Automatische Parallelisierung

SS 1995

Skript zum 2. Teil der Vorlesung

Christoph W. Keßler
Universität Trier
FB 4 – Informatik
Zi. V 217, Tel. 2815
email: kessler@ti.uni-trier.de

Inhaltsverzeichnis

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 Vorlesung von Bedeutung erscheinen.

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

1.1 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.

[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.

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 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: 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.

1.2 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.

1.3 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 [LT93a], EXPRESS [Par], PARMACS [CHHW94], MPI [CGH94], LINDA [CGMS94]) enthalten einfache (SEND und 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.

1.4 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).

1.5 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.*

2 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`.

2.1 Ausführungsreihenfolge und Datenabhängigkeiten

Betrachten wir zunächst einen Grundblock[2] 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: 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.

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)`[3] 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;
```

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

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

wird die Anweisung S insgesamt bis zu $N_{\mathtt{i1}} \cdot N_{\mathtt{i2}} \cdot ... \cdot N_{\mathtt{ik}}$ mal[4] 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 L, gilt $S_1(\vec{i}) \prec S_2(\vec{j})$ genau dann, wenn entweder S_1 textuell vor S_2 liegt und $(\vec{i} \unlhd \vec{j})$, oder S_2 textuell vor S_1 liegt und $\vec{i} \lhd \vec{j}$.

Bemerkung: S_1 und S_2 können selbst wieder Schleifenschachtelungen sein. $\vec{i}$ und $\vec{j}$ beziehen sich dann aber nur auf die tiefste gemeinsame Schleifenebene, d.h. bis Schleifenebene k.

Notation: Sei $S_1(\vec{i}) \prec S_2(\vec{j})$. Mit $S_1(\vec{i}) \prec_q S_2(\vec{j})$ bezeichnen wir, daß $i_r = j_r$ für $1 \leq r < q$ und $i_q < j_q$.

Definition: *Distanzvektor* $dis(\vec{i}, \vec{j}) = \vec{j} - \vec{i}$

Definition: *Richtungsvektor* $\theta = dir(\vec{i}, \vec{j}) = sgn(\vec{j} - \vec{i})$, die Komponenten werden dargestellt als '<' für -1, '=' für 0 und '>' für +1. Den inversen Richtungsvektor θ^{-1} erhält man durch Umdrehen der Ungleichheitszeichen (bzw. Multiplikation mit -1).

Lassen wir Feldzugriffe als Variablen zu, so enthalten $DEF(S)$ und $USE(S)$ nun auch symbolische Feldzugriffe $A[exp_1, exp_2, ..., exp_d]$. Jede Variable, die in einem Indexausdruck exp_j vorkommt, wird auch in $USE(S)$ aufgeführt.

Im Innern einer Schleifenschachtelung erhalten wir

$$\begin{aligned} DEF(S(\vec{i})) = \{v \in DEF(S), v \text{ skalar}\} \quad \cup \quad & \{A[eval(exp_1, \vec{i}), ..., eval(exp_d, \vec{i})] \\ & \text{für alle } A[exp_1, ..., exp_d] \in DEF(S)\}, \end{aligned}$$

wobei $eval(exp, \vec{i})$ für alle Vorkommen von Schleifenvariablen in exp die entsprechende Komponente aus $\vec{i}$ einsetzt.

Analoges gilt für $USE(S(\vec{i}))$.

Definition: 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 $S_1(\vec{i}) \delta S_2(\vec{i})$ genau dann, wenn
(1) $S_1(\vec{i}) \prec S_2(\vec{j})$, und
(2) es existiert eine Variable $v \in VAR$
mit $v \in DEF(S_1(\vec{i})) \cap USE(S_2(\vec{j})) \neq \emptyset$ (Flußabhängigkeit)
oder $v \in USE(S_1(\vec{i})) \cap DEF(S_2(\vec{j})) \neq \emptyset$ (Antiabhängigkeit)
oder $v \in DEF(S_1(\vec{i})) \cap DEF(S_2(\vec{j})) \neq \emptyset$ (Ausgangsabhängigkeit), und
(3) es existiert kein $\vec{k} \in Range(L)$ und keine Anweisung S' im Rumpf von L mit $S_1(\vec{i}) \prec S'(\vec{k}) \prec S_2(\vec{j})$ und $v \in DEF(S'(\vec{k}))$.

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*, in Zeichen $S_1(\vec{i}) \delta_h S_2(\vec{j})$, falls $S_1(\vec{i}) \prec_h S_2(\vec{j})$, d.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.

Es gilt $S_1 \delta S_2$, sobald es zwei (nicht notwendig verschiedene) Iterationsvektoren $\vec{i}, \vec{j}$ gibt mit $S_1(\vec{i}) \delta S_2(\vec{j})$.

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

Definition: 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.

2.2 Verfahren zum Testen von Datenabhängigkeit

Beim Testen der Möglichkeit, alle Datenabhängigkeiten zwischen zwei Anweisungen S_1, S_2 in einer Schleifenverschachtelung von k 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 $e_s = a_{s,0} + \sum_{q=1}^{k} a_{s,q} i_q$, $1 \leq s \leq d_A$, in den Schleifenvariablen $i_1, ..., i_k$ und alle $a_{s,q}$ konstant sind.

Seien für S_1 textuell vor 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 $e_s = a_{s,0} + \sum_{q=1}^{k} a_{s,q} i_q$, $1 \leq s \leq d_A$ und $f_s = b_{s,0} + \sum_{q=1}^{k} b_{s,q} i_q$, $1 \leq s \leq d_A$.

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

$$a_{s,0} + \sum_{q=1}^{k} a_{s,q} i_q \quad = \quad b_{s,0} + \sum_{q=1}^{k} b_{s,q} j_q, \qquad 1 \leq s \leq d_A \tag{1}$$

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 im Iterationsraum liegen, d.h. bei in $\vec{i}$, $\vec{j}$ linearen Unter– und Obergrenzen das lineare Ungleichungssystem

$$\begin{array}{rl} l_1 \leq i_1 \leq u_1 & \qquad l_1 \leq j_1 \leq u_1 \\ l_2(i_1) \leq i_2 \leq u_2(i_1) & \qquad l_2(j_1) \leq j_2 \leq u_2(j_1) \\ \vdots & \qquad \vdots \\ l_k(i_1, ..., i_{k-1}) \leq i_k \leq u_k(i_1, ..., i_{k-1}) & \qquad l_k(j_1, ..., j_{k-1} \leq j_k \leq u_k(j_1, ..., j_{k-1}) \end{array} \tag{2}$$

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*: Sei θ ein Richtungsvektor mit $EQ = \{q \in \{1, ..., k\} : \theta_q = \text{`='}\}$. Sei für alle s, $1 \leq s \leq d_A$,

  $$G_s = \mathrm{ggT}(a_{s,q} - b_{s,q} : q \in EQ\}, a_{s,q'} - b_{s,q'} : q' \notin EQ, a_{s,q'} - b_{s,q'} : q' \notin EQ) \tag{3}$$

  Eingeschränkt auf Iterationsvektoren $\vec{i}, \vec{j}$ mit $dir(\vec{i}, \vec{j}) = \theta$ besitzt die Abhängigkeitsgleichung 1 eine ganzzahlige Lösung $(\vec{i}, \vec{j})$ genau dann, wenn für alle $s \in \{1, ..., d_A\}$ entweder $G_s | (b_{s,0} - a_{s,0})$ (falls G_s definiert) oder $a_{s,0} = b_{s,0}$.

Aus der damit festgestellten Existenz einer Lösung kann aber nicht geschlossen werden, ob diese auch Gleichung 2 erfüllt.

Bemerkung: Je größer EQ, desto weniger Koeffizienten sind im ggT–Argument, desto größer kann der ggT werden, desto größer wird die Wahrscheinlichkeit, daß der ggT das absolute Glied nicht teilt, desto eher kann Unabhängigkeit gezeigt werden.

- *Banerjee–Test*: Im Gegensatz zum ggT–Test bestimmt der Banerjee–Test [Ban88], ob es eine *rationale* Lösung $(\vec{i'}, \vec{j'})$ von Gleichung 1 im Bereich 2 gibt. Der Banerjee–Test bestimmt mit Hilfe von Gleichung 2 die untere Schranke $LB_{s,q}$ und die obere Schranke $UB_{s,q}$ jedes Terms $(a_{s,q}i_q - b_{s,q}j_q)$, $1 \leq q \leq k$, $1 \leq s \leq d_A$. Falls $\sum_{q=1}^{k} LB_{s,q} > b_{s,0} - a_{s,0}$ oder $\sum_{q=1}^{k} UB_{s,q} < b_{s,0} - a_{s,0}$ für ein s, $1 \leq s \leq d_A$, kann Gleichung 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 1 in jeder Dimension s, $1 \leq s \leq d_A$, nur eine einzige Schleifenvariable vorkommt, d.h. die $a_{s,q}$, $b_{s,q}$ für jeweils höchstens ein q, $1 \leq q \leq k$, 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 1 und 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].

2.3 Richtungsvektoren

Sei $(\vec{i}, \vec{j})$ eine Lösung der Abhängigkeitsgleichung 1. Der Vektor $dir(\vec{i}, \vec{j}) = \theta = (\theta_1, ..., \theta_n)$ mit $\theta_q \in \{<, >, =\}$ heißt dann der Richtungsvektor zu dieser Datenabhängigkeit.

Zusätzlich zu den schon bekannten Richtungsvektorkomponentenwerten '$<$', '$=$' und '$>$' kann man die Symbole '$\leq$', '$\geq$' und '$*$' einführen, die entsprechende Disjunktionen zweier oder aller drei Richtungssymbole bedeuten.

Analog zum ggT-Test kann man den Banerjee–Test so umformulieren, daß er sich auf den Beweis der Unabhängigkeit bei gleichzeitiger Vorgabe eines bestimmten Richtungsvektors beschränkt (vgl. [ZC90]). Damit kann man bei verschachtelten Schleifen ($k > 1$) 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.

2.4 Linearisierung

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

1. auf die jeweils in Abschnitt 2.2 beschriebene Variante, bei der alle Dimensionen des Feldes A separat auf Abhängigkeit geprüft werden, und

2. auf die jeweilige *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 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 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].

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.

3.1 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 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 Konstantenpropagation (constant propagation)

Man betrachte das folgende Beispiel:

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

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.

3.3 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:

```
   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].

3.4 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:

```
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. ◇

3.5 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 3.4 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.

3.6 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.

3.7 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:

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

```
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:

```
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*).

3.8 Abwickeln einzelner Iterationen (loop peeling)

Möchte man sich in Beispiel 3.7 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.

3.9 Abrollen von Schleifen (loop unrolling)

Ist wie in Beispiel 3.7 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.

3.10 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: 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. ◇

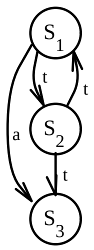
Generell ist Schleifenvertauschung unzulässig, falls es zwischen zwei (nicht notwendig verschiedenen) Anweisungen des Rumpfes eine Datenabhängigkeit gibt, in deren Richtungsvektor, wie in obigem Beispiel, sowohl '<' als auch '>' vorkommen [Wol89]. Eine allgemeine Diskussion der Schleifenvertauschung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

3.11 Schleifenaufgliederung (loop distribution)

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

Beispiel:

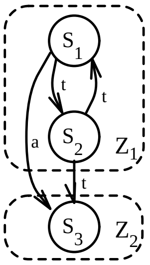
```
   for (i=1; i<=n; i++) {
S1:  A[i+1] = B[i-1] + C[i];
S2:  B[i] = A[i] * K;
S3:  C[i] = B[i] - 1;
   }
```



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).

3.12 Vektorisierung

Betrachten wir nochmals Beispiel 3.11 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.

3.13 Vektorisierung bedingter Anweisungen (`IF` conversion)

Für bedingte vektorisierbare Anweisungen, wie etwa im folgenden Beispiel

```
for (i=1; i<=n; i++)
  if (C[i] != 0.0)
     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] != 0.0);
A[1:n] = (MASK[1:n])?  B[1:n] / C[1:n]  :  A[1:n];
```

transformiert werden [AK87][ZC90].

3.14 Skalar–Expansion (scalar expansion)

Beispiel: 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].

3.15 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:

```
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] = ... ;
```

◇

4 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 ($\mathtt{SIGNAL}(S(\vec{i}))$), 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 ($\mathtt{WAIT}(S(\vec{i}))$). 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].

5 Halbautomatische Parallelisierung für DMS

Die Parallelisierungsstrategie für DMS unterscheidet sich grundlegend von der in Abschnitt 4 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.

5.1 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.

   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: 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

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

```
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 2 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: (Fortsetzung von Beispiel 2): 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: In Beispiel 3 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.

5.2 Lokale Iterations–, Index– und Kommunikationsmengen

Die *lokale Iterationsmenge* $Exec_S(p)$ für einen Prozessor p und eine Zuweisung S der Form

$$S: \;\; \mathtt{A[}f(\vec{i})\mathtt{]} \;=\; \ldots \;\; \mathtt{B[}g(\vec{i})\mathtt{]} \;\; \ldots$$

innerhalb einer Schleifenverschachtelung L ist die Menge der Iterationsvektoren $\vec{i}$, für die die Maske *owned*($\mathtt{A[}f(\vec{i})\mathtt{]}$) auf p zu TRUE evaluiert. $Exec_S$ 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 $Exec_S(p)$ 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* $Ref_{S,B}(p)$ für einen Prozessor p und ein Feld B ist die Menge der Iterationsvektoren $\vec{i}$, für die für *alle* Feldvorkommen $\mathtt{B[}g(\vec{i})\mathtt{]}$ in S die Maske *owned*($\mathtt{B[}g(\vec{i})\mathtt{]}$) auf p zu TRUE evaluiert. $Ref_{S,B}$ ist die Umkehrfunktion der Indizierung g des Vorkommens von `B`, angewandt auf die Datenaufteilung von `A`. Wir definieren $Ref_S(p) = \cap_B Ref_{S,B}(p)$ als Menge der Iterationen, die in S nur zu p lokale Werte lesen.

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* $L_{S,A}(p) = f(Exec_S(p))$ für das Feld A auf der linken Seite und $L_{S,B} = g(Ref_{S,B}(p) \cap Exec_S(p))$ für jedes Feld B, das in S auf der rechten Seite vorkommt; ferner bestimmt man für jedes B die *referenzierte Indexmenge* $M_{S,B}(p) = g(Exec_S(p))$, d.h. explizite Darstellungen der Mengen von Indexwerten, die die Indizierung des jeweiligen Feldvorkommens aufgrund der sie indizierenden umgebenden Schleifen annehmen kann.

Für diese Felder B sind die Indexmengen $L_{S,B}(p)$ auf p lokal und die Indexmengen $M_{S,B}(p) \setminus L_{S,B}(p)$ auf p nichtlokal.

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

Entsprechend sei $Comm_S(p,p') = \cup_B Comm_{S,B}(p,p')$ die Kommunikationsmenge für alle vorkommenden Felder B. Mit einer expliziten Darstellung von $Comm_S(p,p')$ 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 $Comm_S(p,p') \neq \emptyset$ **then** versende Nachrichten gemäß der Listen $Comm_S(p,p')$
2. für alle Iterationen $\vec{i} \in Exec_S(p) \cap Ref_S(p)$:
   führe $S(\vec{i})$ aus
3. für alle Prozessoren $p'' \neq p$:
   if $Comm_S(p'',p) \neq \emptyset$ **then** empfange Nachrichten gemäß der Listen $Comm_S(p'',p)$
4. für alle Iterationen $\vec{i} \in Exec_S(p) - Ref_S(p)$:
   führe Iterationen in $Exec_S(p) - Ref_S(p)$ aus.

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

5.3 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: [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 "richtigen" Spaltenindex Diese Matrix `cmatrx` ist komprimiert gespeichert, d.h. enthält nur die von Null verschiedenen Einträge. Die Indizierung des Vektors `vec` gewährleistet, daß jeweils mit dem richtigen Vektorelement 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], die beide von jedem Prozessor p ausgeführt werden.

Der *inspector* bestimmt (zur Laufzeit) in jeder Iteration $\vec{i} = (i,j)$ den lokalen Iterationsbereich $Exec_S^{\vec{i}}(p)$, sammelt die von ihm für $Ref_{S,\mathtt{vec}}^{\vec{i}}(p)$ benötigten indirekten Feldzugriffe (`vec[ind[i][j]]`) und baut daraus in einer globalen Kommunikationsphase seine lokalen Sende– und Empfangslisten $Comm_S^{\vec{i}}(p,*)$ und $Comm_S^{\vec{i}}(*,p)$ 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].

6 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].

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.

[8]Vgl. hierzu Abschnitt 8, Formel 5.

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 $v = (n_1, \ldots, n_D)$ 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 Prozessoren entlang dieser Dimension (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

$$dist_k(A, t) = \begin{cases} \lfloor \frac{t - offset}{block} \rfloor [\mod N_{map(A_k)}] & \text{falls } A_k \text{ aufgeteilt} \\ X & \text{falls } A_k \text{ repliziert} \end{cases} \quad (4)$$

wobei der Term $\mod N_{map(A_k)}$ 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 2.

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.

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 nicht-zyklische Aufteilungen beschränkt. Umverteilungen werden durch weitere Verzweigungen im Suchbaum

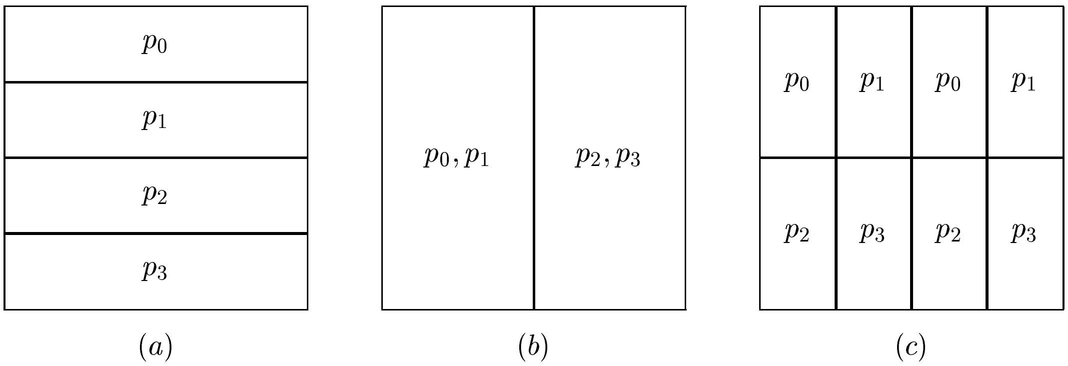


Abbildung 2: Drei Beispiele für Datenaufteilungen eines zweidimensionalen Feldes A mit Extents 16 x 16 und $N = 4$ (in Anlehnung an [Gup92]): (a) $N_1 = 4$, $N_2 = 1$, $dist_1(A,t) = \lfloor \frac{t-1}{4} \rfloor$, $dist_2(A,t) = 1$ (zeilenweise, zusammenhängend); (b) $N_1 = 2$, $N_2 = 2$, $dist_1(A,t) = X$, $dist_2(A,t) = \lfloor \frac{t-1}{8} \rfloor$ (spaltenweise, teilrepliziert); (c) $N_1 = 2$, $N_2 = 2$, $dist_1(A,t) = \lfloor \frac{t-1}{8} \rfloor$, $dist_2(A,t) = \lfloor \frac{t-1}{4} \rfloor \bmod 2$ (blockweise in der ersten, zyklisch in der zweiten Dimension). *map* ist jeweils die Identität.

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.

6.1 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.

6.2 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

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

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.

6.3 Statische Umverteilung

Definition: (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 [LT93b], 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.

7 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: [LC91] Seien `A, B` in beiden Dimensionen aufgeteilt.

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

[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'.

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 7 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 7 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.

8 Laufzeitvorhersage

Der übliche Ansatz zur Laufzeitvorhersage bei mit der in Abschnitt 5 beschriebenen Methode zu gegebenen Feldaufteilungen parallelisierten DMS–Programmen ist ein bottom–up–Durchlaufen 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.

[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.

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 5.2 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 $L_{S,A}$, die referenzierten Indexmengen $M_{S,B}$, $M_{S,C}$,... und daraus die Kommunikationsmengen $Comm_{S,B}(p,p')$, $Comm_{S,C}(p,p')$,... 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, $Comm_{S,B}(p,p')$ explizit vorliegen.

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

Die Menge $Comm_S(p) = Comm_{S,B}(p) \cup Comm_{S,C}(p) \cup \ldots$ gibt dann alle Feldelemente an, die p bei der Kommunikation vor dieser Anweisung empfängt. Die Menge der von p empfangenen Nachrichten ist $SN(p) = SN_B(p) \cup SN_C(p) \cup \ldots$. Durch die Vektorisierung von Kommunikationsanweisungen ist im allgemeinen $|SN(p)| \ll |Comm_S(p)|$.

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

$$t(p_1,p_2,n) = \sigma + \beta \cdot \rho \cdot n + \gamma \cdot dist(p_1,p_2) \tag{5}$$

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]).

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

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 $Comms_S(p_i, p_j)$ 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 $|Comms_S(p')|$ 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 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.

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.

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

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]).

9 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.

Das PARAMAT–System zur vollautomatischen DMS–Parallelisierung baut auf diesem Gedanken auf. 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.

Literatur

[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.

[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.

[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].

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[BC86] Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization. In *SIGPLAN Symposium on Compiler Construction*, 1986.

[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.

[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.

[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.

[Bok90] S. Bokhari. Communication Overhead on the Intel iPSC/860 Hypercube. Technical Report Interim Report 10, ICASE, May 1990.

[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.

[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.

[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.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[Cyt86] Ron Cytron. Doacross: beyond vectorization for multiprocessors. In *Int. Conference on Parallel Processing*, pages 226–234, 1986.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Hin92] Didier Y. Hinz. A run-time load balancing strategy for highly parallel systems. *Acta Informatica*, 29:63–94, 1992.

[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.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Scientific Programming*, 2, 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.

[INT90] INTEL Corp. iPSC/2 and iPSC/860 Programmer's Reference Manual, June 1990.

[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.

[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.

[Keß94] 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.

[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.

[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.

[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.

[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.

[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.

[Lov77] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, Jan. 1977.

[LT93a] Oak Ridge National Laboratory and University of Tennessee. Parallel Virtual Machine Reference Manual, Version 3.2. Technical report, August 1993.

[LT93b] 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.

[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.

[McM86] Frank McMahon. The Livermore Fortran Kernels: A Test of the Numeric Performance Range. Technical report, Lawrence Livermore National Laboratory, 1986.

[MJ81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[MR87] P. Mehrotra and J. Van Rosendale. The BLAZE Language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Sho81] Robert Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the ACM*, 28(4):769–779, Oct. 1981.

[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.

[TP93] Peng Tu and David Padua. Array Privatization for Shared and Distributed Memory Machines (Extended Abstract). *SIGPLAN Notices*, 28(1):64–67, 1993.

[TW91] Arthur Trew and Greg Wilson (Eds.). *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. Springer, 1991.

[Who91] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[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.

[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.

[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.