `VMAXVLM` (2 Schablonen)

$\boxed{\mathsf{MMINVL}^{(2)}}$ analog aus `VMINVLM` (2 Schablonen, mit denen zu $\mathsf{MMAXVL}^{(2)}$ gemeinsam realisiert)

$\boxed{\mathsf{VVMAXVAL}^{(2)}}$

Schablone 1: `VVMAXVAL(i,j,v,A,c)` aus

```
for (i=LB; i<=UB; i+=ST)
   VMAXVAL(j, v[i], A[i][j], c );  -- c optional indiziert: c[i]
```

Schablone 2: `VVMAXVAL(i,j,v,A,v)` aus

```
for (j=LB; j<=UB; j+=ST)
   VMAX(i,v[i],v[i],A[i][j]);  -- elementweise Maximierung
```

Schablone 3 (cross, FLOW): `VVMAXVAL(i,j,v,A,c)` aus

```
VINIT(i',v,c);     -- auch VCOPY oder VASSIGN moeglich
...                -- kein Schreibzugriff auf v,c
VVMAXVAL(i,j,v[i],A[i][j],v[i]);
```

Bemerkung: Die neue $\mathsf{VVMAXVAL}^{(2)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `v` lesen, muß die $\mathsf{VINIT}^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\mathsf{VVMINVAL}^{(2)}}$ analog aus `VMINVAL` bzw. `VMIN` (3 Schablonen, mit denen zu $\mathsf{VVMAXVAL}^{(2)}$ gemeinsam realisiert)

$\boxed{\mathsf{VVMAXLOC}^{(2)}}$ analog aus `VMAXLOC` (2 Schablonen)

$\boxed{\mathsf{VVMINLOC}^{(2)}}$ analog aus `VMINLOC` (2 Schablonen, mit denen zu $\mathsf{VVMAXLOC}^{(2)}$ gemeinsam realisiert)

$\boxed{\mathsf{VVMAXVL}^{(2)}}$ analog aus `VMAXVL` (3 Schablonen)

$\boxed{\mathsf{VVMINVL}^{(2)}}$ analog aus `VMINVL` (3 Schablonen, mit denen zu $\mathsf{VVMAXVL}^{(2)}$ gemeinsam realisiert)

2D–Relaxationen

$\boxed{\mathsf{MJACOBI}^{(2)}}$

Schablone: `MJACOBI(i,j,y,...,starvar,F,...)` aus

```
for (i=LB; i<=UB; i+=ST)
   VJACOBI(j, y[i][j], ..., starvar[i][j], F[i][j], ... );
```

$\boxed{\mathsf{MGAUSSSEIDEL}^{(2)}}$ analog aus `VGAUSSSEIDEL`

$\boxed{\mathsf{VVGAUSSSEIDEL}^{(2)}}$

Schablone 1: `VVJACOBI(i,j,y,...,starvar,F,...)` aus

```
for (i=LB; i<=UB; i+=ST)
   VJACOBI(j, y[i][j],..., starvar[i][j], F[i][j],...,j,...);
```

`j` spannt eine Dimension eines Differenzensterns auf, nicht aber `i`.

Schablone 2: `VVJACOBI(i,j,y,...,starvar,F,...)` aus

```
for (j=LB; j<=UB; j+=ST)
  GVOP(i, y[i][j],
       (H)STAR(i,y[i][j],...,starvar[i][j],F[i][j],...,j,...),
       ... );
```

`j` spannt eine Dimension eines Differenzensternes auf, nicht aber `i`.

$\boxed{\text{VVGAUSSSEIDEL}^{(2)}}$ analog aus `VGAUSSSEIDEL` bzw. `GVOP`

B.4 Schablonen zu Mustern der Ordnung 3

Matrix–Matrix–Multiplikation und verwandte Muster

$\boxed{\text{MM}^{(3)}}$

Schablone 1: `MM(i,j,k, C,A,B, c` aus

```
for (i=LB; i<=UB; i+=ST)
   MV(j,k, C[i][j],B[k][j],A[i][k],c); -- optional c[i], c[i][j]
```

Schablone 2: `MM(i,j,k, C,A,B, c` aus

```
for (k=LB; k<=UB; k+=ST)
   MAADDVV(i,j, C[i][j], B[k][j], A[i][k], C[i][j]);
```

Schablone 3 (cross, FLOW): `MM(i,j,k, C,A,B, c` aus

```
MINIT(C,c);  -- auch MCOPY oder MASSIGN moeglich
...          -- kein Schreibzugriff auf C,c;
MM(i,j,k, C[i][j],A[i][k],B[k][j], C[i][j]);
```

Bemerkung: Die neue $\text{MM}^{(3)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `C` lesen, muß die $\text{MINIT}^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\text{SMM}^{(3)}}$ analog aus `SMV` bzw. `MAADDSVV` (3 Schablonen).

Eine $\text{SMM}^{(3)}$–Instanz wird vor der Codeerzeugungsphase mittels eines temporären Vektors `temp` in die Bestandteile `SM(i,temp,...)` und `MM(i,j,a,temp,...` zerlegt.

Konvolutionen (Ordnung 3)

$\boxed{\text{V2CONV}^{(3)}}$

Schablone 1: `V2CONV(i,k,l,y,init,u,v,d)` aus

```
for (i=LB; i<=UB; i++)
 S2CONV(l,k,y[i],init,u[k][l],v[i-k][d-l],i,d); -- optional init[i]
```

Schablone 2: `V2CONV(i,k,l,y,init,u,v,d)` aus

```
for (k=LB; k<=UB; k++)
 V1CONV(i,l,y[i],init,u[k][l],v[i-k][d-l],d); -- optional init[i]
```

Schablone 3: `V2CONV(i,k,l,y,init,u,v,d)` aus

```
for (l=LB; l<=UB; l++)
 VCONV(i,k,y[i],init,u[k][l],v[i-k][d-l]);   -- optional init[i]
```

Schablone 4 (cross, FLOW): `V2CONV(i,k,l,y,init,u,v,d)` aus

```
VINIT(i',y,init);   -- auch VCOPY, VASSIGN moeglich
...                 -- kein Schreibzugriff auf y[]
V2CONV(i,k,l,y[i],y[i],u[k][l],v[i-k][d-l]);
```

Bemerkung: Die neue V2CONV$^{(3)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die y lesen, muß die VINIT$^{(1)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

$\boxed{\text{M1CONV}^{(3)}}$

Schablone 1: `M1CONV(i,j,l,y,y,u,v)` aus

```
for (j=LB; j<=UB; j++)
   V1CONV(i,l,y[i][j],y[i][j],u[l],v[j-l][i`-..'],j);
```

Schablone 2: `M1CONV(i,j,l,y,init,u,v)` aus

```
for (i=LB; i<=UB; i++)
   VCONV(j,l,y[i][j],init,u[l],v[i`-..'][j-l]);
    -- optional init[i][j]
```

Schablone 3: `M1CONV(i,j,l,y,y,u,v)` aus

```
for (l=LB; l<=UB; l++)
   MAADDSM(i,j,y[i][j],y[i][j],v[i`-..'][j-l],u[l]);
```

Schablone 4 (cross, FLOW): `M1CONV(i,j,l,y,init,u,v)` aus

```
MINIT(i',j',y,init);   -- auch MCOPY, MASSIGN moeglich
...                    -- kein Schreibzugriff auf y[][]
M1CONV(i,j,l,y[i][j],y[i][j],u[l],v[i`-..'][j-l]);
```

Bemerkung: Die neue M1CONV$^{(3)}$–Instanz überschreibt die alte. Falls an der Stelle ... noch andere Anweisungen stehen, die y lesen, muß die MINIT$^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

LR–Zerlegung

$\boxed{\text{LUD}^{(3)}}$

`LUD(k,i,j,A[i][j],Q[i][k],A[i][j],diagval,colsign,A[i][k],A[k][j]` aus

```
for (k=LB; k<=UB; k++)
   MLUD(i=[k`+1':n], j=[k:n], A[i][j],Q[i][k], A[i][j],A[i][k],
                A[k][k], Q[i][k], A[k][j], diagval, colsign, k );
```

Relaxationsalgorithmen

$\boxed{\mathsf{JACOBI}^{(3)}}$

Schablone: `JACOBI(k,i,j,y[i][j],...` aus

```
for (k=LB; k<=UB; k++)
  MJACOBI(i,j,y[i][j],....);
```

$\boxed{\mathsf{GAUSSSEIDEL}^{(3)}}$ analog aus `MGAUSSSEIDEL`

B.5 Schablonen zu Mustern der Ordnung 4

$\boxed{\mathsf{MCONV}^{(4)}}$

Schablone 1: `MCONV(i,j,k,l,y,init,u,v)` aus

```
for (j=LB; j<=UB; j++)
   V2CONV(i,k,l,y[i][j],init,u[k][l],v[i-k][j-l],j);
    -- optional init[i][j]
```

Schablone 2: `M1CONV(i,j,k,l,y,init,u,v)` aus

```
for (k=LB; k<=UB; k++)
   M1CONV(i,j,l,y[i][j],init,u[k][l],v[i-k][j-l]);
    -- optional init[i][j]
```

Schablone 3 (cross, FLOW): `M1CONV(i,j,l,y,init,u,v)` aus

```
MINIT(i',j',y,init);   -- auch MCOPY, MASSIGN moeglich
...                    -- kein Schreibzugriff auf y[][], init
MCONV(i,j,k,l,y[i][j],y[i][j],u[k][l],v[i-k][j-l]);
```

Bemerkung: Die neue $\mathsf{MCONV}^{(4)}$–Instanz überschreibt die alte. Falls an der Stelle `...` noch andere Anweisungen stehen, die `y` lesen, muß die $\mathsf{MINIT}^{(2)}$–Instanz (vorerst) stehenbleiben. Zum Testen auf Schreib– oder Lesezugriff benutzen wir *disjdescr*().

B.6 Musterhierarchiegraph

Für die eben aufgelisteten realisierten Schablonen geben wir nun noch den Musterhierarchiegraphen an. Wir stellen die Kanten im Musterhierarchiegraphen nach Triggermustern geordnet dar. Die Obermuster (und damit die den Kanten entsprechenden Schablonen) sind innerhalb eines Triggermusters so angeordnet, daß Schablonen für speziellere Muster immer vor denen der allgemeineren getestet werden.

B.6.1 Vertikale Musterhierarchie

Für alle Muster der Ordnung 1 oder größer wird jeweils auch die Möglichkeit zur Entblockung einer Schleife erwogen. Alle skalaren Muster besitzen außerdem als Obermuster VASSIGNSP$^{(1)}$, CONDASS$^{(0)}$ und STRANGE$^{(0)}$ (auf STRANGE$^{(0)}$ wird immer zuletzt getestet); alle quasiskalaren Muster der Ordnung 1 haben VCONDASS$^{(1)}$, alle quasiskalaren Muster der Ordnung 2 MCONDASS$^{(2)}$ als Obermuster.

Triggermuster	mögliche Obermuster
SINIT$^{(0)}$	VINIT$^{(1)}$
SCOPY$^{(0)}$	VINV$^{(1)}$ VSHIFT$^{(1)}$ VCOPY$^{(1)}$ VASSIGN$^{(1)}$ MAXVAL$^{(0)}$ MAXLOC$^{(0)}$, MINLOC$^{(0)}$ MAXLOCM$^{(0)}$, MINLOCM$^{(0)}$ MAXVL$^{(0)}$, MAXVLM$^{(0)}$
ADD	ADD AADD FSUM$^{(1)}$ VADD$^{(1)}$ PREVSUM$^{(1)}$ VINC$^{(1)}$ MULTIADD$^{(0)}$ *stargazer*
AADD	VAADD$^{(1)}$ VAINC$^{(1)}$ VSUM$^{(1)}$ *stargazer*
MUL	MUL AMUL ADDMUL$^{(0)}$ MULMUL$^{(0)}$ SV$^{(1)}$ VMUL$^{(1)}$ MULTIMUL$^{(0)}$
AMUL	VAMUL$^{(1)}$ VPROD$^{(1)}$ POWER
MOD$^{(0)}$	VMOD$^{(1)}$
AMOD$^{(0)}$	VAMOD$^{(1)}$
MULTIADD$^{(0)}$	MULTIADD$^{(0)}$ *stargazer* FSUM$^{(1)}$ GVSUM$^{(1)}$ GVOP$^{(1)}$
MULTIMUL$^{(0)}$	ADDMULTIMUL$^{(0)}$ MULTIMUL$^{(0)}$ MULTIADD$^{(0)}$ GVPROD$^{(1)}$ GVOP$^{(1)}$
SIN$^{(0)}$, COS$^{(0)}$, TAN$^{(0)}$, EXP$^{(0)}$, LOG$^{(0)}$, MAXVAL$^{(0)}$, MINVAL$^{(0)}$	VSIN$^{(1)}$, VCOS$^{(1)}$, VTAN$^{(1)}$, VEXP$^{(1)}$, VLOG$^{(1)}$, VMAX$^{(1)}$ VMIN$^{(1)}$
AND$^{(0)}$	VRAND$^{(1)}$ VAND$^{(1)}$
OR$^{(0)}$	VROR$^{(1)}$ VOR$^{(1)}$
POWER	POWER
ADDMUL$^{(0)}$	ADDMUL$^{(0)}$ AADDMUL$^{(0)}$ MULTIADD$^{(0)}$ MULADDMUL$^{(0)}$ *stargazer* VADDMUL$^{(1)}$ VADDSV$^{(1)}$ FOLR$^{(1)}$
AADDMUL$^{(0)}$	SSP$^{(1)}$ VQSUM$^{(1)}$ SVSUM$^{(1)}$ VAADDSS$^{(1)}$ S1CONV$^{(1)}$ VAADDSV$^{(1)}$ *stargazer*
ADDMULTIMUL$^{(0)}$	ADDMULTIMUL$^{(0)}$ AADDMULTIMUL$^{(0)}$ GVOP$^{(1)}$
AADDMULTIMUL$^{(0)}$	VAADDMULTISV$^{(1)}$ SSSP$^{(1)}$ GVSUM$^{(1)}$ GVOP$^{(1)}$
MULADDMUL$^{(0)}$	MULADDMUL$^{(0)}$ FOLR$^{(1)}$ GVOP$^{(1)}$
MULMUL$^{(0)}$	MULMUL$^{(0)}$ MULTIADD$^{(0)}$ *stargazer*

	VMULMUL$^{(1)}$
ROT	VROT$^{(1)}$
SWAP$^{(0)}$	SWAP$^{(0)}$
	VSWAP$^{(1)}$
HSTAR$^{(0)}$	VJACOBI$^{(1)}$
	VGAUSSSEIDEL$^{(1)}$
	GVOP$^{(1)}$
	GVSUM$^{(1)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
STAR$^{(0)}$	VJACOBI$^{(1)}$
	VGAUSSSEIDEL$^{(1)}$
	GVOP$^{(1)}$
	GVSUM$^{(1)}$
	STAR$^{(0)}$
MAXVAL$^{(0)}$	MAXVAL$^{(0)}$
	VMAXVAL$^{(1)}$
	VMAX$^{(1)}$
MINVAL$^{(0)}$	MINVAL$^{(0)}$
	VMINVAL$^{(1)}$
	VMIN$^{(1)}$
MAXLOC$^{(0)}$	VMAXLOC$^{(1)}$
MINLOC$^{(0)}$	VMINLOC$^{(1)}$
MAXLOCM$^{(0)}$	VMAXLOCM$^{(1)}$
MINLOCM$^{(0)}$	VMINLOCM$^{(1)}$
MAXVL$^{(0)}$	VMAXVL$^{(1)}$
MINVL$^{(0)}$	VMINVL$^{(1)}$
MAXVLM$^{(0)}$	VMAXVLM$^{(1)}$
MINVLM$^{(0)}$	VMINVLM$^{(1)}$
CONDASS$^{(0)}$	VCONDASS$^{(1)}$
	VINV$^{(1)}$
STRANGE$^{(0)}$	GVOP$^{(1)}$
	GVSUM$^{(1)}$
	STRANGE$^{(0)}$
VCOPY$^{(1)}$	VVSHIFT$^{(2)}$
	MSHIFT$^{(2)}$
	MCOPY$^{(2)}$
	MASSIGNV$^{(2)}$
VINIT$^{(1)}$	MINIT$^{(2)}$
VINITSP$^{(1)}$	MINITSP$^{(2)}$
VASSIGN$^{(1)}$	MINITSP$^{(2)}$
	MASSIGN$^{(2)}$
	MASSIGNV$^{(2)}$
VASSIGNSP$^{(1)}$	MASSIGNSP$^{(2)}$
VADD$^{(1)}$	MADD$^{(2)}$
VAADD$^{(1)}$	VVSUM$^{(2)}$
	MAADD$^{(2)}$
VMUL$^{(1)}$	MMUL$^{(2)}$
VAMUL$^{(1)}$	VVPROD$^{(2)}$
	MAMUL$^{(2)}$
VINC$^{(1)}$	MINC$^{(2)}$

VAINC$^{(1)}$	MAINC$^{(2)}$
VINV$^{(1)}$	MINV$^{(2)}$
SV$^{(1)}$	SM$^{(2)}$
VAADDSS$^{(1)}$	MAADDVV$^{(2)}$
	MAADDSS$^{(2)}$
	VCONV$^{(2)}$
	V1CONV$^{(2)}$
VAADDSV$^{(1)}$	MAADDVV$^{(2)}$
	MAADDSM$^{(2)}$
	MV$^{(2)}$
	VCONV$^{(2)}$
VADDSV$^{(1)}$	VADDSV$^{(1)}$
VADDMUL$^{(1)}$	V1CONV$^{(2)}$
VAADDMULTISV$^{(1)}$	MAADDSVV$^{(2)}$
	SMV$^{(2)}$
VMAX$^{(1)}$	VVMAXVAL$^{(2)}$
VMIN$^{(1)}$	VVMINVAL$^{(2)}$
VLUD$^{(1)}$	MLUD$^{(2)}$
GVOP$^{(1)}$	GMOP$^{(2)}$
	VVGAUSSSEIDEL$^{(2)}$
	VVGAUSSSEIDEL$^{(2)}$
VCONDASS$^{(1)}$	MCONDASS$^{(2)}$
VSHIFT$^{(1)}$	VVSHIFT$^{(2)}$
	MSHIFT$^{(2)}$
VSUM$^{(1)}$	VVSUM$^{(2)}$
	MSUM$^{(2)}$
SSP$^{(1)}$	MV$^{(2)}$
SSSP$^{(1)}$	SMV$^{(2)}$
VMAXVAL$^{(1)}$	MMAXVAL$^{(2)}$
	VVMAXVAL$^{(2)}$
VMINVAL$^{(1)}$	MMINVAL$^{(2)}$
	VVMINVAL$^{(2)}$
VMAXLOCM$^{(1)}$	MMAXLOC$^{(2)}$
VMAXLOC$^{(1)}$	VVMAXLOC$^{(2)}$
VMINLOCM$^{(1)}$	MMINLOC$^{(2)}$
VMINLOC$^{(1)}$	VVMINLOC$^{(2)}$
VMAXVL$^{(1)}$	VVMAXVL$^{(2)}$
VMAXVLM$^{(1)}$	MMAXVL$^{(2)}$
VMINVL$^{(1)}$	VVMINVL$^{(2)}$
VMINVLM$^{(1)}$	MMINVL$^{(2)}$
VJACOBI$^{(1)}$	MJACOBI$^{(2)}$
	VVGAUSSSEIDEL$^{(2)}$
VGAUSSSEIDEL$^{(1)}$	MGAUSSSEIDEL$^{(2)}$
	VVGAUSSSEIDEL$^{(2)}$
MAADDSM$^{(2)}$	MAADDSM$^{(2)}$
	M1CONV$^{(3)}$
MAADDVV$^{(2)}$	MM$^{(3)}$
MAADDSVV$^{(2)}$	SMM$^{(3)}$
	LUD$^{(3)}$

MV$^{(2)}$	MM$^{(3)}$
MLUD$^{(2)}$	LUD$^{(3)}$
S2CONV$^{(2)}$	V2CONV$^{(3)}$
VCONV$^{(2)}$	V2CONV$^{(3)}$
	M1CONV$^{(3)}$
V1CONV$^{(2)}$	V2CONV$^{(3)}$
	M1CONV$^{(3)}$
V2CONV$^{(3)}$	MCONV$^{(4)}$
M1CONV$^{(3)}$	MCONV$^{(4)}$

B.6.2 Horizontale Musterhierarchie

Triggermuster bei der Mustererkennung entlang einer Querkante ist immer das Ziel der Querkante (d.h. die textuell spätere Anweisung). Eine Ausnahme bilden hierbei die speziellen, gemeinsamen Schablonen zum Aufrollen von Schleifen.

SCOPY$^{(0)}$ kann als Triggermuster zu allen Operationen vorkommen, die einen Skalar schreiben (Umbenennung von Variablen, vgl. Abschnitt 6.6.2). Analog kann VCOPY$^{(1)}$ Triggermuster zu allen einen Vektor schreibenden Operationen sein, MCOPY$^{(2)}$ entsprechend zu Operationen, die eine Matrix schreiben.

Triggermuster	mögliche Obermuster
SCOPY$^{(0)}$	SWAP$^{(0)}$
	ROT
ADD	ADDMUL$^{(0)}$
	MULTIADD$^{(0)}$
	ADDMULTIMUL$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
AADD	AADDMUL$^{(0)}$
	MULTIADD$^{(0)}$
	AADDMULTIMUL$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
ADDMUL$^{(0)}$	MULMUL$^{(0)}$
	ADDMULTIMUL$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
AADDMUL$^{(0)}$	AADDMULTIMUL$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
MUL	MULADDMUL$^{(0)}$
	MULTIADD$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
MULTIADD$^{(0)}$	MULTIADD$^{(0)}$
	HSTAR$^{(0)}$/STAR$^{(0)}$
MULTIMUL$^{(0)}$	MULTIMUL$^{(0)}$
FSUM$^{(1)}$	FSUM$^{(1)}$
VCOPY$^{(1)}$	VSWAP$^{(1)}$
	VROT$^{(1)}$
VADD$^{(1)}$	VADDSV$^{(1)}$
	VADDMUL$^{(1)}$
VAADD$^{(1)}$	VAADDSV$^{(1)}$
VAADDSS$^{(1)}$	VAADDSS$^{(1)}$
VAADDSV$^{(1)}$	VLUD$^{(1)}$
	VAADDMULTISV$^{(1)}$
VSUM$^{(1)}$	VSUM$^{(1)}$
	SSP$^{(1)}$
	VQSUM$^{(1)}$
	GVSUM$^{(1)}$
SVSUM$^{(1)}$	SVSUM$^{(1)}$
VPROD$^{(1)}$	VPROD$^{(1)}$
VROR$^{(1)}$	VROR$^{(1)}$
VRAND$^{(1)}$	VRAND$^{(1)}$
VQSUM$^{(1)}$	VQSUM$^{(1)}$
SSP$^{(1)}$	SSP$^{(1)}$
SSSP$^{(1)}$	SSSP$^{(1)}$
GVSUM$^{(1)}$	GVSUM$^{(1)}$
GVPROD$^{(1)}$	GVPROD$^{(1)}$
S1CONV$^{(1)}$	VCONV$^{(2)}$
	V1CONV$^{(2)}$
	S2CONV$^{(2)}$
PREVSUM$^{(1)}$	PREVSUM$^{(1)}$
FOLR$^{(1)}$	FOLR$^{(1)}$
VMAXVAL$^{(1)}$	VMAXVAL$^{(1)}$
VMINVAL$^{(1)}$	VMINVAL$^{(1)}$
VMAXLOC$^{(1)}$	VMAXLOC$^{(1)}$
VMINLOC$^{(1)}$	VMINLOC$^{(1)}$
VMAXLOCM$^{(1)}$	VMAXLOCM$^{(1)}$
VMINLOCM$^{(1)}$	VMINLOCM$^{(1)}$
VMAXVL$^{(1)}$	VMAXVL$^{(1)}$
VMINVL$^{(1)}$	VMINVL$^{(1)}$
VMAXVLM$^{(1)}$	VMAXVLM$^{(1)}$
VMINVLM$^{(1)}$	VMINVLM$^{(1)}$
MAADD$^{(2)}$	MAADDVV$^{(2)}$
	MAADDSM$^{(2)}$
MAADDVV$^{(2)}$	MAADDVV$^{(2)}$
	MAADDSVV$^{(2)}$
MAADDSVV$^{(2)}$	MAADDSVV$^{(2)}$
MLUD$^{(2)}$	MLUD$^{(2)}$
MV$^{(2)}$	MV$^{(2)}$
VVSUM$^{(2)}$	VVSUM$^{(2)}$
MSUM$^{(2)}$	MSUM$^{(2)}$
VVPROD$^{(2)}$	VVPROD$^{(2)}$
MPROD$^{(2)}$	MPROD$^{(2)}$
MROR$^{(2)}$	MROR$^{(2)}$
MRAND$^{(2)}$	MRAND$^{(2)}$
S2CONV$^{(2)}$	S2CONV$^{(2)}$
VCONV$^{(2)}$	VCONV$^{(2)}$
V1CONV$^{(2)}$	V1CONV$^{(2)}$
MMAXVAL$^{(2)}$	MMAXVAL$^{(2)}$

MMINVAL$^{(2)}$	MMINVAL$^{(2)}$
MMAXLOC$^{(2)}$	MMAXLOC$^{(2)}$
MMINLOC$^{(2)}$	MMINLOC$^{(2)}$
MMAXVL$^{(2)}$	MMAXVL$^{(2)}$
MMINVL$^{(2)}$	MMINVL$^{(2)}$
VVMAXVAL$^{(2)}$	VVMAXVAL$^{(2)}$
VVMINVAL$^{(2)}$	VVMINVAL$^{(2)}$
VVMAXLOC$^{(2)}$	VVMAXLOC$^{(2)}$
VVMINLOC$^{(2)}$	VVMINLOC$^{(2)}$
VVMAXVL$^{(2)}$	VVMAXVL$^{(2)}$
VVMINVL$^{(2)}$	VVMINVL$^{(2)}$
MV$^{(2)}$	MV$^{(2)}$
SMV$^{(2)}$	SMV$^{(2)}$
MM$^{(3)}$	MM$^{(3)}$
SMM$^{(3)}$	SMM$^{(3)}$
MLUD$^{(2)}$	MLUD$^{(2)}$
V2CONV$^{(3)}$	V2CONV$^{(3)}$
M1CONV$^{(3)}$	M1CONV$^{(3)}$
MCONV$^{(4)}$	MCONV$^{(4)}$

C Weitere Quellen und Ergebnisse

C.1 CG–Verfahren

Das nachfolgend abgedruckte Quellprogramm eines least–squares CG–Verfahrens wurde uns freundlicherweise von Prof. A. K. Louis vom Institut für angewandte Mathematik an der Universität des Saarlandes überlassen.

```
main() {
/* modifiziert: - Uebersetzung Fortran77 -> C
                - Procedure Inlining
                - temporary elimination for return values of procedures
                - I/O-Anweisungen auskommentiert
                - durch GOTO implementierte aeussere While-Schleife
                  auskommentiert
*/
/*
C      BERECHNET ITMAX CG-SCHRITTE MIT STARTWERT X
C      ERGEBNIS STEHT WIEDER AUF X
C      RESTART NACH IREST SCHRITTEN
*/
double A[100][100], X[100], B[100];
int M, N, I, J, ITM, IRE, ITM, IRE;
double R[100], Y[100], D[100], S, RM, ALPHA, BETA;

/*   ITM = ITMAX;
     IRE = IREST;
50   IF (ITM.LT.IRE) IRE = ITM; */
for (I = 1; I<= M; I++) {
   Y[I] = 0.0;
   for (J = 1; J<= N; J++)
      Y[I] = Y[I] + A[I][J] * X[J];
}
for (I = 1; I<= M; I++)
   Y[I] = Y[I] - B[I];
for (I = 1; I<= N; I++) {
   R[I] = 0.0;
for (J = 1; J<= M; J++)
   R[I] = R[I] + A[J][I] * Y[J];
}
for (I = 1; I<= N; I++)
   D[I] = - R[I];
RM = 0.0;
for (I = 1; I<= N; I++)
   RM = RM + R[I] * R[I];

for (IM = 1; IM <= IRE; IM++) {
   /*write(6,*) IM */

   for (I = 1; I<= M; I++) {
      Y[I] = 0.0;
      for (J = 1; J<= N; J++)
         Y[I] = Y[I] + A[I][J] * D[J];
   }
   S = 0.0;
   for (I = 1; I<= M; I++)
      S = S + Y[I] * Y[I];
   ALPHA = RM / S;

   for (I = 1; I<= N; I++)
      X[I] = X[I] + ALPHA*D[I];
```

```
   for (I = 1; I<= M; I++) {
      Y[I] = 0.0;
      for (J = 1; J<= N; J++)
         Y[I] = Y[I] + A[I][J] * X[J];
   }
   for (I = 1; I<= M; I++)
      Y[I] = Y[I] - B[I];
   for (I = 1; I<= N; I++) {
      R[I] = 0.0;
      for (J = 1; J<= M; J++)
         R[I] = R[I] + A[J][I] * Y[J];
   }
   BETA = 1.0 / RM;
   RM = 0.0;
   for (I = 1; I<= N; I++)
      RM = RM + R[I] * R[I];
   BETA = RM * BETA;
   for (I = 1; I<= N; I++)
      D[I] = -R[I] + BETA * D[I];
}
/*   ITMAX = ITM - IRE;
     IF(ITM.GT.0) GOTO 50 */
}
```

Unser Mustererkenner–Prototyp liefert folgende Ausgabe[1]:

```
main()
  {
 MV (i,j,y[1:m],a[1:m][1:n],x[1:n],0.000000);
 VAADD (i,y[1:m],y[1:m],-(b[1:m]));
 MV (i,j,r[1:n],a[1:m][1:n],y[1:m],0.000000);
 VCOPY (i,d[1:n],-(r[1:n]));
 VQSUM (i,rm,r[1:n],0.000000);
  for (im=1; im<=ire; im++)
  {
 MV (i,j,y[1:m],a[1:m][1:n],d[1:n],0.000000);
 VQSUM (i,s,y[1:m],0.000000);
 * (alpha,1/(s),rm);
 VAADDMULTISV (i,x[1:n],x[1:n],1/(s),rm,d[1:n]);
 MV (i,j,y[1:m],a[1:m][1:n],x[1:n],0.000000);
 VAADD (i,y[1:m],y[1:m],-(b[1:m]));
 MV (i,j,r[1:n],a[1:m][1:n],y[1:m],0.000000);
 SCOPY (beta,1/(rm));
 VQSUM (i,rm,r[1:n],0.000000);
 * (beta,rm,beta);
 VADDSV (i,d[1:n],beta,d[1:n],-(r[1:n]));
  }
  }
```

Für dieses Programm wurde das Ziel, alle Phasen vollständig als Instanzen von Mustern zu erkennen, voll erreicht.

C.2 Livermore Loops

In diesem Abschnitt geben wir die C–Quellen[2] sowie die von unserem Mustererkennungsprototypen produzierten Ergebnisse zu den Livermore Loops an. Eine Übersicht gibt Tabelle 5.1. Auf

[1] Nicht alle Information, die der Mustererkenner hergeleitet hat, kann hier dargestellt werden; diese Ausgabe ist eher als übersichtlicher Kontrollausdruck zu interpretieren. Die Musterinstanzen als eigentliches Ergebnis der Mustererkennung sind als Datenstrukturen in die Syntaxbaum–Struktur eingehängt.

[2] Die C–Quellen zu den Livermore Loops sind über `netlib` frei erhältlich.

die Auflistung der Variablendeklarationen haben wir aus Platzgründen verzichtet. Ebenfalls haben wir nur in einigen Fällen mehr der vom Mustererkenner ausgegebenen Diagnose–Information abgedruckt, die wir für interessant erachten.

GVOP$^{(1)}$– und GMOP$^{(2)}$–Instanzen wurden dabei noch nicht in ihre Bestandteile zerlegt (diese Zerlegung ist noch nicht implementiert).

Kernel 1 – hydro fragment

```
main(){
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<=n-1 ; k++ )
            x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
}
```

Die Mustererkennung generiert:

```
main()
  for (l=1; l<=loop; l++)
 GVOP (k,x[0:(-1+n)], (q + (y[k] * (z[(10+k)] * r + z[(11+k)] * t)),
       y[0:(-1+n)],z[10:(9+n)],z[11:(10+n)],q,r,t);
```

Kernel 3 – inner product

```
main() {
    for ( l=1 ; l<=loop ; l++ ) {
        q = 0.0;
        for ( k=0 ; k<=n-1 ; k++ )
            q = q + z[k]*x[k];
    }
}
```

Die Mustererkennung liefert:

```
main()
  for (l=1; l<=loop; l++)
 SSP (k,q,z[0:(-1+n)],x[0:(-1+n)],0.000000);
```

Kernel 5 – tri-diagonal elimination, below diagonal

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( i=1 ; i<=n-1 ; i++ )
            x[i] = z[i]*( y[i] - x[i-1] );
}
```

```
main()
 for ( l=1 ; l<=loop ; l++ )
 {
 FOLR (j,x[1:n],y[1:n],1.000000,z[1:100],1.000000,x[0]);
 }
```

Kernel 7 – equation of state fragment

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<=n-1 ; k++ )
            x[k] = u[k] + r*( z[k] + r*y[k] ) +
                   t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
                      t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
}
```

Die Mustererkennung liefert:

```
main()
  for (l=1; l<=loop; l++)
 GVOP (k,x[0:(-1+n)],(r*(z[k]+r*y[k]) +(t *(u[(3+k)]+r*(u[(2+k)]+r*u[(1+k)]))
                      +(t*(u[(6+k)] + r*(u[(5+k)]+r*u[(4+k)]))) +u[k])),
       u[0:(-1+n)],z[0:(-1+n)],y[0:(-1+n)],u[3:(2+n)],u[2:n],
       u[1:n],u[6:(5+n)],u[5:(4+n)],u[4:(3+n)],r,r,t,r,r,t,r,r);
```

Kernel 8 – A.D.I. Integration

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( kx=1 ; kx<=2 ; kx++ )
            for ( ky=1 ; ky<=n-1 ; ky++ ) {
               du1[ky] = u1[0][ky+1][kx] - u1[0][ky-1][kx];
               du2[ky] = u2[0][ky+1][kx] - u2[0][ky-1][kx];
               du3[ky] = u3[0][ky+1][kx] - u3[0][ky-1][kx];
               u1[1][ky][kx]=
                  u1[0][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
                     (u1[0][ky][kx+1]-2.0*u1[0][ky][kx]+u1[0][ky][kx-1]);
               u2[1][ky][kx]=
                  u2[0][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
                     (u2[0][ky][kx+1]-2.0*u2[0][ky][kx]+u2[0][ky][kx-1]);
               u3[1][ky][kx]=
                  u3[0][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
                     (u3[0][ky][kx+1]-2.0*u3[0][ky][kx]+u3[0][ky][kx-1]);
            }
}
```

Der Mustererkenner wendet u.a. Skalar–Expansion und Schleifenaufgliederung an. Er liefert

```
main()
  for (l=1; l<=loop; l++)
  {
 VVJACOBI (ky,kx,du1[1:(-1+n)][1:2],NULL,NULL,NULL,NULL,u1[0][1:(-1+n)][1:2],
           NULL,NULL,-1.000000,NULL,1.000000,NULL,NULL,NULL,NULL,NULL,NULL,
           ky,2,1,NULL,NULL,NULL);
 VVJACOBI (ky,kx,du2[1:(-1+n)][1:2],NULL,NULL,NULL,NULL,u2[0][1:(-1+n)][1:2],
           NULL,NULL,-1.000000,NULL,1.000000,NULL,NULL,NULL,NULL,NULL,NULL,
           ky,2,1,NULL,NULL,NULL);
 VVJACOBI (ky,kx,du3[1:(-1+n)][1:2],NULL,NULL,NULL,NULL,u3[0][1:(-1+n)][1:2],
           NULL,NULL,-1.000000,NULL,1.000000,NULL,NULL,NULL,NULL,NULL,NULL,
           ky,2,1,NULL,NULL,NULL);
 GMOP (kx,ky,u1[1][1:(-1+n)][1:2], ...gekuerzt... , u1[0][1:(-1+n)][1:2],
       du1[1:(-1+n)][1:2],du2[1:(-1+n)][1:2],du3[1:(-1+n)][1:2],
       u1[0][1:(-1+n)][2:3],u1[0][1:(-1+n)][1:2],u1[0][1:(-1+n)][0:1],
       a11,a12,a13,sig);
 GMOP (kx,ky,u2[1][1:(-1+n)][1:2], ...gekuerzt... , u2[0][1:(-1+n)][1:2],
       du1[1:(-1+n)][1:2],du2[1:(-1+n)][1:2],du3[1:(-1+n)][1:2],
       u2[0][1:(-1+n)][2:3],u2[0][1:(-1+n)][1:2],u2[0][1:(-1+n)][0:1],
```

```
       a21,a22,a23,sig);
 GMOP (kx,ky,u3[1][1:(-1+n)][1:2], ...gekuerzt... , u3[0][1:(-1+n)][1:2],
       du1[1:(-1+n)][1:2],du2[1:(-1+n)][1:2],du3[1:(-1+n)][1:2],
       u3[0][1:(-1+n)][2:3],u3[0][1:(-1+n)][1:2],u3[0][1:(-1+n)][0:1],
       a31,a32,a33,sig);
  }
```

Kernel 9 – integrate predictors

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( i=0 ; i<=n-1 ; i++ )
            px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                       dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                       dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][5]) + px[i][2];
}
```

Die Mustererkennung generiert

```
main()
  for (l=1; l<=loop; l++)
 GVOP (i,px[0:(-1+n)][0],
       ((((((dm28*px[i][12])+(dm27*px[i][11])) +((c0 *
        (px[i][5] +px[i][4]))*c0)) +(dm28*px[i][12]))
        +px[i][2])+(dm28*px[i][12])),
       px[0:(-1+n)][12],px[0:(-1+n)][11],px[0:(-1+n)][10],px[0:(-1+n)][9],
       px[0:(-1+n)][8],px[0:(-1+n)][7],px[0:(-1+n)][6],px[0:(-1+n)][4],
       px[0:(-1+n)][5],px[0:(-1+n)][2],dm28,dm27,dm26,dm25,dm24,dm23,dm22,c0);
```

Kernel 10 – numerical differentiation

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( i=0 ; i<n ; i++ )
            ar        =      cx[i][ 4];
            br        = ar - px[i][ 4];
            px[i][ 4] = ar;
            cr        = br - px[i][ 5];
            px[i][ 5] = br;
            ar        = cr - px[i][ 6];
            px[i][ 6] = cr;
            br        = ar - px[i][ 7];
            px[i][ 7] = ar;
            cr        = br - px[i][ 8];
            px[i][ 8] = br;
            ar        = cr - px[i][ 9];
            px[i][ 9] = cr;
            br        = ar - px[i][10];
            px[i][10] = ar;
            cr        = br - px[i][11];
            px[i][11] = br;
            px[i][13] = cr - px[i][12];
            px[i][12] = cr;
}
```

Dies ist ein eindrucksvolles Beispiel zur Anwendung von Skalar–Expansion und Schleifenaufgliederung. Alle Phasen werden erkannt:

```
main()
  for (l=1; l<=loop; l++)
  {
 VCOPY (i,ar[0:(-1+n)],cx[0:(-1+n)][4]);
 VCOPY (i,px[0:(-1+n)][4],ar[0:(-1+n)]);
 VADD (i,br[0:(-1+n)],-(px[0:(-1+n)][4]),ar[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][5],br[0:(-1+n)]);
 VADD (i,cr[0:(-1+n)],-(px[0:(-1+n)][5]),br[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][6],cr[0:(-1+n)]);
 VADD (i,ar[0:(-1+n)],-(px[0:(-1+n)][6]),cr[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][7],ar[0:(-1+n)]);
 VADD (i,br[0:(-1+n)],-(px[0:(-1+n)][7]),ar[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][8],br[0:(-1+n)]);
 VADD (i,cr[0:(-1+n)],-(px[0:(-1+n)][8]),br[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][9],cr[0:(-1+n)]);
 VADD (i,ar[0:(-1+n)],-(px[0:(-1+n)][9]),cr[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][10],ar[0:(-1+n)]);
 VADD (i,br[0:(-1+n)],-(px[0:(-1+n)][10]),ar[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][11],br[0:(-1+n)]);
 VADD (i,cr[0:(-1+n)],-(px[0:(-1+n)][11]),br[0:(-1+n)]);
 VCOPY (i,px[0:(-1+n)][12],cr[0:(-1+n)]);
 VADD (i,px[0:(-1+n)][13],-(px[0:(-1+n)][12]),cr[0:(-1+n)]);
  }
```

Kernel 11 – first sum

```
main() {
    for ( l=1 ; l<=loop ; l++ ) {
        x[0] = y[0];
        for ( k=1 ; k<=n-1 ; k++ )
            x[k] = x[k-1] + y[k];
    }
}
```

Die Mustererkennung liefert

```
main()
  for (l=1; l<=loop; l++)
 PREVSUM (k,x[1:(-1+n)],t[1:(-1+n)],y[1:(-1+n)],y[0]);
```

Kernel 12 – first difference

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<n ; k++ )
            x[k] = y[k+1] - y[k];
}
```

Wir erhalten:

```
main()
  for (l=1; l<=100; l++)
 VJACOBI (k,NULL,x[2:n],NULL,NULL,NULL,NULL,y[2:n],NULL,NULL,
           NULL, -1.000000,1.000000,NULL, NULL,NULL,NULL,NULL,NULL,
           k,1,1,NULL,NULL,NULL);
```

Kernel 13 – 2-D PIC (Particle In Cell)

```
main () {
    for ( l=1 ; l<=loop ; l++ ) {
        for ( ip=0 ; ip<=n-1 ; ip++ ) {
            i1 = p[ip][0];
            j1 = p[ip][1];
            i1 = i1 % 64;
            j1 = j1 % 64;
            p[ip][2] = p[ip][2] + b[j1][i1];
            p[ip][3] = p[ip][3] + c[j1][i1];
            p[ip][0] = p[ip][0] + p[ip][2];
            p[ip][1] = p[ip][1] + p[ip][3];
            i2 = p[ip][0];
            j2 = p[ip][1];
            i2 = ( i2 % 64 ) - 1 ;
            j2 = ( j2 % 64 ) - 1 ;
            p[ip][0] = p[ip][0] + y[i2+32];
            p[ip][1] = p[ip][1] + z[j2+32];
            i2 = i2 + e[i2+32];
            j2 = j2 + f[j2+32];
            h[j2][i2] = h[j2][i2] + 1.0;
        }
    }
}
```

Trotz einiger indirekter Feldzugriffe wird die Schleifenaufgliederung nicht behindert. Auf `i1,j1,i2` und `j2` wird Skalar–Expansion angewendet. Wir erhalten:

```
main()
  {
  for (l=1; l<=loop; l++)
  {
 VAINC (ip,p[0:(-1+n)][1],p[0:(-1+n)][1],z[(32+j2)]);
 VAINC (ip,p[0:(-1+n)][0],p[0:(-1+n)][0],y[(32+i2)]);
 VCOPY (ip,j2[0:(-1+n)],p[0:(-1+n)][1]);
 VAMOD (ip,j2[0:(-1+n)],j2[0:(-1+n)],63);
  for ( ip=0 ; ip<=n-1 ; ip++ )
  j2[ip]=f[(32+j2[ip])]+j2[ip];
 VCOPY (ip,i2[0:(-1+n)],p[0:(-1+n)][0]);
 VAMOD (ip,i2[0:(-1+n)],i2[0:(-1+n)],63);
  for ( ip=0 ; ip<=n-1 ; ip++ )
  i2[ip]=e[(32+i2[ip])]+i2[ip];
 VAADD (ip,p[0:(-1+n)][1],p[0:(-1+n)][1],p[0:(-1+n)][3]);
 VAADD (ip,p[0:(-1+n)][0],p[0:(-1+n)][0],p[0:(-1+n)][2]);
  for ( ip=0 ; ip<=n-1 ; ip++ )
  p[ip][3]=p[ip][3]+c[j1[ip]][i1[ip]];
  for ( ip=0 ; ip<=n-1 ; ip++ )
  p[ip][2]=p[ip][2]+b[j1[ip]][i1[ip]];
 VCOPY (ip,j1[0:(-1+n)],p[0:(-1+n)][1]);
 VAMOD (ip,j1[0:(-1+n)],j1[0:(-1+n)],64);
 VCOPY (ip,i1[0:(-1+n)],p[0:(-1+n)][0]);
 VAMOD (ip,i1[0:(-1+n)],i1[0:(-1+n)],64);
  }
  for (l=1; l<=loop; l++)
  for ( ip=0 ; ip<=n-1 ; ip++ )
  h[j2[ip]][i2[ip]]=1.000000+h[j2[ip]][i2[ip]];
  }
```

Kernel 14 – 1-D PIC (Particle In Cell)

```
main() {
```

```
    for ( l=1 ; l<=loop ; l++ ) {
        for ( k=0 ; k<=n-1 ; k++ ) {
            vx[k] = 0.0;
            xx[k] = 0.0;
            ix[k] = grd[k];
            xi[k] = ix[k];
            ex1[k] = ex[ ix[k] - 1 ];
            dex1[k] = dex[ ix[k] - 1 ];
        }
        for ( k=0 ; k<=n-1 ; k++ ) {
            vx[k] = vx[k] + ex1[k] + ( xx[k] - xi[k] )*dex1[k];
            xx[k] = xx[k] + vx[k]  + flx;
            ir[k] = xx[k];
            rx[k] = xx[k] - ir[k];
            ir[k] = ( ir[k] % 2048-1 ) + 1;
            xx[k] = rx[k] + ir[k];
        }
        for ( k=0 ; k<=n-1 ; k++ ) {
            rh[ ir[k]-1 ] = rh[ ir[k]-1 ] + 1.0 - rx[k];
            rh[ ir[k]   ] = rh[ ir[k]   ] + rx[k];
        }
    }
}
```

Auch hier werden alle Phasen erkannt, die nicht indirekte Feldindizierungen enthalten:

```
main()
  for (l=1; l<=loop; l++)
  {
 VCOPY (k,ix[0:(-1+n)],grd[0:(-1+n)]);
 VCOPY (k,xi[0:(-1+n)],ix[0:(-1+n)]);
  for (k=0; k<=(-1+n); k++)
 SCOPY (dex1[k],dex[(-1+ix[k])]);
  for (k=0; k<=(-1+n); k++)
 SCOPY (ex1[k],ex[(-1+ix[k])]);
 VINIT (k,xx[0:(-1+n)],0.000000);
 VINIT (k,vx[0:(-1+n)],0.000000);
 GVOP (k,vx[0:(-1+n)],
       (((vx[k]+ex1[k]) +(((xx[k]+-(xi[k])) *dex1[k])*xx[k]))+vx[k]),
       vx[0:(-1+n)],ex1[0:(-1+n)],xx[0:(-1+n)],-(xi[0:(-1+n)]),dex1[0:(-1+n)]);
 GVOP (k,xx[0:(-1+n)],(vx[k]+xx[k]),xx[0:(-1+n)],vx[0:(-1+n)],flx);
 VCOPY (k,ir[0:(-1+n)],xx[0:(-1+n)]);
 VADD (k,rx[0:(-1+n)],-(ir[0:(-1+n)]),xx[0:(-1+n)]);
 GVOP (k,ir[0:(-1+n)],(0+(2048%ir[k])),ir[0:(-1+n)]);
 VADD (k,xx[0:(-1+n)],rx[0:(-1+n)],ir[0:(-1+n)]);
  for (k=0; k<=(-1+n); k++)
  {
  rh[ir[k]]=rx[k]+rh[ir[k]];
  rh[(-1+ir[k])]=rh[(-1+ir[k])]+-(rx[k]);
  }
  }
```

Kernel 18 - 2-D explicit hydrodynamics fragment

```
main() {
    for ( l=1 ; l<=loop ; l++ ) {
        t = 0.0037;
        s = 0.0041;
        kn = 6;
        jn = n;
        for ( k=1 ; k<=kn-1 ; k++ )
          for ( j=1 ; j<=jn-1 ; j++ ) {
```

```
              za[k][j] = (zp[k+1][j-1]+zq[k+1][j-1]-zp[k][j-1]-zq[k][j-1])*
                         (zr[k][j]+zr[k][j-1])/(zm[k][j-1]+zm[k+1][j-1]);
              zb[k][j] = (zp[k][j-1]+zq[k][j-1]-zp[k][j]-zq[k][j] ) *
                         (zr[k][j]+zr[k-1][j])/(zm[k][j]+zm[k][j-1]);
            }
        for ( k=1 ; k<=kn-1 ; k++ )
            for ( j=1 ; j<=jn-1 ; j++ ) {
                zu[k][j] = zu[k][j] +
                            s*( za[k][j]   *( zz[k][j] - zz[k][j+1] ) -
                                za[k][j-1] *( zz[k][j] - zz[k][j-1] ) -
                                zb[k][j]   *( zz[k][j] - zz[k-1][j] ) +
                                zb[k+1][j] *( zz[k][j] - zz[k+1][j] ) );
                zv[k][j] = zv[k][j] +
                            s*( za[k][j]   *( zr[k][j] - zr[k][j+1] ) -
                                za[k][j-1] *( zr[k][j] - zr[k][j-1] ) -
                                zb[k][j]   *( zr[k][j] - zr[k-1][j] ) +
                                zb[k+1][j] *( zr[k][j] - zr[k+1][j] ) );
            }
        for ( k=1 ; k<=kn-1 ; k++ )
            for ( j=1 ; j<=jn-1 ; j++ ) {
                zr[k][j] = zr[k][j] + t*zu[k][j];
                zz[k][j] = zz[k][j] + t*zv[k][j];
            }
    }
}
```

Alle Phasen werden erkannt:

```
main()
  {
  for (l=1; l<=loop; l++)
 SCOPY (jn,n);
  for (l=1; l<=loop; l++)
 SINIT (kn,6);
  for (l=1; l<=loop; l++)
  {
 SINIT (t,0.003700);
 SINIT (s,0.004100);
 GMOP (k,j,za[1:(-1+kn)][1:(-1+jn)],
       (((((zp[(1+k)][(-1+j)]+zq[(1+k)][(-1+j)]) *(zr[k][(-1+j)]+zr[k][j]))
       *zp[(1+k)][(-1+j)]) *1/(zm[(1+k)][(-1+j)]+zm[k][(-1+j)]))
       *zp[(1+k)][(-1+j)]),
       zp[2:(0+kn)][0:(-2+jn)],zq[2:(0+kn)][0:(-2+jn)],
       -(zp[1:(-1+kn)][0:(-2+jn)]),-(zq[1:(-1+kn)][0:(-2+jn)]),
       zr[1:(-1+kn)][1:(-1+jn)],zr[1:(-1+kn)][0:(-2+jn)],
       zm[1:(-1+kn)][0:(-2+jn)],zm[2:(0+kn)][0:(-2+jn)]);
 GMOP (k,j,zb[1:(-1+kn)][1:(-1+jn)],
      (((((zp[k][(-1+j)]+zq[k][(-1+j)]) *(zr[(-1+k)][j]+zr[k][j]))
        *zp[k][(-1+j)]) *1/(zm[k][(-1+j)]+zm[k][j]))*zp[k][(-1+j)]),
      zp[1:(-1+kn)][0:(-2+jn)],zq[1:(-1+kn)][0:(-2+jn)],
      -(zp[1:(-1+kn)][1:(-1+jn)]),-(zq[1:(-1+kn)][1:(-1+jn)]),
      zr[1:(-1+kn)][1:(-1+jn)],zr[0:(-2+kn)][1:(-1+jn)],
      zm[1:(-1+kn)][1:(-1+jn)],zm[1:(-1+kn)][0:(-2+jn)]);
 GMOP (k,j,zu[1:(-1+kn)][1:(-1+jn)], (....gekuerzt.......)
      zu[1:(-1+kn)][1:(-1+jn)],za[1:(-1+kn)][1:(-1+jn)],
      zz[1:(-1+kn)][1:(-1+jn)],-(zz[1:(-1+kn)][2:(0+jn)]),
      za[1:(-1+kn)][0:(-2+jn)],zz[1:(-1+kn)][1:(-1+jn)],
      -(zz[1:(-1+kn)][0:(-2+jn)]),zb[1:(-1+kn)][1:(-1+jn)],
      zz[1:(-1+kn)][1:(-1+jn)],-(zz[0:(-2+kn)][1:(-1+jn)]),
      zb[2:(0+kn)][1:(-1+jn)],zz[1:(-1+kn)][1:(-1+jn)],
      -(zz[2:(0+kn)][1:(-1+jn)]),s);
 GMOP (k,j,zv[1:(-1+kn)][1:(-1+jn)],(....gekuerzt.......),
      zv[1:(-1+kn)][1:(-1+jn)],za[1:(-1+kn)][1:(-1+jn)],
```

```
      zr[1:(-1+kn)][1:(-1+jn)],-(zr[1:(-1+kn)][2:(0+jn)]),
      za[1:(-1+kn)][0:(-2+jn)],zr[1:(-1+kn)][1:(-1+jn)],
      -(zr[1:(-1+kn)][0:(-2+jn)]),zb[1:(-1+kn)][1:(-1+jn)],
      zr[1:(-1+kn)][1:(-1+jn)],-(zr[0:(-2+kn)][1:(-1+jn)]),
      zb[2:(0+kn)][1:(-1+jn)],zr[1:(-1+kn)][1:(-1+jn)],
      -(zr[2:(0+kn)][1:(-1+jn)]),s);
 MAADDSM (j,k,zr[1:(-1+kn)][1:(-1+jn)],t,
          zu[1:(-1+kn)][1:(-1+jn)],zr[1:(-1+kn)][1:(-1+jn)]);
 MAADDSM (j,k,zz[1:(-1+kn)][1:(-1+jn)],t,
          zv[1:(-1+kn)][1:(-1+jn)],zz[1:(-1+kn)][1:(-1+jn)]);
  }
  }
```

*Kernel 21 – matrix*matrix product*

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<=24 ; k++ )
            for ( i=0 ; i<=24 ; i++ )
                for ( j=0 ; j<=n-1 ; j++ )
                    px[j][i] = px[j][i] + vy[k][i] * cx[j][k];
}
```

Das folgende Exzerpt aus der Diagnoseausgabe der Mustererkennung offenbart den durch den Musterhierarchiegraphen genommenen Pfad:

```
Starte Matching

 found matching *
 found matching ADDMUL
 found matching AADDMUL
 found matching VAADDSV
 found matching MAADDVV
 found matching MM
```

Wir erhalten:

```
main()
  for (l=1; l<=loop; l++)
 MM (j,i,k,px[0:(-1+n)][0:24],cx[0:(-1+n)][0:24],
     vy[0:24][0:24],px[0:(-1+n)][0:24]);
```

Kernel 22 – Planckian distribution

```
main() {
    expmax = 20.0;
    u[n-1] = 0.99*expmax*v[n-1];
    for ( l=1 ; l<=loop ; l++ ) {
        for ( k=0 ; k<=n-1 ; k++ ) {
            y[k] = u[k] / v[k];
            w[k] = x[k] / ( exp( y[k] ) -1.0 );
        }
    }
}
```

Alle Phasen werden erkannt:

```
main()
  {
 SINIT (expmax,20.000000);
 MULTIMUL (u[(-1+n)],0.990000,expmax,v[(-1+n)]);
  for (l=1; l<=loop; l++)
  {
 VMUL (k,y[0:(-1+n)],1/(v[0:(-1+n)]),u[0:(-1+n)]);
 GVOP (k,w[0:(-1+n)],((x[k] *1/((1/((exp(y[k]) +-1.000000))+y[k])))*x[k]),
       x[0:(-1+n)],y[0:(-1+n)]);
  }
  }
```

Kernel 23 – 2-D implicit hydrodynamics fragment

```
main() {
    for ( l=1 ; l<=loop ; l++ )
        for ( j=1 ; j<6 ; j++ )
            for ( k=1 ; k<n ; k++ ) {
                qa = za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k] +
                     za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k] + zz[j][k];
                za[j][k] += 0.175*( qa - za[j][k] );
            }
}
```

Per Querkante wird der Differenzenstern, der `qa` berechnet, in den Ausdruck für `za[j][k]` expandiert (vgl. Abschnitt 6.2.8), woraus sich ein neuer Differenzenstern für `za[j][k]` ergibt. Die Berechnung von `qa` fällt der Elimination nutzlosen Codes zum Opfer. Wir erhalten schließlich

```
main()
  for (l=1; l<=lp; l++)
 MGAUSSSEIDEL (k,j,za[2:n][2:6],0.175000,NULL,NULL,NULL, za[2:n][2:6],NULL,
              NULL,NULL,zb[2:n][2:6],NULL,zz[2:n][2:6],4.714286,
              zu[2:n][2:6],NULL,zr[2:n][2:6],NULL,k,1,1,j,2,1);
```

Kernel 24 – find location of first minimum in array

```
main() {
    x[n/2] = -1.0e+10;
    for ( l=1 ; l<=loop ; l++ ) {
        m = 0;
        for ( k=1 ; k<=n-1 ; k++ ) {
            if ( x[k] < x[m] ) m = k;
        }
    }
}
```

Wir erhalten

```
main()
  {
 x[(0.500000*n)]=-1.000000e+10);
  for (l=1; l<=loop; l++)
 VMAXLOC (k,m,x[1:(-1+n)],0);
  }
```

Tabelle C.1 gibt die Laufzeiten des Prototyps auf den beschriebenen Livermore Loops an.

Kernel	Anzahl Knoten im Syntaxbaum	Laufzeit
LL 1	47	0.2 sec.
LL 3	35	0.2 sec.
LL 5	45	0.1 sec.
LL 7	88	0.3 sec.
LL 8	320	1.3 sec.
LL 9	91	0.3 sec.
LL 10	242	1.1 sec.
LL 11	48	0.2 sec.
LL 12	32	0.1 sec.
LL 13	258	0.9 sec.
LL 14	229	0.7 sec.
LL 18	608	2.5 sec.
LL 21	58	0.1 sec.
LL 22	80	0.2 sec.
LL 23	105	0.2 sec.
LL 24	47	0.1 sec.

Tabelle C.1 Livermore Loops: Größen der Syntaxbäume und Laufzeit des Mustererkennungs–Prototypen auf einer Low–End SUN SparcStation SLC. Die meiste Zeit wird dabei für die Ausgabe von Diagnose–Informationen auf eine Datei verbraucht. Obwohl der Prototyp in mancherlei Hinsicht suboptimal programmiert ist, sind die benötigten Zeiten durchaus akzeptabel.

C.3 Sonstige Testprogramme

Matrix-Matrix-Multiplikation

```
main () {
for (l=1; l<=m; l++)
 for (i=1; i<=99; i++)
  for (j=1; j<=99; j++) {
    c[i][j][l] = 0.0;
    for (k=1; k<=100; k++)
      c[i][j][l] = c[i][j][l] + a[i][l][k] * b[l][k][j];
  }
}

Syntaxbaum mit 71 Knoten
```

Wir drucken diesmal die gesamte Diagnoseausgabe ab:

```
Starte Matching

--> main0
--> for13
--> for20
--> for27
--> =34
 found matching SINIT
  {
 SINIT (c[i][j][l],0.000000);
  for (k=1; k<=100; k++)
  c[i][j][l] = (c[i][j][l] +(a[i][l][k] *b[l][k][j]));
  }
--> for41
--> =48
--> +59
```

```
--> *65
 found matching *
 cannot match vertical edge on trigger pattern 1 = VAR
 found matching ADDMUL
 cannot match vertical edge on trigger pattern 1 = VAR
 found matching AADDMUL
 AADDMUL (c[i][j][l],a[i][l][k],b[l][k][j],c[i][j][l]);
 found matching SSP
 SSP (k,c[i][j][l],a[i][l][1:100],b[l][1:100][j],c[i][j][l]);
   Cross-Edge 0: wegen c[i][j][l] zu Knoten =34 in Dim.0, Crosstyp FLOW
 Contracte SINIT34 mit rechts SSP41
 cross matching SSP
 SSP (k,c[i][j][l],a[i][l][1:100],b[l][1:100][j],0.000000);
 found matching VM
 VM (j,k,c[i][1:99][l],a[i][l][1:100],b[l][1:100][1:99],0.000000);
 found matching MM
 MM (i,j,k,c[1:99][1:99][l],a[1:99][l][1:100],b[l][1:100][1:99],0.000000);
 cannot match vertical edge on trigger pattern 150 = MM
 cannot match vertical edge on trigger pattern 1 = VAR
  for (l=1; l<=m; l++)
 MM (i,j,k,c[1:99][1:99][l],a[1:99][l][1:100],b[l][1:100][1:99],0.000000);
 Nix gefunden zu for13
 main()
  for (l=1; l<=m; l++)
 MM (i,j,k,c[1:99][1:99][l],a[1:99][l][1:100],b[l][1:100][1:99],0.000000);
 Nix gefunden zu main0
```

Man kann anhand dieser Diagnoseausgabe gut nachvollziehen, welchem Pfad der Mustererkenner im Musterhierarchiegraphen gefolgt ist.

```
main()
  for (l=1; l<=m; l++)
 MM (i,j,k,c[1:99][1:99][l],a[1:99][l][1:100],b[l][1:100][1:99],0.000000);
```

Dieses Beispiel zeigt auch, daß die Mustererkennung sich durch weitere, von äußeren Schleifen indizierte Indexausdrücke nicht stören läßt.

NAS–Benchmark, Kernel 1

```
/*
C     SUBROUTINE MXM (A, B, C, L, M, N)
C     DIMENSION A(L,M), B(M,N), C(L,N)
C
C
C   4-WAY UNROLLED MATRIX MULTIPLY ROUTINE FOR VECTOR COMPUTERS.
C   M MUST BE A MULTIPLE OF 4.  CONTIGUOUS DATA ASSUMED.
C   D H BAILEY  11/15/84
C
*/
main() {
  for (k = 1; k <= n; k++)
      for (i = 1; i <= l; i++) {
          c[i][k] = 0.0;
          for (j = 1; j <= m; j += 4)
            c[i][k] = c[i][k]  +  a[i][j]   * b[j][k]
                               +  a[i][j+1] * b[j+1][k]
                               +  a[i][j+2] * b[j+2][k]
                               +  a[i][j+3] * b[j+3][k];
  }
}

Syntaxbaum mit 95 Knoten
```

Hier wird die j–Schleife wieder aufgerollt. Wir erhalten

```
main()
 MM (i,k,j,c[1:l][1:n],a[1:l][1:m],b[1:m][1:n],0.000000);
```

NAS–Benchmark, Kernel 5

```
/*
  GAUSS ELIMINATION aus NAS Kernel 5
*/
main(){
 int i,j,k,matdim;
 double rmatrx[100][100];

      for ( i = 1; i<= matdim; i++) {
        rmatrx[i][i] = 1.0 / rmatrx[i][i];
        for( j = i+1; j<= matdim; j++) {
          rmatrx[j][i] = rmatrx[j][i] * rmatrx[i][i];
          for ( k = i+1; k<= matdim; k++)
            rmatrx[j][k] = rmatrx[j][k] - rmatrx[j][i] * rmatrx[i][k];
        }
      }
}
```

```
Syntaxbaum mit 82 Knoten
```

Die Mustererkennung produziert

```
main()
 LUD (i,j,k,-(rmatrx[1:matdim][2:matdim]),rmatrx[2:matdim][1:matdim],
      -(rmatrx[1:matdim][2:matdim]),1/(rmatrx[1:matdim][1:matdim]),
      rmatrx[1:matdim][1:matdim],rmatrx[2:matdim][1:matdim],
      -(rmatrx[1:matdim][2:matdim]));
```

Jacobi–Relaxation

```
main () {
for (i=2; i<=N-1; i++)
 for (j=2; j<=M-1; j++)
  uhelp[i][j] = u[i][j]
             + 0.25*( f[i][j] + up[i][j]*u[i-1][j] + down[i][j]*u[i+1][j]
                        + left[i][j]*u[i][j-1] + right[i][j]*u[i][j+1] );
}
```

Wir erhalten:

```
main()
 MJACOBI (i,j,uhelp[2:(-1+n)][2:(-1+m)],0.250000,NULL,NULL,NULL,
          u[2:(-1+n)][2:(-1+m)],f[2:(-1+n)][2:(-1+m)],0.250000,
          NULL,up[2:(-1+n)][2:(-1+m)],NULL,
          left[2:(-1+n)][2:(-1+m)],5.000000,right[2:(-1+n)][2:(-1+m)],
          NULL,down[2:(-1+n)][2:(-1+m)],NULL,
          i,1,1,j,2,1);
```

Jacobi–Verfahren zur Eigenwertberechnung

```
/* JACOBI-EIGENWERT-Iterative Solver aus [LT93] */

main(){
for (k=1; k<=23; k++) {      /* iteriere */
  for (i=1; i<=m; i++) {
    V[i] = 0.0;
    for (j=1; j<=m; j++)
      V[i] = V[i] + A[i][j] * X[j];
  }
  for (i=1; i<=m; i++)
    X[i] = X[i] + (B[i] - V[i]) / A[i][i];
}
}

Syntaxbaum mit 109 Knoten
```

Die der zweiten i–Schleife entsprechende $\mathsf{GVOP}^{(1)}$–Instanz wird hier bereits als in ihre Bestandteile aufgespalten dargestellt:

```
main()
  for (k=1; k<=23; k++)
  {
 MV (i,j,v[1:m],a[1:m][1:m],x[1:m],0.000000);
 VADD (i,temp_233[1:m],b[1:m],-(v[1:m]));
 SVDIAG (i,temp_234[1:m],temp_233[1:m],1/(a[1:m][1:m]));
 VAADD (i,x[1:m],x[1:m],temp_234[1:m]);
  }
```

C.4 EISPACK–Routine tred2

Die Routine tred2 aus der Numerik–Bibliothek EISPACK ([Smi76]; vgl. Abschnitt 5.4.4) haben wir von FORTRAN nach C umformuliert. Die Induktionsvariablen haben wir manuell ersetzt (die Vorabtransformation ist noch nicht implementiert), die GOTOs in if-then-else umgesetzt:

```
main() {
  int i,j,k,l,n;
  double z[100][100], a[100][100];
  double d[100], e[100];
  double f, g, h, hh, scale;

 for (i=1; i<=n; i++) {
   for (j=i; j<=n; j++)
     z[j][i] = a[j][i];
   d[i] = a[n][i];
 }
 if (n!=1) {
   for (i=n; i>=2; i+=-1) {
     h = 0.0;
     scale = 0.0;
     if (i>=3)
       for (k=1; k<=i-1; k++)
         scale = scale + d[k];
     e[i] = d[i-1];
     for (j=1; j<=i-1; j++) {
       d[j] = z[i-1][j];
       z[i][j] = 0.0;
       z[j][i] = 0.0;
```

```
    }
    if (scale != 0) {
      for (k=1; k<=i-1; k++) {
        d[k] = d[k] / scale;
        h = h + d[k] * d[k];
      }
      f = d[i-1];
      g = sqrt(h);
      if (f>=0) g = -g;
      e[i] = scale * g;
      h = h - f * g;
      d[i-1] = f - g;
      for (j=1; j<=i-1; j++)
        e[j] = 0.0;
      for (j=1; j<=i-1; j++) {
        f = d[j];
        z[j][i] = f;
        g = e[j] + z[j][j] * f;
        if (i-1 >= j+1)
          for (k=j+1; k<=i-1; k++) {
            g = g + z[k][j] * d[k];
            e[k] = e[k] + z[k][j] * f;
          }
        e[j] = g;
      }
      f = 0.0;
      for (j=1; j<=i-1; j++) {
        e[j] = e[j] / h;
        f = f + e[j] * d[j];
      }
      hh = f/(h+h);
      for (j=1; j<=i-1; j++)
        e[j] = e[j] - hh * d[j];
      for (j=1; j<=i-1; j++) {
        f = d[j];
        g = e[j];
        for (k=j; k<=i-1; k++)
          z[k][j] = z[k][j] - f * e[k] - g * d[k];
        d[j] = z[i-1][j];
        z[i][j] = 0.0;
      }
      d[i] = h;
    }
  }
  for (l=1; l<=n-1; l++) {
    z[n][l] = z[l][l];
    z[l][l] = 1.0;
    h = d[l+1];
    if (h!=0.0) {
      for (k=1; k<=l; k++)
        d[k] = z[k][l+1] / h;
      for (j=1; j<=l; j++) {
        g = 0.0;
        for (k=1; k<=l; k++)
          g = g + z[k][l+1] * z[k][j];
        for (k=1; k<=l; k++)
          z[k][j] = z[k][j] - g * d[k];
      }
    }
    for (k=1; k<=l; k++)
      z[k][l+1] = 0.0;
  }
} else {
  for (i=1; i<=n; i++) {
```

```
      d[i] = z[n][i];
      z[n][i] = 0.0;
    }
    z[n][n] = 1;
    e[1] = 0.0;
  }
}

Syntaxbaum mit 632 Knoten
```

Die Mustererkennung liefert[3]:

```
main()
  {
 VCOPY (i,d[1:n],a[n][1:n]);
 MCOPY (i,j,z[1:n][1:n],a[1:n][1:n]);
  if ((n !=1))
  {
  for (i=n; i>=2; i+=-1)
  {
 SINIT(h,0.000000);
 SINIT (scale,0.000000);
  if ((i >=3))
 VSUM (k,scale,d[1:(-1+i)],scale);
 SCOPY (e[i],d[(-1+i)]);
 VINIT (j,z[1:(-1+i)][i],0.000000);
 VINIT (j,z[i][1:(-1+i)],0.000000);
 VCOPY (j,d[1:(-1+i)],z[(-1+i)][1:(-1+i)]);
  if ((scale !=0))
  {
 GVOP (k,d[1:(-1+i)],(d[k]*1/(scale)),d[1:(-1+i)],1/(scale));
 VQSUM (k,h,d[1:(-1+i)],h);
 SCOPY (f,d[(-1+i)]);
 SQRT (g,h);
  if ((f >=0))
 SCOPY (g,-(g));
 MUL (e[i],g,scale);
 AADDMUL (h,-(f),g,h);
 ADD (d[(-1+i)],f,-(g));
 VINIT (j,e[1:(-1+i)],0.000000);
  for (j=1; j<=(-1+i); j++)
  {
 SCOPY (f,d[j]);
 SCOPY (z[j][i],f);
 ADDMUL (g,e[j],z[j][j],f);
  if (((i +-1) >=(j +1)))
 VAADDSV (k,e[(1+j):(-1+i)],f,z[(1+j):(-1+i)][j],e[(1+j):(-1+i)]);
  if (((i +-1) >=(j +1)))
 SSP (k,g,z[(1+j):(-1+i)][j],d[(1+j):(-1+i)],g);
 SCOPY (e[j],g);
  }
 GVOP (j,e[1:(-1+i)],(e[j]*1/(h)),e[1:(-1+i)],1/(h));
 SSP (j,f,d[1:(-1+i)],e[1:(-1+i)],0.000000);
  hh = (f *1/((h +h)));
 VAADDSV (j,e[1:(-1+i)],hh,-(d[1:(-1+i)]),e[1:(-1+i)]);
 VINIT (j,z[i][1:(-1+i)],0.000000);
  for (j=1; j<=(-1+i); j++)
  {
 SCOPY (f,d[j]);
 SCOPY (g,e[j]);
 GVOP (k,z[j:(-1+i)][j],(z[k][j]+(-(e[k])*f)),
       z[j:(-1+i)][j],e[j:(-1+i)],d[j:(-1+i)],f,g);
```

[3] Der Prototyp der Mustererkennung benötigt dazu 5.4 sec. auf einer Low–End SUN SLC.

```
 }
VCOPY (j,d[1:(-1+i)],z[(-1+i)][1:(-1+i)]);
SCOPY (d[i],h);
 }
 }
 for (l=1; l<=(-1+n); l++)
 {
SCOPY (z[n][l],z[l][l]);
SINIT (z[l][l],1.000000);
SCOPY (h,d[(1+l)]);
 if ((h !=0.000000))
 {
GVOP (k,d[1:l],(z[k][(1+l)]*1/(h)),z[1:l][(1+l)],1/(h));
SSP (k,g,z[1:l][l],z[1:l][(1+l)],0.000000);
MV (j,k,temp_223[1:l],z[1:l][1:l],z[1:l][(1+l)],0.000000);
MAADDVV (j,k,z[1:l][1:l],temp_223[1:l],-(d[1:l]),z[1:l][1:l]);
 }
VINIT (k,z[1:l][(1+l)],0.000000);
 }
 }
 else
 {
VCOPY (i,d[1:n],z[n][1:n]);
VINIT (i,z[n][1:n],0.000000);
SINIT (z[n][n],1);
SINIT (e[1],0.000000);
 }
 }
```

C.5 Perfect Club Benchmark–Routine EFLUX

Die Routine EFLUX (der Programmtext ist aus [Gup92] entnommen) entstammt der Perfect–Club–Benchmark–Routine flo52 und ist dort für einen Hauptanteil der verbrauchten Rechenzeit verantwortlich. Wiederum haben wir die FORTRAN77–Quelle nach C transformiert und die Konstanten propagiert.

```
main() {
 double dw[5][195][100], fs[195][100][5],
        w[195][100][5], p[194][100], x[195][100][5];
 double xx, xy, yx, yy, pa, qsp, qsm;
 int j, i, n;

for (j=2; j<=33; j++)
  for (i=1; i<= 194; i++) {
    xy = x[i][j][1] - x[i][j-1][1];
    yy = x[i][j][2] - x[i][j-1][2];
    pa = p[i+1][j] + p[i][j];
    qsp = (yy*w[i+1][j][2] - xy*w[i+1][j][3]) / w[i+1][j][1];
    qsm = (yy*w[i][j][2] - xy*w[i][j][3]) / w[i][j][1];
    fs[i][j][1] = qsp*w[i+1][j][1] + qsm*w[i][j][1];
    fs[i][j][2] = qsp*w[i+1][j][2] + qsm*w[i][j][2] + yy*pa;
    fs[i][j][3] = qsp*w[i+1][j][3] + qsm*w[i][j][3] - xy*pa;
    fs[i][j][4] = qsp*(w[i+1][j][4]+p[i+1][j]) + qsm*(w[i][j][4]+p[i][j]);
  }
for (n=1; n<=4; n++)
  for (j=2; j<=33; j++)
    for (i=2; i<= 194; i++)
      dw[i][j][n] = fs[i][j][n] - fs[i-1][j][n];
for (i=2; i<= 194; i++) {
  xx = x[i][1][1] - x[i-1][1][1];
```

```
  yx = x[i][1][2] - x[i-1][1][2];
  pa = p[i][2] + p[i][1];
  fs[i][1][1] = 0.0;
  fs[i][1][2] = -yx*pa;
  fs[i][1][3] = xx*pa;
  fs[i][1][4] = 0.0;
}
for (j=2; j<=33; j++)
  for (i=2; i<= 194; i++) {
    xx = x[i][j][1] - x[i-1][j][1];
    yx = x[i][j][2] - x[i-1][j][2];
    pa = p[i][j+1] + p[i][j];
    qsp = (xx*w[i][j+1][3] - yx*w[i][j+1][2]) / w[i][j+1][1];
    qsm = (xx*w[i][j][3] - yx*w[i][j][2]) / w[i][j][1];
    fs[i][j][1] = qsp*w[i][j+1][1] + qsm*w[i][j][1];
    fs[i][j][2] = qsp*w[i][j+1][2] + qsm*w[i][j][2] - yx*pa;
    fs[i][j][3] = qsp*w[i][j+1][3] + qsm*w[i][j][3] + xx*pa;
    fs[i][j][4] = qsp*(w[i][j+1][4]+p[i][j+1]) + qsm*(w[i][j][4]+p[i][j]);
  }
for (n=1; n<=4; n++)
  for (j=2; j<=33; j++)
    for (i=2; i<= 194; i++)
      dw[i][j][n] = dw[i][j][n] + fs[i][j][n] - fs[i][j-1][n];
}

Syntaxbaum mit 906 Knoten
```

Dieses Beispiel demonstriert die Wichtigkeit von Skalar– und Vektor–Expansion für die Effektivität der Schleifenaufgliederung. Alle Phasen werden bis in Dimension 2 vollständig erkannt:

```
main()
  {
 VVJACOBI (i,j,xy[1:194][2:33],...,x[1:194][2:33][1],NULL,NULL,
          -1.000000,1.000000,NULL,...,i,1,1,NULL,NULL,NULL);
 VVJACOBI (i,j,yy[1:194][2:33],...,x[1:194][2:33][2],NULL,NULL,
          -1.000000,1.000000,NULL,...,i,1,1,NULL,NULL,NULL);
 GMOP (j,i,qsp[1:194][2:33],
      ((MULMUL (NULL,w[(1+i)][j][2],yy[i][j],-(w[(1+i)][j][3]),xy[i][j])
      *1/(w[(1+i)][j][1]))*w[(1+i)][j][2]), yy[1:194][2:33],w[2:195][2:33][2],
      xy[1:194][2:33],w[2:195][2:33][3],1/(w[2:195][2:33][1]));
 GMOP (j,i,qsm[1:194][2:33],
      (( MULMUL (NULL,w[i][j][2],yy[i][j],-(w[i][j][3]),xy[i][j])
      *1/(w[i][j][1]))*w[i][j][2]),yy[1:194][2:33],w[1:194][2:33][2],
      xy[1:194][2:33],w[1:194][2:33][3],1/(w[1:194][2:33][1]));
 GMOP (j,i,fs[1:194][2:33][1],
      MULMUL (NULL,w[(1+i)][j][1],qsp[i][j],w[i][j][1],qsm[i][j]),
      qsp[1:194][2:33],w[2:195][2:33][1],qsm[1:194][2:33],w[1:194][2:33][1]);
 GMOP (j,i,fs[1:194][2:33][4],
      ((((qsp[i][j] *(w[(1+i)][j][4]+p[(1+i)][j]))*qsp[i][j])
      +((qsm[i][j] *(w[i][j][4]+p[i][j]))*qsm[i][j]))+qsp[i][j]),
      qsp[1:194][2:33],w[2:195][2:33][4],p[2:195][2:33],
      qsm[1:194][2:33],w[1:194][2:33][4],p[1:194][2:33]);
 VVJACOBI (i,j,pa[1:194][2:33],...,p[1:194][2:33],NULL,NULL,
          NULL,1.000000,1.000000,NULL,...,i,1,1,NULL,NULL,NULL);
 VVJACOBI (i,j,fs[1:194][2:33],...,w[1:194][j][2],yy[1:194][j],pa[1:194][j],
          NULL,qsm[1:194][j],qsp[1:194][j],NULL,...,NULL,i,1,1,NULL,...);
 VVJACOBI (i,j,fs[1:194][2:33],...,w[1:194][j][3],-(xy[1:194][j]),pa[1:194][j],
          NULL,qsm[1:194][j],qsp[1:194][j],NULL,...,NULL,i,1,1,NULL,...);
  for (n=1; n<=4; n++)
 VVJACOBI (i,j,dw[2:194][2:33][n],...,fs[2:194][2:33][n],NULL,NULL,
          -1.000000,1.000000,NULL,...,NULL,i,1,1,NULL,NULL,NULL);
 VINIT (i,fs[2:194][1][4],0.000000);
 VINIT (i,fs[2:194][1][1],0.000000);
```

```
VADD (i,pa[2:194],p[2:194][2],p[2:194][1]);
VJACOBI (i,NULL,yx[2:194],NULL,...,NULL,x[2:194][1][2],NULL,NULL,
         -1.000000,1.000000,NULL,...,NULL,i,1,1,NULL,NULL,NULL);
VMUL (i,fs[2:194][1][2],pa[2:194],-(yx[2:194]));
VJACOBI (i,NULL,xx[2:194],...,x[2:194][1][1],NULL,NULL,
         -1.000000,1.000000,NULL,...,NULL,i,1,1,NULL,...);
VMUL (i,fs[2:194][1][3],pa[2:194],xx[2:194]);
VVJACOBI (i,j,xx[2:194][2:33],...,x[2:194][2:33][1],NULL,NULL,
          -1.000000,1.000000,NULL,...,NULL,i,1,1,NULL,...);
VVJACOBI (i,j,yx[2:194][2:33],...,x[2:194][2:33][2],NULL,NULL,
          -1.000000,1.000000,NULL,...,NULL,i,1,1,NULL,...);
GMOP (j,i,qsp[2:194][2:33],
      (( MULMUL (NULL,w[i][(1+j)][3],xx[i][j],-(w[i][(1+j)][2]),yx[i][j])
      *1/(w[i][(1+j)][1]))*w[i][(1+j)][3]),xx[2:194][2:33],w[2:194][3:34][3],
      yx[2:194][2:33],w[2:194][3:34][2],1/(w[2:194][3:34][1]));
GMOP (j,i,qsm[2:194][2:33],
      (( MULMUL (NULL,w[i][j][3],xx[i][j],-(w[i][j][2]),yx[i][j])
      *1/(w[i][j][1]))*w[i][j][3]),xx[2:194][2:33],w[2:194][2:33][3],
      yx[2:194][2:33],w[2:194][2:33][2],1/(w[2:194][2:33][1]));
GMOP (j,i,fs[2:194][2:33][1],
      MULMUL (NULL,w[i][(1+j)][1],qsp[i][j],w[i][j][1],qsm[i][j]),
      qsp[2:194][2:33],w[2:194][3:34][1],qsm[2:194][2:33],w[2:194][2:33][1]);
VVJACOBI (j,i,fs[2:194][2:33][2],...,w[2:194][2:33][2],
          -(yx[2:194][2:33]),pa[2:194][2:33],
          NULL,qsm[2:194][2:33],qsp[2:194][2:33],NULL,...,NULL,j,2,1,NULL,...);
VVJACOBI (j,i,fs[2:194][2:33][3],NULL,NULL,NULL,NULL,w[2:194][2:33][3],
          xx[2:194][2:33],pa[2:194][2:33],
          NULL,qsm[2:194][2:33],qsp[2:194][2:33],NULL,...,NULL,j,2,1,NULL,...);
GMOP (j,i,fs[2:194][2:33][4],
      ((((qsp[i][j] *(w[i][(1+j)][4]+p[i][(1+j)]))*qsp[i][j])
      +((qsm[i][j] *(w[i][j][4]+p[i][j]))*qsm[i][j]))+qsp[i][j]),
      qsp[2:194][2:33],w[2:194][3:34][4],p[2:194][3:34],qsm[2:194][2:33],
      w[2:194][2:33][4],p[2:194][2:33]);
VVJACOBI (j,i,pa[2:194][2:33],NULL,NULL,NULL,NULL,p[2:194][2:33],NULL,NULL,
          NULL,1.000000,1.000000,NULL,...,NULL,j,2,1,NULL,...);
 for (n=1; n<=4; n++)
VVJACOBI (j,i,dw[2:194][2:33][n],...,-(fs[2:194][2:33][n]),
          dw[2:194][2:33][n],1.000000,
          -1.000000,1.000000,NULL,...,NULL,j,2,1,NULL,...);
 }
```

D Muster für Operationen auf dünnbesetzten Matrizen

Dieser Anhang enthält eine vorläufige Sammlung von Mustern, die auf dünnbesetzten Matrizen mit indirekten Feldzugriffen arbeiten.

Problematisch für die Mustererkennung sind Operationen auf dünnbesetzten Matrizen, weil es mehrere konkurrierende und völlig verschiedene Datenstrukturen zum Speichern der von Null verschiedenen Einträge einer dünnbesetzten Matrix gibt, von denen jede Vor– und Nachteile besitzt und keine davon sich als *die* Darstellung dünnbesetzter Matrizen etabliert hat. Einige dieser Datenstrukturen arbeiten mit indirekten Feldzugriffen auf einem linearen *workspace*–Feld, die übrigen mit Pointern und verketteten Listen. Sie werden im nächsten Abschnitt kurz vorgestellt.

Da das PARAMAT–Mustererkennungswerkzeug derzeit weder indirekte Feldzugriffe noch Pointer verarbeiten kann, sind diese Muster nicht implementiert. Die Datenabhängigkeits– und –flußanalyse erscheint uns für indirekte Feldzugriffe (worst-case–Annahmen) noch etwas leichter als für Pointer, obwohl die exakte Information über die referenzierten Daten in beiden Fällen erst zur Laufzeit zur Verfügung steht. Eine Erweiterung der PARAMAT–Mustererkennung für indirekte Feldzugriffe erscheint uns naheliegender.

D.1 Datenstrukturen für dünnbesetzte Matrizen

Implementierung durch indirekte Feldzugriffe

Wir stellen verschiedene Möglichkeiten am Beispiel der Matrix–Vektor–Multiplikation einer dünnbesetzten Matrix vor.

1. Standardansatz mit Speicherung der Zeilenlängen:

   ```
   for i=1 to n
     for j=1 to grad[i]
       b[i] = b[i] +- A[i][j] * x[col[i][j]]
   ```

2. (vgl. [Ben86], S. 96) Speicherung der Zeilenanfänge:

   ```
   for i=1 to n
     for k=FirstInCol[i] to FirstInCol[i+1]-1
       b[i] = b[i] +- A[k] * x[Row[k]]
   ```

3. Die Sparse–Routinen der *Numerical Recipes* [PTVF92] benutzen ebenfalls das „row indexed sparse storage format", wobei jedoch die Diagonale der Matrix insgesamt gespeichert wird, und die übrigen Matrixelemente nur, falls sie ungleich Null sind.

   Routine `sprsax()`:

   ```
   for i=1 to n {
     b[i] = A[i] * x[i];                  /* Diagonalelemente */
       for k=ija[i] to ija[i+1]-1
         b[i] = b[i] + A[k] * x[ija[k]]
   }
   ```

   Routine `sprstx()` (Matrix A ist transponiert)

```
for i=1 to n   b[i] = A[i] * x[i];  /* Diagonalelemente */
for i=1 to n {
  for k=ija[i] to ija[i+1]-1
    b[ija[k]] = b[ija[k]] + A[k] * x[i]
}
```

Während die Matrix–Vektor–Multiplikation recht einfach aussieht und offenbar mit der Mustererkennung erkennbar wäre, sehen die Implementierungen der Matrix–Matrix–Multiplikation oder der LR–Zerlegung ziemlich unstrukturiert aus, weil sich in diesen Fällen die Struktur der Ergebnismatrix in jedem Schritt verändern kann, wenn bisherige Null–Einträge plötzlich ungleich Null werden und eingefügt werden müssen (*fill–in*). Für diese Fälle gibt es eine geeignetere Datenstruktur:

4. Die unsortierte Gustavson–Datenstruktur (vgl. [SBV93]) benutzt zwei Offset–Felder, deren Einträge den Anfang jeder neuen Zeile bzw. jeder neuen Spalte im linearen *workarray* angeben, und für jeden *workarray*–Eintrag den zugehörigen Zeilen– und Spaltenindex.

   Die sortierte Variante der Gustavson–Struktur erhält zusätzlich eine Reihenfolge der *workarray*–Einträge nach aufsteigenden Zeilen– und Spaltenindexwerten aufrecht.

Implementierung durch verkettete Listen

Für die Implementierung dünnbesetzter Matrizen durch verkettete Listen gibt es folgende vier wichtigen Datenstrukturen [SBV93], die alle ein effizientes *fill–in* unterstützen:

1. Die unsortierte, in zwei Dimensionen einfach verkettete Listenstruktur benutzt zwei Felder von Pointern, die auf die ersten Einträge jeder Zeile bzw. die ersten Einträge jeder Spalte im *workarray* zeigen, und zu jedem workarray–Eintrag den zugehörigen Zeilen– und Spaltenindex sowie je einen Pointer auf das nächste Element in jeder Dimension.
2. das sortierte Analogon zu Nr. 1.
3. Die unsortierte, in zwei Dimensionen doppelt verkettete Listenstruktur hält zusätzlich zu Nr. 1 zu jedem workarray–Eintrag noch einen Pointer auf das vorangehende Element in jeder Dimension.
4. das sortierte Analogon zu Nr. 3.

Indirekte Feldzugriffe sind für gepipelinete (Knoten–)Prozessoren besser geeignet [SBV93], während sich Pointer eher bei RISC–Architekturen anbieten. Die (halb-)automatische Parallelisierung indirekter Feldzugriffe ist besser erforscht (siehe Abschnitt 3.4.3), und die Datenaufteilung kann zumindest teilweise zur Übersetzungszeit erfolgen.

Da verschiedene Datenstrukturen jeweils verschiedene Sätze von Mustern erfordern, haben wir uns auf wenige Datenstrukturen auf der Basis indirekter Feldzugriffe entschieden, und dabei insbesondere die Varianten Nr. 1 und 2.

D.2 Muster für Operationen mit indirekten Feldzugriffen

In Analogie zur den Mustern aus Abschnitt 5.3 entwerfen wir folgende Muster für Operationen mit indirekten Feldzugriffen:

Quasiskalare Vektoroperationen

```
VGATHER ( i, x, y, ix )
Sem.:  DO i  x[i] = y[ix[i]]
Slot 0: I rng       i           Schleifenvariable, Schleifengrenzen
Slot 1: W vector    x           issimpleidx(x, i)
Slot 2: R vector    y
Slot 3: R vector    ix          issimpleidx(ix, i)


VSCATTER ( i, x, ix, y )
Sem.:  DO i  x[ix[i]] = y[i]
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix, i)
Slot 3: R vector    y           issimpleidx(y, i)


VXASSIGN ( i, x, ix, v )
Sem.:  DO i  x[ix[i]] = v
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix, i)
Slot 3: R scalarvar v           notoccurin(v, i), isvar(v)


VXASSIGNSP ( i, x, ix, s, ... )
Sem.:  DO i  x[ix[i]] = s
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix, i)
Slot 3: I expr      s           notoccuridx(s, i)
Slot 4: R scalarvar v1          erste Variable in s
.... .  . ..        ..          ....


VXINIT ( i, x, ix, c )
Sem.:  DO i  x[ix[i]] = c
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix,i)
Slot 3: R ival/dval c           isconst(s,i)


VXINITSP ( i, x, s )
Sem.:  DO i  x[ix[i]] = s
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix,i)
Slot 3: I expr      s           notoccuridx(s,i), not(isconst(x))
Slot 4: R scalarvar v1          erste Variable in s
....
```

```
VXAADDSS ( i, x, ix, x, u, v )
Sem.:  DO i  x[ix[i]] = x[ix[i]] +- u * v
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix,i)
Slot 3: R vector    x           eqfex(x auf lhs, x auf rhs)
Slot 4: R scalar    u           notoccurin(u,i)
Slot 5: R scalar    v           notoccurin(v,i)
```

```
VXAADDSV ( i, x, ix, x, u, v )
Sem.:  DO i  x[ix[i]] = x[ix[i]] +- u * v[i]
Slot 0: I rng       i
Slot 1: W vector    x
Slot 2: R vector    ix          issimpleidx(ix,i)
Slot 3: R vector    x           eqfex(x auf lhs, x auf rhs)
Slot 4: R scalar    u           notoccurin(u,i)
Slot 5: R vector    v           issimpleidx(v,i)
```

Weitere quasiskalare Operationen mit indirekten Feldzugriffen können in eine Folge aus den beschriebenen Operationen und Mustern mit direkter Indizierung zerlegt werden.

Indirekte 1D–Reduktionen

```
VXSUM ( i, s, y, ix, c )
Sem.:  ‘s = c;’ DO i  s = s + ‘abs’y[ix[i]]
Slot 0: I rng       i
Slot 1: W ivar/dvar s
Slot 2: R vector    y
Slot 3: R vector    ix         issimpleidx(ix, i)
Slot 4: R scalar    c          Init-Wert oder s, falls uninitialisiert
```

`VXPROD ( i, s, y, ix, c )` analog

```
SSXP ( i, x, y, z, iz, c )
Sem.:  ‘s = c;’ DO i  s = s + y[i] * z[iz[i]]
Slot 0: I rng       i
Slot 1: W ivar/dvar s
Slot 2: R vector    y
Slot 3: R vector    z
Slot 4: R vector    iz         issimpleidx(iz, i)
Slot 5: R scalar    c          Init-Wert oder s falls uninitialisiert
```

```
SXSXP ( i, x, y, iy, z, iz, c )
Sem.:  ‘s = c;’ DO i  s = s + y[iy[i]] * z[iz[i]]
Slot 0: I rng       i
Slot 1: W ivar/dvar s
Slot 2: R vector    y
Slot 3: R vector    iy         issimpleidx(iy, i)
Slot 4: R vector    z
Slot 5: R vector    iz         issimpleidx(iz, i)
Slot 6: R scalar    c          Init-Wert oder s falls uninitialisiert
```

```
VQXSUM ( i, s, y, c )
Sem.:  ‘s = c;’ DO i  s = s + y[iy[i]] * y[iy[i]]
Slot 0: I rng       i
Slot 1: W ivar/dvar s
Slot 2: R vector    y
Slot 3: R vector    iy         issimpleidx(iy, i)
Slot 4: R scalar    c          Init-Wert oder s falls uninitialisiert
```

Indirekte 1D–Maximierungen/Minimierungen

```
VXMAXVAL ( i, s, x, ix, c )
Sem.:  ‘s = c;’  DO i  if (s>{=}x[ix[i]])  s = x[ix[i]];
Slot 0: I rng       i
Slot 1: W ivar/dvar s          notoccurin(s,i)
Slot 2: R vector    x
Slot 3: R vector    ix         issimpleidx(ix, i)
Slot 4: R scalar    c          Init-Wert oder s falls uninitialisiert
```

`VXMININVAL` analog

```
VXMAXLOC ( i, k, x, ix, t )
Sem.:  ‘k = t;’  DO i  if (x[k] < x[i])  k = i;
Slot 0: I rng       i
Slot 1: W ivar      k          notoccurin(k,i)
Slot 2: R vector    x
Slot 3: R vector    ix         issimpleidx(ix, i)
Slot 4: R ivar/ival t          k falls uninitialisiert
```

`VXMINLOC` analog

```
VXMAXVL ( i, k, x, ix, s, c, t )
Sem.:  ‘k=t;’ ‘s=c;’  DO i  if (s < x[i]) { k = i; s = x[i] }
Slot 0: I rng       i
Slot 1: W ivar      k          notoccurin(k,i)
Slot 2: R vector    x
Slot 3: R vector    ix         issimpleidx(ix, i)
Slot 4: W ivar/dvar s          notoccurin(s,i)
Slot 5: R scalar    c          s falls uninitialisiert
Slot 6: R ivar/ival t          k falls uninitialisiert
```

`VXMINVL` analog

Matrix–Vektor–Multiplikation bei dünnbesetzter Matrix

```
MXV ( i, j, a, b, C, ix, c )
Sem.:  ‘DO i  a[i]=c;’
       DO i DO j[1:grad[i]] { a[i] = a[i] +- b[col[i][j]] * C[i][j]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W vector    a          (lhs) issimpleidx(a,i), notoccurin(a,j)
Slot 3: R vector    b          issimpleidx(b,i), notoccurin(b,j)
Slot 4: R vector    C
Slot 5: R matrix    ix         issimpleidx(ix,i), issimpleidx(ix,j),
Slot 5: R matrix    ib         issimpleidx(ix,i), issimpleidx(ix,j),
Slot 6: R expr      c          Init-Wert bzw. -Vektor
```

SPARSE-BLAS–Routine	entsprechendes Muster
DAXPYI	VXAADDSV(1)
DDOTI	SSXP(1)
DGTHR	VGATHER(1)
DGTHRZ	VGATHER(1) + VXINIT(1)(nach Schleifenaufgliederung)
DROTI	—
DSCT	VSCATTER(1)

Tabelle D.1 Die `SPARSEBLAS`–Routinen und die ihnen entsprechenden Muster mit indirekten Feldzugriffen

```
MXVX ( i, j, a, ia, b, C, ix, c )
Sem.:  ‘DO i  a[i] = c;’
       DODO i,j  a[ia[i]] = a[ia[i]] +- b[i] * C[ix[i][j]]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W vector    a          (lhs)
Slot 3: R vector    ia         issimpleidx(ia,i), notoccurin(ia,j)
Slot 4: R vector    b          issimpleidx(b,i), notoccurin(b,j)
Slot 5: R vector    C
Slot 6: R matrix    ix         issimpleidx(ix,i), issimpleidx(ix,j),
Slot 7: R expr      c          Init-Wert bzw. -Vektor oder a (rhs)
```

Indirekte 2D–Reduktion

```
MXSUM ( i, j, s, y, ix, c )
Sem.:  ‘s = c;’  DO i DO j[1:grad[i]]  s = s + ‘abs’y[ix[i][j]]
Slot 0: I rng       i
Slot 1: I rng       j
Slot 2: W ivar/dvar s
Slot 3: R vector    y
Slot 4: R matrix    ix         issimpleidx(ix,i), issimpleidx(ix,j)
Slot 5: R expr      c          Init-Wert bzw. -Vektor oder s (rhs)
```

D.3 Analyse der SPARSE-BLAS–Routinen

Tabelle D.1 enthält eine Aufstellung der Routinen aus `SPARSEBLAS`, der BLAS–Variante für dünnbesetzte Matrizen, die über `netlib` verfügbar ist.

Literaturverzeichnis

[AC75] A.V. Aho and M.J. Corasick. Efficient String Matching: An aid to bibliographic search. *Journal of the ACM*, 18(6):333–340, June 1975.

[ACK87] J.R. Allen, David Callahan, and Ken Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. In *ACM SIGPLAN Principles of Programming Languages*, pages 63–76, 1987.

[AG85] Alfred V. Aho and Mahadevan Ganapathi. Efficient Tree Pattern Matching: an Aid to Code Generation. In *ACM SIGPLAN Principles of Programming Languages*, pages 334–340, 1985.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[AH90] Zahira Ammarguellat and W.L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *ACM SIGPLAN Programming Language Design and Implementation*, 1990.

[AJ76] A.V. Aho and S.C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[AK84] J.R. Allen and Ken Kennedy. Automatic Loop Interchange. *SIGPLAN Notices*, 19(6):233–246, June 1984. ACM SIGPLAN '84 Symposium on Compiler Construction.

[AK87] J.R. Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[AKLS88] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Compiling Fortran 8x Array Features for the Connection Machine Computer Systems. In *ACM Symposium on Parallel Programming (PPEALS)*, pages 42–56, 1988.

[AKP90] F. Abolhassan, J. Keller, and W.J. Paul. On Physical Realizations of the Theoretical PRAM Model. Technical Report 21/1990, Sonderforschungsbereich 124 VLSI Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1990.

[APT90] F. Andre, J.-L. Pazat, and H. Thomas. PANDORE: a system to manage data distribution. In *Int. Conference on Supercomputing*, pages 380–388, June 1990.

[ASU86] Alfred S. Aho, Ravi Sethi, and Jeffrey D. Ullman. *COMPILERS: Principles, Techniques, and Tools.* Addison–Wesley, 1986.

[ASU88] Alfred S. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau.* Addison–Wesley, 1988. Englische Originalausgabe: [ASU86].

[Bab91] Marc Baber. Hypertasking: Automatic Array and Loop Partitioning on the iPSC. In *24th Annual Hawaii Int. Conference on System Sciences*, pages 438–447, 1991.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[BBC+93] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[BC86] Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization. In *SIGPLAN Symposium on Compiler Construction*, 1986.

[Ben86] J. Bentley. *Programming Pearls.* Addison–Wesley, Reading, MA, 1986.

[Ber92] Michael Berry (Editor). Scientific Workload Characterization by Loop Based Analyses. *Performance Evaluation Review*, 19:17–29, February 1992.

[BFKK91] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, volume 3, pages 213–223, 1991.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. Steele Jr. Fortran at ten gigaflops: The connection machine convolution compiler. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 145–156, 1991.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 41–53, 1989.

[BKK93] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic Data Layout Using 0–1 Integer Programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1993.

[BKM90] S. Burson, G. Kotik, and L. Markosian. A Program Transformation Approach to Automating Software Re-engineering. In *14th IEEE Int. Computer Software and Applications Conference (COMPSAC), Chicago*, pages 314–322, 1990.

[BKN85] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of Matching Problems. In J.P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 417–429. Springer LNCS vol. 202, 1985.

[BL92] Ralph Butler and Ewing Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

[Ble90] Guy Blelloch. *Vector Models for Data–Parallel Computing.* MIT press, 1990.

[BMW91] Jürgen Börstler, Ulrich Möncke, and Reinhard Wilhelm. Table Compression for Tree Automata. *ACM Transactions on Programming Languages and Systems*, 13(3), July 1991.

[Bok90] S. Bokhari. Communication Overhead on the Intel iPSC/860 Hypercube. Technical Report Interim Report 10, ICASE, May 1990.

[Bos88a] Pradip Bose. Heuristic Rule-Based Program Transformations for Enhanced Vectorization. In *Proc. of Int. Conf. on Parallel Processing*, 1988.

[Bos88b] Pradip Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. In *Proc. Int. Conf. on Supercomputing*, pages 119–130, July 1988.

[BPS92] L.C. Breebaart, E.M. Paalvast, and H.J. Sips. A Rule Based Transformation System for Parallel Languages. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[Bra88] Thomas Brandes. *Formale Methoden zur Spezifizierung automatischer Parallelisierung.* Hüthig Verlag Heidelberg, 1988.

[Bre93] Leo Breebaart. Experiences with Rule-based Compilation. In *Fourth Workshop on Compilers for Parallel Computers*, pages 468–475, Dec. 1993.

[BS87] Thomas Brandes and Manfred Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. In *16th Int. Conf. on Parallel Processing*, pages 446–448, 1987.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods.* Prentice-Hall, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *ACM SIGPLAN Principles of Programming Languages*, Jan. 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM SIGPLAN Principles of Programming Languages*, pages 269–282, Jan. 1979.

[CC92] Ian A. Carmichael and James R. Cordy. *TXL - Tree Transformational Language Syntax and Informal Semantics.* Dept. of Computing and Information Science, Queen's University at Kingston, Canada, February 1992.

[CC93] Ian A. Carmichael and James R. Cordy. *The TXL Programming Language Syntax and Informal Semantics Version 7.* Dept. of Computing and Information Science, Queen's University at Kingston, Canada, June 1993.

[CDPW92] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[CFZ93] Barbara M. Chapman, Thomas Fahringer, and Hans P. Zima. Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon.* Springer LNCS, Aug. 1993.

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland. Birkhäuser*, pages 21–1–6, April 1994.

[CGL+93] Siddhartha Chatterjee, John R. Gilbert, Fred J.E. Long, Robert Schreiber, and Shang-Hua Teng. Generating Local Addresses and Communication Sets for Data-Parallel Programs. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 149–158, 1993.

[CGMS94] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–656, April 1994.

[Cha87] D.R. Chase. An improvement to bottom-up tree pattern matching. In *ACM SIGPLAN Principles of Programming Languages*, pages 168–177, 1987.

[CHHW94] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994.

[CHZ91a] Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Automatic Support for Data Distribution. Technical Report ACPC/TR 91-14, Austrian Center for Parallel Computation, July 1991.

[CHZ91b] Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Knowledge-based Parallelization for Distributed Memory Systems. Technical Report ACPC/TR 91-11, Austrian Center for Parallel Computation, April 1991.

[CK87] David Callahan and Ken Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. In *First Int. Conf. on Supercomputing, Athen, Greece, Springer LNCS vol. 297*, June 1987.

[CK88] David Callahan and Ken Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[CKKM94] Alan Carle, Ken Kennedy, Ulrich Kremer, and John Mellor–Crummey. Automatic Data Layout for Distributed Memory Machines in the D Programming Environment. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 136–152. Verlag Vieweg, 1994.

[CM84] W.F. Clocksin and C.S. Mellish. *Programming in PROLOG*. Springer, 1984.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[CRT87] M. Cosnard, Y. Robert, and M. Tchuente. Matching Parallel Algorithms with Architectures: A Case Study. In G.L. Reijns and M.H. Barton, editors, *Highly Parallel Computers*, pages 127–143. Elsevier Science Publishers, 1987.

[CS88] Vladimir Cherkassky and Ross Smith. Efficient Mapping and Implementation of Matrix Algorithms on a Hypercube. *The Journal of Supercomputing*, 2:7–27, 1988.

[Cyt86] Ron Cytron. Doacross: beyond vectorization for multiprocessors. In *Int. Conference on Parallel Processing*, pages 226–234, 1986.

[DDHH88] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. on Math. Software*, 14(1):1–32, 1988.

[DHR94] Anne Dierstein, Roman Hayer, and Thomas Rauber. Automatic parallelization for distributed memory multiprocessors. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 192–217. Verlag Vieweg, 1994.

[Die93] Anne Dierstein. Parallelisierung mit automatischer Datenaufteilung für imperative Programmiersprachen, Teil 1: Parallelisierungsstrategie. Diplomarbeit, Universität des Saarlandes, Germany, 1993.

[Don79] Jack Dongarra et al. *LINPACK User's Guide*. Philadelphia, 1979.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. In *Proc. of 4th Int. Workshop on Languages and Compilers for Parallel Computing*, pages 65–83. Pitman/MIT press, Aug. 1991.

[ER88] G. Engeln-Müllges and F. Reutter. *Formelsammlung zur numerischen Mathematik mit Standard-FORTRAN 77-Programmen*. Mannheim, 1988.

[ESH93] Kalluri Eswar, P. Sadayappan, and Chua-Huang Huang. Compile-Time Characterization of Recurrent Patterns in Irregular Computations. In *Int. Conference on Parallel Processing*, pages II–148–155, 1993.

[Fah93] Thomas Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Universität Wien, 1993.

[Fah94] Thomas Fahringer. The Weight Finder, an Advanced Profiler for Fortran Programs. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 1–24. Verlag Vieweg, 1994.

[Fea91] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *Int. Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[Fer90] Christian Ferdinand. Pattern Matching in TRAFOLA. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1990.

[Fer94] Christian Ferdinand. private communication, 1994.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, vol. 1: General Techniques and Regular Problems.* Prentice–Hall, 1988.

[FMPB94] Arno Formella, Silvia Müller, Wolfgang Paul, and Anke Bingert. Isolating the Reasons for the Performance of Parallel Machines on Numerical Programs. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 34–64. Verlag Vieweg, 1994.

[FOP+92] A. Formella, A. Obe, W. Paul, T. Rauber, and D. Schmidt. The SPARK 2.0 system – a special purpose vector processor with a VectorPASCAL compiler. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences, vol. 1*, pages 547–558, 1992.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FP92] T.L. Freeman and C. Philips. *Parallel Numerical Algorithms.* Prentice Hall, 1992.

[Fri91] Stephen S. Fried. Personal Supercomputing with the Intel i860. *BYTE*, pages 347–357, Jan. 1991.

[FSW92] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree Automata for Code Selection. In *Code Generation – Concepts, Tools, Techniques*, pages 30–50, 1992.

[GB90] Manish Gupta and Prithviraj Banerjee. Automatic Data Partitioning on Distributed Memory Multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Oct. 1990.

[Ger89] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Universität Bonn, 1989.

[Ger90] Michael Gerndt. Parallelization of Multigrid Programs in SUPERB. Technical Report ACPC/TR 90-6, Austrian Center for Parallel Computation, October 1990.

[Ger93] Michael Gerndt. Methoden der automatischen Parallelisierung für massiv parallele Rechner. In W. Rönsch and J. Schüle, editors, *Parallelisierung im Wissenschaftlichen Rechnen, Informatik-Bericht 93-04, TU Braunschweig*, pages 23–30, June 1993.

[GG78] R.S. Glanville and S.L. Graham. A New Method for Compiler Code Generation. In *ACM SIGPLAN Principles of Programming Languages*, pages 231–240, Jan. 1978.

[GHN+90] K.A. Gallivan, M.T. Heath, E. Ng, J.M. Ortega, B.W. Peyton, R.J. Plemmons, C.H. Romine, A.H. Sameh, and R.G. Voigt. *Parallel Algorithms for Matrix Computations.* SIAM, Philadelphia, 1990.

[Gie92] Robert Giegerich. Considerate Code Selection. In *Code Generation – Concepts, Tools, Techniques*, pages 51–65, 1992.

[GKM+93] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.H. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on Distributed Memory Machines. In *Int. Conference on Parallel Processing*, pages II–301–305, 1993.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 15–29, 1991.

[Got92] Allan Gottlieb. Architectures for Parallel Supercomputing. Usenet comp.parallel, Oct. 1992.

[Gro92] Josef Grosch. Transformation of Attributed Trees using Pattern Matching. In U. Kastens and P. Pfahler, editors, *Fourth Int. Conf. on Compiler Construction (CC'92), Springer LNCS vol. 641*, pages 1–15, Oct. 1992.

[GS90] Thomas Gross and Peter Steenkiste. Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler. *Software – Practice and Experience*, 20(2):133–155, Feb. 1990.

[Gup92] Manish Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Technical Report UILU-ENG-92-2237 or CRHC-92-19, 1992.

[GV91] Elana D. Granston and Alexander D. Veidenbaum. Detecting Redundant Accesses to Array Data. In *Proc. Supercomputing '91, Albuquerque. An extended version is available as Technical Report No. CSRD-TR1053 from the Center for Supercomputing Research and Development at the University of Illinois at Urbana–Champaign.*, pages 854–865, Nov. 1991.

[HA90] David E. Hudak and Santosh G. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Loops. In *Int. ACM Conference on Supercomputing*, pages 187–200, 1990.

[Hä94] Rolf Hänisch. Snap! Prototyping a Sequential and Numerical Application Parallelizer. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 83–88. Verlag Vieweg, 1994.

[HACZ93] Jan Hulman, Stefan Andel, Barbara M. Chapman, and Hans P. Zima. Intelligent Parallelization within the Vienna Fortran Compilation System. In *Fourth Workshop on Compilers for Parallel Computers*, pages 455–467, Dec. 1993.

[Han87] M. Hanus. *Problemlösen mit PROLOG.* B.G. Teubner Verlag, Stuttgart, 1987.

[Hay93] Roman Hayer. Automatische Parallelisierung, Teil 2: Automatische Datenaufteilung für Parallelrechner mit verteiltem Speicher. Master thesis, Universität Saarbrücken, 1993.

[HB84] Kay Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[HHL90] Lorenz Huelsbergen, Douglas Hahn, and James Larus. Exact Data Dependence Analysis Using Data Access Descriptors. Technical Report # 945, University of Wisconsin-Madison, Computer Science Department, 1990.

[Hin92] Didier Y. Hinz. A run-time load balancing strategy for highly parallel systems. *Acta Informatica*, 29:63–94, 1992.

[HK91] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):350–359, July 1991.

[HKS94] Rainer W. Hartenstein, Christoph W. Keßler, and Karin Schmidt. Knowledge–Based Compilation and Restructuring Transformations for Xputers, submitted (1994).

[HKT91a] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler–Support for Machine–Independent Parallel Programming in Fortran-D. Technical Report Rice COMP TR91-149, Rice University, March 1991.

[HKT91b] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler Optimizations for Fortran D on MIMD Distributed Memory Machines. In *Int. ACM Conference on Supercomputing*, pages 86–100, Nov. 1991.

[HN90a] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, January 1990.

[HN90b] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[HN93] Paul D. Hovland and Lionel M. Ni. A Model for Automatic Data Partitioning. In *Int. Conference on Parallel Processing*, pages II–251–259, 1993.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM*, 29(1):68–95, Jan. 1982.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification, Final Version 1.0, May 1993.

[HRC+90] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C. Karathanasis, P.N. Papachiou, M.K. Samartzis, E.A. Vavalis, Ko Yang Wang, and S. Weerawarana. //ellpack: A numerical simulation programming environment for parallel mimd machines. In *Int. Conference on Supercomputing*, pages 96–107, 1990.

[HS93] Reinhold Heckmann and Georg Sander. TrafoLa-H Reference Manual. In B. Hoffmann and B. Krieg-Brückner, editors, *Program Development by Specification and Transformation: The PROSPECTRA Methodology, Language Family, and System*, pages 275–313. Springer LNCS Vol. 680, 1993.

[IFKF90] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. In *Fifth Distr. Memory Computing Conf. (DMCC5), IEEE Computer Society Press*, pages 1105–1114, 1990.

[IMS] IMSL. User's manual. Customer Relations, 2500 Park West Tower One, 2500 City West Boulevard, Houston, TX 77042-3020, USA.

[INT90] INTEL Corp. iPSC/2 and iPSC/860 Programmer's Reference Manual, June 1990.

[Ive62] Kenneth E. Iverson. *A Programming Language.* John Wiley & sons, 1962.

[KD94] Kathleen Knobe and William J. Dally. Subspace Optimizations. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 153–176. Verlag Vieweg, 1994.

[Ken81] Ken Kennedy. A Survey of Data Flow Analysis Techniques. in [MJ81], 1981.

[Keß90] C.W. Keßler. Code–Optimierung quasiskalarer vektorieller Grundblöcke. Diplomarbeit, Universität des Saarlandes, 1990.

[Keß93] Christoph W. Keßler. Pattern Recognition Enables Automatic Parallelization of Numerical Codes. In *Proc. of the Fourth Int. Workshop on Compilers for Parallel Computers, Delft, NL*, Dec. 1993.

[Keß94a] Christoph W. Keßler. Knowledge–Based Automatic Parallelization by Pattern Recognition. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 110–135. Verlag Vieweg, 1994.

[Keß94b] Christoph W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, Basel, Switzerland, April 1994.

[KKP+81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *ACM*, 1981.

[KKP91] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KMR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 177–186, 1990.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD machines. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 416–423, 1990.

[KNE93] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12), Dec. 1993.

[KNS92] Wojtek Kozaczynski, Jim Ning, and Tom Sarver. Program concept recognition. In *Proc. of KBSE'92 Seventh Knowledge-Based Software Engineering Conference*, pages 216–225, 1992.

[Köc90] Norbert Köckler. *Numerische Algorithmen in Softwaresystemen.* Teubner–Verlag Stuttgart, 1990.

[KP93] Christoph W. Keßler and Wolfgang J. Paul. Automatic Parallelization by Pattern Matching. In *Proc. of Second Int. Conference of the Austrian Center for Parallel Computation, Springer LNCS 734*, pages 166–181, Oct. 1993.

[KPR91] C.W. Keßler, W.J. Paul, and T. Rauber. A Randomized Heuristic Approach to Register Allocation. In *Proceedings of the Third Symposium on Programming Language Implementation and Logic Programming (PLILP91), Passau (Germany)*, pages 195–206. Springer LNCS vol. 528, 1991.

[KPR92] C.W. Keßler, W.J. Paul, and T. Rauber. Scheduling Vector Straight Line Code on Vector Processors. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 77–91. Springer Workshops in Computing Series, 1992.

[KR93] C.W. Keßler and T. Rauber. Efficient register allocation for large basic blocks. In *Proceedings of the Fifth Symposium on Programming Language Implementation and Logic Programming (PLILP93), Tallinn (Estonia)*, volume Springer LNCS Vol. 714, pages 418–419, 1993.

[Kre93a] Ulrich Kremer. Automatic Data Layout for Distributed Memory Machines. Technical Report CRPC-TR93299, Center for Research on Parallel Computation, Rice University, Houston, TX, Feb. 1993.

[Kre93b] Ulrich Kremer. NP-Completeness of Dynamic Remapping. Technical Report CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Aug. 1993. see also: Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, Dec. 1993.

[Kro75] H.H. Kron. *Tree Templates and Subtree Transformational Grammars.* PhD thesis, UC Santa Cruz, Dec. 1975.

[KS73] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[Kuc] Kuck Associates and Partners, Inc. KAP User's Guide.

[LC90] Jingke Li and Marina Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[LC91] Jingke Li and Marina Chen. Compiling Communication–Efficient Programs for Massively Parallel Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. on Math. Software*, 5:308–325, 1979.

[LMW88] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. Optran: A Language/System for the Specification of Program Transformations: System Overview and Experiments. In D. Hammer, editor, *Compiler Compilers and High–Speed Compilation*, pages 52–65. Springer LNCS vol. 371, 1988.

[Lou92] A. K. Louis. Parallele Numerik. Course script and selected programs, unpublished, Universität Saarbrücken, 1992.

[Lov77] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, Jan. 1977.

[LT93] PeiZong Lee and Tzung-Bow Tsai. Compiling Efficient Programs for Tightly-Coupled Distributed Memory Computers. In *Int. Conference on Parallel Processing*, pages II–161–165, 1993.

[LYZ90] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[LZ93] Shun-tak Leung and John Zahorjan. Improving the Performance of Runtime Parallelization. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 83–91, 1993.

[Mac87] Mary E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, 1987.

[Mac94] Neil B. Macdonald. Predicting the Execution Time of Sequential Scientific Codes. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 25–33. Verlag Vieweg, 1994.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM SIGPLAN Principles of Programming Languages*, Jan. 1993.

[McM86] Frank McMahon. The Livermore Fortran Kernels: A Test of the Numeric Performance Range. Technical report, Lawrence Livermore National Laboratory, 1986.

[Meh86] Kurt Mehlhorn. *Datenstrukturen und Effiziente Algorithmen*. Teubner–Verlag, 1986.

[MFL+92] A. Mohamed, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Application Benchmark Set for Fortran D and High Performance Fortran. Technical Report 327, Northeast Parallel Architectures Center, 1992.

[MJ81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[Mol] C.B. Moler. Matlab – User's Guide. The Math Works Inc. 21 Eliot Street, South Natick, MA 01760, USA.

[MR87] P. Mehrotra and J. Van Rosendale. The BLAZE Language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[Mül89] Silvia M. Müller. Die Auswirkung der Startup-Zeit auf die Leistung paralleler Rechner bei numerischen Anwendungen. Master thesis, Universität Saarbrücken, 1989.

[NAG] NAG. Fortran library. The Numerical Algorithm Group Ltd., Mayfield House, 256 Banbury Road, Oxford OX2 7DE, UK.

[Paa92] Edwin M. Paalvast. *Programming for Parallelism and Compiling for Efficiency*. PhD thesis, Technische Universiteit Delft, June 1992.

[Par] ParaSoft Corp. The EXPRESS Parallel Toolkit. 2500 E. Foothill Blvd., Suite 205, Pasadena, CA 91107.

[PB85] Paul Walton Purdom Jr. and Cynthia A. Brown. Fast Many-To-One Matching Algorithms. In J.P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 407–416. Springer LNCS vol. 202, 1985.

[Pet93] Paul M. Petersen. *Evaluation of Programs and Parallelizing Compilers using Dynamic Analysis Techniques*. PhD thesis, University of Illinois at Urbana-Champaign, CSRD-Report 1273, 1993.

[PG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. In *ACM SIGPLAN Principles of Programming Languages*, pages 294–308, 1988.

[PM94] Michael Philippsen and Markus U. Mock. Data and Process Alignment in Modula-2*. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 177–191. Verlag Vieweg, 1994.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[PP91] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In *ACM SIGPLAN Principles of Programming Languages*, pages 79–92, 1991.

[PS92] John Palmer and Guy L. Steele jr. Connection Machine Model CM-5 System Overview. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 474–483, 1992.

[PSvG91] Edwin M. Paalvast, Henk J. Sips, and A.J. van Gemund. Automatic Parallel Program Generation and Optimization from Data Decompositions. In *Int. Conference on Parallel Processing*, pages II–124–131, 1991.

[PT91] Michael Philippsen and Walter F. Tichy. Compiling for Massively Parallel Machines. In *Code Generation – Concepts, Tools, Techniques*, pages 92–111. Springer Workshops in Computing, 1991.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition*. Cambridge University Press, 1992.

[PW86] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[PW93] William Pugh and David Wonnacott. An Exact Method for Analysis of Value-based Array Data Dependences. Technical Report CS-TR-3196, Dept. of Computer Science, University of Maryland, 1993.

[RF93] Xavier Redon and Paul Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *PARLE 93, Springer LNCS vol. 694*, pages 132–145, 1993.

[RP89] Anne Rogers and Keshav Pingali. Process Decomposition Through Locality of Reference. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 69–89, 1989.

[RS87] J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.

[RS93] Hubert Ritzdorf and Klaus Stüben. Adaptive multigrid on distributed memory computers. Technical Report GMD Arbeitspapier 781, GMD, Sept. 1993.

[RSW91] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[RW90] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph–Parsing Approach. *IEEE Software*, pages 82–89, Jan. 1990.

[Sab92a] Gary Sabot. A compiler for a massively parallel distributed memory mimd computer. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[Sab92b] Gary W. Sabot. Optimized CM Fortran Compiler for the Connection Machine Computer. In *Proceedings of Hawaii Int. Conf. on System Sciences, HICSS-25*, pages 161–172. IEEE Computer Society, 1992.

[SBV93] A. Frank van der Stappen, Rob H. Bisseling, and Johannes G.G. van de Vorst. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Runtime Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[SCSS92] C.D. Scarbnick, M.C. Chang, M.H. Schultz, and A.B. Sherman. A Parallel Software Package for Solving Linear Systems. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 397–401, 1992.

[SH91] Jaswinder Pal Singh and John L. Hennessy. An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization. In *Int. Symposium on Shared Memory Multiprocessing, Tokyo, Japan*, pages 25–36, April 1991.

[Sho81] Robert Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the ACM*, 28(4):769–779, Oct. 1981.

[SLY90] Z. Shen, Z. Li, and P. Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Trans. Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Smi76] B.T. Smith et al. Matrix Eigensystem Routines – EISPACK Guide, 1976.

[Sny82] Lawrence Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[ST82] Klaus Stüben and Ulrich Trottenberg. Multigrid methods: Fundamental algorithms, model problem analysis and applications. In *Springer Lecture Notes in Mathematics, Vol. 960*, 1982.

[Sta94] Stanford Compiler Group. SUIF Compiler System, version 1.0: The SUIF Parallelizing Compiler Guide. accessible by anonymous ftp from `suif.stanford.edu` in `/pub/suif`, 1994.

[SW93] Gary Sabot and Skef Wholey. Cmax: a Fortran Translator for the Connection Machine System. In *Int. ACM Conference on Supercomputing*, pages 147–156, 1993.

[SWW91] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and Data decomposition. In *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, 1991.

[Tar72] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[Thi92] Thinking Machines Corp. CMSSL for CM Fortran, Quick Reference Guide, V. 3.0, 1992.

[Thi93] Thinking Machines Corp. The CMSSL Library (V. 3.1: Optimizing Numeric Operations Across High Volume Microprocessors, 1993.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct Parallelization of CALL Statements. In *SIGPLAN Symposium on Compiler Construction*, pages 176–185, July 1986.

[TP93] Peng Tu and David Padua. Array Privatization for Shared and Distributed Memory Machines (Extended Abstract). *SIGPLAN Notices*, 28(1):64–67, 1993.

[Tre94] Joachim Trescher. private communication on the integration of PARAMATs pattern library into the booster/v-cal compiler, March 1994.

[TW68] J.W. Thatcher and J.B. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second–Order Logic. *Mathematical Systems Theory*, 2(1), 1968.

[TW91] Arthur Trew and Greg Wilson (Eds.). *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. Springer, 1991.

[Ull84] Jeffrey D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[Wan92] Ko-Yang Wang. Knowledge–Engineering Issues in Intelligent Parallel Compilers. In *Third Workshop on Compilers for Parallel Computers*, pages 1–12, July 1992.

[WG89] Ko-Yang Wang and Dennis Gannon. Applying AI Techniques to Program Optimization for Parallel Computers. In Kai Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, pages 441–485, 1989.

[Who91] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[Wil79] Reinhard Wilhelm. Computation and Use of Data Flow Information in Optimizing Compilers. *Acta Informatica*, 12:209–225, 1979.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 30–44, 1991.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, 1992.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.

[Wol91] Michael Wolfe. The Tiny Loop Restructuring Research Tool, 1991.

[WS90] Debbie Whitfield and Mary Lou Soffa. An Approach to Ordering Optimizing Transformations. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 137–146, 1990.

[WT92] Michael Wolfe and Chau-Wen Tseng. The Power Test for Data Dependence. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):591–601, Sept. 1992.

[WW88] Beatrix Weisgerber and Reinhard Wilhelm. Two Tree Pattern Matchers for Code Generation. In *Springer LNCS vol. 371*, pages 215–229, 1988.

[Zap94] Emilio L. Zapata. private communication on the extension of PARAMAT to sparse matrix operations, Jan. 1994.

[ZBG88] Hans Zima, Heinz Bast, and Michael Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison–Wesley, 1990.

Index

0-1 integer programming, 32

AADD, 53, 132
AADDMUL, 53, 133
AADDMULTIMUL, 54, 134
Abhängigkeitsgleichung, 14
Abhängigkeitstests, 14
—linearisierte Variante, 15
Abrollen von Schleifen, 20
Absorption, 100
abstrakter Syntaxbaum, 9, 83
Abwickeln von Iterationen, 19
Achsenbündel, 31
Adaptierung, 24
ADD, 53, 132
ADDAP-System, 30
ADDMUL, 53, 133
ADDMULTIMUL, 53, 134
Adreßumsetzung, 26
Algorithmen-Ersetzung, 113–122
Alignment, 30
—-funktion, 31
—-konflikte, 31
Alphabet, 39
AMOD, 53, 132
AMUL, 53, 132
AND, 53, 132
ANTI cross edge, 97
Anti-Abhängigkeit, 12
Assoziativität, 46
Aufgliederung
—von bedingten Anweisungen, 18
—von Schleifen, 20
Aufgliederung bedingter Anweisungen, 105
Aufgliederung von Schleifen, 106
Aufrollen von Schleifen, 100
Aufteilungsfunktion, 29
Ausführungsfrequenz, 36
Ausführungsreihenfolge, 13
Ausgangs-Abhängigkeit, 12

Banerjee–Test, 14
Baumgrammatik, 43
Baumtransformationssysteme, 45
Bereichsausdruck, 9
BLAS-Routinen, 78
Blocken von Schleifen, 19, 119
BOOSTER, 126
bottom-up rewrite system, 44
branch-and-bound, 30
BSUBST, 69, 148
BURG, 44

C, 7, 116
Cache, 119
CG-Verfahren, 80, 159
CMSSL, 79
Code-Erzeugung, 113–124
Codeerzeugergenerator, 43
Comm(), 27, 35
CONDASS, 57, 135
constraints, 30
COS, 53, 133
cross edge, 97

dünnbesetzte Matrizen, 81, 179
Data Access Descriptor (DAD), 96
Datenabhängigkeit, 12
—schleifengetragen, 13
—schleifenunabhängig, 13
Datenabhängigkeitsgraph, 13
Datenaufteilung, 122
Datenaufteilungen, 28
—repliziert, 29
—zusammenhängend, 28
—zyklisch, 28
datenparallele Primitive, 31
Datentransfergraph, 36
DEF(), 12
Deskriptor, 90–96
Diagonale, 92
Differenzenstern, 54, 89
disjdescr, 130
disjdescr(), 94
DISJIN cross edge, 97
DISJOUT cross edge, 97
dist(), 29
DMS (Distributed-Memory-Parallelrechner), 2

EAVE, 124
echte Datenabhängigkeit, 12
EFLUX, 176
Eigen-Präferenz, 31
einhüllender Indexbereich, 91
EISPACK, 79
endlicher Baumautomat, 41, 44–45
ENORM, 63, 141
Entblocken von Schleifen, 88
EQ, 53, 132
eqdescr(), 94
eqex(), 10
eqfex(), 10
eqsym(), 10
eval(), 11
EXCH(), 25, 27
Exec(), 26
executor, 28
EXP, 53, 133

exprdescend, 99
exprdescend(), 85
exprtodescr(), 94

Feldachse, 30
Feldreferenz, 9
Feldzugriff, 8
—indirekt, 8
Feldzugriffsdeskriptor, 90–96
Flag-Bits, 9
FLOW cross edge, 97
Fluß-Abhängigkeit, 12
FOLR, 64, 143
for-Schleife, 8
FORTRAN 77, 7
Fortran 90, 77
FSUBST, 69, 148
FSUM, 57, 136

GAUSSSEIDEL, 74, 154
GE, 53, 132
Gebiet, 92
Gebietsanzeiger, 92
gebundene Schleifenvariable, 9
Gegendiagonale, 92
generate_code, 116
geqdescr, 129
geqdescr(), 94
geqex(), 11
ggT-Test, 14
Gitterhierarchie, 104, 117
GMOP, 67, 146
grand challenges, 1
Grundblock, 12
GT, 53, 132
gtex(), 11
GVOP, 60, 139
GVPROD, 63, 142
GVSUM, 63, 142

halbautomatische Parallelisierung, 4
hierarchischer Abhängigkeitstest, 15
High Performance Fortran, 121
HSTAR, 54, 89

IMSL, 78
Indexausdruck, 8
indirekter Feldzugriff, 179
indizierende Schleife, 9
Indizierung, 8
Induktionsvariable, 9
—Eliminierung, 16
Inferenz, 100
Inkarnation eines Musters, 49, 83
Inkarnationsschablone, 47
INPUT cross edge, 97
input-pattern, 43
inspector, 28
instabile Muster, 75, 108
Instanz eines Musters, 49
invex(), 11
isconst(), 10
isdoubleidx(), 11
isleaf(), 10
isoffset(), 11
isscalar(), 10
issimpleidx(), 11
isstandard(), 11
isvar(), 10
Iterationsvektor, 13

JACOBI, 74, 154
Jacobi-Relaxation, 172
Jacobi-Verfahren, 173

Kettenregel, 44
Kommunikationsanweisung höherer Ordnung, 33
Kommunikationsmenge, 27
Kommutativität, 46
Konform-Präferenz, 31
Konstantenpropagation, 16
Konstantensymbol, 39
Kontrollabhangigkeitsgraph, 12
Kontrollflußgraph, 111

LAPACK, 79
Lastbalance, 32, 35
LATINUS, 8
Laufzeitvorhersage, 34
—analytisch, 34
—synthetisch, 37, 123
LE, 53, 132
leqex(), 11
lineare Rekurrenz, 118
Linearisierung von Schleifen, 23
LINPACK, 79
Livermore Loops, 76, 111, 160
LOG, 53, 133
lokale Indexmenge, 27
lokale Iterationsmenge, 26
Lokalreferenziterationsmenge, 27
LT, 53, 132
ltex(), 11
LUD, 74, 153

M1CONV, 73, 153
MAADD, 66, 146
MAADDSM, 67, 146
MAADDSS, 67, 146
MAADDSVV, 68, 147
MAADDVV, 68, 147
MADD, 66, 145

MAINC, 67, 146
MAMUL, 66, 146
Maschinengrammatik, 43
Maske, 24
Masken-Slots, 76
maskierte Vektoroperation, 22
MASSIGN, 66, 145
MASSIGNSP, 66, 145
MASSIGNV, 145
match(), 40
Match-Set, 40
Match-Tabelle, 40
Matrix, 9
Matrix-Multiplikation, 120, 170
Matrix-Vektor-Multiplikation, 47
Matrixmultiplikation, 86
maxeval(), 93
MAXLOC, 56, 135
MAXLOCM, 56, 135
MAXVAL, 56, 135
MAXVL, 56, 135
MAXVLM, 57, 135
MCONDASS, 68, 146
MCONV, 74, 154
MCOPY, 65, 144
Mehrgitterverfahren, 104, 117
message-passing, 4
MGAUSSSEIDEL, 72, 90, 151
MIMD, 1
MINC, 66, 146
mineval(), 93
MINIT, 66, 145
MINITSP, 65, 145
MINLOC, 56, 135
MINLOCM, 56, 135
MINV, 67, 146
MINVAL, 56, 135
MINVL, 57, 135
MINVLM, 57, 135
MJACOBI, 72, 151
MLUD, 69, 147
MM, 73, 152
MMAXLOC, 71, 150
MMAXVAL, 71, 150
MMAXVL, 71, 151
MMINLOC, 71, 151
MMINVAL, 71, 150
MMINVL, 71, 151
MMUL, 66, 146
MOD, 53, 132
MPROD, 70, 149
MRAND, 75, 149
MROR, 75, 149
MSHIFT, 68, 147
MSUM, 70, 149
MUL, 53, 132
MULADDMUL, 54, 134
MULMUL, 54, 134
multex(), 11
MULTIADD, 54, 89
multigrid
—programs, 104
MULTIMUL, 54, 89
Muster, 49
Musterbibliothek, 50, 52–76
Mustererkennung, 50, 82–112
—auf Konzeptebene, 38
—auf semantischer Ebene, 38, 47
—auf syntaktischer Ebene, 38, 46
—auf Textebene, 38
—horizontal, 99
—vertikal, 83
Musterhierarchie
—horizontal, 157
—vertikal, 155
Musterhierarchiegraph, 84, 155–158
—vollständig, 84
MV, 69, 148
MXSUM, 184
MXV, 183
MXVX, 184

NAG, 78
NAS-Benchmark, 171
NE, 53, 132
negex(), 11
neighbdescr, 131
neighbdescr(), 95
network contention, 37
nichtlineares Treepattern, 39, 42, 46
NoParallel[](), 116
NoReplace[], 113
numerische Stabilität, 121
nutzloser Code, 18, 108

Obermuster, 84
occuridx(), 11
occurin(), 11
Operatorsymbol, 39
OPTRAN, 45
OR, 53, 132
Ordnungszahl ord(), 52, 84
output-pattern, 43
Overlap-Konzept, 26
owner–computes rule, 4
owner-computes-Regel, 24, 121

PALATINUS, 50
parallele Implementierung, 116
Parallelisierung
—für DMS (halbautomatisch), 24
—für indirekte Feldzugriffe, 27

—für SMS, 23
Partitionierung von Feldern, 28, 32
PARTOOL, 126
PAT, 111
PDG, 125
Phase, 32
pipelining, 1
plusex(), 11
POWER, 53, 133
Präferenzengraph, 31
PRAM, 2
precedes(), 11
PREVSUM, 63, 142
Programmabhängigkeitsgraph, 125
Programmtransformationen, 15
Prozedur-Expansion, 16
PUMA, 46
Purdue Benchmark Set, 76, 111

quasiskalar, 119
Querkante, 97

Realisierung einer Schablone, 82, 84
Reduktion, 43
Reduktionsregel, 43
Ref(), 27
referenzierte Indexmenge, 27
Referenzmuster, 34
replizierte Implementierung, 113
ReplSeq[], 113
Richtungsvektor, 15
ROT, 56, 134
Rumpfmuster, 84

S1CONV, 63, 142
S2CONV, 70, 149
ScaLAPACK, 79
Schablone, 84
Schablonen, 132–154
scheduling, 24
Schleifenblockung, 19
Schleifennormalisierung, 18
Schleifenverschmelzung, 21
Schleifenvertauschung, 20
SCOPY, 53, 132
semantische Äquivalenz, 47
semantische Hierarchie, 47
Separabilitätstest, 14
SeqDebug[], 113
Sequentialisierung, 36
sequentielle Implementierung, 113
Σ-Baum, 39
Σ-Term, 39
SIMD, 1
SIN, 53, 133
SINIT, 52, 132
Skalar, 9
Skalar-Expansion, 22, 105
Skalarprodukt, 47
Slot, 49
SM, 67, 146
SMM, 73, 75, 152
SMS (Shared-Memory-Parallelrechner), 1
SMV, 75, 148
SPARSEBLAS, 184
speed-up, 1
Splitting, 24
SPMD, 4
SQRT, 53, 133
SSP, 62, 141
SSSP, 75
SSXP, 182
Standardaufteilungen, 28
Standardparallelisierung, 113
STAR, 54, 89, 104
stargazer(), 90
Status, 49
Stelligkeit, 39
stmtdescend(), 85, 99
STRANGE, 136
string-matching, 38, 41
subspace optimizations, 119
SUFVSUM, 63, 143
superlinearer Speedup, 119
SV, 59, 138
SVDIAG, 61, 138
SVSUM, 75, 141
SWAP, 56, 134
SXSXP, 182
syntaktische Prädikate, 10

TAN, 53, 133
task, 24
temporäre Variable, 17
Tensor, 9
Termersetzungssysteme, 45
toter Code, 18
TraFoLa, 46
trapezoidaler Feldzugriff, 92
tred2-Routine, 173
tree parsing, 38
tree pattern matching, 38–42
—bottom-up, 39
—top-down, 41
Treepattern, 39
Treepatternmatching-Problem, 39
Trigger-Muster, 84
Triggermuster, 84
TWIG, 44
TWOTO, 53, 133
TXL, 46

Umbenennen von Ergebnissen, 103
Umverteilung, 32
Unifikation, 39
Untermuster, 84
Unterpattern, 40
USE(), 12

V1CONV, 71, 150
V2CONV, 73, 152
VAADD, 58, 137
VAADDMULTISV, 60, 139
VAADDSS, 59, 138
VAADDSV, 60, 138
VADD, 58, 137
VADDMUL, 59, 138
VADDSV, 60, 139
VAINC, 59, 137
VAMOD, 59, 137
VAMUL, 58, 137
VAND, 58, 137
Variablensymbol, 39
VASSIGN, 57, 136
VASSIGNSP, 58, 136
VCONDASS, 62, 139
VCONV, 70, 149
VCOPY, 57, 136
VCOS, 58, 137
Vektor, 9
Vektor-DAG, 108, 119
Vektorisierung, 21
—von Kommunikationsanweisungen, 26
Vektoroperation, 1
Vektorrechner, 1
Vektortriade, 47
Verschmelzung, 100
VEXP, 58, 137
VGATHER, 181
VGAUSSSEIDEL, 65, 144
VINC, 59, 137
VINIT, 58, 137
VINITSP, 58, 137
VINV, 59, 137
virtually shared memory (VSM), 2
VJACOBI, 65, 144
VLIW, 1
VLOG, 58, 137
VLUD, 61, 139
VMAX, 58, 137
VMAXLOC, 64, 143
VMAXLOCM, 64, 144
VMAXVAL, 64, 143
VMAXVL, 64, 144
VMAXVLM, 64, 144
VMIN, 58, 137
VMINLOC, 64, 144
VMINLOCM, 64, 144
VMINVAL, 64, 143
VMINVL, 64, 144
VMINVLM, 65, 144
VMOD, 58, 137
VMUL, 58, 137
VMULMUL, 60, 139
VOR, 58, 137
Vorabtransformationen, 83
VPROD, 62, 141
VQSUM, 62, 141
VQXSUM, 182
VRAND, 75, 141
VROR, 75, 141
VROT, 61, 140
VSCATTER, 181
VSHIFT, 62, 140
VSIN, 58, 137
VSUM, 62, 141
VSWAP, 61, 140
VTAN, 58, 137
VVGAUSSSEIDEL, 72, 152
VVJACOBI, 72, 151
VVMAXLOC, 72, 151
VVMAXVAL, 71, 151
VVMAXVL, 72, 151
VVMINLOC, 72, 151
VVMINVAL, 72, 151
VVMINVL, 72, 151
VVPROD, 70, 149
VVSHIFT, 68, 147
VVSUM, 70, 148
VXAADDSS, 181
VXAADDSV, 182
VXASSIGN, 181
VXASSIGNSP, 181
VXINIT, 181
VXINITSP, 181
VXMAXLOC, 183
VXMAXVAL, 183
VXMAXVL, 183
VXMINLOC, 183
VXMINVAL, 183
VXMINVL, 183
VXPROD, 182
VXSUM, 182

workspace-Konzept, 104, 117
workstation cluster, 2

Xputer, 126

Zielmuster, 84
Zwischendarstellung, 9