DIE ALIGNMENTANALYSE

Eine pragmatische Methode zur Erzeugung effizienter datenparalleler Fortranprogramme und ihre Realisierung im VFCS

eingereicht von

Erwin LAURE

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Magister rerum socialium oeconomicarumque (Mag. rer. soc. oec.)
Magister der Sozial- und Wirtschaftswissenschaften

Sozial- und Wirtschaftswissenschaftliche Fakultät
Universität Wien

Studienzweig: *Betriebsinformatik*

Betreuer/Begutachter:
O.Univ.-Prof. Dr. Hans P. Zima

Wien, im Jänner 1996

INHALTSVERZEICHNIS

LEBENSLAUF

Name:	LAURE Erwin Andreas
Geburtsdatum:	04. Februar 1970
Geburtsort:	Wolfsberg
Wohnort:	A-1030 Wien, Boerhaavegasse 3/1/24

Eltern:	LAURE Otto
	LAURE Hildegard

Ausbildungsweg:

1976 - 1980	Volksschule St. Paul i. Lav.
1980 - 1984	Stiftsgymnasium St. Paul i. Lav.
1984 - 1989	Bundeshandelsakademie Wolfsberg
1989	Ablegung der Matura an der Bundeshandelsakademie Wolfsberg
1989 - 1990	Ableistung des Präsenzdienstes
1990 - 1996	Studium der Wirtschaftsinformatik (Betriebsinformatik) an der Universität Wien

Studienverlauf:

1993	Ablegung der 1. Diplomprüfung
1995 - 1996	Abfassung der Diplomarbeit

Es sei mir an dieser Stelle erlaubt, Herrn Prof. Dr. Hans P. Zima für die Betreuung dieser Diplomarbeit sowie den Mitgliedern seiner Forschungsgruppe, die mir im Zuge meiner Arbeiten oftmals hilfreich und beratend zur Seite standen, zu danken.
Ein besonderer Dank gebührt Frau Barbara M. Chapman, die durch zahlreiche Diskussionen und Anregungen wesentlich zum Gelingen dieser Arbeit beitrug.

0. EINLEITUNG

Parallelrechner haben in den letzten Jahren stetig an Bedeutung gewonnen und werden heute als erfolgsversprechendste Entwicklung zur Lösung von vor allem wissenschaftlichen, numerischen Problemen angesehen. Trotz dieser Beliebtheit paralleler Rechnerarchitekturen steckt die Programmierung derselben noch in den Kinderschuhen und man ist noch weit davon entfernt, Parallelrechner, insbesondere Parallelrechner mit verteiltem Speicher (*distributed memory multi-processors DMMP*), ebenso komfortabel wie normale Einprozessormaschinen programmieren zu können.

Eine weitverbreitete Methode zur Programmierung solcher DMMPs ist das *Single Program Multiple Data (SPMD) Modell*, bei welchem mehrere Prozessoren an der Lösung des selben Problems arbeiten. Als eines der größten Probleme erweist sich hierbei jedoch der Umstand, daß über die Zuteilung der Programmdaten zu den Speichereinheiten der einzelnen Prozessoren (die *Datenaufteilung*) ein entscheidender Einfluß auf das Performanceverhalten dieser Rechner genommen wird, da bei ungünstigen Zuteilungen Prozessoren oft auf Daten zugreifen müssen, welche im Speicherbereich anderer Prozessoren befindlich sind, was zu zeitaufwendiger Kommunikation zwischen diesen beiden Prozessoren führt. Eine optimale Datenaufteilung zu finden, ist eine schwierige und fehleranfällige Aufgabe, welche vom Programmierer vorgenommen werden muß. Eines der großen Forschungsziele ist es daher, den Programmierer bei dieser Aufgabe zu unterstützen bzw. die Datenaufteilung von entsprechenden Systemen (seien dies parallelisierende Compiler oder andere Tools) automatisch vornehmen zu lassen.

In der vorliegenden Arbeit wird zu diesem Themenkomplex ein Analysetool vorgestellt, welches sich zum Ziel setzt, dem Benutzer von Parallelrechnern Informationen über die Datenzugriffe im Zuge einer Programmexekution zur Verfügug zu stellen und ihn damit bei der Wahl der geeigneten Datenaufteilung zu unterstützen. Diese Analyseverfahren beschäftigen sich mit der Vermeidung von unnötiger Kommunikation bei Arrayzugriffen innerhalb von Schleifen, indem versucht wird, gemeinsam genutzte Datenbereiche unterschiedlicher Arrays festzustellen und diese für eine nach Abschluß der Analyse vorzunehmende Datenaufteilung aneinander zu koppeln. Dieses Problem wird im allgemeinen als das **Alignmentproblem** bezeichnet. Die Lösung desselben erleichtert die Aufgabe der Datenaufteilung in Hinblick auf die Vermeidung von Kom-

munikation, indem für viele Arrays eine implizite Datenaufteilung generiert wird, da diese an die Datenaufteilung anderer Arrays gekoppelt wird.
Die Modellierung dieses Problems erfolgt mit Hilfe des von Li et.al. eingeführten graphentheoretischen Modells des *Component Alignment Graph* (CAG), die Lösung über eine Variante des ebenfalls von Li et.al. eingeführten Heuristischen Algorithmus. Die Alignmentanalyse ist auf jeden Fall als heuristisches Hilfsmittel anzusehen und kann keinesfalls einen Anspruch auf Optimalität der Lösung erheben, wiewohl in vielen Fällen sehr hilfreiche Informationen über die Datenzugriffe innerhalb eines Programms gewonnen werden können. In Hinblick auf den heuristischen Charakter wird stets auf die Flexibilität und Praktikabilität der angewandten Methoden Bedacht genommen, ohne jedoch auf ein größtmögliches Maß an Genauigkeit und Aussagekräftigkeit der gewonnenen Informationen zu verzichten.

Die Vorteile der hier vorzustellenden Methoden liegen neben diesem Praxisbezug auch darin, daß durch Anwendung einfacher Methoden die Wirksamkeit der Analysen in Hinblick auf die Anzahl der erkannten Fälle signifikant erhöht werden kann, wie auch die Bedeutung der erkannten Fälle in Hinblick auf deren Kommunikationsverhalten durch Entwicklung eines einfachen, jedoch wirkungsvollen Kommunikationskostenmodells festgestellt wird. Ein weiterer Vorteil dieser Arbeit liegt darin begründet, daß neben dem interdimensionalen Alignmentproblem, welches sich mit der gemeinsamen Nutzung von Arraydimensionen beschäftigt und oft generalisierend als Alignmentproblem bezeichnet wird, auch das intradimensionale Alignmentproblem, welches sich mit der relativen Position der gemeinsam genutzten Arrayelemente innerhalb der Dimensionen beschäftigt, modelliert wird und in die Behandlung des interdimensionalen Alignmentproblems einfließt. Durch die Modellierung des intradimensionalen Alignmentproblems ist es auch möglich, bei Programmen, die hauptsächlich aus eindimensionalen Arrays bestehen (und damit eine triviale Lösung des interdimensionalen Alignmentproblems haben), trotzdem einen gewissen Grad an Information über die anzuwendenden Datenaufteilungen zu erlangen. Die Praktikabilität der vorzustellenden Methoden wird auch anhand eines im Rahmen eines parallelisierenden Compilers implementierten Prototypen untermauert, mit dessen Hilfe auch die vorzustellenden Analyseergebnisse zweier Benchmarkprogramme generiert wurden.

Der weitere Verlauf dieser Arbeit gliedert sich wie folgt:
In Kapitel 1 wird eine Einführung in die Parallelverarbeitung vorgenommen, bei welcher parallele Rechnerarchitekturen und parallele Programmiermodelle vorgestellt werden. Ein besonderer Schwerpunkt wird dabei auf die Problematik der Datenaufteilung gelegt.

Kapitel 2 stellt sodann das *Vienna Fortran Compilation System (VFCS)*, einen semiautomatischen parallelisierenden Compiler vor, der die Grundlage für das bei der Formulierung des Alignmentproblems herangezogene Compilermodell bildet und in dessen Rahmen auch der Prototyp des Alignmentanalysetools implementiert wurde.
Die Definition des Alignmentproblems sowie dessen heuristischer Lösungsansatz findet sich in Kapitel 3.
In Kapitel 4 wird das Alignmentanalysetool vorgestellt, seine Arbeitsweise anhand eines einfachen Beispiels demonstriert sowie auf einige Implementierungsdetails eingegangen. Zwei Anwendungsbeispiele beenden dieses Kapitel.
Ein Überblick über weitere Forschungsansätze auf diesem Gebiet wird in Kapitel 5 gegeben, bevor eine kurze Zusammenfassung sowie ein Ausblick auf mögliche Erweiterungen und Adaptionen der vorzustellenden Methoden in Kapitel 6 diese Arbeit beenden.

1. PARALLELVERARBEITUNG

In den letzten Jahren hat die Parallelverarbeitung, vor allem am Gebiet der numerischen Anwendungen (als Beipiele für kommerzielle Anwendungen wären hier Programme zur Wettervorhersage, aber auch Echtzeitsimulationen wie z.B. Crash-Tests zu nennen) an immer größerer Bedeutung gewonnen. Der Grund dafür liegt vor allem darin, daß bei numerischen Anwendungen sehr große Datenmengen verarbeitet werden müssen, was auf sequentiellen Rechnern unter Umständen nur mit einem nicht vertretbaren Zeitaufwand realisierbar wäre. Aber auch der Genauigkeitsgrad der Lösungen läßt sich bei vielen Problemen durch die Einbeziehung größerer Datenmengen signifikant erhöhen. Ein weiterer Grund, warum gerade in die Entwicklung von Strategien für Parallelverarbeitung ein großer Forschungsaufwand gesteckt wurde und wird, ist, daß eine Leistungssteigerung von herkömmlichen Prozessoren kaum noch möglich ist, da die Prozessorentechnik an ihre physikalischen Grenzen zu stoßen droht. Die Verknüpfung mehrerer Prozessoren zu einer parallel arbeitenden Einheit verspricht hingegen die Möglichkeit, die Leistungsfähigkeit von Rechnern ohne große Weiterentwicklung der Prozessoren signifikant zu steigern.

Trotz angestrengter Forschungsarbeiten ist die allgemeine Nutzbarkeit der Parallelverarbeitung noch nicht gegeben, da einerseits die unterschiedlichen Architekturen von Parallelrechnern die Portabilität von parallelen Programmen beeinträchtigen, andererseits die explizite parallele Programmierung selbst noch sehr unkomfortabel und am ehesten mit der Assemblerprogrammierung von sequentiellen Rechnern vergleichbar ist.

In diesem Kapitel soll ein kurzer Einblick in den momentanen Forschungsstand am Gebiet der Parallelverarbeitung gegeben werden, wobei zuerst einige Typen von Parallelrechnern vorgestellt werden und sodann auf die Problematik der parallelen Programmierung und hier im speziellen auf die automatische Parallelisierung eingegangen werden soll.

1.1. Parallelrechner

Um unterschiedliche Rechnerarchitekturen einteilen zu können, wurde eine Vielzahl von Klassifikationssystemen entwickelt. Die gebräuchlichste Klassifikationsmethode stellt trotz oder vielleicht gerade wegen ihrer Einfachheit die Klassifikation von Flynn

[Flynn72] dar. Flynn unterscheidet anhand der Anzahl von Instruktions- und Datenströmen vier Klassen von Rechnerarchitekturen:

- *SISD (single-instruction stream, single-data stream):*
  Diese Klasse repräsentiert die gebräuchlichsten Rechnerarchitekturen, welche auf einem, sequentiell arbeitenden Prozessor beruhen.
- *SIMD (single-instruction stream, multiple-data stream):*
  Zu dieser Klasse zählen vor allem Vektormaschinen, mit denen bereits ein gewisser Grad an Parallelität erzielbar ist.
- *MISD (multiple-instruction stream, single-data stream):*
  Diese Klasse wurde von Flynn nur der Vollständigkeit halber aufgeführt und hat kaum praktische Relevanz.
- *MIMD (multiple-instruction stream, multiple-data stream):*
  In diese Klasse fallen die üblicherweise als *Multiprozessoren* bezeichneten Rechnerarchitekturen.

Unglücklicherweise umfaßt die Klasse der MIMD-Maschinen die unterschiedlichsten parallelen Rechnerarchitekturen, weshalb hierbei weitere Unterscheidungen unabdingbar sind. Die wichtigste Unterteilung der MIMD-Maschinen erfolgt nach der Lokalität des Speichers in Systeme mit gemeinsamem Speicher (Shared Memory Multi-Processors SMMP) und in Systeme mit verteiltem Speicher (Distributed Memory Multi-Processors DMMP). Letztere Klasse ist aufgrund ihrer Skalierbarkeit (und damit theoretischen Unbegrenztheit) die wichtigere der beiden Architekturen. Auch dieser Arbeit liegt das Modell eines DMMP zugrunde.

Die nachfolgend angeführten Beispiele für unterschiedliche Rechnerarchitekturen wurden aus [Hert95] entnommen:

SIMD-Rechner - und deren wichtigste Vertreter, die Vektorrechner, basieren auf dem Prinzip der Datenparallelität, das heißt, das auszuführende Programm ist weiterhin ein normales sequentielles Programm. Aufgrund ihrer sehr spezialisierten Architektur wird bei den neueren Rechnerentwicklungen kaum mehr versucht, reine Vektorrechner zu entwickeln; man ist vielmehr dazu übergegangen, Multiprozessoren mit zusätzlichen Vektoreinheiten zu versehen, die diese spezialisierten Berechnungen gegebenenfalls ausführen.

Beispiele für Vektorrechner wären die *Connetcion Machine CM-200* der Thinking Machines Corporation mit 4096 bis 65536 Prozessoren mit einer Wortlänge von 1 Bit und Gleitkomma-Koprozessoren für jeweils 32 Prozessoren oder die *MasPar-2* der Firma MasPar mit bis zu 16K 32-bit Prozessoren und mikrokodierten Gleitkommaoperationen.

Die wesentlich bedeutendere Kategorie ist jene der MIMD-Rechner, bei denen viele verschiedene Handlungsfäden (threads) eines Programmes parallel ablaufen können und die daher allgemein als *Parallelrechner* bezeichnet werden. Wie bereits erwähnt, kann man aufgrund der physischen Beschaffenheit des Speichers zwischen SMMP und DMMP unterscheiden. Aufgrund ihrer Skalierbarkeit sind DMMPs die bevorzugte Architektur, wobei diese Rechner auch als *message passing Maschinen* bezeichnet werden, da Daten zwischen den einzelnen Prozessoren über ein effizientes Verbindungsnetzwerk ausgetauscht werden müssen. Dieser Nachrichtenaustausch erfolgt über ungleichförmige Speicherzugriffe (*non-uniform memory access*), was bedeutet, daß die Prozessoren nicht auf einen virtuell einheitlichen Speicher zurückgreifen können, sondern unter Umständen auf mehrere getrennte Speicherplätze zugreifen müssen.
Die Effizienz des Verbindungsnetzwerkes ist natürlich für die Performance von entscheidender Bedeutung, wobei sich hierbei einige effiziente Topologien herauskristallisiert haben: der Hypercube, das Gitter oder Mehrstufennetze.

Ein Beispiel für die Hypercube-Architektur wäre der *Intel iPSC/860* mit Standard-i860 Prozessoren. Als Vertreter der Gitterarchitektur wäre der *Intel Paragon XP/S* zu nennen, der ebenfalls über Standard-i860 Prozessoren verfügt, welche jedoch mit einem 2D-Gitter Verbindungsnetz ausgestattet sind, was die Erweiterung im Gegensatz zu Hypercube Architekturen erleichtert. Der *IBM SP2* ist ein Beispiel für Mehrstufennetzwerke, wobei jeweils 16 RS/6000 Prozessoren zu einem 8x8 Schalter-Chip zusammengefügt werden, welche wiederum zu größeren Netzwerken zusammengestellt werden können.

Der große Nachteil bei den genannten Architekturen besteht darin, daß message passing Routinen sehr zeitaufwendig sind und damit eine große Rolle im Performanceverhalten von parallelen Programmen spielen. In neueren Rechnerentwicklungen ist man daher dazu übergegangen, neben dem expliziten message passing, welches ja auf ungleichförmigen Speicherzugriffen basiert, auch die Möglichkeit anzubieten, durch die selbe Adressierung zusammengehöriger Elemente in den einzelnen Speichereinheiten auf wesentlich schnellere Kommunikationsroutinen zurückgreifen zu können. Solche hochentwickelten Kommunikationsroutinen erlauben zwar eine wesentliche Verkleinerung des Zeitaufwandes gegenüber dem expliziten massage passing, sind aber noch immer um ein Vielfaches langsamer als der Datenzugriff bei einem gleichförmigen Speicher (*uniform memory access*).

[Hert95] bezeichnet solche Maschinen aufgrund dieser speziellen Speicherzugriffsmöglichkeiten als "shared memory Maschinen", was in Hinblick auf die Programmierung dieser Maschinen jedoch als nicht zutreffend eingestuft werden muß, da ja lediglich effizientere Kommunikationsroutinen angeboten werden, der Programmierer diese jedoch wie beim herkömmlichen message passing anzuwenden hat und es letztlich ihm obliegt, den erzielbaren Grad an Parallelität auszuschöpfen und für die entsprechende Datenlokalität zu sorgen, um teure Kommunikation zu vermeiden.

Beispiele für diese Architekturen wären die *CRAY T3D* von Cray Research, welche aus bis zu 2048, über ein torusförmiges dreidimensionales Netz verbundenen Prozessoren besteht, oder die *Meiko CS-2*, die über ein mehrstufiges Verbindungsnetzwerk Knoten, welche aus SPARC RISC-Prozessoren, zwei Vektor-Prozessoren und einer Speichereinheit bestehen, verbindet.

1.2. Parallele Programmierung

Die effiziente Programmierung der in 1.1. angeführten Maschinen ist zur Zeit ein vorrangiges Forschungsgebiet. Die potentielle Leistung paralleler Rechensysteme kann mit herkömmlichen Programmiermethoden nicht ausgeschöpft werden und daher erfordert eine zielführende Anwendung dieser neuen Architekturen eine zum Teil vollkommene Neuorientierung sowohl am Gebiet der Programmierung als auch am Gebiet der Compilertechnik.
Grundsätzlich wird davon ausgegangen, daß man bei Systemen mit hunderten, ja tausenden parallel arbeitenden Prozessoren die Leistungsfähigkeit nicht dadurch ausschöpfen kann, indem jeder Prozessor an einem unterschiedlichen Programm arbeitet. Dies ist einerseits nicht möglich, da normalerweise nicht derart viele Anwendungen vorhanden sind, andererseits die Architekturen aufgrund ihrer zum Teil beschränkten I/O-Fähigkeit solche Programmierungen nicht unterstützen. Deshalb hat sich ein Modell durchgesetzt, bei dem versucht wird, ein und das selbe Problem durch die parallele Arbeit der verfügbaren Prozessoren zu lösen; dieses Modell wird als ***Single Program Multiple Data*** **(SPMD)** Modell bezeichnet.

Parallele Programmiermodelle können je nach Art des zur Verfügung gestellten Parallelismus in verschiedene Ebenen eingeteilt werden, wobei [Bemm92] zwischen 6 Ebenen unterscheidet:

1) Parallelismus auf der Ebene der parallelen Verarbeitung mehrerer Operanden eines Maschinenbefehls; dieser Parallelismus wird auch von sequentiellen Maschinen zur Verfügung gestellt.
2) Parallelismus auf Instruktionsebene: hier werden mehrere Maschinenbefehle in einem Maschinenzyklus ausgeführt. Diese Möglichkeit wird bei nahezu allen modernen Prozessorentwicklungen angewandt.
3) Parallelismus auf Schleifenebene: es wird versucht, voneinander unabhängige Schleifeniterationen oder Prozedurinkarnationen parallel auszuführen, was vor allem bei numerischen Anwendungen zu guten Resultaten führt.
4) Parallelismus auf Ebene von Handlungsfäden (Threads): Handlungsfäden, die einer Ausführung einer beliebigen Folge von Maschinenbefehlen entsprechen, werden parallel ausgeführt.
5) Parallelismus auf Prozeßebene: Funktional zusammengehörige Teilaufgaben eines Programmes werden parallel ausgeführt.
6) Parallelismus auf Programmebene: Voneinander unabhängige Programme werden parallel ausgeführt (*multiprogramming*).

Eine weitere Unterscheidungsmöglichkeit des zur Verfügung gestellten Parallelismus bietet die *Granularität* oder *Körnigkeit*, welche bei den genannten 6 Ebenen von der ersten zur sechsten Ebene ständig zunimmt. Die erste und zweite Ebene wird als feinkörniger Parallelismus bezeichnet, ab der dritten Ebene spricht man von mittel- bzw. grobkörnigem Parallelismus.
Entsprechend dem SPMD-Modell wird in dieser Arbeit das Augenmerk vor allem auf den mittelkörnigen Parallelismus und hier im speziellen auf die Parallelität auf Schleifenebene gelegt.

Die größten Probleme bei der Programmierung von DMMPs liegen darin, daß der Programmierer für die Effizienz und Korrektheit der Anwendung zu sorgen hat, was er über die Festlegung der Arbeitsaufteilung (welcher Prozessor welche Berechnungen ausführt) in Hinblick auf die Optimierung der Parallelität, die Festlegung der Datenaufteilung (welcher Prozessor besitzt welche Daten in seinem lokalen Speicher) in Hinblick auf die

Minimierung der teuren Kommunikation und die entsprechende Synchronisierung der Prozessoren (einzelne Prozessoren müssen gegebenenfalls darauf warten, bis andere Prozessoren ihre Berechnungen abgeschlossen haben, um auf deren lokale Daten zugreifen zu können) in Hinblick auf die semantische Korrektheit der Anwendung erreichen muß.
Besonders die Synchronisierung eines parallelen Programmes, welche bei DMMPs über blockierende Kommunikationsroutinen erfolgt, ist eine schwierige und fehleranfällige Aufgabe, da diese nur mit sehr großem Wissen über die internen Abläufe in einem Programm korrekt erfolgen kann. Es ist daher ein großes Ziel, die Programmierung paralleler Maschinen dahingehend zu erleichtern, daß möglichst viele der angesprochenen Probleme nicht mehr vom Programmierer bearbeitet werden müssen, sondern automatisch gelöst werden und die Programmierung von Multiprozessoren ähnlich der üblichen Programmierung von sequentiellen Maschinen zu gestalten. Zu diesem Zweck wurden parallele Programmiersprachen mit entsprechenden Compilern entwickelt, die dem Programmierer gestatten, explizit parallele Konstrukte zu verwenden (z.B. forall-Schleifen), sowie die Arbeits- und Datenaufteilung mittels einfacher Routinen zu bestimmen. Weiters wird versucht, die Synchronisation und Kommunikation basierend auf der vom Programmierer angegebenen Arbeits- und Datenaufteilung automatisch zu generieren. Beispiele für derartige Programmiersprachen wären *High Performance Fortran* [HPF94], *High Performance C* [VDBF94], *Crystal* [CCL89], *Fortran D* [HKet.al.91] sowie *Vienna Fortran* [CMZ92] bzw. *Vienna Fortran 90* [Benk94].

Die für die Performance eines parallelen Programmes entscheidenden Aufgaben der Arbeits- und Datenaufteilung obliegen jedoch bei allen diesen Ansätzen weiterhin dem Programmierer!
Aus diesem Grund werden zur Zeit große Anstrengungen unternommen, diese Aufgaben zu automatisieren, bzw. den Programmierer bei der geeigneten Wahl zu unterstützen. Auch in dieser Arbeit wird eine Analysemethode entwickelt, die dem Programmierer bei dieser schwierigen Aufgabe hilfreiche Informationen bieten kann, doch soll zunächst im folgenden Abschnitt noch näher auf die Problematik der automatischen Parallelisierung eingegangen werden.

1.3. Automatische Parallelisierung

Wie bereits im vorangegangenen Abschnitt festgehalten wurde, ist die Programmierung von Multiprozessoren eine schwierige und fehleranfällige Aufgabe, weshalb versucht wird, mit Hilfe von parallelen Programmiersprachen und Compilern die parallele Programmierung zu vereinfachen. Solche Compiler (wie z. Bsp. der *D-Editor* [HKTW94] für Fortran D oder das *Vienna Fortran Compilation System* [CBBet.al.93] für Vienna Fortan) werden oft auch als *parallelisierende Compiler* bezeichnet, da sie ein in den Grundzügen sequentielles Programm automatisch in ein paralleles Äquivalent umwandeln. Diese automatische Parallelisierung erfolgt durch die automatische Generierung von Synchronisierungs- und Kommunikationsanweisungen, basierend auf den vom Benutzer/Programmierer angegebenen Arbeits- und Datenaufteilungen.[1]

Hierin liegt jedoch auch eines der größten Probleme der automatischen Parallelisierung, da die Effizienz eines parallelen Programmes in hohem Maße von der getroffenen Arbeits- und Datenaufteilung abhängt. In vielen Fällen wird von einer getrennten Spezifikation der Arbeits- und Datenaufteilung abgesehen, indem die Arbeitsaufteilung durch die Datenaufteilung bestimmt wird. Dies wird erreicht, indem festgelegt wird, daß jeder Prozessor, der ein zu schreibendes Element in seinem lokalen Speicher hat, die zugehörigen Berechnungen ausführt. - Diese Art der Arbeitsaufteilung wird üblicherweise als das ***"owner-computes-paradigm"*** bezeichnet.

Da damit durch die Datenaufteilung auch implizit die Arbeitsaufteilung festgelegt wird, steht man bei der Spezifizierung derselben vor dem Problem, die richtige Balance zwischen Datenlokalität (um Kommunikation zu vermeiden) und Parallelität (welche nur durch das Verteilen von Daten auf mehrere Prozessoren erreicht werden kann) zu finden.

Ausgehend von diesem owner-computes-paradigm (das auch manchmal als "compute-iff-owned"-Regel bezeichnet wird), soll nun ein vertiefender Einblick in die Problematik der Datenaufteilung gegeben werden.

1.4. Datenaufteilung

Für diesen Abschnitt, wie auch im restlichen Verlauf dieser Arbeit, wird ein Sprachenmodell verwendet, welches eine Erweiterung von Fortran 77 darstellt. Diese Erweiterungen umfassen die Möglichkeit der Spezifikation von Prozessorenarrays und von Verteilungen:

---

[1] Nähere Details zur Arbeitsweise von parallelisierenden Compilern werden in Kapitel 2 im Rahmen der Einführung in das VFCS, eines semiautomatischen parallelisierenden Compilers, vorgestellt.

- Die Spezifikation von Prozessorenarrays erlaubt die Definition einer Menge (abstrakter) Prozessoren, welche das parallele Programm ausführen. Die Kommunikation zwischen zwei Prozessoren erfolgt mittels message passing, welches jedoch die tatsächliche Topologie des Rechners außer Acht läßt, was zu einem abstrakten Rechnermodell führt, das die Distanzen zwischen zwei kommunizierenden Prozessoren als konstant annimmt.
- Verteilungen weisen Arrays nichtleere Mengen von Prozessoren zu und können als Beifügungen der Datendeklarationen angegeben werden.

Diese hier genutzten Sprachspezifika werden zum Beispiel auch von *Vienna Fortran* [CMZ92], neben weiteren parallelen Konstrukten, angeboten, wodurch Vienna Fortran als Übermenge der hier angewandten Sprache angesehen werden kann.

Zur Modellierung der Aufteilung von Programmdaten (im speziellen beliebig dimensionierter Arrays) auf die zur Verfügung stehenden Prozessoren wird angenommen, daß die Prozessoren virtuell in einem beliebig dimensionierten Array darstellbar sind. Die Datenaufteilung kann sodann als Funktion aufgefaßt werden, die den Indexbereich von beliebig dimensionierten Arrays einer Potenzmenge des Prozessorenarray-Indexbereiches zuordnet, was gemäß [ZiChap92] folgendermaßen definiert wird:

Definition 1.1: Seien **I** und **J** zwei Indexbereiche. Eine **Indexzuordnung** von **I** nach **J** ist eine totale Funktion $\iota : \mathbf{I} \rightarrow P(\mathbf{J}) - \{\emptyset\}$, wobei $P(\mathbf{J})$ eine Potenzmenge von **J** darstellt ▪

Definition 1.2: Sei A ein Array und R ein Prozessorarray. Eine Indexzuordnung δ_R^A von $\mathbf{I}^A$ nach $\mathbf{J}^R$ wird **Aufteilung** von A auf R genannt ▪

Definition 1.3: Sei $\mathcal{E}^A$ die Menge der Elemente von Array A. Die totale Funktion $\lambda^A : R \rightarrow P(\mathcal{E}^A)$ bestimmt für jeden Prozessor $r \in R$ die Menge der Elemente von Array A, welche dieser in seinem lokalen Speicher besitzt.
$\lambda^A(r)$ wird das **Distributionssegment** von Array A auf Prozessor r genannt ▪

Natürlich kann Definition 1.2 auch dergestalt verstanden werden, daß die Aufteilungsfunktion δ_R^A für jedes Element des Arrays A eine nichtleere Menge von Prozessoren bestimmt, welche dieses Element ihrem lokalen Speicher zugeordnet bekommen.

Die in der Praxis häufigsten Aufteilungsfunktionen sind:

- Die **(partielle) Replikation**: Hierbei werden die Elemente eines Arrays mehreren Prozessoren zugewiesen, also repliziert; üblicherweise wird dabei eine Dimension eines Arrays allen Prozessoren einer Prozessorarraydimension zugeordnet. Wird ein Array allen Prozessoren zugewiesen, d.h. $\delta_R^A(\mathbf{i}) = R$ für alle $\mathbf{i} \in \mathbf{I}^A$, so spricht man von **vollständiger Replikation**.
- Die **Blockaufteilung**: Bei dieser Aufteilungsvariante wird eine Arraydimension in gleich großen Teilen auf die unterschiedlichen Prozessoren einer Prozessorarraydimension distributiert.
- Die **Zyklische Aufteilung**: Hierbei wird jedes Element einzeln in zyklischer Form dem Prozessorarray zugeordnet. Das heißt, wenn das Arrayelement a(i) dem Prozessor p(j) zugeordnet wird, so befindet sich das Arrayelement a(i+1) im Speicher von Prozessor p(j+1).
- Die **Block-Zyklische Aufteilung**: Diese Aufteilungsvariante ist die Generalisierung der Block- und Zyklischen Aufteilung, die im engeren Sinn nur die beiden Extremfälle der Block-Zyklischen Aufteilung darstellen. Im Gegensatz zur Zyklischen Aufteilung, die immer nur ein Element einem Prozessor zyklisch zuordnet, werden hier die Elemente in variablen Blockgrößen zyklisch aufgeteilt. Entspricht diese Blockgröße dem Quotienten aus dem Indexbereich der Arraydimension und dem Indexbereich der Prozessorarraydimension, so erhält man eine Blockaufteilung, ist diese Blockgröße gleich 1, spricht man von einer Zyklischen Aufteilung.

Im nachfolgenden Beispiel 1.1 sind exemplarische Beispiele für die genannten Aufteilungsfunktionen dargestellt, wobei der PROCESSORS-Befehl das Prozessorenarray deklariert und die Anweisung DIST für jedes Array die Aufteilungsvariante angibt. In Abbildung 1.1 werden diese Aufteilungsvarianten visualisiert.

Beispiel 1.1 - Aufteilungsfunktionen

a) Vollständige Replikation:

```
PROCESSORS R(4)
REAL A(12) DIST (:) TO R          /* ":" wird für Replikation verwendet*/
```

Die Aufteilungsfunktion lautet: $\delta_R^A(i) = R$

b) Blockaufteilung:

```
PROCESSORS R(4)
REAL A(12) DIST (BLOCK) TO R
```

Die Aufteilungsfunktion lautet: $\delta_R^A(i) = \left\{\left\lceil \frac{i}{3} \right\rceil\right\}$

c) Blockaufteilung und partielle Replikation:

```
PROCESSORS R(4,4)
REAL A(12,12) DIST (BLOCK,:) TO R
```

Die Aufteilungsfunktion lautet: $\delta_R^A(i,j) = \left\{\left\lceil \frac{i}{3} \right\rceil\right\}, R$

d) Zyklische Aufteilung:

```
PROCESSORS R(4)
REAL A(12) DIST (CYCLIC) TO R
```

Die Aufteilungsfunktion lautet: $\delta_R^A(i) = \left\{\mathrm{MODULO}\left(i-1,4\right)+1\right\}$

e) Block-Zyklische Aufteilung:

```
PROCESSORS R(4)
REAL A(16) DIST (BLOCKCYCLIC(2)) TO R
```

Die Aufteilungsfunktion lautet: $\delta_R^A(i) = \left\{\mathrm{MODULO}\left(\left\lceil \frac{i}{2} \right\rceil - 1,4\right)+1\right\}$

Graphisch können die Aufteilungsfunktionen von Beispiel 1.1 folgendermaßen dargestellt werden:

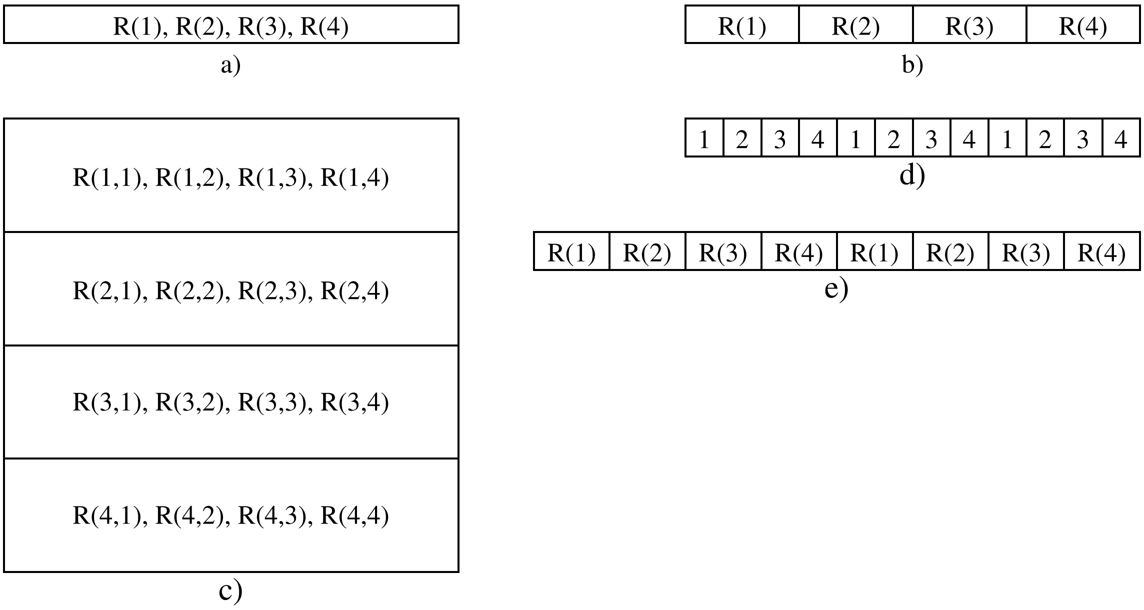


Abb. 1.1 - Datenaufteilungen zu Bsp. 1.1

In Hinblick auf die Auswirkungen der angeführten Aufteilungsvarianten kann festgehalten werden, daß bei vollständiger Replikation zwar keine Kommunikation anfallen kann, jedoch auch kein Parallelismus vorhanden ist. Die Blockaufteilung hat den Vorteil, daß sie sehr einfach zu realisieren ist und bei Anwendungen, deren Berechnungen gleichmäßig über ganze Arrays fortschreiten, auch genügend Parallelismus bereit stellt. Finden hingegen die Berechnungen nur auf kleinen Teilen eines Arrays statt, so kann mit einer Blockaufteilung unter Umständen kein Parallelismus erzielt werden, weshalb in solchen Fällen oft zyklische Aufteilungsvarianten angewandt werden. Diese haben jedoch den Nachteil, daß sie schwierig und zeitaufwendig zu realisieren sind, weshalb im Normalfall Blockaufteilungen bevorzugt angewandt werden sollten. (Mit der Komplexität zyklischer Aufteilungsvarianten beschäftigt sich beispielsweise [Benk94a].) Um jedoch den möglicherweise auftretenden Mangel an Parallelismus beseitigen zu können, wurde in Vienna Fortran eine Generalisierung der Blockaufteilung, die *general block distribution* [CBBet.al.93] entwickelt, die eine Spezifizierung der Anzahl der Blöcke wie auch der individuellen Größe eines jeden Blockes ermöglicht.

Auf der Basis der angeführten Möglichkeiten zur Datenaufteilung wird nun anhand des Beispiels der Matrixmultiplikation die Performanceauswirkung der Datenaufteilung, unter Annahme des owner-computes-Paradigmas, kurz angerissen:

Beispiel 1.2 - Matrixmultiplikation

```
...
PROCESSORS R(1:N)
REAL a(N,N), c(N,N) DIST (:,BLOCK) TO R
REAL b(N,N) DIST (BLOCK,:) TO R
INTEGER i,j,k
...
DO 10 i=1,N
  DO 10 j=1,N
        DO 10 k=1,N
              c(i,j)=c(i,j)+a(i,k)*b(k,j)
10 CONTINUE
...
```

Im angeführten Beispiel wird eine Matrixmultiplikation der Form c = a * b durchgeführt, wobei es sich jeweils um N x N Matrizen handelt. (Die im Beispiel angeführten Datenaufteilungen werden für N=4 in Abb. 1.2. dargestellt.) Da die Matrizen c und a spaltenweise, die Matrix b jedoch zeilenweise aufgeteilt wurde, muß jeder Prozessor I ∈ R alle anderen Prozessoren ansprechen, um die I-te Zeile von a zu bekommen; dann müssen in weiterer Folge wiederum alle anderen Prozessoren angesprochen werden, um die J-te Spalte von b zu bekommen. Werden jedoch die Matrizen c und a zeilenweise und die Matrix b spaltenweise aufgeteilt, so müssen nur die Spalten von b allen Prozessoren zugesandt werden, was in N Kommunikationsschritten bewerkstelligt werden kann. Wählt man weiters für Matrix c eine Blockaufteilung in beiden Dimensionen, so ist keinerlei Kommunikation nötig.

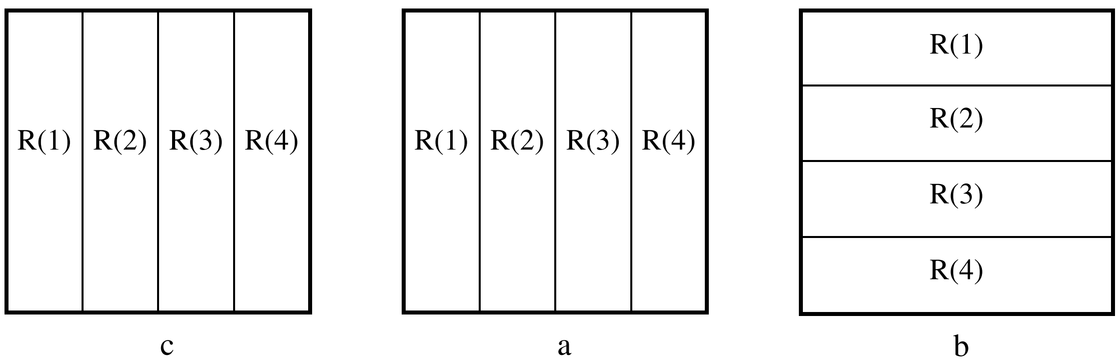


Abb. 1.2. - Datenaufteilungen zu Bsp. 1.2

Bereits aus diesem einfachen Beispiel ist ersichtlich, welchen eminenten Einfluß die richtige Wahl der Datenaufteilung auf die letztendlich erzielbare Performance eines parallelen Programmes hat. In manchen Fällen ist es sogar möglich, daß die durch die Verwendung mehrerer Prozessoren erzielte Leistungssteigerung durch eine falsch gewählte Datenaufteilung (und dem daraus resultierenden Kommunikationsaufwand) wieder zunichte gemacht wird. Zwar ist es möglich, durch die Anwendung verschiedener Strategien den Kommunikatiosaufwand zu optimieren (dies wird in 2.2.3 näher behandelt), doch ist das Vermeiden von unnötiger Kommunikation letztlich jeder Optimierung vorzuziehen.

Da jedoch, wie bereits erwähnt, neben dem Vermeiden von Kommunikation auch das Ausnützen eines möglichst hohen Grades an Parallelität wünschenswert ist, muß bei der Spezifizierung der Datenaufteilung auf beide Gesichtspunkte Rücksicht genommen werden. Diese schwierige Aufgabe liegt jedoch bei heutigen Parallelisierungssystemen beim Benutzer und eine optimale Wahl der Datenaufteilung ist für diesen, speziell bei größeren Programmen, nahezu unmöglich. Es wäre daher anzustreben, auch die Datenaufteilung im Kontext eines Parallelisierungssystems automatisch vorzunehmen, was jedoch, unter anderem in [Krem93] als der Klasse der NP-vollständigen Probleme zugehörig bewiesen wurde, weshalb versucht wird, dem Benutzer möglichst große Unterstützung bei dieser Aufgabe zuteil werden zu lassen. Ziel solcher Systeme wäre die automatische Generierung von Datenaufteilungsanweisungen bzw. von Vorschlägen einiger möglicher Varianten.

Diese Arbeit präsentiert zu diesem Zwecke einen heuristischen Ansatz, der im allgemeinen als das **Alignmentproblem** bekannt ist. Dieses Problem wurde erstmalig von [LiChen89] als das Problem des Auffindens von im Programmverlauf gemeinsam genutzten Arraydimensionen (diese Beziehungen zweier Arraydimensionen wird als Alignmentrelation bezeichnet) formuliert. Der große Nutzen, der aus solchen Informationen gezogen werden kann, ist die Möglichkeit, die Datenaufteilungsfunktionen dieser beiden Arraydimensionen zu vereinheitlichen, wodurch einerseits Kommunikationskosten eingespart werden können, andererseits nur mehr für wenige Arrays Datenaufteilungsfunktionen spezifiziert werden müssen, da die restlichen durch Alignmentrelationen impliziert werden.
Li et. al. versuchen auch die Bedeutung von Alignmentrelationen für das Performanceverhalten festzustellen, wozu sie jedoch sehr vereinfachende Annahmen treffen (vgl. Abschnitt 5.1.). In [Gupta92] wurden die Arbeiten von Li et. al. aufgegriffen und erweitert, was seinen Niederschlag vor allem in der verbesserten Gewichtung der

Alignmentrelationen in Hinblick auf deren Performanceauswirkungen findet. Gupta versucht erstmals, Kommunikationskosten abzuschätzen und dadurch die Bedeutung von Alignmentrelationen widerzuspiegeln (vgl. Abschnitt 5.2.).
In der Folge wurden diese Arbeiten als Ausgangsbasis für weiterführende Forschungen am Gebiet der automatischen Datenaufteilung herangezogen, weshalb in Kapitel 5 weitere Methoden und Forschungsansätze vorgestellt werden, welche versuchen, die Spezifizierung der Datenaufteilung zu vereinfachen bzw. zu automatisieren.

Der hier vorgestellte Ansatz versucht auf Basis der Arbeiten von Li et. al. und Gupta das Alignmentproblem in einer für reale Programme anwendbaren Art und Weise zu lösen. Die genannten Arbeiten werden zu diesem Zwecke abgewandelt und erweitert, wobei ein Schwergewicht auf das Auffinden von Alignmentrelationen gelegt und dadurch eine signifikante Steigerung der erkennbaren Alignmentrelationen erreicht wird. Als weiteres Hauptgewicht kann die Erreichung einer großen Flexibilität im Analyseverfahren genannt werden, wodurch Praxisnähe und die Anwendbarkeit der Methoden für reale Programme gewährleistet werden kann. Diese Flexibilität zeigt ihren Ausschlag auch in der Gewichtung der Alignmentrelationen, zu welcher einfache, aber wirkungsvolle Abschätzungen von Kommunikationskosten vorgenommen werden.
Auch für jene Fälle, in denen die klassische Alignmentanalyse keine Lösungen anbieten kann (dies ist vor allem der Fall, wenn im Programmverlauf hauptsächlich eindimensionale Arrays verwendet werden), kann durch die Methoden dieser Arbeit ein gewisses Maß an Information über anzuwendende Datenaufteilungsfunktionen erlangt werden.

Die Alignmentinformationen, welche durch die vorzustellenden Analyseverfahren erlangt werden, kann der Benutzer in seiner Spezifikation der Datenaufteilung entsprechend umsetzen. Es wäre jedoch auch möglich, automatisch "Align"-Anweisungen (wie sie von HPF [HPF94] oder Vienna Fortran 90 [Benk94] zur Verfügung gestellt werden) zu generieren, oder diese Informationen auch in weitere Analysesysteme einfließen zu lassen.

Bevor jedoch direkt auf das Alignmentproblem eingegangen werden wird, soll zunächst der parallelisierende Compiler *VFCS* vorgestellt werden, da in dieser Arbeit das Alignmentproblem im Rahmen dieses Compilers definiert und dessen Funktionalität für die Lösung des Alignmentproblems herangezogen wird.

2. DAS VFCS

Das Vienna Fortran Compilation System ist ein semi-automatischer Parallelisierer für in Vienna Fortran, einer Erweiterung von Fortran 90 bzw. High Performance Fortran (HPF) [HPF94] verfaßte Programme. Diese Erweiterungen umfassen vor allem die Spezifikation der Datenaufteilung, explizit parallele Forall-loops sowie die Möglichkeit des expliziten Alignments zweier Variablen.[CMZ92]

Weiters bietet das VFCS dem Benutzer die Möglichkeit, Programmtransformationen wie Scalar Forward Substitution, Constant Propagation, Dead Code Elimination, Loop Distribution, Loop Unrolling, Loop Interchange etc. [ZiChap90] durchzuführen und so den erzielbaren Grad an Parallelität zu erhöhen. In Hinblick auf die Alignmentanalyse stellen vor allem die (auch automatisch ausführbaren) Transformationen *Scalar Forward Substiution* und *Induction Variable Substitution* notwendige Voraussetzungen dar, da nach Anwendung dieser Transformationen eine wesentlich detailiertere Analyse möglich ist. In Abschnitt 4.4 wird darauf noch näher eingegangen werden.

Die Parallelisierung des Eingabeprogrammes erfolgt automatisch auf Basis der vom Benutzer spezifizierten Aufteilungsfunktionen, wobei die Arbeitsaufteilung gemäß dem *owner-computes-paradigm* [ZiChap92] erfolgt. Der Kommunikationsaufwand des resultierenden Programmes wird durch Verfahren zur Kommunikations- und Maskenoptimierung [Gern89],[Gern91] verbessert.

Die folgende detailierte Vorstellung des VFCS ist in diesem Rahmen vor allem deshalb vonnöten, da der Definition des Alignmentproblems ein Compilermodell und eine Entwicklungsumgebung zugrunde liegen, welche jenen des VFCS angeglichen wurden. Auch wurde ein Prototyp zur Durchführung der Alignmentanalyse in das VFCS integriert, mit welchem die in Kapitel 4 präsentierten Testdaten ermittelt wurden.

Eine schematische Darstellung des VFCS ist Abb. 2.1 zu entnehmen:

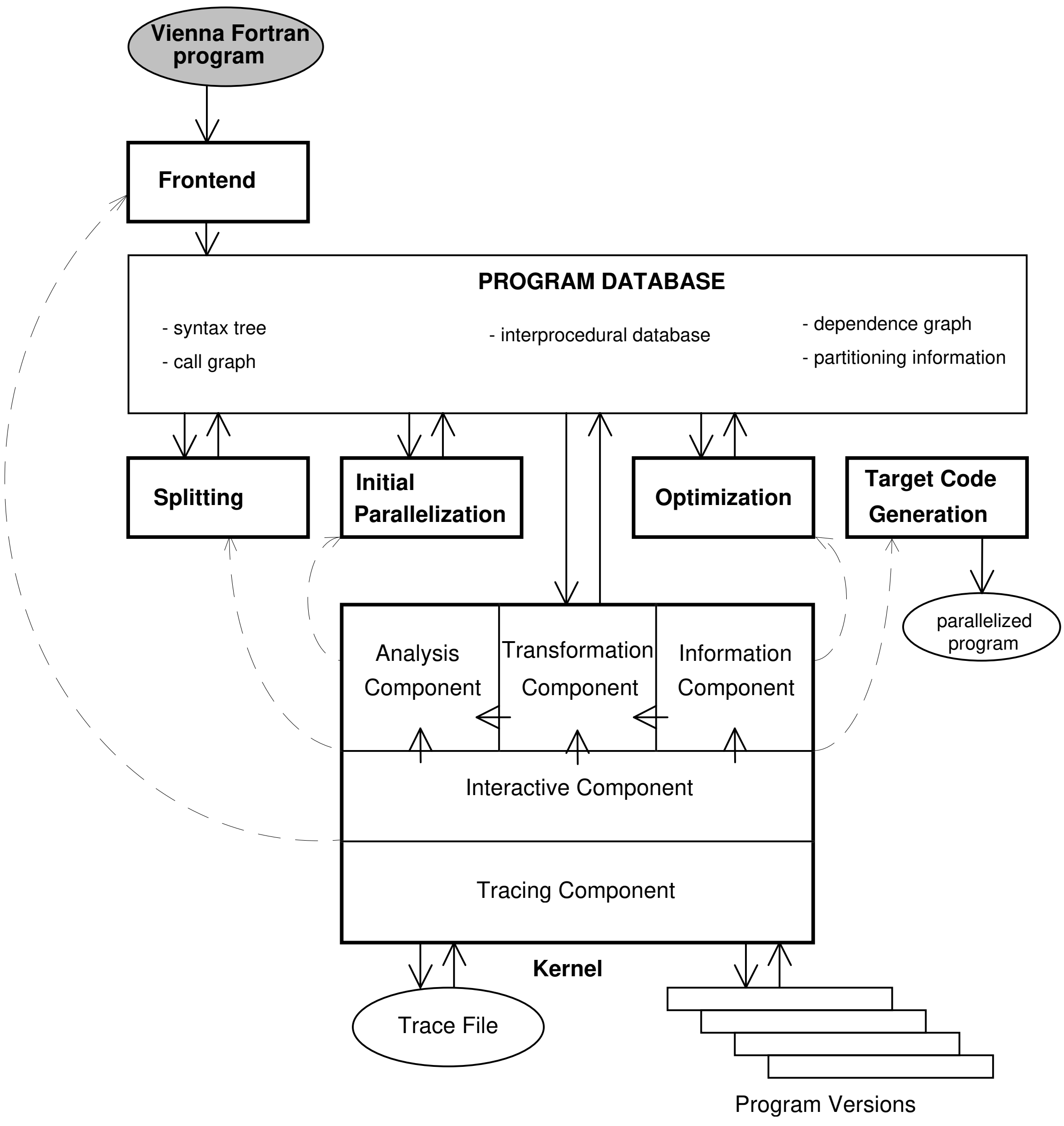


Abb. 2.1 VFCS [ZiChap92]

2.1. Der Parallelisierungsvorgang

Der Parallelisierungsvorgang erfolgt in 4 unterschiedlichen Phasen, in denen das Quellenprogramm (Q^0) (in Vienna Fortran, VF) in ein Fortran Programm (Q^4) mit expliziten message-passing Konstrukten (Message Passing Fortran, MPF) für die jeweilige zu spezifizierende Zielmaschine (z.B. iPSC 860, Meiko CS-2, etc.) übersetzt wird [ZiChap92].

1. Phase (Front End):

VF Source Program (Q^0) → VF Normalized Program (Q^1)

Diese Phase erfüllt 3 Aufgaben: Die Transformation des Quellenprogramms in eine interne Darstellung für weitere Compilermanipulationen, die Analyse und die Normalisierung des Programms. Diese Vorgänge beinhalten die syntaktische und semantische Analyse, die Kontrollfluß-, Datenfluß- und Abhängigkeitsanalyse sowie die Konstruktion des Aufrufgraphen. Transformationen wie die interprozedurale constant propagation und dead code elimination werden durchgeführt.

Für die Alignmentanalyse wichtige Transformationen, die ebenfalls in dieser Phase durchgeführt werden, sind die Do-Loop-Normalization, welche bewirkt, daß jede Schleife eine Schrittweite von 1 erhält, und die If-Conversion, die Kontrollabhängigkeiten in Datenabhängigkeiten umwandelt und bewirkt, daß in jedem If-Zweig nur eine einzige Anweisung steht.

2. Phase (Splitting):

VF Normalized Program (Q^1) → Host Program (Q^H) und Node Program (Q^2)

In dieser Phase wird das Programm in ein Host Program und ein Node Program geteilt, wobei das Host Program am Host-Rechner ausgeführt wird und die globalen Managementaufgaben übernimmt (wie z.B. Ressourcenzuteilung, I/O-Handling etc.)

Das Node Program beinhaltet die eigentlichen Berechnungen, I/O wird als Kommunikation mit dem Host-Rechner gekennzeichnet.

3. Phase (Initial Parallelization):

Node Program (Q^2) → Defining Parallel Program (Q^3)

In dieser Phase wird aufgrund der im VF-Programm angegebenen Datenaufteilung die Arbeitsaufteilung (mittels Maskierung) sowie das Einfügen der Kommunikationsroutinen durchgeführt. Die Maskierung verfolgt das "compute iff owend"-Prinzip mittels Einfügung eines boolean guard, der Maske. Für alle nicht lokalen Datenzugriffe werden geeignete Kommunikationskommandos eingefügt.

Da jeder Knoten das selbe Programm ausführt (SPMD-Modell), wird nur ein Knotenprogramm generiert, wobei jeder Knoten das Programm auf seinem, durch die Datenaufteilung spezifizierten Datenbereich durchführt. Durch die einfache Vorgangsweise bei der Kommunikationsgenerierung ist die Performance des erzielten, bereits parallelen Programmes meist sehr schlecht, wodurch die in der letzten Phase vorgenommenen Optimierungen notwendig sind:

4. Phase (Optimierung und Zielcodegenerierung):
Defining Parallel Program (Q^3) → Optimized Target Program (Q^4)

In dieser Optimierungsphase wird die Kommunikation und die Maskierung optimiert:
Kommunikationsanweisungen werden gebündelt und wenn möglich, aus Schleifen herausgezogen, die Information der Masken wird in die Schleifengrenzen gezogen. Grundlage für diese Optimierungen bilden die in Phase 1 durchgeführten Analysen.
Sodann wird der Zielcode für die jeweilige zu spezifizierende Zielmaschine generiert.

2.2. Die Entwicklungsumgebung

Zur Durchführung der Parallelisierung stellt das VFCS eine integrierte graphische Oberfläche zur Verfügung, die dem Benutzer eine Vielzahl von Informationen über das zu parallelisierende Programm liefert (wie z.B. Abhängigkeitsinformationen, Basisblöcke, Use-Definition-Ketten etc.) und die Möglichkeit bietet, Programmtransformationen (wie bereits erwähnt) automatisch durchzuführen.

Daneben beinhaltet das System des VFCS noch umfangreiche Analysetools, welche sich hauptsächlich mit der Performance-Prediction beschäftigen, wie z. B. den *Weight Finder* [Fahr93], einen Profiler für sequentielle Fortran 77 Programme, dessen Ergebnisse auch direkt in die Alignmentanalyse einfließen, weshalb eine nähere Beschreibung desselben noch folgen wird, oder das *P^3T* [Fahr94], ein Tool zur Analyse des Laufzeitverhaltens paralleler Programme zur Compilezeit.

Neben den Performance-Prediction Tools ist ein weiterer Forschungsschwerpunkt die Automatisierung des Parallelisierungsvorganges [CHZ91],[CFZ93], wobei versucht wird, neben der automatischen Generierung entsprechender Datenaufteilungsfunktionen auch performanceverbessernde Transformationen automatisch durchzuführen. Die Basis hierfür stellt, neben dem Ausschöpfen von analytischen Möglichkeiten, ein expertensystembasiertes Tool, der *ExpertAdviser* [HACZ93], dar, über dessen Entwicklungsstand [ACHZ95] Auskunft gibt.

Nachfolgend wird ein kurzer Einblick in jene Teile des Compilers gegeben, die Einfluß auf das Alignmentanalysetool nehmen, worunter im Speziellen der Weight Finder, die Abhängigkeitsanalyse sowie die Kommunikationsoptimierung fallen:

2.2.1. Der Weight Finder

Der Weight Finder [Fahr93] ist ein eigens für das VFCS entwickeltes Profilierungstool, welches die Aufgabe hat, runtime intensive Programmteile eines Fortran 77 Programmes zu entdecken. Als Input erhält der Weight Finder das normalisierte Programm, welches bereits das Front-End des VFCS durchlaufen hat. Mittels eines Probelaufes des sequentiellen Programmes (wofür der Benutzer entsprechend aussagekräftige Inputdaten spezifizieren muß) können folgende, selektiv wählbare, Parameter bestimmt werden:

* Anweisungsfrequenz: diese gibt an, wie oft eine Anweisung während eines Programmlaufes exekutiert wird.

* TRUE-Anteil von if-Anweisungen (*True ratio)*:
  diese gibt die Wahrscheinlichkeit an, daß die Bedingung eines if-statements zu TRUE evaluiert.

* Dynamischer outcome von if-expressions:
  Der dynamische outcome der if-expressions jeder Bedingungsanweisung wird erhalten.

* Schleifenparameter:
  Die durchschnittlichen Schleifenparameter (obere und untere Grenze) werden bestimmt.

* Zeitanteil (*profile time*): Zeitbedarf einer Schleife, einer Prozedur oder des gesamten Programms.

Um diese Parameter bestimmen zu können, werden 6 Phasen durchlaufen:

1. Front End:

Zuerst durchläuft das Fortran 77 Programm das VFCS Front End, welches wie in 2.1. beschrieben, intra- und interprozedurale Analysen, Normalisierung und

Standardisierung umfaßt. Eine Kopie des dabei erzielten Programmes wird für die weitere Bearbeitung in Phase 6 gespeichert.

2. Parameterauswahl:

In dieser Phase kann der Benutzer die zu berechnenden Parameter, wie oben beschrieben, auswählen.

3. Instrumentalisierung:

Soll für ein bestimmtes Statement S ein Parameter errechnet werden, so wird S ein *instrumentalisiertes Statement* genannt. Das heißt, die relevanten Statements werden für den weiteren Verlauf markiert.

4. Optimierung:

Durch die Instrumentalisierung entsteht ein großer overhead, welcher unter Umständen eine weitere Profilierung unmöglich macht. Beispielsweise werden zur Häufigkeitsmessung von Basisblöcken alle Basisblöcke herangezogen, auch wenn sie aufgrund der Programmstruktur die gleiche Häufigkeit aufweisen müssen. Ziel dieser Optimierungsphase ist es daher, diesen Instrumentalisierungs-overhead abzubauen.

5. Compilierung und Exekution:

Das nach der Optimierung erhaltene Programm wird compiliert und exekutiert, wobei die notwendigen Messungen vorgenommen werden. Die während dieses Programmlaufes erhaltenen Daten werden in eigene *profile files* abgespeichert.

6. Attributierung und Visualisierung:

Die in Phase 2 gespeicherte Kopie des ursprünglichen Programms wird mit den in Phase 5 errechneten Parametern attributiert (sie werden an den Flowgraphen des Programms angefügt); diese Parameter werden auch am Bildschirm visualisiert.

Der Weight Finder ist dergestalt konzipiert, daß nachträglich durchgeführte Programmtransformationen keinen nennenswerten Einfluß auf die erzielten Resultate haben; ein neuerlicher Durchlauf des Weight Finders ist somit auch nach nachträglichen Programmtransformationen nicht notwendig.

2.2.2. Die Abhängigkeitsanalyse

Die Abhängigkeitsanalyse im VFCS-Compiler [Grin87][CBBet.al.93][ZiChap90] stellt für jede Anweisung des untersuchten Programmes fest, welche Arrayreferenzen

Abhängigkeiten verursachen. Es werden weiters die Arten der Abhängigkeit (*true*, *anti* oder *output*), der Abhängigkeitslevel (0 für schleifenunabhängige Abhängigkeiten, >0 für schleifengetragene Abhängigkeiten) und jene Anweisung, welche die zweite abhängigkeitsverursachende Arrayreferenz aufweist, berechnet.
Die relevanten Anweisungspaare werden mittels eines hierarchischen Tests auf Abhängigkeit untersucht:

Zuerst wird der *seperability test* durchgeführt. Dieser Test liefert eine hinreichende und notwendige Bedingung für das Vorliegen von Abhängigkeiten, ist jedoch nur anwendbar, wenn in den untersuchten Arrayindizes nicht mehr als eine Schleifenvariable verwendet wird.
Konnte durch den seperability test das Vorliegen von Abhängigkeiten nicht bewiesen werden, wird der *gcd test* durchgeführt. Dieser Test liefert eine notwendige Bedingung für das Vorliegen von Abhängigkeiten, ignoriert jedoch die Beschränktheit des Iterationsraumes, weshalb der gcd test zwar das Nichtvorliegen von Abhängigkeiten beweisen kann, bei Vorliegen von Abhängigkeiten jedoch um den *Banerjee test* ergänzt werden muß.
Der Banerjee test liefert ebenfalls nur eine notwendige Bedingung für das Vorliegen von Abhängigkeiten, beachtet jedoch den Iterationsraum. Die exakten mathematischen Formulierungen der verwendeten Testmethoden finden sich in [ZiChap90].

2.2.3. Die Kommunikationsoptimierung

In der dritten Phase des Parallelisierungsvorganges, der Initial Parallelization (vgl. 2.1.), werden Masken, welche das owner-compute-paradigm umsetzen, vor jede Anweisung und einfache Kommunikationsroutinen vor jedes Zuweisungsstatement, dessen Element auf der linken Seite nicht repliziert wurde, eingefügt. [CBBet.al.93][ZiChap92]

Diese Transformationen würden zwar ausreichen, um das Programm korrekt auf Parallelrechnern auszuführen, die Laufzeit wird jedoch in den meisten Fällen bedeutend länger sein, als bei sequentieller Ausführung, da sowohl für jede Anweisung die zugehörige Maske evaluiert werden muß, als auch die Kommunikationsroutinen sehr ineffizient sind, weil auch sie bei jeder Schleifeniteration ausgewertet werden müssen und nur jeweils ein Element pro Kommunikationsschritt übersenden.

Um dennoch befriedigende Ergebnisse am Parallelrechner erreichen zu können, müssen weitere Transformationen angewandt werden, die den parallelen Programmcode optimieren. Diese Optimierungen werden in der 4. Phase des Parallelisierungsvorganges, der Optimierung und Zielcodegenerierung, vorgenommen und umfassen die *mask optimization* und die *communication optimization*.

Die *mask optimization* [Gern91] optimiert die Maskierung der Anweisungen durch zwei Transformationen:

In der *iteration elimination* wird festgestellt, ob es Schleifeniterationen gibt, bei welchen alle Masken des Schleifenrumpfes zu *false* evaluieren; solche Iterationen können eliminiert werden. Im Idealfall führt dies dazu, daß eine Schleife komplett aufgelöst wird, indem jeder Prozessor eine einzige Iteration durchführt.
Die zweite Transformation ist die *mask simplification*. Hierbei werden Masken, welche für alle Schleifeniterationen zu *true* evaluieren, durch *always* ersetzt. Dadurch wird einerseits erreicht, daß Masken nicht mehr ausgewertet werden müssen, andererseits wird die Generierung von Vektoranweisungen erleichtert.

In der *communication optimization* [Gern89] wird versucht, die Kommunikationsroutinen dahingehend zu optimieren, daß ein Blocken von zu übertragenden Daten möglich wird und die Kommunikationsroutinen aus den Schleifenkörpern herausgezogen werden.

Ein hiezu dienliches Konzept stellt die *overlap analysis* dar. Dieses Konzept bedient sich der Lokalität der Berechnungen und berechnet für jeden Prozessor p und jedes partitionierte array A mit $\lambda^A(p)$ als entsprechendem Distributionssegment die kleinste zusammenhängende Umgebung von $\lambda^A(p)$, welche alle nichtlokalen Datenzugriffe beinhaltet. Diese Umgebung wird *overlap area*, *OA(A,p)*, genannt. Die OA stellt eine Übermenge der tatsächlich benötigten Daten dar. Mit Hilfe dieser overlap analysis können die Kommunikationsroutinen vereinfacht werden, indem durch die jeweiligen OAs festgestellt werden kann, welche Elemente ein Prozessor von anderen benötigen könnte und welche Elemente in seinem privaten Bereich verbleiben (d.h., die Elemente werden von keinem anderen Prozessor benötigt).

Auf Basis des Konzeptes der overlap analysis können die eigentlichen Kommunikationsoptimierungen vorgenommen werden: die *Vektorisierung von Kommunikationsanweisungen* und die *Fusion von Kommunikationsanweisungen*.

Die Vektorisierung von Kommunikationsanweisungen erfolgt ähnlich der Vorgangsweise bei Vektorisierungen von Schleifen [ZiChap90] und hat das Ziel, Datenvektoren anstelle von einzelnen Daten zu übertragen. Eine Kommunikationsanweisung kann vektorisiert werden, wenn

a) sie die einzige Anweisung im Schleifennest ist.
b) die zu übertragenden Arrayelemente durch eine Arraysektion beschrieben werden können.
c) im Falle des Vorliegens einer reinen Sende- bzw. Empfangsanweisung die korrespondierende Anweisung ebenfalls vektorisiert werden kann.

Die Anforderung a) kann durch *loop distribution* [ZiChap90] erreicht werden; die Anforderung b) entspricht der Berechnung der overlap area.

Die Fusion von Kommunikationsanweisungen verfolgt das Ziel, mehrere Kommunikationsanweisungen zu einer einzigen zusammenzufassen und so vor allem den Zeitaufwand, welcher zur Einleitung der Datenübertragung notwendig ist, zu reduzieren. Dies kann bei Kommunikationsanweisungen, welche das gleiche Array betreffen, durch die Kombination der entsprechenden overlap areas erfolgen. Dieser Ansatz ist jedoch nicht immer vorteilhaft, da einerseits die zu übertragende Datenmenge explodieren kann und andererseits unter Umständen Kommunikationsanweisungen zwischen zwei Prozessen eingefügt werden müssen, welche vorher nicht kommunizieren mußten.
Betreffen Kommunikationsanweisungen unterschiedliche Arrays, so können die benötigten Elemente mittels eigener Kommunikationsdeskriptoren ermittelt und zu einer einzigen Nachricht zusammengestellt werden.

Nachdem nun ein kurzer Einblick in den der Alignmentanalyse zugrundeliegenden Compiler gegeben wurde und auch die direkt in die Analyse einfließenden Komponenten vorgestellt wurden, kann nunmehr das Alignmentproblem definiert werden.

3. DEFINITION DES ALIGNMENTPROBLEMS

Das Alignmentproblem stellt sich als das Problem der Minimierung des Kommunikationsaufwandes, welcher durch cross-referenzen zwischen Arraydimensionen entstehen kann, durch Überführen dieser Arraydimensionen in eine gemeinsame, entsprechend zu distributierende Dimension.

Gemäß dem *owner-computes-paradigm* führt jener Prozessor die Berechnung eines Zuweisungsstatements aus, welcher das zu schreibende Element in seinem lokalen Speicher hat. Sind nun die Daten, auf welche in der rechten Seite des Zuweisungsstatements zugegriffen wird, nicht in dessen lokalem Speicher vorhanden, so ist Kommunikation notwendig. Das Dimension-Alignment-Problem kann somit als das auf Analysen der Zugriffsmuster basierende Erkennen von gemeinsam gebrauchten Arraydimensionen (*interdimensionales Alignment*) sowie der relativen Position der gemeinsam genutzten Arrayelemente (*intradimensionales Alignment*) formuliert werden. Durch das Erkennen dieser Alignmentrelationen ist es möglich, Dimensionen mehrerer Arrays mittels einer einzigen, für alle diese Dimensionen in Anwendung zu bringenden Aufteilungsfunktion zu distributieren. Der Hauptvorteil des Alignments liegt also darin, daß durch die Koppelung von Arrays implizite Aufteilungen in Relation zu einem Basisarray generiert werden und dadurch nur mehr für eine wesentlich kleinere Zahl von Arrays explizite Aufteilungsfunktionen spezifiziert werden müssen, was den Lösungsraum des Datenaufteilungsproblems signifikant verkleinert. [LiChen89][Gupta92] [KenKrem94]

Im hier verwendeten Ansatz zur Modellierung und Lösung des Alignmentsproblems wird hauptsächlich auf die Arbeiten von [LiChen89] und [Gupta92] (vgl. Kapitel 5) zurückgegriffen, deren Ansätze erweitert und modifiziert werden. Die Erweiterungen umfassen vor allem die Möglichkeiten zur Erkennung von Alignmentrelationen sowie die Modellierung des intradimensionalen Alignments, was hilfreiche Hinweise für die Wahl der Aufteilungsfunktionen liefert. Weiters wurde stets auf die Flexibilität der angewandten Methoden Bedacht genommen, um einerseits deren Anwendbarkeit und Aussagekräftigkeit für reale Programme zu gewährleisten und andererseits auch mögliche Adaptionen für unterschiedliche Zielsetzungen zu ermöglichen. Dieser Praxisbezug des Modells zeigt sich vor allem in der Definition eines einfachen, aber wirkungsvollen Kommunikationskostenmodells zur Gewichtung von Alignmentpräferenzen.

Die nachstehende theoretische Definition des Alignmentproblems bedient sich zu dessen Modellierung des von [LiChen89] eingeführten *Component Alignment Graph* (CAG), welcher die erkannten Alignmentrelationen und deren Bedeutung veranschaulicht und die Lösung des Alignmentproblems mit Hilfe von graphentheoretischen Ansätzen ermöglicht.

Im ersten Abschnitt wird das nötige Rüstzeug zur Konstruktion eines CAG bereitgestellt; der zweite Abschnitt beschäftigt sich mit der Attributierung der Kanten eines CAG, welche entscheidenden Einfluß auf die Güte des im dritten Abschnitt vorgestellten heuristischen Algorithmus zur Lösung des Alignmentproblems hat.

Für die nachfolgenden Definitionen werden folgende typographischen Konventionen vereinbart:

Großbuchstaben symbolisieren Arrays (A), Kleinbuchstaben Arrayreferenzen (a).

Indexausdrücke einer Arrayreferenz a in der Dimension d werden $expr_d(a)$ genannt und deren Koeffizienten mit griechischen Buchstaben bezeichnet.

Schleifenvariablen werden durch Kleinbuchstaben dargestellt.

Alignmentknoten werden durch den Arraybezeichner (Großbuchstaben) mit tiefgestellter Dimensionsangabe dargestellt.

3.1. Grundlegende Definitionen

Wie bereits angemerkt, basiert die Alignmentanalyse auf eingehenden Analysen von Zugriffsmustern in Zuweisungsstatements, wobei diese Analysen auf reguläre Zugriffsmuster eingeschränkt werden. Man sagt, daß zwei Dimensionen unterschiedlicher Arrays eine Alignmentpräferenz aufweisen, wenn sie gleichförmige (d.h. sich nur um konstante Faktoren unterscheidende) lineare Funktionen eines einzigen Schleifenindexes darstellen. In den Definitionen 3.1. bis 3.6. werden die Voraussetzungen für das Vorliegen einer derartigen Alignmentpräferenz in Anlehnung an den Ansatz Guptas, welcher verallgemeinert und um wichtige Fälle erweitert wird, definiert:

Definition 3.1: Sei a eine Referenz auf ein n-dimensionales Array A in einem beliebigen Schleifennest und $expr_d(a)$ dessen subscript expression in der Dimension d. Wenn $expr_d(a)$ die Form $\alpha * i + \gamma$ hat, wobei α und γ schleifeninvariant[2] und i ein Schleifenindex der umgebenden Schleifen seien, so heißt die subscript expression **single-index** ▪
[Gupta92]

Definition 3.2: Sei a eine Referenz auf ein n-dimensionales Array A in einem beliebigen Schleifennest und $expr_d(a)$ dessen subscript expression in der Dimension d. Wenn $expr_d(a)$ die Form $\alpha_1 * i_1 + \alpha_2 * i_2 + ... + \alpha_k * i_k + \gamma$, $k \geq 2$ hat, wobei $\alpha_1,...,\alpha_k$ und γ schleifeninvariant und $i_1,...,i_k$ Schleifenindizes der umgebenden Schleifen seien, so heißt die subscript expression **multiple-index** ▪
[Gupta92]

Definition 3.3: Sei a eine Referenz auf ein n-dimensionales Array A in einem beliebigen Schleifennest und $expr_d(a)$ die subscript expression des Arrays A in der Dimension d, wobei $expr_d(a)$ der Form single-index oder multiple-index ist. Die Tiefe der innersten Schleife, welche den Wert von $expr_d(a)$ verändert, heißt **variation-level** ▪
[Gupta92]

Auf Basis dieser Definitionen, welche die notwendigen Eigenschaften der jeweiligen Indexausdrücke hinreichend beschreiben, definiert Gupta in Anlehnung an Li und Chen die Alignmentpräferenz:

Definition 3.4: Sei S ein Zuweisungsstatement in einem Schleifennest der Form a = F(b), a und b seien Referenzen auf beliebig dimensionierte Arrays A und B, $A \neq B$, mit $expr_d(a)$ als subscript expression der Referenz a in der Dimension d und $expr_t(b)$ als subscript expression der Referenz b in der Dimension t. Wenn $expr_d(a)$ und $expr_t(b)$ vom Typ single-index sind und den gleichen variation-level besitzen, so heißen diese beiden Dimensionen von A und B **alignment-prefering** ▪
[Gupta92],[LiChen89]

---

[2] Gupta fordert, daß α und γ Integerkonstanten seien. Dies stellt jedoch eine nicht notwendige Einschränkung dar, da schleifeninvariante Koeffizienten bei der isolierten Betrachtung einer Schleife die gleichen Effekte wie konstante Koeffizienten zeigen, daß nämlich der Wert eines Indexausdruckes ausschließlich von Schleifenindizes verändert werden kann.

Diese von Li und Chen bzw. Gupta angewandte Definition berücksichtigt jedoch nur Alignmentpräferenzen zwischen Arrayzugriffen auf der linken respektive rechten Seite eines Zuweisungsstatements; Alignmentpräferenzen, welche auf zwei Arrayzugriffen auf der rechten Seite eines Zuweisungsstatements basieren (beispielsweise bei Reduktionen, aber auch bei Matrixmultiplikationen etc.), werden nicht erkannt, was unter Umständen zu falschen Resultaten führen kann. Auch werden Alignmentpräferenzen zwischen zwei Arraydimensionen mit identen Indexausdrücken, was auch ohne nähere Kenntnis über deren Gestalt eine Alignmentpräferenz ausdrückt, nicht erkannt. Es ist daher notwendig, die Identität zweier Indexausdrücke zu definieren, sowie Definition 3.4. leicht abzuwandeln, um auch diese Fälle berücksichtigen zu können:

Definition 3.5: Sei $expr_d(a)$ eine subscript expression der Form $\alpha_j * i_j + \gamma$ ($1 \leq j \leq L$; L...Anzahl der umgebenden Schleifen) und $expr_t(b)$ eine subscript expression der Form $\beta_j * i_j + \delta$ ($1 \leq j \leq L$) mit i_j als den entsprechenden Schleifenindizes.
Wenn gilt: $\alpha_j = \beta_j \ \forall j, 1 \leq j \leq L$ und $\gamma - \delta = c$ (c sei schleifeninvariant), so heißen $expr_d(a)$ und $expr_t(b)$ **ident** ▪

Mit Hilfe der Identität kann man nun Definition 3.4 abwandeln und man erhält nachstehende Definition der Alignmentpräferenz, welche die ursprüngliche Definition 3.4 ersetzt:

Definition 3.6: Sei S ein Zuweisungsstatement in einem Schleifennest, a, b und c seien Referenzen auf beliebig dimensionierte Arrays A, B und C, $A \neq B$, $A \neq C$ und $C \neq B$, mit $expr_d(a)$ als subscript expression der Referenz a in der Dimension d und $expr_t(b)$ als subscript expression der Referenz b in der Dimension t.
Wenn
- $expr_d(a)$ und $expr_t(b)$ vom Typ single-index sind und den gleichen variation-level besitzen, oder $expr_d(a)$ und $expr_t(b)$ ident sind

und
- S der Form a = F(b) ist

oder
- S der Form c = F(a,b) ist und a und b keine Alignmentpräferenzen zur Arrayreferenz c aufweisen

so heißen diese beiden Dimensionen von A und B **alignment-prefering** ▪

Die Alignmentpräferenz stellt somit eine Relation zwischen Dimensionen zweier Arrays dar, die besagt, daß diese beiden Dimensionen im Programmverlauf gemeinsam für Berechnungen benötigt werden und daher, um Kommunikation zu vermeiden, entsprechend ihrem intradimensionalen Alignment (vgl. Abschnitt 3.2. - Locality) aufgeteilt werden sollten.
Mit Hilfe der Alignmentpräferenz kann man nun den CAG konstruieren, dessen Knoten die Dimensionen der betrachteten Arrays symbolisieren und bei dem für jedes Knotenpaar, welches eine Alignmentpräferenz ausdrückt, eine ungerichtete Kante gezogen wird. Diese Kante wird mit einem Gewicht versehen, welches die Bedeutung dieser Alignmentpräferenz für die spätere Lösung des Alignmentproblems ausdrückt. Weiters werden diese Kanten auch noch mit Informationen über die Art der Alignmentpräferenz und des intradimensionalen Alignments attributiert, was in 3.2. erläutert wird.

Definition 3.7: Sei G ein ungerichteter, gewichteter Graph, wobei dessen Knoten je eine Dimension eines Arrays des betrachteten Programmteiles repräsentieren und für jedes Knotenpaar, welches alignment-prefering ist, eine Kante (**=Alignmentkante *e***) gezogen wird. Dieser Graph heißt **Component-Alignment-Graph (CAG)** ▪
[Gupta92],[LiChen89]

Mit den Definitionen 3.1 bis 3.7. wurde nun das nötige theoretische Rüstzeug für die Erstellung eines CAG präsentiert. Die Gestalt eines solchen Graphen ist aus den Abbildungen zu den Beispielen in Kapitel 4 ersichtlich. Zur besseren Übersichtlichkeit ist der Graph in Zeilen und Spalten geteilt, wobei in jeder Spalte die Knoten ein und desselben Arrays und in jeder Zeile Knoten, die dieselbe Dimension repräsentieren, angeordnet werden.
Eine besondere Bedeutung bei der Alignmentanalyse kommt der Wahl der geeigneten Gewichtungsparameter der Kanten eines CAG zu. Diese Kanten symbolisieren ja Alignmentpräferenzen, was bedeutet, daß bei Nichteinhaltung dieser im Zuge der Distributierung der Arrays mit einem erhöhten Kommunikationsaufwand zu rechnen ist. Es ist daher naheliegend, Maßzahlen für die potentielle Kommunikation, welche aus den Arrayzugriffen, auf welchen eine Alignmentpräferenz beruht, resultiert, zu entwickeln und diese zur Gewichtung der Kanten zu benutzen.

Li und Chen haben in [LiChen89] die Kanten mit ε bzw. 1 gewichtet, je nachdem, ob mehrere gleich gute Alignment-Möglichkeiten bestehen oder nicht. Diese grobe Gewichtung wurde von Gupta in [GuptBan90] und [Gupta92] durch eine Abschätzung der Kommunikationskosten ersetzt, wobei er versucht, die Zugriffsmuster einigen vordefinierten Kommunikationsroutinen zuzuordnen, was jedoch aufgrund der Tatsache, daß zum Zeitpunkt der Analyse weder die tatsächlichen Aufteilungsfunktionen, noch die Anzahl der verwendeten Prozessoren bekannt sind, nur mit Hilfe umfangreicher Heuristiken bewerkstelligt werden kann.

In dieser Arbeit wird ein Ansatz verwendet, der eine Abschätzung des Kommunikationsaufwandes mit geringem Aufwand ermöglicht, wofür zur Erhöhung der Genauigkeit und Aussagekraft dieser Abschätzungen auf Profiling Daten eines Profilers für sequentielle Fortran-Programme zurückgegriffen wird. Überdies wird die Komponente des intradimensionalen Alignments berücksichtigt.

3.2. Attributierung der Kanten

Optimal wäre es, die Kanten mit dem Kommunikationsaufwand, welcher aus dem der Kante zugrundeliegenden Zugriffsmuster resultiert, zu gewichten, um so das Ziel der Minimierung desselben erreichen zu können. Leider wird der Kommunikationsaufwand durch die tatsächliche Lozierung der Arrayelemente (welche wiederum von der gewählten Aufteilungsfunktion und der Anzahl der Prozessoren abhängig ist) bestimmt - und diese Information ist zum Zeitpunkt der Analyse noch nicht vorhanden, ja die Analyse soll erst eine Hilfestellung zur Lösung dieses Problems geben. [Gupta92] beschäftigt sich ausführlich mit diesem Problem und gelangt zu einer Kommunikationskostenabschätzung unter Anwendung komplexer und umfangreicher Heuristiken, welche auch einen großen Zeitaufwand verursachen.

In diesem Ansatz soll vor allem auf die Einfachheit der Methoden zur Abschätzung des Kommunikationsaufwandes, selbstverständlich unter größtmöglicher Bewahrung der Aussagekraft, Wert gelegt werden, um somit einerseits deren Anwendung ohne großen Zeitbedarf und andererseits auch deren einfache Adaptierung für unterschiedliche Anforderungen zu ermöglichen. Zu diesem Zweck wird nachstehend ein Kommunikationskostenmodell entwickelt, welches auf einem Compilermodell und Kostenparametern eines Maschinenmodells basiert. Daneben wird gleichzeitig die Komponente des intradimensionalen Alignments behandelt.

Dieser pragmatische Ansatz zur Lösung des Gewichtungsproblems führt zu folgenden Überlegungen darüber, welche Informationen über eine Alignmentpräferenz verarbeitet werden sollen:

Im allgemeinen kann festgehalten werden, daß der potentielle Kommunikationsaufwand von der Ausführungshäufigkeit eines Zuweisungsstatements abhängig ist. (Im schlechtesten Fall müssen bei jeder Ausführung des Statements Daten kommuniziert werden). Weiters sollten mögliche Optimierungen der Kommunikation durch den Compiler (im speziellen das Herausziehen aus Schleifen und Blocken von Kommunikationsanweisungen wie in 2.2.3. beschrieben) berücksichtigt werden, welche von den Datenabhängigkeiten, in welche das untersuchte Statement involviert ist, gesteuert werden.
Zuguterletzt soll neben der Abschätzung des potentiellen Kommunikationsaufwandes auch die relative Position der Arrayzugriffe (das intradimensionale Alignment) modelliert werden und in geeigneter Form den Alignmentkanten zugeschrieben werden.

Die Alignmentkanten werden daher mit den folgenden drei, nachstehend definierten Informationen attributiert, wobei auf Ergebnisse eines sequentiellen Profilers sowie Abhängigkeitsberechnungen des Compilers zurückgegriffen wird:

- **Frequency**
- **Locality**
- **Dependency**

Die **Frequency (f)** einer Alignmentkante *e* soll einen Rückschluß darauf zulassen, mit welchem Kommunikationsaufwand (mit welcher Anzahl an Kommunikationsschritten) im worst-case zu rechnen ist. Ohne nähere Information über die Datenaufteilung muß man annehmen, daß im worst-case bei jeder Veränderung des Schleifenindexes, welcher den Wert der subscript-expression bestimmt, Daten kommuniziert werden müssen. Das bedeutet, daß die *Frequency* nicht gleich der Ausführungshäufigkeit eines Statements, welche durch einen sequentiellen Profiler festgestellt wurde, sein muß, sondern der Ausführungshäufigkeit eines fiktiven Statements entspricht, dessen Nestingtiefe dem *variation level* (vgl. Def. 3.3) der Alignmentkante entspricht. Man kann somit die Frequency definieren:

Definition 3.8:

Frequency f(*e*) = (AH(DO-head(varlev)) * Iter(DO-body)) * TR

mit AH(DO-head(varlev)) gleich der Ausführungshäufigkeit des DO-Loop-heads jener Schleife, deren Nestingtiefe dem variation level *varlev* entspricht;
Iter(DO-body) gleich der Anzahl der Iterationen, die diese Schleife bewirkt und
TR gleich der True-ratio des Statements, welches die Alignmentkante verursacht[3] ▪

[3] Alle zur Berechnung dieser Maßzahlen nötigen Werte werden vom sequentiellen Profiler zur Verfügung gestellt.

Die **Locality (loc)** modelliert das intradimensionale Alignment, welches sich, da die Analyse auf Indexausdrücke beschränkt ist, welche affine Funktionen der Schleifenindizes sind, als Quotient der Koeffizienten von Indexvariablen bzw. Differenz der konstanten Faktoren ausdrücken läßt:

Definition 3.9: Seien $expr_d(a)$ und $expr_t(b)$ 2 subscript expressions, beide vom Typ single-index und alignment-prefering. Der Quotient der beiden Indexkoeffizienten (β/α) heißt **Multiplicator** von $expr_t(b)$ in bezug auf $expr_d(a)$. Die Differenz der beiden konstanten Koeffizienten (δ - γ) heißt **Constant Offset** von $expr_t(b)$ in bezug auf $expr_d(a)$. Beide Komponenten zusammen werden als **Locality-Relation (loc)** von $expr_t(b)$ in bezug auf $expr_d(a)$ bezeichnet ▪

Die beiden Komponenten der Locality können als Tupel aufgefaßt werden und mit der Notation **(m,o)** mit m gleich dem Mulitplicator und o gleich dem Constant Offset dargestellt werden.

Lemma 3.1: Sind zwei subscript expressions ident (Def. 3.5.), so besitzen sie einen Multiplicator von 1 und einen konstanten oder invarianten Offset ▪

Beweis: Der Beweis ist direkt von Def. 3.5. und Def. 3.9. ableitbar ▪

In Hinblick auf die Auswirkungen der Localityrelation einer Alignmentkante *e* auf die anzuwendende Aufteilungsfunktion kann man die Menge aller in diesem Rahmen möglichen Localityrelationen in 3 Gruppen unterteilen:

Definition 3.10:

$$\mathrm{loc}(e) = \begin{cases} \textit{perfect}, \text{ wenn das Localitytupel die Form } \mathbf{(1,0)} \text{ hat} \\ \\ \textit{constant}, \text{ für alle Localitytupel:} \\ \{\mathbf{(m,o)} \mid \text{m und o sind Integerkonstanten} \wedge (m \neq 1 \vee o \neq 0)\} \\ \\ \textit{invariant}, \text{ für alle Localitytupel:} \\ \{\mathbf{(m,o)} \mid \text{m oder o sind keine Integerkonstanten,} \\ \text{aber schleifeninvariant für alle umgebenden Schleifen}\} \end{cases}$$

▪

Die Locality gibt Hinweise über die geeignete Wahl von Aufteilungsfunktionen, sodaß bei Vorliegen einer true-Locality jede Aufteilungsvariante gewählt werden kann, bei einer constant-Locality jedoch auf den Multiplicator, der die Blockgröße beeinflußt, und auf den Constant Offset, der einen Shift bewirkt, Bedacht genommen werden muß, bei einer invariant-Locality jedoch ohne weitere Informationen kaum Aussagen über die mögliche Aufteilungsfunktion getätigt werden können.

Die Localityrelation wird aber auch gebraucht, um ein in tatsächlichen Programmen üblicherweise mehrmaliges Auftreten der selben Alignmentrelation zu modellieren, was in 3.2.1. näher erläutert werden wird. Hiezu wird die nachstehend definierte Identität zweier Localityrelationen benötigt:

Definition 3.11: Zwei Localityrelationen loc(e_1) und loc(e_2) heißen **ident**, wenn sie vom Typ perfect oder constant sind, den gleichen Multiplicator besitzen und der Constant Offset nur um eine kleine Konstante[4] differiert ▪

Zur Modellierung der oben beschriebenen Kommunikationsoptimierung ist es nötig, die entsprechende Abhängigkeitsinformation verfügbar zu haben (Abhängigkeitsinformationen können im allgemeinen vom Compiler generiert werden), um Rückschlüsse auf die mögliche Positionierung des Kommunikationsstatements und somit auf die potentielle Datenmenge, welche in einer Ausführung des Kommunikationsstatements versandt wird, sowie auf die Ausführungshäufigkeit des Kommunikationsstatements zu erhalten. Da Kommunikation aus einem Schleifennest gezogen werden kann, wenn dadurch keine true dependency verletzt wird [ZiChap92], wird die Dependency (d) folgend definiert:

Definition 3.12: Ist die auf der rechten Seite des Zuweisungsstatements betrachtete Arrayreferenz in eine schleifengetragene true-dependence involviert, so ist die **Dependency d(*e*)** die Tiefe der innersten, dependence verursachenden Schleife; bei Vorliegen einer schleifenunabhängigen dependence entspricht die Dependency dem nesting-level der Anweisung, andernfalls erhält sie den Wert 0 (d.h. die Kommunikation kann aus dem gesamten Schleifennest gezogen werden). ▪

Es ist nicht nötig, auch die linke Seite des Zuweisungsstatements in Def. 3.12 aufzunehmen, da aufgrund des owner-compute-Paradigmas nur Daten kommuniziert werden müssen, auf welche ein Lesezugriff stattfindet.

[4] Hier wurde ein Wert von ±10 angenommen, um auch Stencils in diese Definition aufnehmen zu können.

Die Dependency entspricht somit jenem Schleifenlevel, auf welches das Kommunikationsstatement gezogen werden kann.

Aus der frequency-Information und der dependency-Information kann das Gewicht der Kante errechnet werden, welches eine heuristische Abschätzung der worst-case-Kommunikationskosten[5] darstellt. Da es sich bei dem gesamten Alignmentproblem um eine heuristische Vorgangsweise handelt und auch die Kommunikationskosten nur heuristisch abgeschätzt werden können, da ja noch keine Datenaufteilungen bekannt sind, wird hier aus pragmatischen Gründen ein einfaches Kommunikationskostenmodell angewandt, welches die Plazierung der Kommunikationsroutinen, die zu übertragende Datenmenge sowie zwei maschinenabhängige Parameter, die startup-time (σ) und die Transferzeit (τ)[6], berücksichtigt. Selbstverständlich könnten auch detailiertere Berechnungen für den Kommunikationsaufwand herangezogen werden, es stellt sich hierbei jedoch die Frage der Zweckmäßigkeit und der Vertretbarkeit des dabei betriebenen Aufwandes.
Hiezu wäre noch anzumerken, daß im allgemeinen auch zu viele entscheidende Informationen, welche für exakte Abschätzungen des Kommunikationsaufwandes vonnöten sind, zum Zeitpunkt der Ausführung der Alignmentanalyse nicht verfügbar sind, was allein schon die Anwendung einfacher Methoden impliziert.

Mit Hilfe dieser beiden maschinenabhängigen Parameter sowie den Werten für die Anzahl der Daten, welche pro Kommunikationsstatement übertragen werden (θ) und der Ausführungshäufigkeit des Kommunikationsstatements (ϕ), läßt sich die Gleichung zur Berechnung des Kantengewichtes erstellen:

Definition 3.13:

$$\textbf{weight } \omega(e) = (\tau * \theta + \sigma) * \phi$$

Die Parameter τ und σ der Gleichung sind maschinenabhängig und werden entsprechend der spezifizierten Zielmaschine aus vordefinierten Bibliotheken entnommen; die

---

[5] Es wird hierbei davon ausgegangen, daß alle gemeinsam genutzten Elemente der beiden Arrays unterschiedlichen Prozessoren zugeordnet, die beschriebenen Optimierungsschritte jedoch durchgeführt wurden.

[6] Da es sich bei dem Kantengewicht nur um einen heuristischen Wert zur Abschätzung der Größenordnung von Kommunikationskosten und keinesfalls um eine exakte Berechnung derselben handelt, ist es auch unerheblich, von welchem Datentyp (und damit Größe) die zu übersendenden Daten sind; d.h. die zu spezifizierenden maschinenabhängigen Daten müssen nur das etwaige Größenverhältnis widerspiegeln.

Parameter θ und ϕ können aus der frequency-Information und der dependency-Information wie folgt errechnet werden:

$$\theta = \begin{cases} \mathbf{1,\ wenn\ varlev}(\boldsymbol{e}) \le \mathbf{d}(\boldsymbol{e}) \\ \mathbf{Iter(DO\text{-}body),\ wenn\ varlev}(\boldsymbol{e}) > \mathbf{d}(\boldsymbol{e}) \end{cases}$$

$$\phi = \textbf{Iterationsanzahl eines Statements auf level } \mathbf{d}(\boldsymbol{e})$$

■

Die Berechnungsformel von Definition 3.13. modelliert die Vektorisierung der Kommunikationsanweisungen wie in 2.2.3. beschrieben; die ebenso mögliche Fusion von Kommunikationsanweisungen wird nicht modelliert, kann jedoch im Bedarfsfalle durch eine einfache Modifikation von Definition 3.13. entsprechend berücksichtigt werden.

Der große Vorteil bei dieser Methode der Berechnung der Kantengewichte liegt darin, daß durch die explizite Einbeziehung der Abhängigkeitsinformationen und das einfache Modell der Kommunikationsoptimierung der potentielle Kommunikationsaufwand nicht nur durch die Ausführungshäufigkeit eines Statements modelliert wird, sondern dieser vielmehr durch die Plazierung (und damit Ausführungshäufigkeit) der Kommunikationsanweisung abgeschätzt wird, wodurch eine signifikante Verbesserung des Resultats erreicht wird.

Beispiel 3.1. demonstriert die praktische Attributierung und Gewichtung von Alignmentkanten:

Beispiel 3.1. - Attributierung von Kanten

In einem zu analysierenden Programm wird folgendes Programmstück gefunden:

```
            ...
    L1:     DO 10 i=1,100
    L2:         DO 10 j=1,100
                    ...
    S:              A(i+15,j)=B(j,i+15)
                    ...
    10      CONTINUE
            ...
```

Unter der Annahme, daß L1 nur einmal im Zuge des Programmablaufes durchlaufen wird und das Statement S keine echten Abhängigkeiten aufweist, kann die Kante $e(A_2,B_1)$ folgendermaßen attributiert werden:

f(e) = 100 * 100 * 1.0 = 10000
loc(e) = *perfect*, da das Localitytupel die Form (1,0) aufweist
d(e) = 0

Die zur Berechnung des Gewichtes nötigen Parameter ergeben sich zu:

$\phi = 1$
$\theta = 100$

und das Gewicht somit:

$\boldsymbol{\omega(e) = (\tau * 100 + \sigma) * 1}$

3.2.1. Behandlung von multiplen Alignmentpräferenzen

Im Laufe eines Programmstückes ist es wahrscheinlich, daß die Dimensionen zweier Arrayreferenzen mehrmals und in unterschiedlichen Statements Alignmentpräferenzen aufweisen. Ohne einer detaillierten Analyse, ob die Werte der involvierten Arrayelemente verändert wurden oder nicht, muß man davon ausgehen, daß man bei jedem neuen Auftreten der Alignmentpräferenz mit Kommunikation rechnen muß.
Unter Berücksichtigung des intradimensionalen Alignments (insbesondere der identen Locality) und des jeweiligen Statements, das die Alignmentpräferenz verursacht, kann man vier Fälle unterschieden:

1. Die Alignmentpräferenzen resultieren vom selben Statement und besitzen eine idente Locality.
2. Die Alignmentpräferenzen resultieren vom selben Statement und besitzen eine unterschiedliche Locality.
3. Die Alignmentpräferenzen resultieren von verschiedenen Statements und besitzen eine idente Locality.
4. Die Alignmentpräferenzen resultieren von verschiedenen Statements und besitzen eine unterschiedliche Locality.

Bei der Erstellung bzw. Gewichtung der Alignmentkanten werden diese vier Fälle unterschiedlich behandelt, da sie auch durchwegs unterschiedliche Auswirkungen auf den Kommunikationsaufwand zeigen:

ad 1) - Gleiches Statement, idente Locality:
Dieser Fall stellt eine redundante Information dar, da, falls nichtlokale Speicherzugriffe notwendig sind, die Daten bereits übersandt wurden und daher bereitstehen. Die Information wird für die weitere Behandlung ignoriert.

ad 2) - Gleiches Statement, unterschiedliche Locality:
Dieser Fall stellt konkurrierende Anforderungen an die Aufteilungsfunktionen dar, da zwar das interdimensionale Alignment konfliktfrei ist, beim intradimensionalen Alignment jedoch Konflikte auftreten können.
Dies wird durch die Generierung einer neuen, zusätzlichen Kante gemäß Def. 3.7. modelliert. Man erhält somit einen Graphen, bei welchem ein Knotenpaar durch mehr als eine Kante verbunden sein kann. Alle diese Kanten sind entsprechend attributiert und modellieren damit auch die Bedeutung des intradimensionalen Alignments.

ad 3) - Anderes Statement, idente Locality:
Da ohne ausführlichere Analyse angenommen werden muß, daß sich die Daten zwischen den beiden Statements verändert haben, muß wieder mit Kommunikation gerechnet werden.
Der Einhaltung dieser Alignmentpräferenz kommt daher eine höhere Bedeutung zu, was dadurch ausgedrückt wird, daß sowohl die Gewichtungen der existierenden Alignmentkante als auch der neuen (fiktiven) Kante kummuliert werden.
Sollte es sich bei den beiden Kanten um solche mit konstanter Locality halten, so wird der maximale Schwankungsbereich des konstanten Offsets gespeichert. Das Localitytupel erhält somit die neue Gestalt $\mathbf{(m,o_{min}/o_{max})}$.

ad 4) - Anderes Statement, unterschiedliche Locality:
Dieser Fall ist analog zu Fall 2) zu behandeln und zeigt die selben Konsequenzen.

Eine Fortführung von Beispiel 3.1. soll die Vorgehensweise bei multiplen Alignmentrelationen veranschaulichen:

Beispiel 3.2. - Multiple Alignmentrelationen

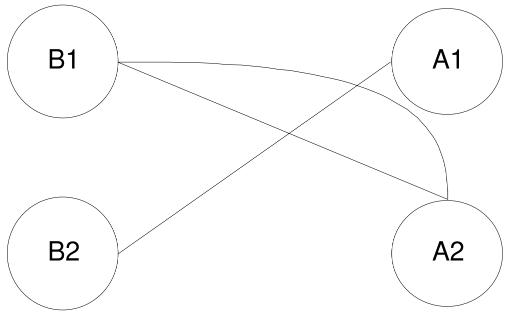
```
        ...
L1:     DO 10 i=1,100
L2:         DO 10 j=1,100
                ...
S1:             A(i+15,j)=B(j,i+15)
                ...
10      CONTINUE
        ...
L3:     DO 20 i=1,100
L4:         DO 20 j=1,100
                ...
S2:             B(j+20,i)=A(i,j)+A(i,3)
                ...
20      CONTINUE
```

In Statement S1 werden folgende Alignmentpräferenzen festgestellt: (A_2,B_1) und (A_1,B_2).
Die Alignmentpräferenzen von S2 stellen ein wiederholtes Auftreten der bereits in S1 festgestellten Präferenzen dar und werden gemäß der besprochenen Vorgangsweise modelliert:

Die Alignmentpräferenz (B_1,A_2) hat eine andere Locality als die entsprechende Präferenz in S1; es tritt daher Fall 2 in Kraft und eine zusätzliche Kante zwischen diesen beiden Knoten wird generiert.

Die Alignmentpräferenz (B_2,A_1) tritt sowohl in S1, als auch zweimal in S2 auf, hat aber immer idente Localitys, was den Fällen 1 und 3 entspricht (d.h., die Gewichte, welche für diese Kante in S1 und S2 berechnet werden, werden kummuliert).

Der resultierende CAG hat folgende Gestalt (wobei die Kanten natürlich entsprechend gewichtet und attributiert sind, was der Übersichtlichkeit halber jedoch nicht angezeigt wird):



3.3. Lösung des Alignmentproblems

Mit den in 3.1. und 3.2. vorgestellten Definitionen ist man in der Lage, aus einem Programm mit Hilfe eines Profilers und geeigneter Datenabhängigkeitsanalysen einen CAG zu erstellen, welcher eine Fülle von Informationen enthält und sowohl das interdimensionale als auch das intradimensionale Alignmentproblem modelliert. Die Merkmale dieses CAG können nochmals kurz zusammengefaßt werden:

- Alignmentpräferenzen zwischen zwei Arraydimensionen (interdimensionales Alignment) werden dargestellt
- Modellierung des intradimensionalen Alignments innerhalb von interdimensionalen Präferenzen
- Worst-case Abschätzungen der Kommunikationskosten
- Modellierung vom mehrfachen Auftreten der selben interdimensionalen Alignmentpräferenzen
- Modellierung von konkurrierenden intradimensionalen Präferenzen innerhalb von gleichen interdimensionalen Alignmentpräferenzen

Das Ziel ist es nun, die Knoten dieses CAG in n disjunkte Mengen (n = maximale Dimensionalität der Arrays) dergestalt aufzuteilen, daß einer Menge nicht zwei Knoten desselben Arrays zugeteilt werden dürfen und die Summe aller Kantengewichte, die eine Menge verlassen, minimiert wird. Die dadurch erzielten Alignmentmengen stellen Mengen von Arraydimensionen dar, welche gemäß der Ergebnisse des intradimensionalen Alignments gemeinsam distributiert werden sollen.

Üblicherweise wird ein solcher CAG jedoch eine Fülle von sowohl konkurrierenden interdimensionalen Alignmentpräferenzen, als auch von konkurrierenden intradimensionalen Alignmentpräferenzen beinhalten, was eine optimale Zuordnung gemäß den obgenannten Kriterien sehr erschwert. Li und Chen bewiesen in [LiChen89], daß diese optimale Zuordnung der Knoten der Klasse der NP-vollständigen-Probleme angehört, weshalb sie einen heuristischen Algorithmus präsentieren. (Welcher jedoch nur für das Problem des interdimensionalen Alignments anwendbar ist und das intradimensionale Alignment außer Acht läßt).

[KenKrem94] und [GAL95] formulieren derartige Zuordnungen ([KenKrem94] bedienen sich des Konzepts des CAG, während [GAL95] das Konzept des *Communication-Parallelism-Graph CPG* einführen) als 0-1 integer programming problem, wobei die Arbeit von [GAL95] auf Distributierung einer einzigen Arraydimension beschränkt ist, während [KenKrem94] die effektive NP-Vollständigkeit des Problems außer Acht lassen und einen möglicherweise großen Zeitbedarf für diese Berechnungen in Kauf nehmen. (vgl. Kapitel 5)

Diese Arbeit bedient sich einer verbesserten Variante des heuristischen Algorithmus von Li und Chen zur Lösung des interdimensionalen Alignments. Die Entscheidung zur Anwendung des heuristschen Algorithmus beruht hauptsächlich auf den guten Ergebnissen und dem befriedigenden Zeitverhalten desselben unter Beibehaltung der theoretischen Unbegrenztheit des CAG. Die am ursprünglichen Algorithmus vorgenommenen Änderungen werden in 3.3.2. beschrieben und begründet.

Zuvor muß jedoch, um den heuristischen Algorithmus anwenden zu können, eine Lösung des intradimensionalen Alignmentproblems gefunden werden.

3.3.1. Lösung des intradimensionalen Alignments

Das intradimensionale Alignment wurde im CAG durch die Beifügung der Localitytupel zu den Alignmentkanten modelliert. Wie bereits erwähnt, kann diese Localityinformation Hinweise über die Wahl der Aufteilungsfunktion bzw. der anzuwendenden Alignmentfunktionen geben, wenn sie von den Kanten direkt auf die Knoten übertragen wird. In 3.2.1. wurde bereits gezeigt, wie intradimensionale Konflikte innerhalb einer interdimensionalen Alignmentrelation modelliert werden können, nämlich durch Generierung einer zusätzlichen Kante. Doch auch zwischen unterschiedlichen interdimensionalen Alignmentrelationen kann es zu intradimensionalen Konflikten kommen;

hiezu ist jedoch ein kurzer Vorgriff auf die nachfolgend behandelte Lösung des interdimensionalen Alignmentproblems notwendig:

Der im folgenden behandelte Lösungsalgorithmus des interdimensionalen Alignmentproblems sieht vor, daß die Knoten des CAGs disjunkten Mengen zugeordnet werden, was über die Alignmentpräferenzen eines Knoten zu den bereits in einer Menge befindlichen anderen Knoten gesteuert wird. Die Lösung ergibt also Mengen von Arraydimensionen, welche gegenseitige Alignmentpräferenzen aufweisen. Unter Ausblendung des intradimensionalen Alignmentproblems könnte aus diesen Lösungen abgeleitet werden, daß alle Arraydimensionen, welche sich in der gleichen Menge befinden, auf die selbe Art und Weise distributiert werden sollten. Nun können die Kanten zwischen den Knoten einer Menge die unterschiedlichsten Localityrelationen aufweisen, welche natürlich in der Aufteilungsfunktion berücksichtigt werden müssen, um optimale Ergebnisse zu erlangen. Hieraus können sich intradimensionale Konflikte ergeben, wenn ein Knoten mehrere Alignmentpräferenzen zu anderen Arraydimensionen aufweist, welche unterschiedliche Localityrelationen besitzen. Das heißt, für einen solchen Knoten ist zwar das interdimensionale Alignmentproblem gelöst, doch aufgrund der unterschiedlichen intradimensionalen Erfordernisse wird keine Aufteilungsfunktion gefunden werden können, welche keine Kommunikation verursacht (außer natürlich der trivialen Aufteilung - der Replikation).

Es müssen daher bereits vor der Modellierung des interdimensionalen Problems intradimensionale Konflikte berücksichtigt werden, was idealerweise durch Berechnung des nicht vermeidbaren Kommunikationsaufwandes erfolgen sollte. Dies ist aber wiederum nicht möglich, da keine Informationen über Aufteilungsfunktionen vorhanden sind, weshalb wieder auf eine worst-case Abschätzung zurückgegriffen wird.

Es ist notwendig, daß die Localityinformation einer Alignmentkante zwischen einem Knoten innerhalb einer Menge und einem Knoten außerhalb dergestalt relativiert wird, daß sie die Localityrelation des Knoten außerhalb der Menge zur gesamten Menge modelliert. Dazu wird zunächst der erste Knoten, welcher einer Alignmentmenge zugeordnet wird, als *Basisknoten* für das intradimensionale Alignmentproblem ausgezeichnet. Das heißt, dieser Knoten wird mit einer *perfect* Locality versehen und die Locality aller anderen, dieser Menge zugeordneten Knoten wird in Bezug zu diesem Knoten gesetzt. Dazu müssen die Localitytupel von Knoten, die einer Menge zugeordnet werden sollen, um die Localityrelationen der bereits zugeordneten Knoten, zu welchen eine Alignmentpräferenz besteht, relativiert werden. Algorithmus 3.1. erläutert diese Vorgangsweise, wonach dies in Bsp. 3.3. veranschaulicht wird.

Algorithmus 3.1. - Relativierung von Localityinformationen

Sei A ein bereits einer Menge zugeordneter Knoten mit loc(A) als seiner Localityinformation, B ein Knoten, welcher zugeordnet werden soll mit einer entsprechenden Alignmentkante *e*(A,B). Die Localityinformation von *e* wird folgendermaßen relativiert:

```
if (loc(e)==perfect)
{
    loc(e)=loc(A);
}
elsif ((loc(e)==invariant)||(loc(A)==invariant))
{
    loc(e)=invariant;
}
elsif ((loc(e)==constant)||(loc(A)==constant))
{
    loc(e)=constant;
    multiplicator(e)=multiplicator(e)*multiplicator(A);
    constant offset(e)=constant offset(e)+constant offset(A);
}
else
{
    loc(e)=loc(e);
}
```

Durch diese Relativierung der Localityinformation wird erreicht, daß diese nicht mehr die Locality zweier Knoten, sondern jene zwischen einem Knoten und einer gesamten Menge modelliert. Danach werden Kanten mit identen Localitys, entsprechend Fall 3 von 3.2.1. zusmmengefaßt, indem deren Gewichtungen kummuliert werden.

Als Endresultat erhält man somit bei der Zuordnung eines Knotens zu einer Menge für jede Locality eine eigene Kante. Entsprechend der worst-case Annahme, daß nur eine der konkurrierenden intradimensionalen Anforderungen vollständig beachtet werden kann, alle anderen jedoch überhaupt nicht, kann bei der weiteren Behandlung im Rahmen des interdimensionalen Alignmentproblems auf die am höchsten gewichtete Kante zurückgegriffen werden.

Beispiel 3.3. - Relativierung der Localityinformation

Ein Teil eines CAG, bestehend aus dem zweidimensionalen Array A und den eindimensionalen Arrays B und C weisen Kanten mit folgenden Localityinformationen auf:

$e(A_1,B_1)$: loc = constant, Localitytupel für B_1: (-1,0)
$e(A_1,C_1)$: loc = constant, Localitytupel für C_1: (-1,0)
$e(B_1,C_1)$: loc = perfect

Wird nun der Knoten A_1 als erster einer Alignmentmenge zugeordnet, so wird er *Basisknoten* und erhält die Localityinformation *perfect*. Als nächster wird Knoten B_1 dieser Menge zugeordnet und erhält die Localityinformation *constant* mit dem Localitytupel (-1,0). Wenn nun versucht wird, C_1 dieser Menge zuzuordnen, so stellt man fest, daß dieser Knoten zwei Alignmentkanten mit unterschiedlicher Localityinformationen aufweist. Dies würde auf intradimensionale Konflikte hindeuten, welche jedoch nicht vorhanden sind. Wird die Localityinformation jedoch relativiert, so erhalten beide Kanten die Localityinformation *constant* mit dem Localitytupel (-1,0) und deren Gewichte können kummuliert werden.

3.3.2. Lösung des interdimensionalen Alignments - Der Heuristische Algorithmus

Die Lösung des interdimensionalen Alignments beruht auf dem bereits angeführten Zuordnungsproblem der Knoten des CAG zu disjunkten Mengen, sodaß die Gewichte jener Kanten, die eine Menge verlassen (das ist jener Kommunikationsaufwand, welcher nicht vermieden werden kann), minimiert werden.

In ihrem heuristischen Algorithmus [LiChen89] lösen Li und Chen das Zuordnungsproblem, indem sie konsekutiv alle Spalten (in zufälliger Reihenfolge) jener Spalte, welche die höchste Dimensionalität aufweist, zuordnen. Somit reduziert sich das Problem auf das Zuordnungsproblem eines bipartiten Graphen, was mit polynomialen Algorithmen zu bewerkstelligen ist. Beim Aufbau des bipartiten Graphen werden alle Wege berücksichtigt, die im ursprünglichen CAG zwischen den beiden Knoten vorhanden sind, und deren Gewichte kummuliert, um transitive Alignmentrelationen (Knoten a ist aligned mit Knoten b, b mit c, somit ist auch a mit c aligned) entsprechend zu berücksichtigen. (vgl. Abb. 5.1. in Abschnitt 5.1.)

Diese Vorgangsweise hat jedoch zwei entscheidende Schwachpunkte: zum einen werden transitiven Alignmentrelationen meistens, aufgrund der Kummulierung der Gewichte der Kanten auf allen Wegen zwischen den beiden Knoten, zu hohe Gewichte beigemessen, zum anderen liefert der Algorithmus bei unterschiedlicher Reihenfolge der ausgewählten Spalten unterschiedliche Ergebnisse.

Dies führt zu dem Schluß, daß die Reihenfolge, in welcher die zuzuordnenden Kanten ausgewählt werden, keine zufällige sein darf, was impliziert, daß eine Ordnung im Graphen geschaffen werden muß. Weiters sind transitive Alignmentrelationen nicht wie bei Li und Chen unabhängig von anderen Zuordnungen zu betrachten, sondern hängen von bereits vorgenommenen Zuordnungen ab. Diese beiden Abänderungen werden durch ein topologisches Sortieren des CAG und die alleinige Berücksichtigung von direkten Alignmentrelationen (allerdings zu sämtlichen bereits angeordneten Knoten) beim Aufbau des bipartiten Graphen erreicht.

Die linke Spalte des bipartiten Graphen stellt somit die Alignment-Sets, die rechte Spalte die den Alignment-Sets zuzuordnenden Knoten dar. Zwischen dem Knotenpaar S_i ($S_i \in$ linken Spalte, somit ein Alignmentset) und B_j ($B_j \in$ rechten Spalte) wird eine Kante (Bigraphkante *be*) gezogen, wenn ein $A_k \in S_i$ mit einer Alignmentpräferenz zu B_j existiert. Das Gewicht dieser Kante ergibt sich aus dem Maximum der kummulierten Gewichte aller dieser Kanten mit identer, relativierter Locality, wie in 3.3.1. beschrieben wurde:

$$\omega(be) = \max\Big(\sum_{\text{ident loc}} \omega(e(A_k,B_j))\ \forall\, A_k \in S_i\Big) \tag{1}$$

Das in Formel (1) berechnete Gewicht stellt die Entscheidungsgrundlage für das Auffinden des optimalen Matchings dar.

Weiters wird diese Kante mit den kummulierten Localityinformationen **aller** betrachteten Alignmentpräferenzen (nicht nur jener, welche für die Gewichtung der Kante ausschlaggebend waren) gemäß Definition 3.14 attributiert, wodurch Widersprüche im intradimensionalen Alignment widergespiegelt werden.

Definition 3.14

$$\mathrm{loc}(be)=\begin{cases} perfect \text{ if } \forall A_k \in S_i \ \mathrm{loc}(e(A_k,B_j))= perfect \mid \exists\, e(A_k,B_j) \\ constant \text{ if } \forall A_k \in S_i \ \mathrm{loc}(e(A_k,B_j))=(constant \vee perfect)\wedge \\ \forall(e_1,e_2)\ \mathrm{loc}(e_1)\ ident\ \mathrm{loc}(e_2) \mid \exists\, e(A_k,B_j) \\ competing \text{ else} \end{cases}$$

▪

Die Localityinformation dieser Kanten kann dem Benutzer hilfreiche Hinweise über die Wahl der geeigneten Aufteilungsfunktionen eines Alignmentsets geben, indem man die einzelnen Knoten eines solchen Sets mit der Localityinformation der für die Zuordnung verantwortlichen Kante versieht. (Der als erstes angeordnete Knoten wird wie beschrieben als Basisknoten mit *perfect* attributiert!).

Ein weiterer Vorteil dieses modifizierten heuristischen Algorithmus besteht darin, daß durch das topologische Sortieren des CAG auch unzusammenhängende Graphen in einem Schritt bearbeitet werden können: in diesem Fall werden zuerst die Knoten der ersten zusammenhängenden Komponente angeordnet und sodann die Knoten der ersten Spalte der nächsten zusammenhängenden Komponente zufällig den Sets zugeschrieben, wonach wieder mit der üblichen Heuristik fortgesetzt werden kann.
Es besteht jedoch natürlich auch die Möglichkeit, die einzelnen zusammengehörigen Komponenten von unzusammenhängenden Graphen getrennt zu betrachten.

Das Konzept dieses modifizierten heuristischen Algorithmus ist nachstehend in Abb. 3.1 angeführt. Alle Komponenten des heuristischen Algorithmus haben polynomiale Laufzeit, daher hat auch der heuristische Algorithmus polynomiale Laufzeit, abhängig von der Anzahl der Spalten des CAG.

```
Heuristic_Algorithm(CAG)
{
 Sort_Graph(CAG);

 Distribute_First_Column(CAG,Alignment_sets);
 /* distribute the nodes of the first column from
    the sorted graph among the Alignment_sets */

 column_to_be_matched=Second_Column(CAG);

 while (column_to_be_matched!=NIL)
 {
   x=Form_Bipartite_Graph(column_to_be_matched, Alignment_sets);

    Mx = Optimal_Matching(x);
    /* the optimal matching of the bipartite graph is computed
       (for this polynomial time algorithms exist) */

    Distribute_Column (column_to_be_matched,Mx, Alignment_sets);
    /* distribute the column among the Alignment_sets according to the
       computed matching */

    column_to_be_matched=Next_Column(CAG);
    /* get the next column from the sorted CAG */
   }
}


Sort_Graph(CAG)
{
 /* sort the CAG according to the following rules:

 1) the first column is the one with the highest weight of all edges
 leaving the column.

 2) compute the acummulated weight of all edges between the nodes of
 the sorted part of the CAG and the columns not yet sorted. The column
 with the maximum of these weights is placed next.

 3) Repeat 2) until the whole graph is sorted.
 */
}
```

```
Form_Bipartite_Graph(column, sets);
{
 bigraph = /*
              form a bipartite graph, whose nodes of the first column
              represent the Alignment_sets, the second column is the
              column, that should be matched. Create an edge, if the
              node of the second column is direct connected with any
              node of an Alignment_set. Attribute this edge with the
              highest acummulated weight of all edges between the node
              and all nodes of the set having the same relative
              locality. Set the locality of this edge according to
              Definition 3.14.
           */

 return bigraph;
}
```

Abb. 3.1 - Heuristischer Algorithmus

In diesem Kapitel wurden nun die theoretischen Grundlagen für eine Alignmentanalyse präsentiert: Es wurde gezeigt, wie man einen CAG generiert und dessen Kanten gewichten kann, wobei sowohl das inter- als auch intradimensionale Alignment modelliert wurde. Für die Gewichtung wurde eine einfache Modellierung theoretisch möglicher Kommunikationskosten vorgeschlagen, wozu auf Daten eines Profilers zurückgegriffen und auch Kommunikationsoptimierung modelliert wird. Letztlich wurde ein heuristischer Algorithmus zur Lösung des Alignmentproblems präsentiert.

Das nachfolgende Kapitel behandelt nun die Anwendung eines auf diesen Grundlagen beruhenden Alignmentanalysetools, von dem ein Prototyp im Rahmen des in Kapitel 2 vorgestellten VFCS entwickelt wurde.

4. DAS ALIGNMENTANALYSETOOL

In diesem Kapitel wird die Anwendung der Alignmentanalyse anhand der Arbeitsweise des im Rahmen des VFCS implementierten Alignmentanalysetools veranschaulicht. Nach der Vorstellung der Struktur des Tools folgt eine Fallstudie, wonach einige Hinweise auf Implementierungsdetails gegeben werden und sodann Anwendungsbeispiele dieses Kapitel beenden.

Das Alignmentanalysetool wurde als integrierter Bestandteil des VFCS entwickelt und bedient sich vieler vom Compiler zur Verfügung gestellter Informationen:

Die Analyse erfolgt mittels eines Durchlaufes des Syntaxbaumes [ASU86], welcher im Frontend des VFCS generiert wird, unter Zuhilfenahme der in Kapitel 2 beschriebenen Abhängigkeitsinformationen und profiling-Daten des Weight Finders. Mittels des in 3.2. vorgestellten Gewichtungsmodells wird die anfallende Kommunikation unter Berücksichtigung der Kommunikationsoptimierung modelliert.

4.1. Struktur des Alignmentanalysetools

Eine schematische Darstellung der Struktur der Alignmentanalyse kann Abb. 4.1 entnommen werden, wobei die einzelnen Schritte in 3 Phasen unterteilt werden können:

1. Phase - Frontend:
In dieser Phase werden die für die Analyse notwendigen Daten, wie die maschinenabhängigen Parameter und die Profile-Informationen, bereitgestellt und hilfreiche Programmtransformationen (*scalar forward substitution* und *induction variable substitution*) durchgeführt.

2. Phase - Alignmentanalyse:
Hier findet die eigentliche Alignmentanalyse statt. Arrayreferenzen werden paarweise auf das Vorliegen von Alignmentpräferenzen getestet und der CAG wird generiert.

3. Phase - Heuristischer Algorithmus:
Das intradimensionale Alignment wird, wie in 3.3.1. beschrieben, gelöst und der Heuristische Algorithmus (vgl. 3.3.2.) ausgeführt. Das Ergebnis dieser Phase sind die Align-

mentsets, deren Knoten mit der kummulierten Localityinformation ihrer Kanten versehen wurden.

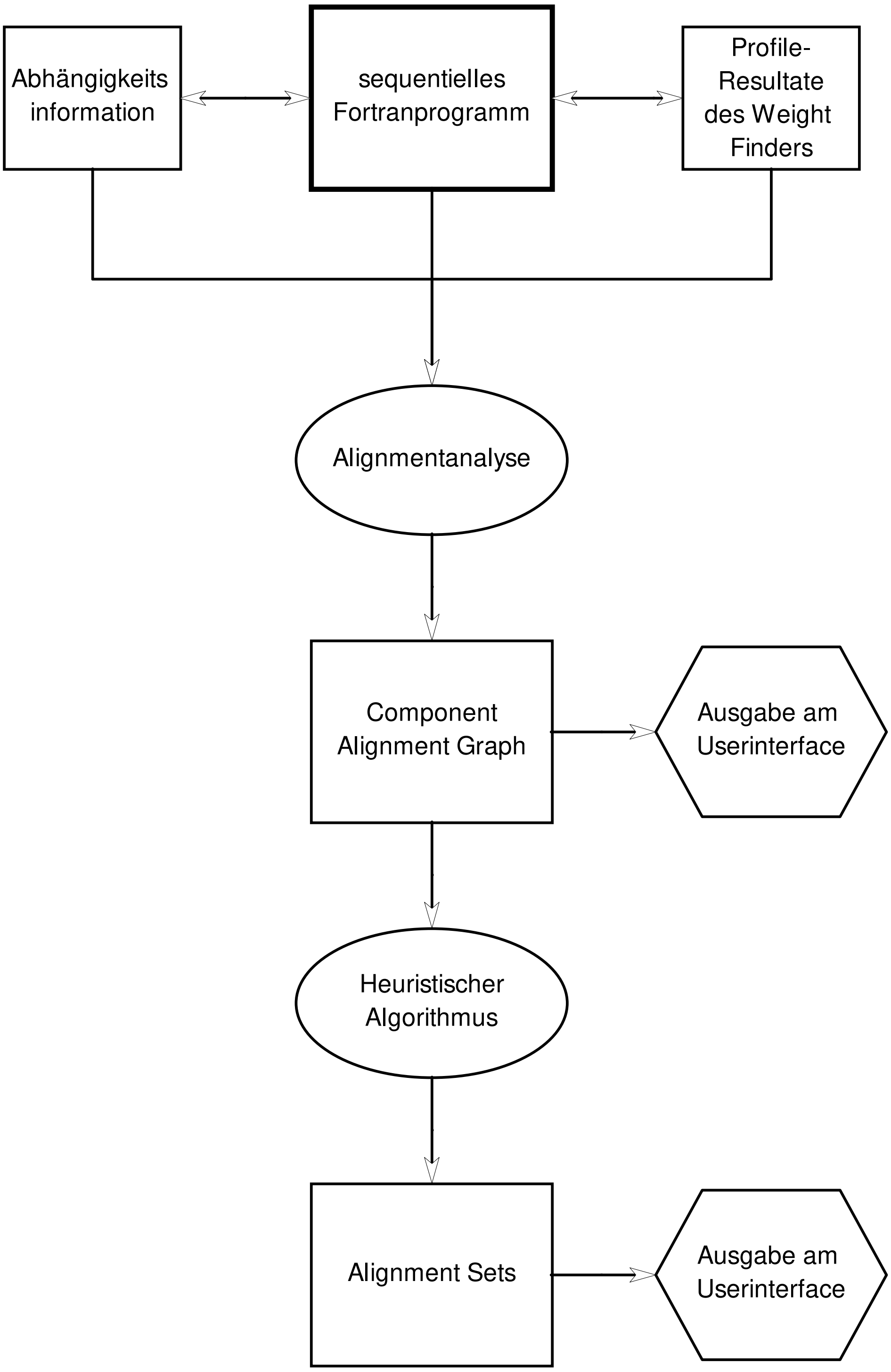


Abb. 4.1 Struktur der Alignmentanalyse

Nach dem Durchlauf dieser drei Phasen ist die Alignmentanalyse beendet und der Benutzer erhält als Information den vollständig attributierten CAG sowie die Alignmentsets ausgegeben.

Die genaue Arbeitsweise dieser drei Phasen wird nachfolgend beschrieben:

4.2. Arbeitsweise des Tools

1. Phase - Frontend:
In dieser ersten Phase wird das Programm für die Ausführung der Alignmentanalyse vorbereitet. Es wird überprüft, ob das Frontend des VFCS durchlaufen wurde, um die nötige interne Repräsentation des Programmes verfügbar zu haben; weiters wird überprüft, ob der Flowgraph [ASU86] des Programmes mit den nötigen Weight Finder Informationen attributiert wurde bzw. gegebenenfalls die Weight Finder Analyse durchgeführt, wobei die Informationen über die Ausführungsfrequenzen, Schleifenparameter und True-ratios erlangt werden.

Die Transformationen *scalar forward substitution* und *induction variable substitution* werden automatisch durchgeführt, da nach Anwendung dieser Transformationen oft bedeutend bessere Aussagen über die vorliegenden Alignmentrelationen getätigt werden können. Ayguade et. al. berichten in [ALGGV94] über empirische Untersuchungen, die bestätigen, daß durch die Anwendung der genannten Transformationen die Anzahl der erkennbaren Alignmentrelationen signifikant gesteigert werden kann. In Abschnitt 4.4 wird dieses Phänomen anhand von Anwendungsbeispielen veranschaulicht.

Daneben erfolgt die Spezifikation der maschinenabhängigen Parameter (*startuptime* und *transfertime*) über die Auswahl der Zielmaschine und die Bereitstellung dieser Daten in entsprechenden Bibliotheken. Vom Benutzer wird in dieser Phase die Auswahl der zu analysierenden Prozedur bzw. des entsprechenden Programmteiles verlangt. Die Analyse ist derart ausgelegt, daß nicht die gesamte Prozedur analysiert werden muß, sondern vom Benutzer ein Start- und Endpunkt hierfür festgelegt werden kann. Die Wahl dieser Punkte unterliegt den Einschränkungen, daß sie der selben Prozedur angehören müssen und beide Punkte von keiner Schleife umgeben sein dürfen. Die letztere Einschränkung ist erforderlich, da für die Analyse die Schleifeninformationen (Parameter, Index) sämtlicher, eine zu analysierende Anweisung umgebender Schleifen verfügbar sein müssen. Die freie Wahl von einzelnen Programmteilen erhöht die

Flexibilität der Analyse signifikant und erleichtert ihre Einbindung und Anwendung in anderen Analysesystemen. Ein weiterer großer Vorteil ist die dadurch mögliche Einteilung eines Programmes in *Phasen*, zwischen denen Realignment nötig ist.
Nach der Ausführung des Frontends ist gewährleistet, daß die nachfolgenden Informationen verfügbar sind:

- Syntaxbaum
- Flowgraph mit Weight Finder Informationen
- Maschinenabhängige Parameter
- Spezifikation der Prozedur und des Programmteils

2. Phase - Alignmentanalyse
In der 2. Phase wird die eigentliche Alignmentanalyse durchgeführt und der Component Alignment Graph konstruiert. Die hierbei angewandte Vorgangsweise, deren Grundlagen die Definitionen 3.1 bis 3.7 bilden, soll anhand des einfachen Programmstückes von Beispiel 4.1 erläutert werden. (Die Weight Finder Informationen sind rechts neben den Anweisungen ausgeworfen). Die Arbeitsweise der nachfolgend angesprochenen Prozeduren kann aus Abschnitt 4.3. entnommen werden.

Beispiel 4.1

```
L1:   DO 10 I=1,100                        <-- FR=1; LP=[1.00;100.00]
L2:       DO 20 J=1,100                  <-- FR=100; LP=[1.00;100.00]
S1:           A(I,J)=B(J,I)+C(I+1,J+2)                  <-- FR=10000
S2:           C(I,J)=A(J,I)                             <-- FR=10000
  20      CONTINUE                                        <-- FR=100
  10  CONTINUE                                              <-- FR=1
```

Der Syntaxbaum dieses Programmstückes zeigt im VFCS folgende (vereinfachte) Gestalt:

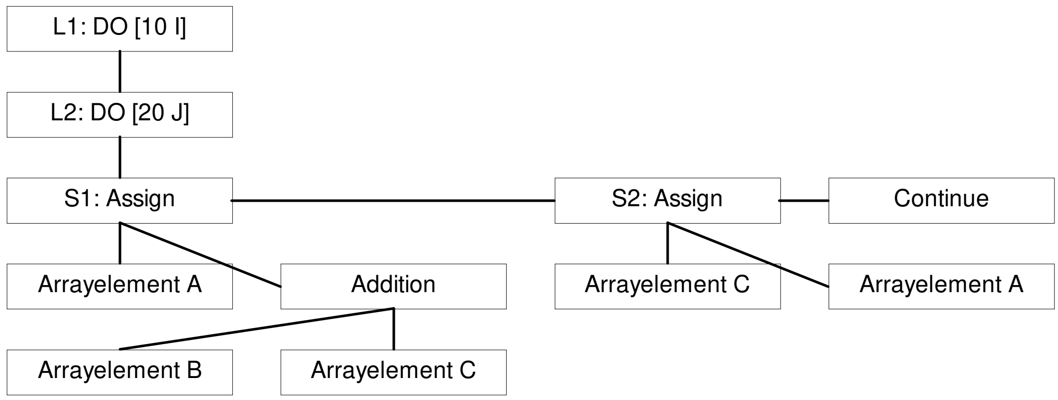


Abb. 4.2 Syntaxbaum

Die Struktur des Syntaxbaumes ermöglicht ein effizientes Durchlaufen, wobei gemäß den Definitionen in Abschnitt 3.1. nur Anweisungen in Schleifenkörpern berücksichtigt werden. Die Analyse dieser Schleifenkörper erfolgt in der Prozedur *analyze loop body*. Wird in einem Schleifenkörper ein Zuweisungsstatement gefunden, so erfolgt dessen Analyse in der Prozedur *analyze statement*, indem zunächst die Abhängigkeitsinformation für diese Anweisung berechnet und sodann alle Dimensionen der Arrayreferenz auf der linken Seite mit allen Dimensionen aller Arrayreferenzen auf der rechten Seite auf Alignmentpräferenzen getestet, entsprechende Knoten im CAG (*create node*) und gegebenenfalls Alignmentkanten (*create edge*) generiert und attributiert werden. Schließlich werden noch jene Dimensionen der Arrayreferenzen der rechten Seite, welche keine Alignmentpräferenzen zur linken Seite aufweisen, auf gegenseitige Alignmentpräferenzen getestet. Der eigentliche Test auf Alignmentpräferenzen findet in der Prozedur *analysis_align* statt, in welcher zuerst auf identes Alignment getestet wird und sodann, sollte ein solches nicht vorliegen, der variation level und die Eigenschaft single index überprüft werden.

Der schematische Ablauf dieser Analyse ist für das Beispielprogramm in Abb. 4.3 dargestellt. Die Arraydimensionen werden dabei durch das Anfügen der entsprechenden Zahl an den Arraybezeichner gekennzeichnet. *A1* bezeichnet demnach die erste Dimension von Array A.

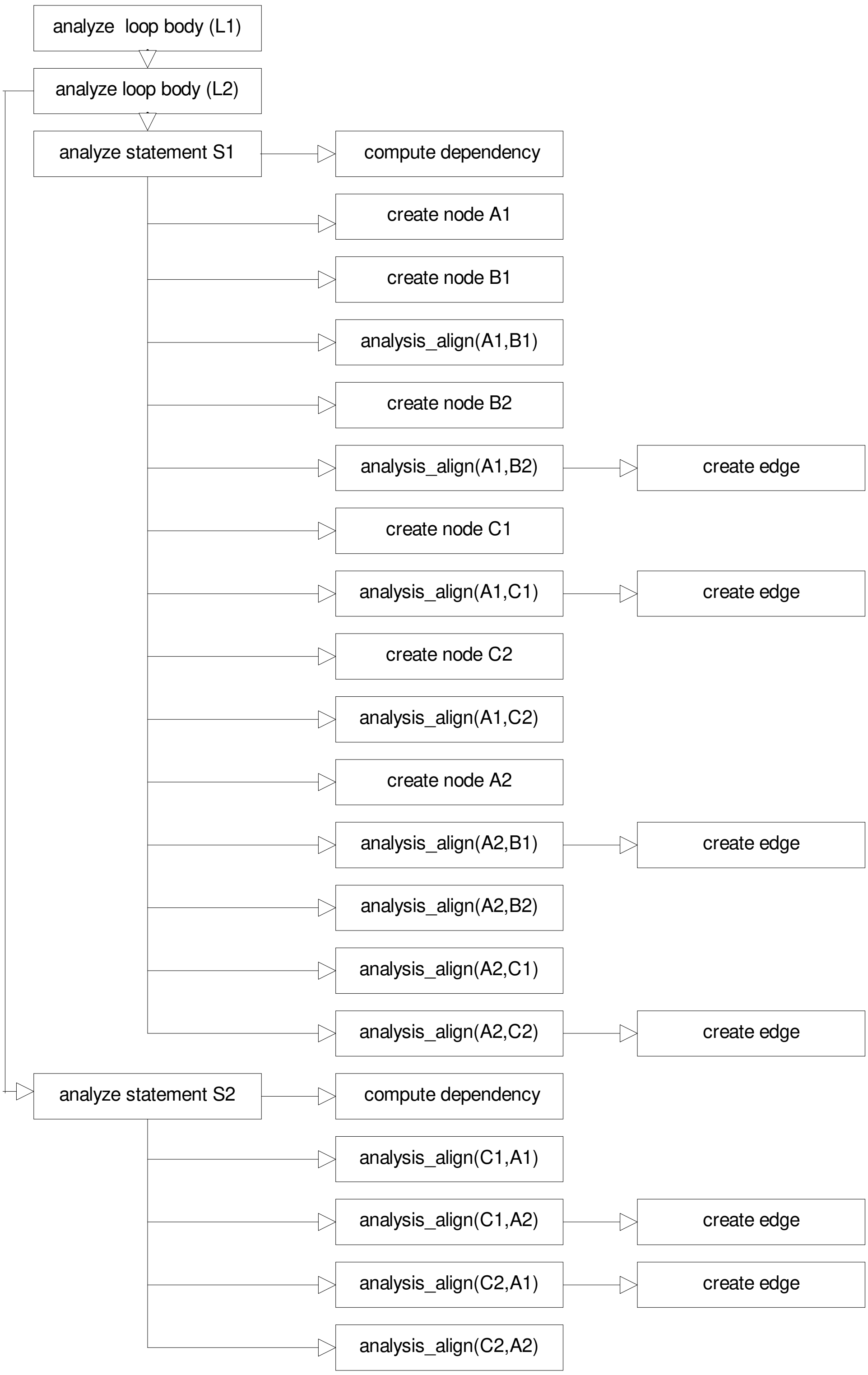


Abb. 4.3 Analysevorgang

Die in der Prozedur *analysis_align* durchgeführten Tests auf Alignmentpräferenzen bedienen sich der im VFCS vorhandenen Koeffizientenmatrizen. Diese Matrizen stellen für jede Arrayreferenz die Koeffizienten der Schleifenindizes sowie den konstanten Faktor der (normalisierten und somit in kanonischer Form befindlichen) subscript expressions aller Dimensionen dar. Überdies liefern sie Informationen über die Linearisierung [ZiChap90] der subscript expression.

Jedes Feld dieser Matrix wird mit zwei Werten belegt: der erste ist ein Integer und liefert einen konstanten Koeffizienten, der zweite ist ein Pointer auf den Syntaxbaum und liefert symbolische Koeffizienten. Zur Illustration ist nachstehend die Koeffizientenmatrix der Arrayreferenz C in S1 von Beispiel 4.1 angeführt, wobei die Linearisierungsfelder ausgelassen wurden, da diese für die Analyse keinerlei Relevanz haben.

		Konstant	Schleife L1	Schleife L2
Linearisierung				
Dimension 1	Pointer to Syntaxtree	NIL	NIL	NIL
	Integer	1	1	0
Dimension 2	Pointer to Syntaxtree	NIL	NIL	NIL
	Integer	2	0	1

Abb. 4.4 Koeffizientenmatrix

Mit Hilfe dieser Koeffizientenmatrizen kann das Vorliegen von Alignmentpräferenzen gemäß den Definitionen in Abschnitt 3.1. leicht überprüft werden.

Wurde eine Alignmentpräferenz festgestellt und eine Alignmentkante generiert, so wird diese, wie in 3.2. beschrieben, attributiert, was anhand der Kante $e(A_2,C_2)$ aus S1 veranschaulicht werden soll:

Beispiel 4.2 - Kantenattributierung:

Die Kante $e(A_2,C_2)$ aus S1 (Beispiel 4.1) wird wie folgt attributiert:

Frequency: Die innerste Schleife, welche die Indexausdrücke verändert ist L2, somit ist der variation level von *e* gleich 2; die Anweisung liegt in keinem If-Zweig, somit ist die true-ratio gleich 1; die Ausführungshäufigkeit des Do-Loop-heads von L2 ist gleich 100 und L2 bewirkt 100 Iterationen ihres Schleifenkörpers. Damit läßt sich die Gleichung gemäß Def. 3.8 aufstellen:

$$\mathbf{f}(\boldsymbol{e}) = \mathbf{(100 * 100) * 1 = 10000}$$

Locality: Die beiden subscript expressions haben folgende Gestalt:

$$expr_2(a) = 1*J$$
$$expr_2(c) = 1*J+2$$

somit ergibt sich gemäß Def. 3.9. der Multiplicator aus (1/1) = 1 und der Constant Offset aus (2 - 0) = 2. Das Localitytupel für C_2 lautet daher (1,2) und gemäß Def. 3.10. ist die locality von *e*:

$$\mathbf{loc}(\boldsymbol{e}) = \boldsymbol{constant}$$

Dependency: Arrayreferenz c in Anweisung S1 ist in keine true-dependence involviert, daher ergibt sich die Dependency gemäß Def. 3.12:

$$\mathbf{d}(\boldsymbol{e}) = \mathbf{0}$$

Somit ist die Kante vollständig attributiert und man kann deren Gewicht gemäß Def. 3.13. errechnen:

Für die maschinenabhängigen Parameter *transfertime* (τ) wurde ein Wert von 0.1 und für die *startuptime* (σ) ein Wert von 100 angenommen, was in etwa den realen Werten für die Meiko CS-2 entspricht.

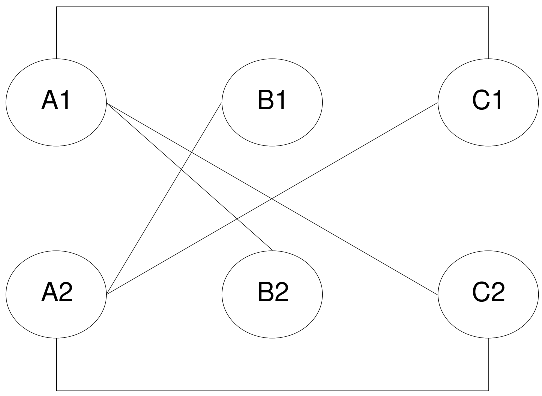
Da der variaton-level (2) größer als die Dependency (0) ist und L2 100 Iterationen bewirkt, ergibt sich die Nachrichtengröße mit $\theta = 100$

und die Ausführungshäufigkeit des Kommunikationsstatements mit $\phi = 1$.

Somit:

$$\boldsymbol{\omega}(\boldsymbol{e}) = \mathbf{(0.1 * 100 + 100) * 1 = 110}$$

Der resultierende CAG ist mit den gesamten Kantenattributen aus Abbildung 4.5 zu entnehmen:



Kantenattribute:

$e(A_1,B_2)$:

f = 100; loc = perfect; d = 0; ω = 110

$e(A_1,C_1)$:

f = 100; loc = constant, Localitytupel für C_1: (1,1); d = 0; ω = 110

$e(A_1,C_2)$:

f = 10000; loc = perfect; d = 2; ω = 1001000

$e(A_2,B_1)$:

f = 10000; loc = perfect; d = 0; ω = 110

$e(A_2,C_1)$:

f = 100; loc = perfect; d = 2; ω = 1001000

$e(A_2,C_2)$:

f = 10000; loc = constant, Localitytupel für C_2: (1,2); d = 0; ω = 110

Abb. 4.5 CAG

Nach Abschluß dieser Phase wird der gesamte CAG dem Benutzer zur Verfügung gestellt und darauf basierend der Heuristische Algorithmus angewandt.

3. Phase - Der Heuristische Algorithmus

Das Konzept des Heuristischen Algorithmus wurde bereits in Abschnitt 3.3.2 vorgestellt und dessen Arbeitsweise ist ebendort aus Abbildung 3.1 zu entnehmen.

Seine Arbeitsweise im Falle von Beispiel 4.1. sieht dergestalt aus, daß zuerst die beiden Knoten von Array A den Alignmentsets zugeteilt werden. Dann wird versucht, Array C dazuzufügen, was aber nicht ohne Konflikte zu bewerkstelligen ist, da beide Dimensionen von C sowohl mit der ersten als auch mit der zweiten Dimension aligned sind. Aufgrund der unterschiedlichen Kantengewichtungen ist es jedoch möglich, eine optimale Zuordnung zu bestimmen, indem C_2 der ersten Menge und C_1 der zweiten Menge zugeordnet werden.

Die Ermittlung dieser optimalen Zuordnung erfolgt mittels eines branch-and-bound Algorithmus [HiLi90]. Schlußendlich wird noch Array B den Mengen zugeordnet, was zu dem in Abb. 4.6. dargestellten Endresultat führt.

Abb. 4.6 - Alignmentsets

Nach Abschluß dieser 3 Phasen ist die Alignmentanalyse beendet. Die Resultate werden dem Benutzer zur Verfügung gestellt und stehen für weiterführende Analysen bereit.

Nach dieser kurzen Einführung in die Arbeitsweise des Alignmentanalysetools folgt noch ein kurzer Abriß über die Implementierung, bevor in Abschnitt 4.4 die Analyse zweier realer Benchmarkprogramme betrachtet wird.

4.3. Implementierungsdetails

In diesem Abschnitt werden einige Implementierungsdetails angeführt, wobei insbesondere auf die Datenstrukturen, die Konstruktion des CAG (2. Phase) und den Heuristischen Algorithmus (3. Phase) eingegangen werden wird. Die 1. Phase (das Frontend) besteht hauptsächlich aus Interaktion mit dem Benutzer sowie dem Testen und Aufrufen anderer Systemkomponenten, weshalb in diesem Rahmen nicht näher darauf eingegangen werden soll.

Die Prozedur *align_main* übernimmt die globalen Managementaufgaben der Alignmentanalyse, ruft die 2. und die 3. Phase auf und organisiert die Interaktion mit der Benutzeroberfläche sowie das Abspeichern der Resultatefiles.

4.3.1. Datenstrukturen

Für die interne Repräsentation des CAG bzw. der Alignmentsets wurden folgende Datenstrukturen deklariert:

```
typedef struct loc                              /* Localitykomponente */
{
   short  type;                 /* {perfect, constant, invariant} */
   float  size_mul;                              /* multiplicator */
   int    offset_min, offset_max;              /* constant-offset */
   struct Align_node *nd;         /* relativer Knoten für Locality */
} loc;


typedef struct Edge_attr                          /* Kantenattribute */
{
   float  weight;
   float  frequency;
   struct loc locality;
   short  dependency;
   int    variation_level;
   FNODE  *stmt;            /* Verweis zu jenem Knoten im Flußgraphen,
                                          der die Kante verursachte */
   struct Edge_attr *next; /* nächstes Attribut bei multiplen Kanten */
} Edge_attr;


typedef struct Edge_list          /* Liste von Kanten eines Knoten */
{
   Edge_attr         *attr;                            /* Attribut */
   struct Align_node *rght;           /* Zweiter Knoten der Kante */
   struct Edge_list  *next;          /* Kante zum nächsten Knoten */
} Edge_list;
```

```
typedef struct Align_node                              /* Knoten im CAG */
{
   ENTRY  *ident;                        /* Verweis auf Symboltabelle */
   short  dimension;                     /* Arraydimension des Knoten */
   struct Align_node *rght;                  /* nächste Spalte im CAG */
   struct Align_node *down;                   /* nächste Zeile im CAG */
   short marked;                            /* Markierungsmöglichkeit */
   Edge_list *edge;                /* Kanten, die vom Knoten ausgehen */
} Align_node;


typedef struct Nodeset   /*Alignmentset, als lineare Liste realisiert*/
{
   Align_node *nd;
   struct loc locality;                        /* Locality des Knoten */
   struct Nodeset *next;
} Nodeset;


Align_node *CAG;                              /* Startpointer des CAG */

Nodeset *Alignset[MAXDIM];                /* Array der Alignmentsets */
```

4.3.2. Konstruktion des CAG

Die Prozedur *align_main* ruft die Prozedur *construct_CAG* auf und übergibt dieser den Anfangs- und Endpunkt (als Referenzen auf den Syntaxbaum - Datenstruktur *ANODE*) der Analyse. *Construct_CAG* durchläuft den Syntaxbaum rekursiv vom Anfangs- bis zum Endpunkt und sucht hierbei das Vorhandensein von Do-Loops (im Syntaxbaum als N_DO dargestellt). Wird eine Do-Loop gefunden, so wird ein Verweis auf den entsprechenden Knoten im Flußgraphen in einem Array gespeichert (welches somit immer Verweise auf aktuelle Schleifen bietet) und die Prozedur *analyze_loop_body* mit dem Anfangspunkt des Schleifenkörpers als Parameter aufgerufen:

```
void
analyze_loop_body(do_body)
ANODE *do_body;
{
 ANODE *assign_stmt;                              /* Verweis auf ein zu
                                 analysierendes Zuweisungsstatement */

 switch (do_body->tp)          /* tp = Typ des Knoten im Syntaxbaum */
 {
 case N_CONTINUE:
    {
       /* Lösche Schleife aus dem Schleifenarray */
       return;
    }
 case N_DO:
    {
       /* Nimm Schleife im Schleifenarray auf */
       analyze_loop_body(/* Schleifenbody */);
       assign_stmt=NIL;
       break;
    }
 case N_ASSIGN:                              /* Zuweisungsstatement */
    {
       assign_stmt=do_body;
       true_ratio=1.0;
       break;
    }
 case N_LIF:                                           /* IF-Zweig */
    {
       if (/* Anweisung im if-Zweig ist ein Zuweisungsstatement*/)
       {
          assign_stmt= /* Anweisung im if-Zweig */
          true_ratio= /* true ratio Information des Weight Finders */
          break;
       }
    }
 default: assign_stmt=NIL;
 }

 if (assign_stmt!=NIL)
 {
    /* Wenn auf der linken Seite des Zuweisungsstatements eine
    Arrayreferenz steht, so berechne die Abhängigkeiten dieser
    Anweisung und analysiere die Alignmentrelationen zwischen der lin-
    ken und rechten Seite (Prozedur analyze_alignment_relations)
    sowie die Alignmentrelationen der rechten Seite (Prozedur analy-
    ze_rhs_relations);
    diese Managementaufgaben übernimmt die Prozedur analyze_statement
    */
 }
 analyze_loop_body(next_stmt(do_body));    /* Analyse der nächsten
                                           Anweisung */
}
```

```
void
analyze_alignment_relations(lhs_nd,rhs_tree)
ANODE *lhs_nd, *rhs_tree;
/* lhs_nd ist ein Pointer auf die Arrayreferenz der linken Seite des
Zuweisungsstatements, rhs_tree ein Pointer auf den
Syntaxbaumabschnitt, der die rechte Seite darstellt */
{
 foreach dimension of lhs_nd do
 {
    /* suche bzw. generiere den Knoten im CAG = lhs
       generiere die zugehörige Koeffizientenmatrix */

    foreach (arrayreference ∈ rhs_tree = rhs_nd) do
    {
        foreach dimension of rhs_nd do
        {
            /* suche bzw. generiere den Knoten im CAG = rhs
               generiere die zugehörige Koeffizientenmatrix */
            if (analysis_align(lhs_nd,rhs_nd)==TRUE)
            {
                /* generiere eine Kante zwischen den beiden Knoten lhs
                   und rhs und attributiere sie */
            }
        }
    }
 }
}


short
analys_align(lhs,rhs)
ANODE *lhs, *rhs;
{
 /* teste auf das Vorliegen einer Alignmentpräferenz:*/
 if (identity(lhs,rhs) ||
    ((single_index(lhs) && single_index(rhs)) &&
     (variation_level(lhs) == variation_level(rhs))))
 {
    return TRUE;
 }
 else
 {
    return FALSE;
 }
}
```

Die Prozedur *analyse_rhs_relations* entspricht der Prozedur *analyze_ alignment_relations*, wobei jedoch nur jene Knoten, die keine Alignmentpräferenz zur linken Seite aufweisen mit allen anderen Knoten der rechten Seite auf das Vorliegen einer solchen getestet werden.

Ist die Prozedur *construct_CAG* am Endpunkt der Analyse angelangt, steht der gesamte CAG in der internen Repräsentation zur Verfügung, und er wird von der Prozedur *align_main* in einem File abgespeichert. Sodann initiiert *align_main* den Heuristischen Algorithmus.

4.3.3. Der Heuristische Algorithmus

Nach der Konstruktion des CAG ruft *align_main* den Heuristischen Algorithmus auf, unter der Bedingung, daß überhaupt ein CAG konstruiert wurde. Die Vorgangsweise erfolgt konform der Darstellung in 3.3.2. durch die Prozedur *heuristic_algorithm*, welche den CAG übergeben bekommt und die nötigen Arbeiten ausführt. Das Finden eines optimalen Matchings im bipartiten Graphen erfolgt mittels eines branch-and-bound Algorithmus, welcher in der Prozedur *branch_and_bound* durchgeführt wird.

```
void
heuristic_algorithm(graph)
Align_node *graph;

{
    /* Initialisiere die Alignmentsets */
    /* Sortiere den Graphen */
    /* Weise die erste Spalte des Graphen den Alignmentsets zu */

    graph=graph->rght; /* Die erste Spalte wird nicht mehr benötigt */

    while (graph!=NIL)
    {
       bipartite_graph = /* Bilde einen Bipartiten Graphen aus den
                            Alignmentsets und der ersten Spalte von
                            graph und konstruiere sowie gewichte die
                            Kanten gemäß den in 3.3.1. und 3.3.2. be-
                            schriebenen Vorgangsweisen. */

       optimal_matching = branch_and_bound(bipartite_graph);

       /* Weise die Knoten gemäß dem optimal_matching den Alignment-
          sets zu und versieh sie mit der berechneten Locality der
          Kante im bipartiten Graphen (vgl. Def. 3.14) */

       graph=graph->rght;    /* Gehe zur nächsten Spalte */
    }
}
```

Die Prozedur *branch_and_bound* führt den branch-and-bound Algorithmus am Bipartiten Graphen durch, wobei solange partielle Lösungen generiert werden, bis keine Verbesserung des Gesamtresultats mehr möglich ist. Als Gewichtungsfunktion für die Güte einer Lösung wird das kummulierte Gewicht aller Kanten, die ein Alignmentset verlassen, berechnet; dies ist die zu minimierende Zielfunktion. Die Prozedur *branch_and_bound* generiert partielle Lösungen über den Aufruf der Prozedur *branch*, welcher als Parameter die Ausgangslösung, die Tiefe dieser Ausgangslösung (das ist die Anzahl der bereits fix angeordneten Elemente), der Bipartite Graph sowie Zeiger auf die bisherige beste Lösung und das bisherige beste Gewicht übergeben werden.

```
Partial_solution *
branch_and_bound(graph)
Bipartite_graph * graph;
{
    /* Initialisierung der Optima */
    best_solution = NIL;
    best_weight = 0.0;
    PS=alloc_partial_solution();
    /*  die Struktur PS ist der Baum, der sämtliche generierte
        partiellen Lösungen speichert */

    /*  Generierung der ersten Lösungen (Tiefe=0, da noch kein Element
        fixiert wurde) */
    branch(PS,graph,0,&best_solution,&best_weight);

    /*  Suche in den generierten Lösungen nach einer, die ein besseres
        Ergebnis als die bisher gespeicherte beste Lösung ergibt */
    help=search_min_solution(PS,best_weight);
    while (help!=NIL)
    {
        branch(PS,graph,help->depth,&best_solution,&best_weight);
        help=search_min_solution(PS,best_weight);
    }
    return best_solution;
}
```

In der Prozedur *branch* wird eine Ausgangslösung erweitert, indem an der nächsten Stelle der übergebenen Tiefe alle noch nicht angeordneten Knoten der Reihe nach eingefügt werden und die Lösung mit den restlichen Knoten aufgefüllt wird. Sodann wird jede erhaltene Lösung evaluiert, was bedeutet, daß ihre Erlaubtheit überprüft und ihre Zielfunktion berechnet wird. Liefert eine solche partielle Lösung ein besseres Ergebnis als die bisherige beste Lösung und ist sie erlaubt, so wird sie als neue beste Lösung gespeichert. Ist eine Lösung nicht erlaubt, liefert jedoch ein besseres Ergebnis als die beste Lösung, so wird sie zur weiteren Bearbeitung in die Struktur der partiellen Lösungen aufgenommen; andernfalls wird der Zweig verworfen.

Nähere Details zur Arbeitsweise und Implementierung von branch-and-bound Algorithmen finden sich in [HiLi90].

Nach Beendigung des Heuristischen Algorithmus wurden alle Knoten des CAG den Alignmentsets zugeordnet, diese werden gespeichert und dem Benutzer zur Verfügung gestellt.

4.4. Anwendungsbeispiele

In diesem Kapitel werden die Anwendung sowie die Resultate des Alignmentanalysetools anhand zweier realer Benchmarkprogramme veranschaulicht. Insbesondere soll gezeigt werden, wie durch die Anwendung der *scalar forward substitution* die Mächtigkeit der Analyse erhöht und ihre Resultate verbessert werden können. Die dazu angeführten Programmcodes werden mit den entsprechenden Weight-Finder Informationen dargestellt, welche am rechten Rand ausgeworfen werden.

Als Testprogramme wurden die Routinen *eflux* und *tred2* gewählt, um verschiedene Merkmale der Analyse darzustellen.
Die Routine *eflux* ist eine der zeitaufwendigsten Prozeduren des Programmes *flo52*, welches sich mit der Auswirkung des Überschalls auf Tragflächen beschäftigt und hiezu Euler'sche Gleichungen löst.
Die Routine *tred2* wurde aus der Eispack Bibliothek entnommen und reduziert eine symmetrische Matrix auf eine symmetrische Dreiecksmatrix.

Die für die Kantengewichtungen notwendigen maschinenabhängigen Parameter wurden mit 0,1 für die Transferzeit und 100 für die Startuptime festgelegt, was in etwa den realen Werten für die Meiko CS-2 entspricht.

4.4.1. EFLUX

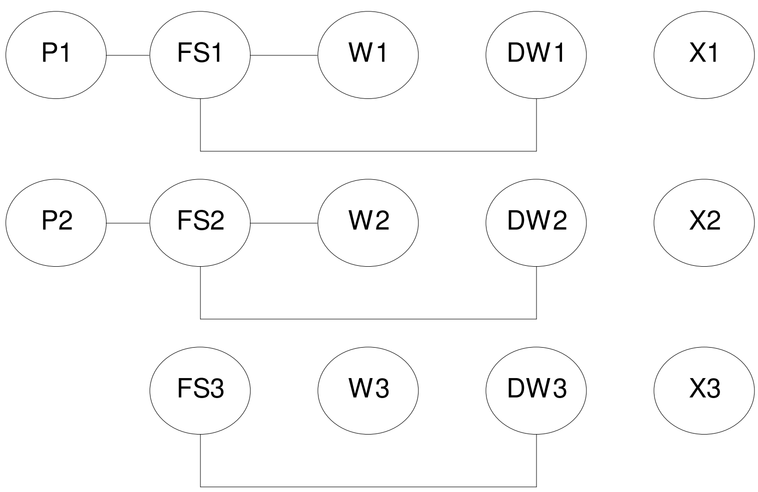
Die Eflux Routine führt regelmäßige Berechnungen auf 3dimensionalen Arrays aus, wobei keine konkurrierenden Alignmentpräferenzen festgestellt werden. Da jedoch zur Berechnung eines neuen Wertes auch seine Umgebung einbezogen wird, kann die Attributierung der Alignmentkanten mit der Locality nachvollzogen werden. Der Programmcode kann Abb. 4.7. entnommen werden.

```
      PROGRAM MAIN                                                          <-- FR=1

      PARAMETER(I2=194,J2=34,IL=193,JL=33)
      PROCESSORS R(4,4)
      INTEGER J1D,I2D,K3D,J1D00,I2D00,N1D,J2D,I3D,I1D,J1D01,I2D01,N1D00,
     *J2D00,I3D00
      REAL DW(194,34,4)
      REAL FS(193,34,4)
      REAL W(I2,J2,4)
      REAL P(I2,J2)
      REAL X(I2,J2,2)
      REAL SUM
      INTEGER I,J,K

      DO 6 J1D=1,194                                      <-- FR=1; LP=[1.00;194.00]
        DO 7 I2D=1,34                                     <-- FR=194; LP=[1.00;34.00]
          P(J1D,I2D) = J1D - 2 * I2D                                     <-- FR=6596
          X(J1D,I2D,1) = (-J1D) + 2 * I2D                                <-- FR=6596
          X(J1D,I2D,2) = J1D + 2 * I2D                                   <-- FR=6596
          DO 5 K3D=1,4                                    <-- FR=6596; LP=[1.00;4.00]
            W(J1D,I2D,K3D) = I2D * K3D                                   <-- FR=26384
5         CONTINUE                                                       <-- FR=26384
7       CONTINUE                                                         <-- FR=6596
6     CONTINUE                                                           <-- FR=194
      DO 10 J1D00=2,33                                    <-- FR=1; LP=[2.00;33.00]
        DO 11 I2D00=1,193                                 <-- FR=32; LP=[1.00;193.00]
          XY = X(I2D00,J1D00,1) - X(I2D00,(-1) + J1D00,1)                <-- FR=6176
          YY = X(I2D00,J1D00,2) - X(I2D00,(-1) + J1D00,2)                <-- FR=6176
          PA = P(1 + I2D00,J1D00) + P(I2D00,J1D00)                       <-- FR=6176
          QSP = (YY * W(1 + I2D00,J1D00,2) - XY * W(1 + I2D00,J1D00,3)) / W(1 + I2D00,J1D00,1)      <-- FR=6176
          QSM = (YY * W(I2D00,J1D00,2) - XY * W(I2D00,J1D00,3)) / W(I2D00,J1D00,1)                  <-- FR=6176
          FS(I2D00,J1D00,1) = QSP * W(1 + I2D00,J1D00,1) + QSM * W(2D00,J1D00,1)                    <-- FR=6176
          FS(I2D00,J1D00,2) = QSP * W(1 + I2D00,J1D00,2) + QSM * W(I2D00,J1D00,2) + YY * PA         <-- FR=6176
          FS(I2D00,J1D00,3) = QSP * W(1 + I2D00,J1D00,3) + QSM * W(I2D00,J1D00,3) - XY * PA         <-- FR=6176
          FS(I2D00,J1D00,4) = (P(1 + I2D00,J1D00) + W(1 + I2D00,J1D00,4)
     *    ) * QSP + (P(I2D00,J1D00) + W(I2D00,J1D00,4)) * QSM                                       <-- FR=6176
11      CONTINUE                                                         <-- FR=6176
10    CONTINUE                                                           <-- FR=32
      DO 20 N1D=1,4                                       <-- FR=1; LP=[1.00;4.00]
        DO 21 J2D=2,33                                    <-- FR=4; LP=[2.00;33.00]
          DO 22 I3D=2,193                                 <-- FR=128; LP=[2.00;193.00]
            DW(I3D,J2D,N1D) = FS(I3D,J2D,N1D) - FS((-1) + I3D,J2D,N1D)    <-- FR=24576
22        CONTINUE                                                       <-- FR=24576
21      CONTINUE                                                         <-- FR=128
20    CONTINUE                                                           <-- FR=4
      DO 25 I1D=2,193                                     <-- FR=1; LP=[2.00;193.00]
        XX = X(I1D,1,1) - X((-1) + I1D,1,1)                              <-- FR=192
        YX = X(I1D,1,2) - X((-1) + I1D,1,2)                              <-- FR=192
        PA = P(I1D,2) + P(I1D,1)                                         <-- FR=192
        FS(I1D,1,1) = 0.                                                 <-- FR=192
        FS(I1D,1,2) = (-YX) * PA                                         <-- FR=192
        FS(I1D,1,3) = XX * PA                                            <-- FR=192
        FS(I1D,1,4) = 0.                                                 <-- FR=192
25    CONTINUE                                                           <-- FR=192
```

```
      DO 30 J1D01=2,33                                                     <-- FR=1; LP=[2.00;33.00]
        DO 31 I2D01=2,193                                                 <-- FR=32; LP=[2.00;193.00]
          XX = X(I2D01,J1D01,1) - X((-1) + I2D01,J1D01,1)                                <-- FR=6144
          YX = X(I2D01,J1D01,2) - X((-1) + I2D01,J1D01,2)                                <-- FR=6144
          PA = P(I2D01,1 + J1D01) + P(I2D01,J1D01)                                       <-- FR=6144
          QSP = (XX * W(I2D01,1 + J1D01,3) - YX * W(I2D01,1 + J1D01,2)) / W(I2D01,1 + J1D01,1)      <-- FR=6144
          QSM = (XX * W(I2D01,J1D01,3) - YX * W(I2D01,J1D01,2)) / W(I2D01,J1D01,1)       <-- FR=6144
          FS(I2D01,J1D01,1) = QSP * W(I2D01,1 + J1D01,1) + QSM * W(I2D01,J1D01,1)        <-- FR=6144
          FS(I2D01,J1D01,2) = QSP * W(I2D01,1 + J1D01,2) + QSM * W(I2D01,J1D01,2) - YX * PA         <-- FR=6144
          FS(I2D01,J1D01,3) = QSP * W(I2D01,1 + J1D01,3) + QSM * W(I2D01,J1D01,3) + XX * PA         <-- FR=6144
          FS(I2D01,J1D01,4) = (P(I2D01,1 + J1D01) + W(I2D01,1 + J1D01,4)
     *    ) * QSP + (P(I2D01,J1D01) + W(I2D01,J1D01,4)) * QSM                             <-- FR=6144
 31     CONTINUE                                                                          <-- FR=6144
 30   CONTINUE                                                                              <-- FR=32
      DO 40 N1D00=1,4                                                       <-- FR=1; LP=[1.00;4.00]
        DO 41 J2D00=2,33                                                   <-- FR=4; LP=[2.00;33.00]
          DO 42 I3D00=2,193                                              <-- FR=128; LP=[2.00;193.00]
            DW(I3D00,J2D00,N1D00) = DW(I3D00,J2D00,N1D00) + FS(I3D00,
     *      J2D00,N1D00) - FS(I3D00,(-1) + J2D00,N1D00)                                  <-- FR=24576
 42       CONTINUE                                                                       <-- FR=24576
 41     CONTINUE                                                                           <-- FR=128
 40   CONTINUE                                                                               <-- FR=4
      WRITE(*,*) 'finished'                                                                  <-- FR=1
      STOP
      END
```

Abb. 4.7 - Eflux

Der aus diesem Programm generierte CAG ist mit seinen Kantenattributen in Abb. 4.8. dargestellt.



Kantenattribute:

$e(P_1,FS_1)$:

f = 12320; loc = constant, Localitytupel für P_1: (1,0/1); $\omega = 238,5$

$e(P_2,FS_2)$:

f = 64; loc = constant, Localitytupel für P_2: (1,0/1); $\omega = 206,4$

$e(W_1,FS_1)$:

f = 49280; loc = constant, Localitytupel für W_1: (1,0/1); $\omega = 954$

$e(W_2,FS_2)$:

f = 256; loc = constant, Localitytupel für W_2: (1,0/1); $\omega = 825,6$

$e(FS_1,DW_1)$:

f = 49152; loc = constant, Localitytupel für FS_1: (1,-1/0); $\omega = 238,4$

$e(FS_2,DW_2)$:

f = 256; loc = constant, Localitytupel für DW_2: (1,-1/0); $\omega = 206,4$

$e(FS_3,DW_3)$:

f = 8; loc = perfect; $\omega = 200,8$

Abb. 4.8 - CAG von Eflux

Aus diesem Ergebnis kann man einerseits die Darstellung des intradimensionalen Alignments ersehen (die meisten Knoten haben eine konstante Locality mit schwankenden Offsets), andererseits kann das Phänomen festgestellt werden, daß das Array X keine Alignmentpräferenzen aufweist. Dies ist dadurch bedingt, daß die Werte von Array X ausschließlich temporären Variblen zugewiesen werden und nie anderen Arrayelementen. Entsprechend den vorgestellten Vorgangsweisen beim Heuristischen Algorithmus können die drei Dimensionen von Array X den Alignmentsets zufällig zugeordnet werden, was zu den in Abb. 4.9. dargestellten Lösungsmengen führt.

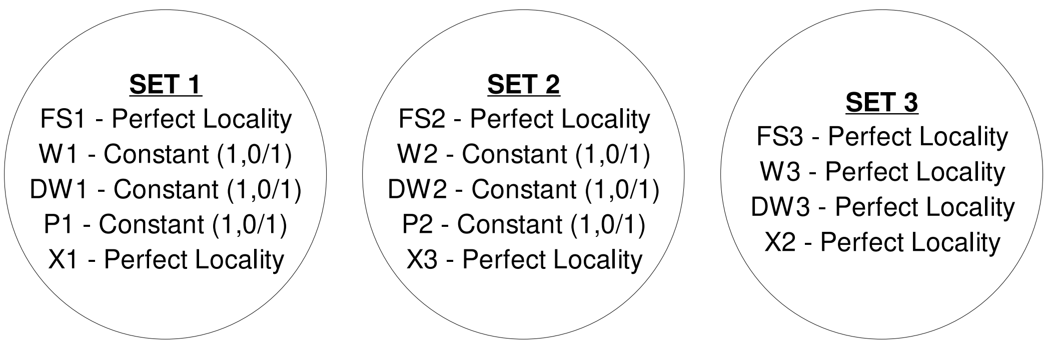


Abb. 4.9 - Alignmentsets von Eflux

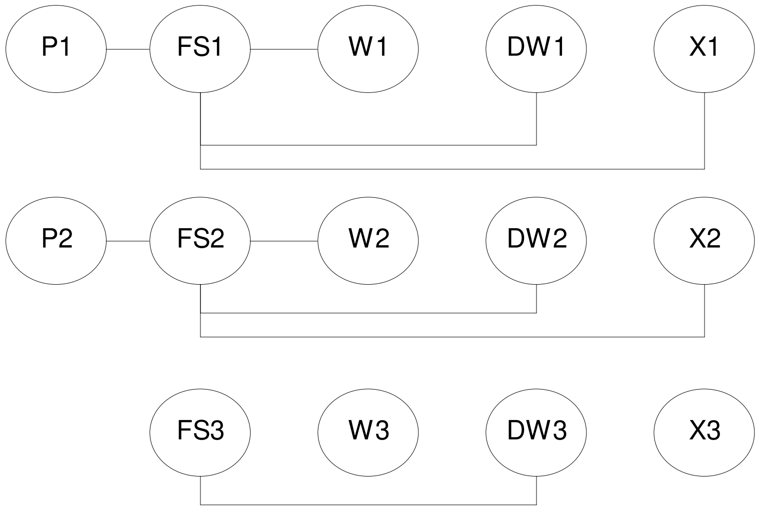
Wird hingegen vor Durchführung der Alignmentanalyse die *scalar forward substitution* durchgeführt, können auch für Array X Alignmentpräferenzen festgestellt und dessen Elemente somit eindeutig den Alignmentsets zugeordnet werden. Der modifizierte Programmcode samt den daraus resultierenden Ergebnissen der Alignmentanalyse ist in Abb. 4.10. dargestellt.

```
      PROGRAM MAIN                                                          <-- FR=1

      PARAMETER(I2=194,J2=34,IL=193,JL=33)
      PROCESSORS R(4,4)
      INTEGER J1D02,I2D02,K3D00,J1D03,I2D03,N1D01,J2D01,I3D01,I1D00,
     *J1D04,I2D04,N1D02,J2D02,I3D02
      INTEGER J1D,I2D,K3D,J1D00,I2D00,N1D,J2D,I3D,I1D,J1D01,I2D01,N1D00,
     *J2D00,I3D00
      REAL DW(194,34,4) DIST(BLOCK,BLOCK,:)
      REAL FS(193,34,4) DIST(BLOCK,BLOCK,:)
      REAL W(I2,J2,4) DIST(BLOCK,BLOCK,:)
      REAL P(I2,J2) DIST(BLOCK,BLOCK)
      REAL X(I2,J2,2) DIST(BLOCK,BLOCK,:)
      REAL SUM
      INTEGER I,J,K

      DO 6 J1D02=1,194                                   <-- FR=1; LP=[1.00;194.00]
        DO 7 I2D02=1,34                                 <-- FR=194; LP=[1.00;34.00]
          P(J1D02,I2D02) = J1D02 - 2 * I2D02                               <-- FR=6596
          X(J1D02,I2D02,1) = (-J1D02) + 2 * I2D02                          <-- FR=6596
          X(J1D02,I2D02,2) = J1D02 + 2 * I2D02                             <-- FR=6596
          DO 5 K3D00=1,4                                <-- FR=6596; LP=[1.00;4.00]
            W(J1D02,I2D02,K3D00) = I2D02 * K3D00                          <-- FR=26384
5         CONTINUE                                                        <-- FR=26384
7       CONTINUE                                                           <-- FR=6596
6     CONTINUE                                                              <-- FR=194
      DO 10 J1D03=2,33                                     <-- FR=1; LP=[2.00;33.00]
        DO 11 I2D03=1,193                                <-- FR=32; LP=[1.00;193.00]
          FS(I2D03,J1D03,1) = (W(1 + I2D03,J1D03,2) * X(I2D03,J1D03,2) -
     *     W(1 + I2D03,J1D03,2) * X(I2D03,(-1) + J1D03,2) - W(1 + I2D03,
     *    J1D03,3) * X(I2D03,J1D03,1) + W(1 + I2D03,J1D03,3) * X(I2D03,(
     *    -1) + J1D03,1)) / W(1 + I2D03,J1D03,1) * W(1 + I2D03,J1D03,1)
     *    + (W(I2D03,J1D03,2) * X(I2D03,J1D03,2) - W(I2D03,J1D03,2) * X(
     *    I2D03,(-1) + J1D03,2) - W(I2D03,J1D03,3) * X(I2D03,J1D03,1) +
     *    W(I2D03,J1D03,3) * X(I2D03,(-1) + J1D03,1)) / W(I2D03,J1D03,1)
     *     * W(I2D03,J1D03,1)                                              <-- FR=6176
          FS(I2D03,J1D03,2) = (W(1 + I2D03,J1D03,2) * X(I2D03,J1D03,2) -
     *     W(1 + I2D03,J1D03,2) * X(I2D03,(-1) + J1D03,2) - W(1 + I2D03,
     *    J1D03,3) * X(I2D03,J1D03,1) + W(1 + I2D03,J1D03,3) * X(I2D03,(
     *    -1) + J1D03,1)) / W(1 + I2D03,J1D03,1) * W(1 + I2D03,J1D03,2)
     *    + (W(I2D03,J1D03,2) * X(I2D03,J1D03,2) - W(I2D03,J1D03,2) * X(
     *    I2D03,(-1) + J1D03,2) - W(I2D03,J1D03,3) * X(I2D03,J1D03,1) +
     *    W(I2D03,J1D03,3) * X(I2D03,(-1) + J1D03,1)) / W(I2D03,J1D03,1)
     *     * W(I2D03,J1D03,2) + (X(I2D03,J1D03,2) - X(I2D03,(-1) +
     *    J1D03,2)) * (P(1 + I2D03,J1D03) + P(I2D03,J1D03))               <-- FR=6176
          FS(I2D03,J1D03,3) = (W(1 + I2D03,J1D03,2) * X(I2D03,J1D03,2) -
```

```
     *     W(1 + I2D03,J1D03,2) * X(I2D03,(-1) + J1D03,2) - W(1 + I2D03,
     *    J1D03,3) * X(I2D03,J1D03,1) + W(1 + I2D03,J1D03,3) * X(I2D03,(
     *    -1) + J1D03,1)) / W(1 + I2D03,J1D03,1) * W(1 + I2D03,J1D03,3)
     *    + (W(I2D03,J1D03,2) * X(I2D03,J1D03,2) - W(I2D03,J1D03,2) * X(
     *    I2D03,(-1) + J1D03,2) - W(I2D03,J1D03,3) * X(I2D03,J1D03,1) +
     *    W(I2D03,J1D03,3) * X(I2D03,(-1) + J1D03,1)) / W(I2D03,J1D03,1)
     *     * W(I2D03,J1D03,3) - (X(I2D03,J1D03,1) - X(I2D03,(-1) +
     *    J1D03,1)) * (P(1 + I2D03,J1D03) + P(I2D03,J1D03))                     <-- FR=6176
28:           FS(I2D03,J1D03,4) = (P(1 + I2D03,J1D03) * W(1 + I2D03,J1D03,2)
     *     * X(I2D03,J1D03,2) - P(1 + I2D03,J1D03) * W(1 + I2D03,J1D03,
     *    2) * X(I2D03,(-1) + J1D03,2) - P(1 + I2D03,J1D03) * W(1 +
     *    I2D03,J1D03,3) * X(I2D03,J1D03,1) + P(1 + I2D03,J1D03) * W(1 +
     *     I2D03,J1D03,3) * X(I2D03,(-1) + J1D03,1) + W(1 + I2D03,J1D03,
     *    4) * W(1 + I2D03,J1D03,2) * X(I2D03,J1D03,2) - W(1 + I2D03,
     *    J1D03,4) * W(1 + I2D03,J1D03,2) * X(I2D03,(-1) + J1D03,2) - W(
     *    1 + I2D03,J1D03,4) * W(1 + I2D03,J1D03,3) * X(I2D03,J1D03,1) +
     *     W(1 + I2D03,J1D03,4) * W(1 + I2D03,J1D03,3) * X(I2D03,(-1) +
     *    J1D03,1)) / W(1 + I2D03,J1D03,1) + (P(I2D03,J1D03) * W(I2D03,
     *    J1D03,2) * X(I2D03,J1D03,2) - P(I2D03,J1D03) * W(I2D03,J1D03,
     *    2) * X(I2D03,(-1) + J1D03,2) - P(I2D03,J1D03) * W(I2D03,J1D03,
     *    3) * X(I2D03,J1D03,1) + P(I2D03,J1D03) * W(I2D03,J1D03,3) * X(
     *    I2D03,(-1) + J1D03,1) + W(I2D03,J1D03,4) * W(I2D03,J1D03,2) *
     *    X(I2D03,J1D03,2) - W(I2D03,J1D03,4) * W(I2D03,J1D03,2) * X(
     *    I2D03,(-1) + J1D03,2) - W(I2D03,J1D03,4) * W(I2D03,J1D03,3) *
     *    X(I2D03,J1D03,1) + W(I2D03,J1D03,4) * W(I2D03,J1D03,3) * X(
     *    I2D03,(-1) + J1D03,1)) / W(I2D03,J1D03,1)                            <-- FR=6176
29: 11      CONTINUE                                                           <-- FR=6176
30: 10    CONTINUE                                                               <-- FR=32
31:       DO 20 N1D01=1,4                                         <-- FR=1; LP=[1.00;4.00]
32:         DO 21 J2D01=2,33                                      <-- FR=4; LP=[2.00;33.00]
33:           DO 22 I3D01=2,193                                <-- FR=128; LP=[2.00;193.00]
34:             DW(I3D01,J2D01,N1D01) = FS(I3D01,J2D01,N1D01) - FS((-1) +
     *      I3D01,J2D01,N1D01)                                                <-- FR=24576
35: 22        CONTINUE                                                        <-- FR=24576
36: 21      CONTINUE                                                            <-- FR=128
37: 20    CONTINUE                                                                <-- FR=4
38:       DO 25 I1D00=2,193                                    <-- FR=1; LP=[2.00;193.00]
39:         FS(I1D00,1,1) = 0.                                                  <-- FR=192
40:         FS(I1D00,1,2) = (-X(I1D00,1,2)) * P(I1D00,2) - X(I1D00,1,2) * P(
     *  I1D00,1) + X((-1) + I1D00,1,2) * P(I1D00,2) + X((-1) + I1D00,1,2) * P(I1D00,1)       <-- FR=192
41:         FS(I1D00,1,3) = X(I1D00,1,1) * P(I1D00,2) + X(I1D00,1,1) * P(
     *  I1D00,1) - X((-1) + I1D00,1,1) * P(I1D00,2) - X((-1) + I1D00,1,1) * P(I1D00,1)       <-- FR=192
42:         FS(I1D00,1,4) = 0.                                                  <-- FR=192
43: 25    CONTINUE                                                              <-- FR=192
44:       DO 30 J1D04=2,33                                     <-- FR=1; LP=[2.00;33.00]
45:         DO 31 I2D04=2,193                                 <-- FR=32; LP=[2.00;193.00]
46:           FS(I2D04,J1D04,1) = (W(I2D04,1 + J1D04,3) * X(I2D04,J1D04,1) -
     *     W(I2D04,1 + J1D04,3) * X((-1) + I2D04,J1D04,1) - W(I2D04,1 +
     *    J1D04,2) * X(I2D04,J1D04,2) + W(I2D04,1 + J1D04,2) * X((-1) +
     *    I2D04,J1D04,2)) / W(I2D04,1 + J1D04,1) * W(I2D04,1 + J1D04,1)
     *    + (W(I2D04,J1D04,3) * X(I2D04,J1D04,1) - W(I2D04,J1D04,3) * X(
     *    (-1) + I2D04,J1D04,1) - W(I2D04,J1D04,2) * X(I2D04,J1D04,2) +
     *    W(I2D04,J1D04,2) * X((-1) + I2D04,J1D04,2)) / W(I2D04,J1D04,1)
     *     * W(I2D04,J1D04,1)                                                <-- FR=6144
47:           FS(I2D04,J1D04,2) = (W(I2D04,1 + J1D04,3) * X(I2D04,J1D04,1) -
```

```
     *     W(I2D04,1 + J1D04,3) * X((-1) + I2D04,J1D04,1) - W(I2D04,1 +
     *    J1D04,2) * X(I2D04,J1D04,2) + W(I2D04,1 + J1D04,2) * X((-1) +
     *    I2D04,J1D04,2)) / W(I2D04,1 + J1D04,1) * W(I2D04,1 + J1D04,2)
     *    + (W(I2D04,J1D04,3) * X(I2D04,J1D04,1) - W(I2D04,J1D04,3) * X(
     *    (-1) + I2D04,J1D04,1) - W(I2D04,J1D04,2) * X(I2D04,J1D04,2) +
     *    W(I2D04,J1D04,2) * X((-1) + I2D04,J1D04,2)) / W(I2D04,J1D04,1)
     *     * W(I2D04,J1D04,2) - (X(I2D04,J1D04,2) - X((-1) + I2D04,
     *    J1D04,2)) * (P(I2D04,1 + J1D04) + P(I2D04,J1D04))                <-- FR=6144
48:           FS(I2D04,J1D04,3) = (W(I2D04,1 + J1D04,3) * X(I2D04,J1D04,1) -
     *     W(I2D04,1 + J1D04,3) * X((-1) + I2D04,J1D04,1) - W(I2D04,1 +
     *    J1D04,2) * X(I2D04,J1D04,2) + W(I2D04,1 + J1D04,2) * X((-1) +
     *    I2D04,J1D04,2)) / W(I2D04,1 + J1D04,1) * W(I2D04,1 + J1D04,3)
     *    + (W(I2D04,J1D04,3) * X(I2D04,J1D04,1) - W(I2D04,J1D04,3) * X(
     *    (-1) + I2D04,J1D04,1) - W(I2D04,J1D04,2) * X(I2D04,J1D04,2) +
     *    W(I2D04,J1D04,2) * X((-1) + I2D04,J1D04,2)) / W(I2D04,J1D04,1)
     *     * W(I2D04,J1D04,3) + (X(I2D04,J1D04,1) - X((-1) + I2D04,
     *    J1D04,1)) * (P(I2D04,1 + J1D04) + P(I2D04,J1D04))                <-- FR=6144
49:           FS(I2D04,J1D04,4) = (P(I2D04,1 + J1D04) * W(I2D04,1 + J1D04,3)
     *     * X(I2D04,J1D04,1) - P(I2D04,1 + J1D04) * W(I2D04,1 + J1D04,
     *    3) * X((-1) + I2D04,J1D04,1) - P(I2D04,1 + J1D04) * W(I2D04,1
     *    + J1D04,2) * X(I2D04,J1D04,2) + P(I2D04,1 + J1D04) * W(I2D04,
     *    1 + J1D04,2) * X((-1) + I2D04,J1D04,2) + W(I2D04,1 + J1D04,4)
     *    * W(I2D04,1 + J1D04,3) * X(I2D04,J1D04,1) - W(I2D04,1 + J1D04,
     *    4) * W(I2D04,1 + J1D04,3) * X((-1) + I2D04,J1D04,1) - W(I2D04,
     *    1 + J1D04,4) * W(I2D04,1 + J1D04,2) * X(I2D04,J1D04,2) + W(
     *    I2D04,1 + J1D04,4) * W(I2D04,1 + J1D04,2) * X((-1) + I2D04,
     *    J1D04,2)) / W(I2D04,1 + J1D04,1) + (P(I2D04,J1D04) * W(I2D04,
     *    J1D04,3) * X(I2D04,J1D04,1) - P(I2D04,J1D04) * W(I2D04,J1D04,
     *    3) * X((-1) + I2D04,J1D04,1) - P(I2D04,J1D04) * W(I2D04,J1D04,
     *    2) * X(I2D04,J1D04,2) + P(I2D04,J1D04) * W(I2D04,J1D04,2) * X(
     *    (-1) + I2D04,J1D04,2) + W(I2D04,J1D04,4) * W(I2D04,J1D04,3) *
     *    X(I2D04,J1D04,1) - W(I2D04,J1D04,4) * W(I2D04,J1D04,3) * X((-
     *    1) + I2D04,J1D04,1) - W(I2D04,J1D04,4) * W(I2D04,J1D04,2) * X(
     *    I2D04,J1D04,2) + W(I2D04,J1D04,4) * W(I2D04,J1D04,2) * X((-1)
     *    + I2D04,J1D04,2)) / W(I2D04,J1D04,1)                             <-- FR=6144
50: 31      CONTINUE                                                       <-- FR=6144
51: 30    CONTINUE                                                         <-- FR=32
52:       DO 40 N1D02=1,4                                   <-- FR=1; LP=[1.00;4.00]
53:         DO 41 J2D02=2,33                               <-- FR=4; LP=[2.00;33.00]
54:           DO 42 I3D02=2,193                         <-- FR=128; LP=[2.00;193.00]
55:             DW(I3D02,J2D02,N1D02) = DW(I3D02,J2D02,N1D02) + FS(I3D02,
     *          J2D02,N1D02) - FS(I3D02,(-1) + J2D02,N1D02)                <-- FR=24576
56: 42        CONTINUE                                                     <-- FR=24576
57: 41      CONTINUE                                                       <-- FR=128
58: 40    CONTINUE                                                         <-- FR=4
59:       WRITE(*,*) 'finished'                                            <-- FR=1
60:       STOP
61:       END
```

Der generierte CAG hat folgende Gestalt und Kantenattribute:



Kantenattribute:

$e(P_1,FS_1)$:

f = 37344; loc = constant, Localitytupel für P_1: (1,0/1); $\omega = 953{,}9$

$e(P_2,FS_2)$:

f = 192; loc = constant, Localitytupel für P_2: (1,0/1); $\omega = 619{,}2$

$e(X_1,FS_1)$:

f = 49664; loc = constant, Localitytupel für X_1: (1,-1/0); $\omega = 1192{,}4$

$e(X_2,FS_2)$:

f = 256; loc = constant, Localitytupel für X_2: (1,-1/0); $\omega = 825{,}6$

$e(W_1,FS_1)$:

f = 49280; loc = constant, Localitytupel für W_1: (1,0/1); $\omega = 954$

$e(W_2,FS_2)$:

f = 256; loc = constant, Localitytupel für W_2: (1,0/1); $\omega = 825{,}6$

$e(FS_1,DW_1)$:

f = 49152; loc = constant, Localitytupel für FS_1: (1,-1/0); $\omega = 238{,}4$

$e(FS_2,DW_2)$:

f = 256; loc = constant, Localitytupel für FS_2: (1,-1/0); $\omega = 206{,}4$

$e(FS_3,DW_3)$:

f = 8; loc = perfect; $\omega = 200{,}8$

Die durch den Heuristischen Algorithmus vorgenommene Zuteilung der einzelnen Knoten zu den Alignmentsets führte zu folgendem Resultat:

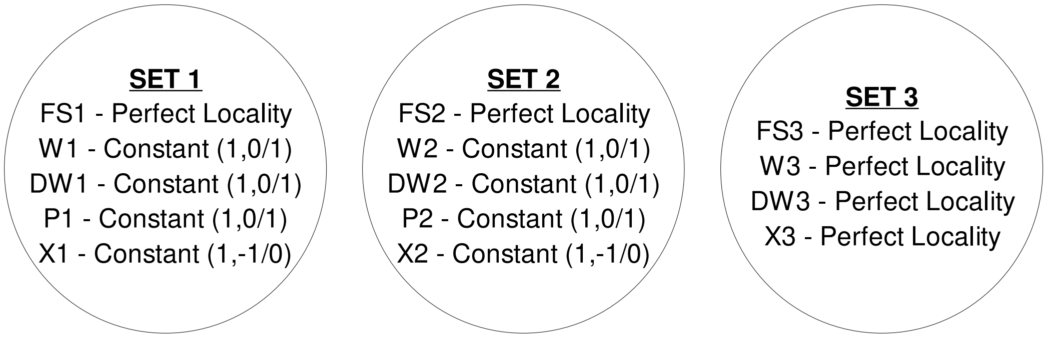


Abb. 4.10 - Eflux nach scalar forward substitution

4.4.2. TRED2

Am Beispiel der Tred2 Routine kann die Vorgangsweise des Heuristischen Algorithmus bei konkurrierenden Alignmentpräferenzen nachvollzogen werden. Weiters kann die Auswirkung der Gewichtungsfunktion gezeigt werden, da Alignmentpräferenzen mit ähnlichen Frequency-Attributen aufgrund der unterschiedlichen Ausführungshäufigkeit der Kommunikationsanweisungen verschieden gewichtet werden. Der Programmcode ist Abb. 4.11. zu entnehmen.

```
      SUBROUTINE TRED2(N,A,D,E,Z)                                   <-- FR=1

      LOGICAL BR1,BR2,BR3,BR4
      INTEGER I1D,J2D,I1D00,K2D,J2D00,K2D00,J2D01,J2D02,K3D,J2D03,J2D04,
     *J2D05,K3D00,I1D01,K2D01,J2D06,K3D01,K3D02,K2D02,I1D02
      INTEGER I,J,K,L,N,II,JP1
      DOUBLE PRECISION A(N,N),D(N),E(N),Z(N,N)
      DOUBLE PRECISION F,G,H,HH,SCALE

      DO 100 I1D=1,48                              <-- FR=1; LP=[1.00;48.00]
        DO 81 J2D=I1D,48                          <-- FR=48; LP=[24.50;48.00]
          Z(J2D,I1D) = A(J2D,I1D)                                <-- FR=1176
 81     CONTINUE                                                 <-- FR=1176
        D(I1D) = A(48,I1D)                                         <-- FR=48
 100  CONTINUE                                                     <-- FR=48
      IF (48 .EQ. 1) GOTO 510                              <-- FR=1; TR=0.00
      DO 300 I1D00=2,48                            <-- FR=1; LP=[2.00;48.00]
        I = 50 - I1D00                                             <-- FR=47
```

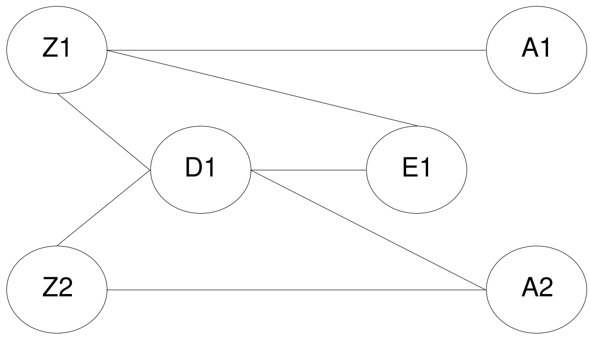
```
      L = I - 1                                                                    <-- FR=47
      H = 0.0D0                                                                    <-- FR=47
      SCALE = 0.0D0                                                                <-- FR=47
      BR1 = L .LT. 2                                                               <-- FR=47
      DO 121 K2D=1,L                                                 <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1) SCALE = SCALE + DABS(D(K2D))                         <-- FR=1128; TR=1.00
121   CONTINUE                                                                   <-- FR=1128
      IF (.NOT.BR1) BR2 = SCALE .NE. 0.0D0                                   <-- FR=47; TR=0.98
      IF (BR1 .OR. .NOT.BR2) E(I) = D(L)                                     <-- FR=47; TR=0.68
      DO 135 J2D00=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (BR1 .OR. .NOT.BR2) D(J2D00) = Z(L,J2D00)                       <-- FR=1128; TR=0.47
        IF (BR1 .OR. .NOT.BR2) Z(I,J2D00) = 0.0D0                          <-- FR=1128; TR=0.47
        IF (BR1 .OR. .NOT.BR2) Z(J2D00,I) = 0.0D0                          <-- FR=1128; TR=0.47
135   CONTINUE                                                                   <-- FR=1128
      DO 150 K2D00=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) D(K2D00) = D(K2D00) / SCALE                <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) H = H + D(K2D00) * D(K2D00)                <-- FR=1128; TR=0.53
150   CONTINUE                                                                   <-- FR=1128
      IF (.NOT.BR1 .AND. BR2) F = D(L)                                       <-- FR=47; TR=0.32
      IF (.NOT.BR1 .AND. BR2) G = (-DSIGN(DSQRT(H),F))                       <-- FR=47; TR=0.32
      IF (.NOT.BR1 .AND. BR2) E(I) = SCALE * G                               <-- FR=47; TR=0.32
      IF (.NOT.BR1 .AND. BR2) H = H - F * G                                  <-- FR=47; TR=0.32
      IF (.NOT.BR1 .AND. BR2) D(L) = F - G                                   <-- FR=47; TR=0.32
      DO 171 J2D01=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) E(J2D01) = 0.0D0                           <-- FR=1128; TR=0.53
171   CONTINUE                                                                   <-- FR=1128
      DO 240 J2D02=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) F = D(J2D02)                               <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) Z(J2D02,I) = F                             <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) G = E(J2D02) + Z(J2D02,J2D02) * F          <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) JP1 = 1 + J2D02                            <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) BR3 = L .LT. JP1                           <-- FR=1128; TR=0.53
        DO 200 K3D=JP1,L                                          <-- FR=1128; LP=[27.09;31.67]
          IF (.NOT.BR1 .AND. BR2 .AND. .NOT.BR3) G = G + Z(K3D,J2D02) * D(K3D)          <-- FR=11775; TR=1.00
          IF (.NOT.BR1 .AND. BR2 .AND. .NOT.BR3) E(K3D) = E(K3D) + Z(K3D,J2D02) * F     <-- FR=11775; TR=1.00
200     CONTINUE                                                                <-- FR=11775
        IF (.NOT.BR1 .AND. BR2) E(J2D02) = G                               <-- FR=1128; TR=0.53
240   CONTINUE                                                                   <-- FR=1128
      IF (.NOT.BR1 .AND. BR2) F = 0.0D0                                      <-- FR=47; TR=0.32
      DO 245 J2D03=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) E(J2D03) = E(J2D03) / H                    <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) F = F + E(J2D03) * D(J2D03)                <-- FR=1128; TR=0.53
245   CONTINUE                                                                   <-- FR=1128
      IF (.NOT.BR1 .AND. BR2) HH = F / (2 * H)                               <-- FR=47; TR=0.32
      DO 251 J2D04=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) E(J2D04) = E(J2D04) - HH * D(J2D04)        <-- FR=1128; TR=0.53
251   CONTINUE                                                                   <-- FR=1128
      DO 280 J2D05=1,L                                               <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.BR1 .AND. BR2) F = D(J2D05)                               <-- FR=1128; TR=0.53
        IF (.NOT.BR1 .AND. BR2) G = E(J2D05)                               <-- FR=1128; TR=0.53
        DO 261 K3D00=J2D05,L                                      <-- FR=1128; LP=[16.33;31.67]
          IF (.NOT.BR1 .AND. BR2) Z(K3D00,J2D05) = Z(K3D00,J2D05) - F *
     *    E(K3D00) - G * D(K3D00)                                         <-- FR=18424; TR=0.67
261     CONTINUE                                                                <-- FR=18424
        IF (.NOT.BR1 .AND. BR2) D(J2D05) = Z(L,J2D05)                      <-- FR=1128; TR=0.53
```

```
          IF (.NOT.BR1 .AND. BR2) Z(I,J2D05) = 0.0D0                           <-- FR=1128; TR=0.53
 280    CONTINUE                                                                 <-- FR=1128
        D(I) = H                                                                   <-- FR=47
 300  CONTINUE                                                                     <-- FR=47
      DO 500 I1D01=2,48                                                 <-- FR=1; LP=[2.00;48.00]
        L = (-1) + I1D01                                                           <-- FR=47
        Z(48,L) = Z(L,L)                                                           <-- FR=47
        Z(L,L) = 1.0D0                                                             <-- FR=47
        H = D(I1D01)                                                               <-- FR=47
        BR4 = H .EQ. 0.0D0                                                         <-- FR=47
        DO 331 K2D01=1,L                                               <-- FR=47; LP=[1.00;24.00]
          IF (.NOT.BR4) D(K2D01) = Z(K2D01,I1D01) / H                     <-- FR=1128; TR=0.53
 331    CONTINUE                                                                 <-- FR=1128
        DO 361 J2D06=1,L                                               <-- FR=47; LP=[1.00;24.00]
          IF (.NOT.BR4) G = 0.0D0                                         <-- FR=1128; TR=0.53
          DO 341 K3D01=1,L                                           <-- FR=1128; LP=[1.00;31.67]
            IF (.NOT.BR4) G = G + Z(K3D01,I1D01) * Z(K3D01,J2D06)       <-- FR=35720; TR=0.68
 341      CONTINUE                                                              <-- FR=35720
          DO 360 K3D02=1,L                                           <-- FR=1128; LP=[1.00;31.67]
            IF (.NOT.BR4) Z(K3D02,J2D06) = Z(K3D02,J2D06) - G * D(K3D02) <-- FR=35720; TR=0.68
 360      CONTINUE                                                              <-- FR=35720
 361    CONTINUE                                                                 <-- FR=1128
        DO 401 K2D02=1,L                                               <-- FR=47; LP=[1.00;24.00]
          Z(K2D02,I1D01) = 0.0D0                                                 <-- FR=1128
 401    CONTINUE                                                                 <-- FR=1128
 500  CONTINUE                                                                     <-- FR=47
 510  CONTINUE                                                                      <-- FR=1
      DO 520 I1D02=1,48                                                 <-- FR=1; LP=[1.00;48.00]
        D(I1D02) = Z(48,I1D02)                                                     <-- FR=48
        Z(48,I1D02) = 0.0D0                                                        <-- FR=48
 520  CONTINUE                                                                     <-- FR=48
      Z(48,48) = 1.0D0                                                              <-- FR=1
      E(1) = 0.0D0                                                                  <-- FR=1
      END
```

Abb. 4.11 - Tred2

Der aus diesem Programm generierte CAG ist mit seinen Kantenattributen in Abb. 4.12. dargestellt.

Besonders bemerkenswert ist die unterschiedliche Gewichtung der Kanten $e(Z_1,A_1)$ und $e(Z_2,D_1)$, welche beide das selbe Frequencyattribut aufweisen. Die unterschiedliche Gewichtung resultiert aus dem Umstand, daß für die Kante $e(Z_1,A_1)$ nur eine Anweisung verantwortlich ist und überdies die Kommunikation optimiert werden kann. Der Kante $e(Z_2,D_1)$ liegen hingegen drei Anweisungen zugrunde, deren Kommunikation zum Teil auch nicht optimiert werden kann, was zu erhöhtem Kommunikationsaufwand, vor allem durch die Startuptime, führt.



Kantenattribute:

$e(Z_1,A_1)$:

f = 1176; loc = perfect; $\omega = 102{,}45$

$e(Z_1,E_1)$:

f = 18657,99; loc = perfect; $\omega = 118205{,}263$

$e(Z_1,D_1)$:

f = 37119; loc = perfect; $\omega = 14438{,}4$

$e(Z_2,A_2)$:

f = 48; loc = perfect; $\omega = 104{,}8$

$e(Z_2,D_1)$:

f = 1176; loc = perfect; $\omega = 117830{,}4$

$e(A_2,D_1)$:

f = 48; loc = perfect; $\omega = 104{,}8$

$e(D_1,E_1)$:

f = 598; loc = perfect; $\omega = 4812{,}8$

Abb. 4.12 - CAG von Tred2

Der generierte CAG weist zwar konkurrierende Alignmentpräferenzen auf, welche jedoch durch den Heuristischen Algorithmus leicht aufgelöst werden können, da die Gewichtungen starke Präferenzen für die Einhaltung bestimmter Alignmentrelationen zeigen. Die vorgenommene Zuordnung ist in Abb. 4.13 dargestellt.

Abb. 4.13 - Alignmentsets von Tred2

Auch bei diesem Programm kann das Ergebnis durch die Anwendung der *scalar forward substitution* signifikant verbessert werden. So kann eine zusätzliche Alignmentpräferenz zwischen den Knoten Z_2 und E_1 festgestellt werden, was schlußendlich dazu führt, daß E_1 im Gegensatz zur ursprünglichen Lösung dem zweiten Alignmentset zugeordnet wird.

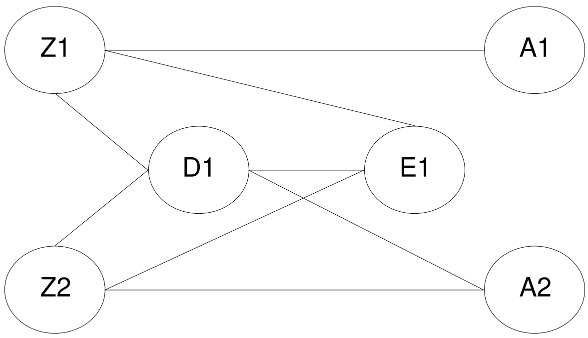
Der modifizierte Programmcode sowie die Resultate der Alignmentanalyse können Abb. 4.14. entnommen werden.

```
      SUBROUTINE TRED2(N,A,D,E,Z)                                  <-- FR=1

      INTEGER I1D03,J2D07,I1D04,K2D03,J2D08,K2D04,J2D09,J2D0A,K3D03,
     *J2D0B,J2D0C,J2D0D,K3D04,I1D05,K2D05,J2D0E,K3D05,K3D06,K2D06,I1D06
      LOGICAL BR1,BR2,BR3,BR4
      INTEGER I1D,J2D,I1D00,K2D,J2D00,K2D00,J2D01,J2D02,K3D,J2D03,J2D04,
     *J2D05,K3D00,I1D01,K2D01,J2D06,K3D01,K3D02,K2D02,I1D02
      INTEGER I,J,K,L,N,II,JP1
      DOUBLE PRECISION A(N,N),D(N),E(N),Z(N,N)
      DOUBLE PRECISION F,G,H,HH,SCALE

      DO 100 I1D03=1,48                               <-- FR=1; LP=[1.00;48.00]
        DO 81 J2D07=I1D03,48                         <-- FR=48; LP=[24.50;48.00]
          Z(J2D07,I1D03) = A(J2D07,I1D03)                         <-- FR=1176
 81     CONTINUE                                                  <-- FR=1176
        D(I1D03) = A(48,I1D03)                                      <-- FR=48
100   CONTINUE                                                      <-- FR=48
      IF (48 .EQ. 1) GOTO 510                               <-- FR=1; TR=0.00
      DO 300 I1D04=2,48                               <-- FR=1; LP=[2.00;48.00]
        H = 0.0D0                                                   <-- FR=47
        SCALE = 0.0D0                                               <-- FR=47
        DO 121 K2D03=1,49 - I1D04                    <-- FR=47; LP=[1.00;24.00]
```

```
        IF (.NOT.49 - I1D04 .LT. 2) SCALE = SCALE + DABS(D(K2D03))          <-- FR=1128; TR=1.00
121   CONTINUE                                                              <-- FR=1128
      IF (.NOT.49 - I1D04 .LT. 2) BR2 = SCALE .NE. 0.0D0                    <-- FR=47; TR=0.98
      IF (49 - I1D04 .LT. 2 .OR. .NOT.BR2) E(50 - I1D04) = D(49 - I1D04)    <-- FR=47; TR=0.68
      DO 135 J2D08=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (49 - I1D04 .LT. 2 .OR. .NOT.BR2) D(J2D08) = Z(49 - I1D04,J2D08) <-- FR=1128; TR=0.47
        IF (49 - I1D04 .LT. 2 .OR. .NOT.BR2) Z(50 - I1D04,J2D08) = 0.0D0    <-- FR=1128; TR=0.47
        IF (49 - I1D04 .LT. 2 .OR. .NOT.BR2) Z(J2D08,50 - I1D04) = 0.0D0    <-- FR=1128; TR=0.47
135   CONTINUE                                                              <-- FR=1128
      DO 150 K2D04=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) D(K2D04) = D(K2D04) / SCALE   <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) H = H + D(K2D04) * D(K2D04)   <-- FR=1128; TR=0.53
150   CONTINUE                                                              <-- FR=1128
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) F = D(49 - I1D04)               <-- FR=47; TR=0.32
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) G = (-DSIGN(DSQRT(H),F))        <-- FR=47; TR=0.32
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) E(50 - I1D04) = SCALE * G       <-- FR=47; TR=0.32
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) H = H - F * G                   <-- FR=47; TR=0.32
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) D(49 - I1D04) = F - G           <-- FR=47; TR=0.32
      DO 171 J2D09=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) E(J2D09) = 0.0D0              <-- FR=1128; TR=0.53
171   CONTINUE                                                              <-- FR=1128
      DO 240 J2D0A=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) F = D(J2D0A)                  <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) Z(J2D0A,50 - I1D04) = D(J2D0A)  <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) G = E(J2D0A) + Z(J2D0A,J2D0A) * D(J2D0A)   <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) JP1 = 1 + J2D0A               <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) BR3 = 49 - I1D04 .LT. JP1     <-- FR=1128; TR=0.53
        DO 200 K3D03=JP1,49 - I1D04                                         <-- FR=1128; LP=[27.09;31.67]
          IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2 .AND. .NOT.BR3) G = G +Z(K3D03,J2D0A) *
     *    D(K3D03)                                                          <-- FR=11775; TR=1.00
          IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2 .AND. .NOT.BR3) E(
     *    K3D03) = E(K3D03) + Z(K3D03,J2D0A) * D(J2D0A)                     <-- FR=11775; TR=1.00
200     CONTINUE                                                            <-- FR=11775
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) E(J2D0A) = G                  <-- FR=1128; TR=0.53
240   CONTINUE                                                              <-- FR=1128
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) F = 0.0D0                       <-- FR=47; TR=0.32
      DO 245 J2D0B=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) E(J2D0B) = E(J2D0B) / H       <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) F = F + E(J2D0B) * D(J2D0B)   <-- FR=1128; TR=0.53
245   CONTINUE                                                              <-- FR=1128
      IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) HH = F / (2 * H)                <-- FR=47; TR=0.32
      DO 251 J2D0C=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) E(J2D0C) = E(J2D0C) - HH * D(J2D0C)   <-- FR=1128; TR=0.53
251   CONTINUE                                                              <-- FR=1128
      DO 280 J2D0D=1,49 - I1D04                                             <-- FR=47; LP=[1.00;24.00]
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) F = D(J2D0D)                  <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) G = E(J2D0D)                  <-- FR=1128; TR=0.53
        DO 261 K3D04=J2D0D,49 - I1D04                                       <-- FR=1128; LP=[16.33;31.67]
          IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) Z(K3D04,J2D0D) = Z(
     *    K3D04,J2D0D) - D(J2D0D) * E(K3D04) - E(J2D0D) * D(K3D04)          <-- FR=18424; TR=0.67
261     CONTINUE                                                            <-- FR=18424
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) D(J2D0D) = Z(49 - I1D04,
     *  J2D0D)                                                              <-- FR=1128; TR=0.53
        IF (.NOT.49 - I1D04 .LT. 2 .AND. BR2) Z(50 - I1D04,J2D0D) =
     *  0.0D0                                                               <-- FR=1128; TR=0.53
```

```
280     CONTINUE                                                                    <-- FR=1128
          D(50 - I1D04) = H                                                         <-- FR=47
300   CONTINUE                                                                      <-- FR=47
      DO 500 I1D05=2,48                                          <-- FR=1; LP=[2.00;48.00]
        Z(48,(-1) + I1D05) = Z((-1) + I1D05,(-1) + I1D05)                           <-- FR=47
        Z((-1) + I1D05,(-1) + I1D05) = 1.0D0                                        <-- FR=47
        H = D(I1D05)                                                                <-- FR=47
        DO 331 K2D05=1,(-1) + I1D05                             <-- FR=47; LP=[1.00;24.00]
          IF (.NOT.H .EQ. 0.0D0) D(K2D05) = Z(K2D05,I1D05) / H       <-- FR=1128; TR=0.53
331     CONTINUE                                                                    <-- FR=1128
        DO 361 J2D0E=1,(-1) + I1D05                             <-- FR=47; LP=[1.00;24.00]
          IF (.NOT.H .EQ. 0.0D0) G = 0.0D0                           <-- FR=1128; TR=0.53
          DO 341 K3D05=1,(-1) + I1D05                         <-- FR=1128; LP=[1.00;31.67]
            IF (.NOT.H .EQ. 0.0D0) G = G + Z(K3D05,I1D05) * Z(K3D05,J2D0E)   <-- FR=35720; TR=0.68
341       CONTINUE                                                                  <-- FR=35720
          DO 360 K3D06=1,(-1) + I1D05                         <-- FR=1128; LP=[1.00;31.67]
            IF (.NOT.H .EQ. 0.0D0) Z(K3D06,J2D0E) = Z(K3D06,J2D0E) - G * D(K3D06)   <-- FR=35720; TR=0.68
360       CONTINUE                                                                  <-- FR=35720
361     CONTINUE                                                                    <-- FR=1128
        DO 401 K2D06=1,(-1) + I1D05                             <-- FR=47; LP=[1.00;24.00]
          Z(K2D06,I1D05) = 0.0D0                                                    <-- FR=1128
401     CONTINUE                                                                    <-- FR=1128
500   CONTINUE                                                                      <-- FR=47
510   CONTINUE                                                                      <-- FR=1
      DO 520 I1D06=1,48                                          <-- FR=1; LP=[1.00;48.00]
        D(I1D06) = Z(48,I1D06)                                                      <-- FR=48
        Z(48,I1D06) = 0.0D0                                                         <-- FR=48
520   CONTINUE                                                                      <-- FR=48
      Z(48,48) = 1.0D0                                                              <-- FR=1
      E(1) = 0.0D0                                                                  <-- FR=1
      END
```

Der aus diesem Programm generierte CAG hat folgende Gestalt und Kantenattribute:



Kantenattribute:

$e(Z_1,A_1)$:

f = 1176; loc = perfect; ω = 102,45

$e(Z_1,E_1)$:

f = 18657,99; loc = perfect; ω = 9502,954

$e(Z_1,D_1)$:

f = 37717; loc = perfect; $\omega = 14540{,}8$

$e(Z_2,A_2)$:

f = 48; loc = perfect; $\omega = 104{,}8$

$e(Z_2,D_1)$:

f = 3061,53; loc = perfect; $\omega = 19356$

$e(Z_2,E_1)$:

f = 757,53; loc = perfect; $\omega = 4812{,}8$

$e(A_2,D_1)$:

f = 48; loc = perfect; $\omega = 104{,}8$

$e(D_1,E_1)$:

f = 630; loc = constant, Localitytupel für D_1: (1,-1/0); $\omega = 9517{,}5$

Der Heuristische Algorithmus liefert folgende Knotenzuordnung:

Abb. 4.14 - Tred2 nach scalar forward substitution

5. Weitere Forschungsansätze

In diesem Kapitel soll ein kurzer Überblick über weitere Forschungsansätze, die sich mit dem Problem des Alignments im speziellen bzw. dem Problem der Datenaufteilung im allgemeinen beschäftigen, gegeben werden. Diesem Forschungsbereich wurde in letzter Zeit immer größere Bedeutung zugemessen, was sich auch in der Fülle diesbezüglicher Arbeiten niederschlägt. Diese komplexe und vielschichtige Thematik kann daher hier nicht umfassend behandelt werden, weshalb nur eine kleine repräsentative Auswahl behandelt werden soll. Bei dieser Auswahl der beschriebenen Arbeiten werden vor allem jene Arbeiten, die Anregungen für das hier entwickelte Modell lieferten oder Methoden präsentieren, die in künftige Weiterentwicklungen dieses Modells einfließen könnten, berücksichtigt.

5.1. Das Crystal-Projekt

Die grundlegenden Arbeiten für den Problemkreis des Alignments, auf welchen auch eine Vielzahl weiterer Ansätze basieren, wurden von Jingke Li und Marina Chen an der Yale-University durchgeführt. In [LiChen89] definieren sie das Alignmentproblem im Rahmen der funktionalen Sprache *Crystal* [CCL89], beweisen die NP-Vollständigkeit dieses Problems und präsentieren einen heuristischen Algorithmus zu dessen Lösung.

Die Arbeit von Li und Chen stellt, neben der nachstehend behandelten Arbeit Guptas, die Basis des in den vorangegangenen Kapiteln entwickelten Ansatzes zur Lösung des Alignmentproblems dar. Sie modellieren Alignmentpräferenzen mit Hilfe des *Component Affinity Graph*, der in seinen Grundzügen dem in Kapitel 3 definierten *Component Alignment Graph* entspricht, wobei in diesem Graph Kanten zwischen zwei Knoten gezogen und damit Alignmentpräferenzen modelliert werden, wenn es im Programm Referenzen auf diese beiden Arraydimensionen gibt, die *affin* sind. Die Affinität zweier Arrayreferenzen wird dahingehend definiert, daß die Indexausdrücke der beiden betrachteten Dimensionen die selbe, nur um einen nicht näher bezeichneten "kleinen" konstanten Faktor verschiedene, kanonische Form aufweisen müssen. Diese Definition der erkannten Alignmentpräferenzen stellt eine Untermenge der durch die hier angewandten Definitionen 3.1 - 3.6 feststellbaren Alignmentpräferenzen dar.

Als Schwachpunkt bei der Modellierung des Component Affinity Graphen erweist sich jedoch die Gewichtung seiner Kanten: Li und Chen definieren zwei oder mehrere

Kanten des Graphen als *competing*, wenn sie vom selben Zugriffsmuster stammen und am selben Knoten auftreten, ansonsten als *non-competing*. Non-competing-Kanten werden mit 1, competing-Kanten mit ε, einem Wert, der "sehr viel kleiner" als 1 ist, gewichtet.

Klarerweise kann mit dieser Gewichtung nur eine sehr grobe Klassifikation und Bewertung von Alignmentpräferenzen erreicht werden, weshalb in der vorliegenden Arbeit, wie auch bereits im Ansatz von Gupta, versucht wurde, tatsächlich mögliche Kommunikationskosten zur Gewichtung der Kanten einzusetzen.

Schlußendlich stellen Li und Chen nach dem Beweis der NP-Vollständigkeit des Alignmentproblems einen heuristischen Algorithmus für dessen Lösung vor, der auch die Ausgangsbasis für den in 3.3.2. präsentierten Algorithmus darstellt, weshalb hier eine kurze Skizzierung des Algorithmus von Li und Chen aus Abb. 5.1. entnommen werden kann.

5.2. Der Constraint-Based Ansatz

Die Arbeit Manish Guptas an der University of Illinois ([Gupta92] und [GuptBan90]) stellt die zweite Basis für den hier verwendeten Ansatz dar. Gupta beschäftigt sich jedoch nicht nur mit dem Alignmentproblem, sondern versucht im Kontext eines parallelisierenden Compilers den gesamten Problemkreis der Datenaufteilung automatisch zu lösen.

Zu diesem Zweck werden vier unterschiedliche Phasen durchlaufen, von welchen jede einen Teilbereich des Datenaufteilungsproblemes zu lösen sucht, wozu in jeder Phase "Einschränkungen" (das sind Anforderungen, die an die letztendlich generierte Datenaufteilungsfunktion gestellt werden) festgestellt und bewertet werden. Daher bezeichnet Gupta seinen Ansatz auch als *constraint-based approach*. Zur Bewertung der Einschränkungen errechnet er *Berechnungskosten* (das sind die Kosten, die zur reinen Berechnung von Daten anfallen) und *Kommunikationskosten* (das sind jene Kosten, die bei allfälliger Kommunikation von Daten anfallen) einer Anweisung, wobei er sich jedoch umfangreicher Heuristiken bedienen muß, da er über keine Informationen über die Ausführungshäufigkeiten, Schleifengrenzen und -iterationen verfügt.

```
Heuristic_Alignment(G)
{
 /* Der Algorithmus benötigt N Schritte, wobei N die Anzahl der
 Spalten im Component Affinity Graph G darstellt. In jedem Schritt
 wird eine beliebige Spalte mit der Zielspalte aligned */

 /* Als Zielspalte C_T wird jene Spalte von G ausgewählt, welche die
 höchste Anzahl an Knoten aufweist*/

 G_1 ← G;    /* Der Graph wird in G_1 kopiert */

 while (G_1 is not empty)
 {
      /* Wähle irgendeine neue Spalte C_x aus G_1 */
      G_x ← Form_Bipartite_Graph(C_T, C_x, G_1);
      M ← Optimal_Alignment(G_x );
      G_1 ← Reduce_Graph(M, C_T, C_x, G_1);
 }
}

procedure Form_Bipartite_Graph(C_T, C_x, G_1)
{
 /* Aus den zwei Spalten C_T und C_x wird ein zweiteiliger Graph
 gebildet, wobei zwei Knoten dieses Graphen mit Kanten verbunden
 werden, wenn es in G_1 einen Weg zwischen diesen gibt. Das Kanten-
 gewicht ist die Summe der Gewichte aller dieser Kanten in G_1. */
}

procedure Optimal_Alignment(G_x)
{
 /* Hier wird das optimale matching des zweiteiligen Graphen G_x
 gefunden, wofür polinomiale Algorithmen existieren */
}

procedure Reduce_Graph(M,C_T, C_x, G_1)
{
 /* Die Spalten C_T und C_x werden gemäß dem in M gefundenen matching
 gemischt. Der Graph wird vereinfacht, indem multiple Kanten sowie
 rekursive Zyklen beseitigt werden. */
}
```

Abb. 5.1 - Heuristischer Algorithmus von [LiChen89]

Die erste Phase ist die **Alignmentphase**, in der sich Gupta an den Grundzügen der Arbeiten von Li und Chen (vgl. 5.1.) orientiert. Eine Alignmentpräferenz zwischen zwei Arraydimensionen stellt in seinem Ansatz eine Einschränkung dar, der ein entsprechendes Gewicht beigemessen wird, wobei, da sich das Alignmentproblem ja mit der Vermeidung von Kommunikation beschäftigt, nur die Kommunikationskosten berücksichtigt werden.

Diese Berücksichtigung der Kommunikationskosten stellt die große Neuerung des Ansatzes Guptas im Gegensatz zu jenem von Li und Chen dar. Gupta versucht, die Kanten des CAG mit jenen zusätzlichen Kommunikationskosten zu gewichten, welche anfallen, wenn diese Alignmenteinschränkung nicht eingehalten werden sollte. Da Gupta über keine Profil-Daten des untersuchten Programmes verfügt, bedient er sich umfangreicher Heuristiken und einer angenommenen Basisaufteilung, um die zusätzlichen Kommunikationskosten abschätzen zu können. Er verwendet jedoch auch ein Compilermodell, welches Kommunikationsoptimierungen vornimmt, indem er versucht, die einzelnen Zugriffsmuster einer Anweisung vordefinierten Kommunikationsroutinen (im Speziellen *Transfer, OneToManyMulticast, ManyToManyMulticast, Scatter, Gather, Shift* und *Reduction*) zuzuordnen und deren Zeitkomplexität in die Berechnung der Kommunikationskosten einfließen läßt.
Die schlußendliche Lösung des Alignmentproblems erfolgt mit dem von Li und Chen vorgeschlagenen heuristischen Algorithmus.

In der zweiten Phase wird die **Art der Aufteilung (block oder zyklisch)** entschieden. Ziel dieser Phase ist die Optimierung der Arbeitsaufteilung (unter Annahme des owner-computes Paradigmas), weshalb als Gewichtungsfaktor für die Einhaltung einer block (zyklischen) Einschränkung die Differenz der jeweiligen Berechnungskosten angesetzt wird.

Da in der zweiten Phase nur die Art der Aufteilung festgelegt wird, ist in der dritten Phase die Spezifizierung der **Blockgröße** notwendig. Gupta modelliert dieses Problem mit einem *Block-size Constraint Graph*, der ähnlich dem CAG aufgebaut ist und als Kantengewichtung die zusätzlichen Kommunikationskosten, die bei Nichteinhaltung der präferierten Blockgröße entstehen, verwendet. Die Lösung dieses Problems ist letztlich ein maximal spannender Baum im Block-size Constraint Graphen.

In der letzten Phase wird entschieden, welche **Anzahl an Prozessoren** jeder Dimension des virtuellen Prozessorenarrays zugeordnet wird. Da zu diesem Zeitpunkt bereits die Aufteilungsfunktionen feststehen, kann man eine heuristische Abschätzung der Ausführungszeit als Funktion der Prozessorenanzahl vornehmen und diese Funktion minimieren.

Der Vorteil von Guptas Ansatz besteht darin, daß er das Problem der Datenaufteilung als Ganzes behandelt und alle notwendigen Faktoren (Alignment, Prozessorenanzahl, Blockgrößen) miteinbezieht. Zur Lösung der einzelnen Teilprobleme wendet er jedoch komplexe Heuristiken an, die, ohne Gewähr auf eine optimale Lösung, einen großen Rechenaufwand verursachen.

5.3. Das Modell des Communication und Access Graph

Im Gegensatz zu den beiden soeben vorgestellten Arbeiten wird in den Arbeiten des *Laboratoire de l'Informatique du Parallélisme (LIP)* der *Ecole Normale Supérieure de Lyon* das Alignmentproblem nicht mit Hilfe eines CAG, sondern mit anderen graphentheoretischen Ansätzen, dem *communication graph* [DaRo93] bzw. dem *access graph* [DiRo94] modelliert. Ein großer Unterschied dieser beiden Arbeiten zu den vorherigen besteht auch darin, daß sie das owner-computes Paradigma brechen und ein getrenntes Mapping der Schleifeniterationen und der Arrayelemente annehmen.

In [DaRo93] wird der *communication graph* eingeführt, welcher ein bipartiter Graph ist, dessen Knoten der einen Spalte die Statements des Programms darstellen, die Knoten der zweiten Spalte die darin verwendeten Arrays. Zwischen beiden Spalten werden gerichtete Schreib- bzw. Lesezugriffskanten gezogen, welche mit der Differenz der in Matrizenform dargestellten Iterationsvektoren- und Zugriffsmuster-Mappingfunktionen gewichtet werden. Die Lösung des Alignmentproblems erfolgt sodann durch den Versuch, die Gewichte möglichst vieler dieser Kanten auf 0 zu bringen (das heißt, die Anweisung wird am selben Prozessor ausgeführt, wo auch die dazu notwendigen Lese- bzw. Schreibzugriffe stattfinden müssen), was mit Hilfe von Matrizengleichungen durchgeführt wird. Jedoch erweist sich auch dieser Ansatz bereits unter den einfachsten Annahmen als NP-vollständig, weshalb ein heuristischer Lösungsansatz vorgestellt wird.

[DiRo94] verfeinern das Konzept des communication graph, indem sie die Dimensionalität des Prozessorenarrays nicht wie im obgenannten Ansatz als variabel anpaßbar, sondern als Inputparameter betrachten, nach dem sich die weiteren Alignmentschritte zu richten haben. Hiezu definieren sie den *access graph*, dessen Knoten jenen des communication graph entsprechen, seine Kanten jedoch nur eine Untermenge der Kanten des communication graph darstellen. Im access graph werden nur Kanten gezogen, wenn sowohl die Nestingtiefe der Anweisung als auch die Dimensionalität der Arrayreferenz größer als die Dimensionalität des Prozessorenarrays sind. Je nach dem Größenverhältnis der Nestingtiefe der Anweisung und der Dimensionalität der Arrayreferenz ziehen sie sodann einseitig oder zweiseitig gerichtete Kanten, welche wie im communication graph mit der Matrizendarstellung der Mappingfunktionen gewichtet werden.

Die Lösung des Alignmentsproblems erfolgt wiederum durch den Versuch, möglichst viele Kantengewichte auf 0 zu bringen, was ebenfalls als NP-vollständiges Problem bewiesen wird. Heuristische Vorgangsweisen zur Lösung des Problems werden angeführt.

5.4. Das Modell des Communication-Parallelism Graph

Ein von den bisherigen Arbeiten abweichender Ansatz wird in [GAL95] vorgestellt, indem versucht wird, sowohl das Problem des Alignments als auch das Problem der optimalen Ausschöpfung des möglichen Parallelismus über die Arbeitsaufteilung (unter Anwendung des owner-computes Paradigmas) in einem Schritt zu lösen. Die bisherigen Modelle praktizierten dies in zwei getrennten Schritten (vgl. insbesondere 5.2.). Garcia et. al. definieren für ihre Zwecke den *Communication-Parallelism Graph (CPG)*, welcher aus *Datenübertragungskanten* zur Modellierung des Alignmentproblems und *Parallelismus-Hyperkanten* zur Modellierung des Arbeitsaufteilungsproblems besteht.

Die Knoten des CPG entsprechen jenen eines CAG, die Datenübertragungskanten drükken jedoch im Gegensatz zum CAG keine Präferenz dafür aus, die beiden verbundenen Knoten auch tatsächlich zu alignen, sondern stellen vielmehr eine prinzipielle Alignmentmöglichkeit dar, welche durch Analysen der Zugriffsmuster von Zuweisungen festgestellt werden kann. Das heißt, alle Knoten einer Spalte des CPG werden mit allen Knoten des zweiten Arrays im entsprechenden Zugriffsmuster durch Datenübertragungskanten verbunden, womit alle theoretisch möglichen Alignmentvarianten zwischen zwei Arrays dargestellt werden. Die Gewichtung dieser Kanten erfolgt nun nicht wie bei den vorhergegangenen Arbeiten durch eine Maßzahl, die Kommunikationsersparnis bei Einhaltung einer Alignmentpräferenz ausdrückt, sondern durch eine Maßzahl, welche die Kosten einer Alignmentpräferenz durch das Zuordnen der Zugriffsmuster zu vordefinierten Kommunikationsroutinen (vergleichbar mit jenen in der Arbeit Guptas, Abschnitt 5.2.) abschätzt.

Die Parallelismus-Hyperkanten stellen Präferenzen dar, welche Arraydimensionen verteilt werden sollen, um einen möglichst hohen Grad an Parallelismus zu erlangen. Um diese Kanten generieren zu können, werden unter Zuhilfenahme von Abhängigkeitsanalysen alle möglichen parallelen Schleifen markiert. Für jede parallele Schleife wird sodann eine Hyperkante gezogen, die alle Arraydimensionen verbindet, auf welche ein Schreibzugriff stattfindet und in deren Indexausdruck der Schleifenparameter vorkommt. Diese Hyperkanten werden mit dem durch die Verteilung der jeweiligen Arraydimensionen erzielbaren Zeitgewinn gewichtet.

Das Lösungsproblem für den CPG ist in der Minimierung der Ausführungszeit gegeben, welche sich aus dem Kommunikationsaufwand (Datenübertragungskanten) abzüglich des Zeitgewinns durch Ausnützung des Parallelismus (Parallelismus-Hyperkanten) ergibt. Garcia et. al. beschränken ihren Lösungsweg auf das Auffinden einer einzigen aufzuteilenden Dimension, indem sie das Problem auf das des kürzesten Pfades zurückführen und als 0-1 integer programming problem formulieren. Lösungsansätze unter Berücksichtigung von mehrdimensionalen Aufteilungen werden ebenfalls kurz angerissen.

5.5. Das Fortran-D Projekt

Im Rahmen des Fortran-D Projektes [HKet.al.91] und des D-Editors [HKTW94] an der Rice University wurde von [KenKrem94] ein Prototyp eines Datenaufteilungstools vorgestellt, welcher versucht, die Datenaufteilung nicht notwendigerweise für ganze Programme bzw. Prozeduren zu spezifizieren, sondern auch die dynamische Umverteilung in Betracht zieht. Zu diesem Zweck werden *Phasen* im Programm erkannt, welche im Prinzip vollständige Schleifennester sind, deren äußerste Schleife Indexausdrücke von Arrayreferenzen verändert. Dadurch ist gewährleistet, daß sich Phasen nicht überlappen und auch nicht geschachtelt sein können. Die einzelnen Phasen eines Programms werden in einem *phase control flow graph (PCFG)* dargestellt, dessen Knoten die Phasen darstellen und dessen Kanten mit Informationen über Schleifengrenzen und Beschreitungswahrscheinlichkeiten (beide Werte werden heuristisch erlangt!) versehen werden.

Für jede Phase wird sodann die Alignmentanalyse unter Verwendung des Konzeptes des CAG von Li et. al. (vgl. 5.1.) durchgeführt, die Lösung des Alignmentproblems jedoch als 0-1 integer programming formuliert. Die möglichen Partitionierungen des CAG werden als Halbverbund dargestellt. Mit Hilfe von Heuristiken werden sodann mögliche Aufteilungsfunktionen festgestellt und die Ergebnisse der Alignment- und Aufteilungsanalysen als Lösungsraum für jede Phase gespeichert.
Das Problem, die Phasen wiederum zusammenzufügen, wird als Graphenproblem mit Hilfe des *data layout graph* modelliert, dessen Kanten Umverteilungen und dessen Knoten Phasen darstellen. Zur Lösung des Aufteilungsproblems muß versucht werden, Kno-

ten dieses Graphen zusammenzufügen (und dadurch mit derselben Aufteilungsfunktion zu versehen), sodaß schlußendlich die gesamten Ausführungskosten aus den Kommunikationskosten innerhalb von Phasen und den Umverteilungskosten entlang der Kanten des data layout graph minimiert werden. Auch dieses Problem wird als 0-1 integer programming problem definiert.

Mit ihren Verfahren können Kennedy et. al. zwar zu optimalen Lösungen des Datenaufteilungsproblems gelangen, die Zeitkomplexität der angewandten Lösungsverfahren beschränkt deren Anwendung jedoch auf kleinere Problemfälle. Die Behandlung realer Programme wird mit diesen Lösungsverfahren meist einen nicht mehr vertretbaren Zeitaufwand beanspruchen.

5.6. Das HPC Projekt

Eine wichtige Arbeit speziell am Gebiet der Programmdekomposition und der Erkennung von Phasen, zwischen denen umverteilt werden sollte, wurde von [NVDG94] vorgestellt. Ning et. al. entwickeln darin ein automatisches Daten- und Arbeitsaufteilungstool als Teil des EPPP-Projekts [HVDG94], welches versucht, einen portierbaren parallelisierenden Compiler für die Programmiersprache HPC [VDBF94] zu realisieren. Der Ansatz von Ning et. al. beschäftigt sich hauptsächlich mit der Optimierung von Schleifen und Arrayzugriffen innerhalb von Schleifen, weshalb diese Arbeiten auch für Fortran-ähnliche Sprachen umgesetzt werden können.
Ein Vorteil dieser Arbeit im Gegensatz zur zuvor behandelten von Kennedy et. al. besteht darin, daß versucht wird, das Daten- und Arbeitsaufteilungsproblem nicht mit zeitaufwendigen Optimierungen zu lösen, sondern dies mit Hilfe effizienter Heuristiken zu erreichen, wodurch der Ansatz auch für reale Programme anwendbar ist.

Im Zentrum ihrer Arbeit stehen zwei heuristische Algorithmen, der *Locality Algorithm (LA)*, welcher das Daten- und Arbeitsaufteilungsproblem zu lösen sucht, und der *Clustering Algorithm (CA)*, welcher eine optimale Programmdekomposition in Clusters (Phasen), zwischen denen umverteilt wird, zu finden sucht. Beide Algorithmen werden jeweils auf einzelne Schleifennester angewandt, die der Einschränkung unterliegen, daß sowohl die Schleifengrenzen als auch die Indizes von Arrayreferenzen affine Funktionen von Schleifenindizes und Schleifeninvarianten sein müssen. Ning et. al. Klassifizieren für ihre Zwecke die Schleifen eines Programms gemäß deren Abhängigkeitsinformation nach dem der Schleife inhärenten Parallelismus. Sie unterscheiden zwischen *Äußersten Parallelen Schleifen*, welche vollkommen parallelisiert werden können, *Inneren Parallelen Schleifen*, welche zwar parallelisiert werden können, aber ihrerseits von nicht parallelisierbaren Schleifen umgeben werden, *Pipelining Schleifen*, die eine Abarbeitung ihrer Iterationen durch Pipelining ermöglichen, und schließlich *Pseudo Sequentiellen Schleifen*, welche nicht parallelisiert werden können.

Der Locality Algorithm versucht für jedes Schleifennest die optimale Daten- und Arbeitsaufteilung zu finden, indem abweichend vom owner-computes Paradigma die Iterationsvektoren und Indizes von Arrayreferenzen, auf denen ein Schreibzugriff erfolgt, als homogene lineare Gleichung angesetzt werden und dadurch alle Zuweisungen einer Schleife zu einem linearen Gleichungssystem zusammengestellt werden, dessen Lösung mit Hilfe des Gauß'schen Eliminationsverfahren die Daten- und Arbeitsaufteilung ergibt. Ausgehend von dieser Basislösung wird sukzessive versucht, den höchstmöglichen Grad an Parallelität (entsprechend den oben angeführten Schleifenklassifikationen) zu erreichen, was schlußendlich zu einer Daten- und Arbeitsaufteilungsfunktion für jedes Schleifennest führt.

Sodann versucht der Clustering Algorithm das Programm in optimale Cluster aufzuteilen, sodaß die Umverteilungskosten minimiert werden. Dazu werden die Schleifennester des Programms in einem *Loop Program Dependence Graph (LPDG)* dargestellt, dessen Knoten die Schleifennester darstellen. Zwischen zwei Knoten wird eine gerichtete Kante gezogen, wenn in einem Schleifennest ein Array definiert wird, welches im zweiten konsumiert wird. Diese Kanten werden mit der abgeschätzten Anzahl der Beschreitungen im Zuge einer Programmexekution und den Umverteilungskosten zwischen beiden Schleifennestern, welche ja jeweils durch den LA mit einer Aufteilungsfunktion versehen wurden, gewichtet. In einem Greedyverfahren ohne Backtracking wird nun versucht, ausgehend von der am höchsten gewichteten Kante im LPDG, jeweils zwei Knoten zu einem Cluster zusammenzufügen. Für dieses neue Cluster wird wiederum der LA angewandt und dessen Ergebnis mit dem vorherigen verglichen. Wird festgestellt, daß das neue Cluster ein schlechteres Ergebnis liefert als die beiden ursprünglichen inklusive der Umverteilungskosten, so bleiben beide Knoten getrennt und zwischen ihnen wird entsprechend ihrer Aufteilungsfunktionen umverteilt; andernfalls bilden beide Knoten einen neuen Cluster. Der CA versucht also, immer mehr Schleifennester zu einem Cluster zusammenzufügen, um so die Umverteilungskosten unter Beibehaltung von möglichst optimalen Daten- und Arbeitsaufteilungsfunktionen zu vermindern.

5.7. Die Arbeiten von Chatterjee et. al.

Chatterjee et. al. beschreiben in [CGST93] Methoden zur Lösung des Alignmentproblems auf Basis von heuristischen Algorithmen der Dynamischen Programmierung. Sie wenden diese Algorithmen auf Basisblöcke eines Programmes an, welche in Form eines gerichteten azyklischen Graphen dargestellt werden. Die Kanten dieses Graphen repräsentieren einen Datenfluß zwischen Programmobjekten und werden durch eine Abschätzung der Kommunikationskosten, welche auf den bekannten Arraygrößen und Abschätzungen der Häufigkeiten beruht, gewichtet. Zur Lösung des Alignmentproblems werden nur Arrayreferenzen herangezogen, deren Indizes affine Funktionen eines einzigen Schleifenindexes sind, und das owner-computes Paradigma wird nicht eingehalten.

Auf Basis dieser Graphendarstellung eines Basisblocks wird für jeden Basisblock das Alignmentproblem mit Hilfe zweier heuristischer Algorithmen der Dynamischen Programmierung gelöst: Zuerst werden jene Arrays bestimmt, welche aligned werden sollen, sodann wird für diese Arrays das inter- und intradimensionale Alignment berechnet; zweiteres bestehend aus dem Offset und der Blockgröße.
Nach Ausführung dieser Algorithmen wird ein Basisblock mit seinen errechneten Alignmentfunktionen versehen; auf ein ganzes Programm umgelegt bedeutet dies, daß jeder Basisblock mit einer eigenen Alignmentfunktion versehen ist, welche die unterschiedlichsten Ausformungen haben kann.

Um auf dieser Basis effiziente Datenaufteilungen zu erhalten, wird in [CGSS94] der *Alignment-Distribution Graph (ADG)* eingeführt, dessen Kanten Definitionen und Konsumationen von Arrays verbinden, welche mit ihren Alignmentfunktionen versehen sind. Mit Hilfe eines Divide-and-Conquer Algorithmus werden im Graphen Mengen von Knoten festgestellt, für die mehrere auf Basis der Alignmentfunktionen mögliche Aufteilungsfunktionen errechnet und evaluiert werden. Bei der Evaluierung der Aufteilungsfunktionen werden, wiederum ausgehend von den Arraygrößen und den geschätzten Ausführungshäufigkeiten, die Kosten für Kommunikation und Redistributierung berücksichtigt.
In [SSPGC95] werden schließlich Methoden zur Vereinfachung eines ADG vorgestellt, indem mehrere Knoten, die eine gleiche Aufteilungsfunktion erhalten dürften, zu einem einzigen "Superknoten" zusammengefaßt werden. Dadurch ist eine signifikante Verkleinerung des Problemraumes möglich. Neben der Präsentation mehrerer Testverfahren für die Gültigkeit der Knotenzusammenfassung wird in dieser Arbeit auch die Effizienz der vorgestellten Methoden anhand einiger Beispiele gezeigt.

5.8. Das Fort Translation System (FTS)

Philip Crooks stellt in seiner Dissertation [Croo94] ein System vor, welches zu einem in Fort (einer speziell für diese Zwecke definierte Untermenge von Fortran 90) verfaßten Programm automatisch entsprechende Aufteilungs- und dynamische Umverteilungsfunktionen findet und diese als Anfügungen an den ursprünglichen Code (vergleichbar mit Vienna Fortran oder HPF) realisiert. Er beschränkt seine Codeanalyse auf Array-Anweisungen (entsprechend der Syntax von Fortran 90) und versucht mittels pattern-matching Zugriffsmuster zu erkennen, für welche in einer Datenbank mögliche Aufteilungsfunktionen vordefiniert sind. Konflikte zwischen möglichen Aufteilungsfunktionen werden mittels einer einfachen Performanceabschätzung aufgelöst, sowie auch die Möglichkeit der dynamischen Umverteilung zwischen Programmphasen in Betracht gezogen wird. Als Phasen definiert Crooks innerste Schleifen und stellt die Bedinung, daß Umverteilungen nur zwischen diesen möglich sind.

Das FTS ist modular aufgebaut und besteht aus 6 Einheiten:
Der *Code Handler* überträgt das Quellenprogramm in eine interne Darstellung für die weitere Bearbeitung. Das *Preference Determination Module* stellt für jeden Datenzugriff innerhalb einer Phase die optimale Aufteilung fest. Das *Preference Combination Module* kombiniert diese, möglicherweise konkurrierenden Aufteilungsfunktionen mit Hilfe eines einfachen *Performance Estimator*, sodaß jedes Array einer Phase mit einer für diese Phase (quasi)optimalen Aufteilungsfunktion versehen wird.
Der *Distribution Scheme Generator* versucht nun, die initialen Aufteilungsfunktionen sowie die notwendigen Umverteilungsfunktionen aus den Informationen der einzelnen Phasen zu bestimmen. Da dies ein NP-vollständiges Problem ist, modelliert Crooks die möglichen Umverteilungen mit Hilfe eines *distribution tree* und löst das Problem mit Hilfe des Performance Estimator auf heuristische Weise.
Schließlich erzeugt der *Code Generator* die Distributions- und Umverteilungsanweisungen im Quellencode.

Seinen Analysen legt Crooks ein Compilermodell zugrunde, welches nach dem SPMD- und owner-computes-Paradigma vorgeht, jedoch keinerlei Optimierungen, seien dies Codeoptimierungen durch Transformationen oder Kommunikationsoptimeriungen, vornimmt. Weiters basieren seine Analysen und Performanceabschätzungen auf einer vordefinierten Zielmaschine mit 16, als 4x4-Gitter organisierten Prozessoren.
In einigen Anwendungsbeispielen werden schließlich die Arbeitsweise und Resultate des FTS demonstriert.

6. ZUSAMMENFASSUNG UND AUSBLICK

6.1. Zusammenfassung

Im Rahmen der Programmierung von Parallelrechnern stellt die geeignete Wahl der Arbeits- und Datenaufteilung eines der größten ungelösten Probleme dar, da diese Wahl einerseits entscheidenden Einfluß auf die erzielbare Performance des parallelen Programmes nimmt, andererseits jedoch in den meisten heutigen Systemen vom Programmierer getroffen werden muß, was eine schwierige und fehleranfällige Aufgabe ist. Es wurden und werden daher große Forschungsanstrengungen unternommen, die Arbeits- und Datenaufteilung im Rahmen von automatischen Parallelisierern zu automatisieren, bzw. dem Benutzer hiezu größtmögliche Unterstützung zu bieten. Die Hauptrichtung liegt dabei sicherlich auf einer möglichen automatischen Lösung des Problems, welche jedoch nur bedingt möglich sein wird, da das Problem der Klasse der NP-vollständigen Probleme angehört.

In dieser Arbeit wurde ein Analysesystem vorgestellt, welches dem Benutzer Informationen über Datenzugriffe im Programmverlauf bietet und darauf basierend versucht, Hinweise über die optimale Ausrichtung der Daten im Speicher zu geben, um den aus nichtlokalen Datenzugriffen resultierenden Kommunikationsaufwand zu minimieren. Das Problem, solche Datenzugriffe festzustellen, wird im allgemeinen als das *Alignmentproblem* bezeichnet, welches in zwei Ausformungen unterteilt werden kann: das *interdimensionale Alignment* und das *intradimensionale Alignment*. Die Lösung des interdimensionalen Alignments versucht gemeinsam genutzte Arraydimensionen festzustellen, welche auf die selben Prozessoren aufgeteilt werden sollten, um teure Kommunikation zu vermeiden, während die Lösung des intradimensionalen Alignments die relative Lage der gemeinsam genutzten Arrayelemente solcher Arraydimensionen festzustellen sucht.

Das Alignmentproblem wurde hier mit Hilfe eines *Component Alignment Graph (CAG)* modelliert, der sowohl Konflikte im inter- als auch im intradimensionalen Alignment darstellen kann. Besonderes Augenmerk wurde auf die Klasse der erkennbaren Alignmentrelationen gelegt, welche im Gegensatz zu vergleichbaren Arbeiten signifikant erweitert werden konnte, indem einerseits die Definition von feststellbaren Alignmentrelationen um wichtige Fälle ergänzt wurde und andererseits Programmtransformationen ausgeführt werden, die ein vermehrtes Feststellen von Alignmentrelationen ermöglichen.

Mit Hilfe dieses CAG ist es möglich, sowohl das interdimensionale als auch das intradimensionale Alignment zu modellieren. Die Modellierung des interdimensionalen Alignments erfolgt wie in der Literatur üblich durch Alignmentkanten zwischen zwei Arraydimensionen, welche eine Alignmentpräferenz ausdrücken. Zur Modellierung des intradimensionalen Alignments wurde eine *Localityrelation* definiert, welche aus einem Multiplikator zur Abbildung der Größenverhältnisse und einem Constant Offset zur Abbildung von Verschiebungen besteht. Mit Hilfe dieser Localityrelation ist es auch möglich, konkurrierende Anforderungen an das intradimensionale Alignment zu modellieren, was ebenfalls gezeigt wurde.

Ein weiterer zentraler Punkt dieser Arbeit ist die angewandte Gewichtung der Alignmentkanten im CAG, welche die Bedeutung dieser Alignmentrelation für die Programmperformance modelliert. Da das Ziel des Alignments ja die Vermeidung von Kommunikation ist, wurde ein einfaches Kommunikationskostenmodell entwickelt, welches worst-case Kommunikationskosten bei Nichteinhaltung der Alignmentpräferenz heuristisch abschätzt. Das Kommunikationskostenmodell wurde bewußt sehr einfach gehalten, da der gesamten Alignmentanalyse ein heuristischer Charakter zukommt und daher versucht wurde, die Berechnungen möglichst zeitschonend durchzuführen, jedoch unter Beibehaltung eines möglichst großen Aussagewertes der Ergebnisse. Eine der wichtigsten Neuerungen bei der Kommunikationskostenabschätzung stellt die Verwendung eines Profilers für sequentielle Programme dar, durch welchen Informationen über Ausführungshäufigkeiten, Schleifengrenzen und True-Anteile von If-Anweisungen erhalten werden können, welche direkt in die Berechnungen einfließen. Weiters wurden Abhängigkeitsinformationen in das Berechnungsmodell aufgenommen, wodurch die vom zugrundeliegenden Compilermodell durchgeführten Kommunikationsoptimierungen modelliert werden konnten. Schließlich wurden in das Kostenmodell noch maschinenabhängige Faktoren aufgenommen, um einen möglichst großen Realitätsbezug bewahren, wie auch die Auswirkungen verschiedener Plattformen untersuchen zu können.

Das Alignmentproblem wurde somit mittels eines CAG modelliert, dessen große Neuerungen vor allem in der Behandlung des intradimensionalen Alignments, sowie in der Gewichtung seiner Kanten liegen.

Zur Lösung des Alignmentproblems wurde ein heuristischer Algorithmus angewandt, welcher sich in seinen Grundzügen an den in der Literatur bekannten Verfahren orientiert. Zwar gibt es in der Literatur auch Ansätze, das Problem mit Hilfe nicht-heuristischer Verfahren zu lösen, doch wird diese Exaktheit der Lösung mit einem immensen

Zeitaufwand erkauft, welcher die Anwendbarkeit dieser Verfahren für reale Programme sehr beschränkt. Im vorgestellten Ansatz wurde jedoch immer auf die effiziente Anwendbarkeit der vorgestellten Methoden bedacht genommen, wodurch die einfachen Verfahren bei der Kantengewichtung, der Modellierung des intradimensionalen Alignments[7] sowie der heuristische Lösungsweg begründet sind.

Diese praktische Anwendbarkeit der vorgestellten Analyseverfahren wurde auch durch die Realisierung eines prototypischen Alignmentanalysetools untermauert. Dieser Prototyp wurde im Rahmen des *Vienna Fortran Compilation System (VFCS)* entwickelt, welches ein semiautomatischer Compiler für in Fortran, bzw. Abarten von Fortran verfaßte Programme ist. Im Mittelpunkt dieses Compilers stehen die Optimierung des Parallelisierungsvorganges durch Programmtransformationen und hochentwickelte Kommunikationsabwicklungen, wie auch die Analyse und Evaluierung der sequentiellen und parallelen Programmversionen.
Anhand der Arbeitsweise dieses Prototypen wurde die Anwendung der Analysemethoden vorgestellt, sowie die Resultate der Analysierung zweier Benchmarkprogramme präsentiert.

Das vorgestellte Alignmentanalysetool stellt somit dem Benutzer Informationen über das Zusammenspiel verschiedener Arrays im Programmverlauf zur Verfügung, wodurch er Hilfestellungen für die Wahl der Aufteilungsfunktionen in Hinblick auf die Vermeidung von Kommunikation erhält. Das Tool ist jedoch so konzipiert, daß Erweiterungen und Einbindungen in andere Systeme leicht möglich sind, weshalb im nachfolgenden Abschnitt noch einige abschließende Bemerkungen über mögliche Weiterentwicklungen dieser Arbeit folgen sollen.

6.2. Ausblick

Das in dieser Arbeit vorgestellte Instrumentarium stellt ein effektives Hilfsmittel zur Programmierung von Multiprozessoren dar und ist auch darauf ausgelegt, Basisinformationen für weiterführende Verfahren zu bieten. Einige Beispiele für solche möglichen Weiterführungen sollen in diesem Abschnitt angerissen werden.

[7] In einigen der vorgestellten Arbeiten wird sowohl das inter- als auch das intradimensionale Alignment mit Hilfe von Funktionen der Indexausdrücke modelliert und ein Ergebnis durch Lösen dieser Gleichungssysteme erlangt. Diese Verfahren schränken jedoch die Anzahl der erkennbaren Alignmentrelationen stark ein und verursachen ebenso einen sehr großen Zeitaufwand.

6.2.1. Automatisches Alignment

Eine Erweiterungsmöglichkeit wäre die automatische Generierung von Alignmentanweisungen, wie sie unter anderem von HPF [HPF94] oder Vienna Fortran 90 [Benk94] zur Verfügung gestellt werden. Die Informationen der Alignmentmengen, welche durch den Heuristischen Algorithmus erlangt werden, können direkt in Alignmentanweisungen umgewandelt werden, wobei auch die Localityrelation ihren entsprechenden Niederschlag findet. Da jedoch die errechneten Alignmentrelationen nicht notwendigerweise ein optimales Ergebnis liefern müssen und für bestimmte Problemkreise (wie z.B. bei ausschließlich eindimensionalen Arrayzugriffen) nur wenig aussagekräftige Informationen bieten, sollte dem Benutzer die Möglichkeit gegeben werde, aus den vorgeschlagenen Alignmentanweisungen die für seine Zwecke als richtig erachteten auswählen zu können.

Die Umsetzung der Informationen der Alignmentsets in Alignmentanweisungen ist nachfolgend in Beispiel 6.1 veranschaulicht, für welches der Code von Beispiel 4.1. und die Alignmentsyntax von Vienna Fortran 90 als Basis herangezogen werden.

Beispiel 6.1

Die Alignmentanalyse für den Code von Bsp. 4.1 lieferte folgende Alignmentsets:
(vgl. Abb. 4.6)

Set 1: A1-Perfect Locality; C2-Perfect Locality; B2-Perfect Locality
Set 2: A2-Perfect Locality; C1-Perfect Locality; B1-Perfect Locality

Die mit den Alignmentanweisungen versehenen Datendeklarationen in Vienna Fortran 90 können aus den obigen Mengen direkt abgeleitet werden (es wird der Basisdatentyp REAL angenommen):

```
REAL, DISTRIBUTED(?,?)[8]            :: A(N,N)
REAL, ALIGNED(C(I,J) WITH A(J,I))  :: C(N,N)
REAL, ALIGNED(B(I,J) WITH A(J,I))  :: B(N,N)
```

[8] Die DISTRIBUTED-Anweisung wird nicht automatisch generiert und ist nur zur Veranschaulichung des Zusammenhanges zwischen den ALIGNED-Anweisungen und der tatsächlichen Aufteilung angegeben. Die "?" symbolisieren Platzhalter für mögliche Aufteilungsfunktionen

6.2.2. Interprozedurales Alignment

Bei den bisherigen Arbeiten am Gebiet der Alignmentanalyse wurde stets davon ausgegangen, daß die Analyse ausschließlich intraprozedural stattfindet - zur Analyse kompletter Programme mußten die Prozeduren mittels inlining aufgelöst werden. Ziel des interprozeduralen Alignments wäre die Durchführung einer Alignmentanalyse für komplette Programme, ohne deren Prozedurenstruktur ändern zu müssen.

Im vorhergegangenen Kapitel wurden bereits zwei Arbeiten vorgestellt, die sich, zwar auf intraprozeduraler Basis, mit der Spezifizierung von Programmphasen, zwischen denen das Alignment geändert, also die Daten physisch umverteilt werden sollen, beschäftigen. Diese Ansätze können auch für das interprozedurale Alignment genutzt werden, indem in einem ersten Schritt die Prozeduren eines Programmes als *Phasen* betrachtet werden.
In diesem Zusammenhang stellt sich auch der Problem der möglichst effizienten Gestaltung der durch das Realignment ausgelösten Redistribution der Arrayelemente, wozu unter anderem in [KHRS94] effiziente Methoden vorgestellt wurden.

Eine Möglichkeit, das interprozedurale Alignment zu realisieren, wäre durch die Verwendung des von [NVDG94] vorgestellten *Clustering Algorithm (CA)* (vgl. 5.6.) gegeben, worauf etwas näher eingegangen werden soll:

Der CA ist ein Greedyalgorithmus ohne Backtracking, der versucht, einzelne Programmphasen zu größeren Phasen zusammenzufassen, sodaß schließlich jede Phase mit entsprechenden Aufteilungsfunktionen versehen ist und beim Wechseln von einer Phase zur nächsten umverteilt werden muß.
Auf die interprozedurale Alignmentanalyse umgelegt bedeutet dies, daß zunächst für jede Prozedur mit Hilfe der bekannten Verfahren ein CAG generiert wird, welcher dem Callgraph des Programmes zugefügt wird. Sodann werden die Kanten des Callgraphen mit der Beschreitungshäufigkeit gewichtet und in der Reihenfolge des Gewichtes der Callgraphkanten jeweils zwei Prozeduren in Hinblick darauf untersucht, ob für sie eine gemeinsame Alignmentinformation generiert werden kann oder ob bei Beschreiten dieser Kante ein Realignment vorgenommen werden muß. Die Gewichtung der Callgraphkanten mit der Beschreitungshäufigkeit ergibt sich aus dem Umstand, daß im Falle des Realignments entlang einer solchen Kante mit häufigen Umverteilungen gerechnet werden muß und daher primär versucht werden soll, die Prozeduren dieser Kante mit der selben Alignmentinformation zu versehen.

Die Alignmentinformation zweier Prozeduren kann dabei durch das Mischen beider CAGs entsprechend ihrem Callvektor errechnet werden. Voraussetzung hierfür ist natürlich, daß der Callgraph azyklisch ist! Zur Evaluierung des Ergebnisses werden die Kosten eines CAG als Summe der Gewichtungen jener Kanten, die zwischen zwei Alignmentmengen verlaufen, berechnet, wodurch die Kosten des neuen, gemischten CAG mit jenen der beiden ursprünglichen CAGs zuzüglich der Umverteilungskosten verglichen werden können und so die Entscheidungsfindung steuern.

Eine Skizzierung des Clustering Algorithm für die interprozedurale Alignmentanalyse ist aus Abb. 6.1. zu entnehmen:

```
Step 1:
      Berechne für jede Prozedur des Callgraphen den CAG;
      Gewichte die Kanten des Callgraphen mit der Häufigkeit ihres Be-
      schreitens und markiere sie als mischbar.

Step 2:
      Suche eine mischbare Kante mit maximalem Gewicht und füge die
      beiden CAGs zusammen. Evaluiere den neuen, gemischten CAG und
      bestimme, ob die beiden ursprünglichen CAGs oder der gemischte
      CAG ein besseres Verhalten zeigen. Markiere je nach Ergebnis der
      Evaluierung den gemischten CAG als permanent oder die Kante des
      Callgraphen als unmischbar.

      Goto 2
```

Abb. 6.1. - Clustering Algorithm nach [NVDG94]
zur interprozeduralen Alignmentanalyse

6.2.3. Einbindung in andere Systeme

Bei der Gestaltung des Analysetools wurde auch besonderes Augenmerk darauf gelegt, die Einbindung in andere Analysesysteme zu ermöglichen. Wie aus den Literaturbeispielen in Kapitel 5 entnommen werden kann, wird die Alignmentanalyse meist als Teil eines Komplexes behandelt, mit welchem das Datenaufteilungsproblem in allen seinen Facetten behandelt werden kann. Wie bereits erwähnt, beschäftigt sich die Alignmentanalyse nur mit der Vermeidung von Kommunikation bei cross-Referenzen, behandelt jedoch die anderen Details der Datenaufteilung wie Aufteilungsfunktionen, Blockgrößen, Prozessorenanzahl etc. nicht. Es wäre daher naheliegend, das präsentierte Konzept der Alignmentanalyse in größere Systeme einzubinden, welche die angesprochenen Probleme zu lösen suchen.

In diesem Zusammenhang soll die Anwendung der Alignmentanalyse in Verbindung mit einer neuen Strategie zur Verbesserung der Parallelisierungsmethoden vorgestellt werden.

In [MaChap95] wird ein Tool vorgestellt, welches sich mit der Anwendung von Pattern-Matching im Zuge der automatischen Parallelisierung beschäftigt. Dieses Tool ist in der Lage, nicht nur die Syntax eines Progammteiles, sondern auch die Semantik desselben zu erkennen, wodurch die erkannten Programmteile auf abstrakte Konzepte zurückgeführt werden können. Für solche Konzepte stehen in Datenbanken verschiedene optimierte parallele Realisierungen mit optimierten Datenaufteilungsmethoden bereit. Natürlich ist dieses Tool nicht in der Lage, die Parallelisierung eines gesamten Programmes zu steuern, da nur eine Anzahl von Konzepten erkannt werden und der verbleibende Rest des Codes nicht analysiert werden kann.
Die Verbindung dieses Ansatzes mit den präsentierten Verfahren zur Alignmentanalyse könnte eine neuartige Methode zur automatischen Parallelisierung ergeben, indem die Datenaufteilung durch das Pattern-Matching Tool gesteuert wird und für jene Programmteile, welche nicht erkannt werden können, mit Hilfe der Alignmentanalyse die entsprechenden Alignmentrelationen errechnet und ausgewertet werden können. Umgekehrt wäre es möglich, bei Vorliegen sehr starker Alignmentpräferenzen, welche sich im Widerspruch zu jenen befinden, die durch das Pattern-Matching Tool gefunden wurden, andere Implementierungen für dieses Konzept zu verwenden, um so die Konkurrenzen der Alignmentpräferenzen abzubauen.

Aber auch für andere Analyseverfahren, wie zum Beispiel das zur Zeit an der Universität Wien entwickelte, auf Basis von Expertensystemen operierende Tool *Expert-Adviser* [CHZ91],[HACZ93],[CFZ93],[ACHZ95] kann die Alignmentanalyse Basisdaten liefern bzw. Programmstücke, welche mit den spezialisierten Verfahren nicht analysiert werden können, behandeln.

Grundsätzlich kann festgehalten werden, daß sich die Alignmentanalyse aufgrund ihrer allgemein gefaßten Definition sehr gut zur Ergänzung spezialisierter Verfahren, die jedoch nur auf bestimmte Probleme anwendbar sind, eignet. Umgekehrt können solche spezialisierten Verfahren die weiteren Schritte zur Datenaufteilung steuern und dabei auf die Ergebnisse der Alignmentanalyse zurückgreifen.

BIBLIOGRAPHIE

[ACHZ95]
Andel Stefan, Chapman Barbara M., Hulman Jan und Zima Hans: An Expert Advisor for Parallel Programming Environments and Its Realization within the Framework of the Vienna Fortran Compilation System. Technical Report TR 95-4, Institut für Softwaretechnik und Parallele Systeme, Universität Wien, September 1995

[ALGGV94]
Ayguade Eduard, Labarta Jesus, Garcia Jordi, Girones Merce und Valero Mateo: Detecting Affinity for Automatic Data Distribution. In: Massive Parallelism, Hardware, Software and Applications. Herausg. von Furnari M. Mango, World Scientific, London, 1994, S 32-46.

[ASU86]
Aho Alfred, Sethi Ravi und Ullman Jeffrey: Compilers - Principles, Techniques, and Tools. Addison-Wesely, Reading 1986

[Bemm92]
Bemmerl Thomas: Programmierung Skalierbarer Multiprozessoren. Reihe Informatik 84, Wissenschaftsverlag, Mannheim 1992

[Benk94]
Benkner Siegfried: Vienna Fortran 90 and its Compilation. Diss., Technical Report TR 94-8, Institut für Softwaretechnik und Parallele Systeme, Universität Wien, November 1994

[Benk94a]
Benkner Siegfried: Handling Block-Cyclic Distributed Arrays in Vienna Fortran 90. Technical Report TR 94-9, Institut für Softwaretechnik und Parallele Systeme, Universität Wien, November 1994

[CBBet.al.93]
Chapman B., Benkner S., Blasko R., Brezany P., Egg M., Fahringer T., Gerndt H.M., Hulman J., Knaus B., Kutschera P., Moritsch H., Schwald A., Sipkova V. und Zima H.P.: VIENNA FORTRAN Compilation System - Version 1.0 - Users's Guide. Institut für Softwaretechnik und Parallele Systeme, Universität Wien, Jänner 1993

[CCL89]
Chen Marina, Choo Young-il und Li Jingke: Theory and Pragmatics of Compiling Efficient Parallel Code. Technical Report YALEU/DCS/TR-760, Yale University, Dezember 1989

[CFZ93]
Chapman Barbara M., Fahringer Thomas und Zima Hans P.: Automatic Support for Data Distribution on Distributed Memory Mulitprocessor Systems. Technical Report TR 93-2, Institut für Softwaretechnik und Parallele Systeme, Universität Wien, August 1993

[CGST93]
Chatterjee Siddhartha, Gilbert John R., Schreiber Robert und Teng Shang-Hua: Automatic Array Alignment in Data-Parallel Programs. In: Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages. Charleston, Jänner 1993

[CGSS94]
Chatterjee Siddhartha, Gilbert John R., Schreiber Robert und Sheffler Thomas J.: Array Distribution in Data-Parallel Programs. Technical Report 94.09., RIACS, NASA Ames Research Center, Moffett Field 1994

[CHZ91]
Chapman Barbara M., Herbeck Heinz und Zima Hans P.: Automatic Support for Data Distribution. Technical Report ACPC/TR 91-14, Austrian Center for Parallel Computation, Wien, Juli 1991

[CMZ92]
Chapman Barbara M., Mehrotra P. und Zima Hans P.: Programming in Vienna Fortran. Technical Report APCP/TR 92-3, Austrian Center for Parallel Computation, Wien, März 1992

[Croo94]
Crooks Philip: An Automatic Program Translator for Distributed Memory MIMD Machines. Diss., The Queen's University of Belfast, Belfast, November 1994

[DaRo93]
Darte Alain und Robert Yves: A Graph-Theoretic Approach to the Alignment Problem. Research Report Nr. 93-20, Laboratoire de l'Informatique du Parallèlisme, Ecole Normale Supèrieure de Lyon, Lyon, Juli 1993

[DiRo94]
Dion Michèle und Robert Yves: Mapping Affine Loop Nests: New Results. Research Report Nr. 94-30, Laboratoire de l'Informatique du Parallèlisme, Ecole Normale Supèrieure de Lyon, Lyon, November 1994

[Fahr93]
Fahringer Thomas: The Weight Finder, An Advanced Profiler for Fortran Programs. International Workshop of AP '93, Saarbrücken 1993

[Fahr94]
Fahringer Thomas: Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. IEEE Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, Mai 1994

[Flynn72]
Flynn Michael J.: Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, Vol. c-21, Nr. 9, September 1972

[GAL95]
Garcia Jordi, Ayguadè Eduard und Labarta Jesús: A Novel Approach Towards Automatic Data Distribution. Workshop on Automatic Data Layout and Performance Prediction, Center for Research on Parallel Computing, Rice University, Houston, April 1995

[Gern89]
Gerndt Michael: Automatic Parallelization for Distributed-Memory Multiprocessing Systems. Diss., Rheinische Friedrich-Wilhelms-Universität, Bonn 1989

[Gern91]
Gerndt Michael: Work Distribution in Parallel Programs for Distributed Memory Multiprocessors. Proceedings ICS 1991, Köln, ACM 1991

[GuptBan90]
Gupta Manish und Banerjee Prithviraj: Automatic Data Partitioning on Distributed Memory Multiprocessors. Technical Report UILU-ENG-90-2248, Coordinated Science Lab, University of Illinois at Urbana-Champaign, Urbana 1990

[Gupta92]
Gupta Manish: Automatic Data Partitioning on Distributed Memory Mulitcomputers. Diss., Technical Report UILU-ENG-92-2237, Coordinated Science Lab, University of Illinois at Urbana-Champaign, Urbana 1992

[Grin87]
Grindel Michael: Abhängigkeitsanalyse im SUPRENUM Parallelisierer. Diplomarbeit, Universität Bonn, Bonn 1987

[HACZ93]
Hulman Jan, Andel Stefan, Chapman Barbara M. und Zima Hans P.: Intelligent Parallelization within the Vienna Fortran Compilation System. Fourth Workshop on Compilers for Parallel Computers, Delft, December 1993

[Hert95]
Hertweck Friedrich: Realisierung paralleler Architekturen. In: Parallelrechner: Architekturen - Systeme - Werkzeuge. Herausg. von: Waldschmidt Klaus, B.G. Teubner, Stuttgart 1995, S. 215-261.

[HiLi90]
Hillier Frederick S. und Lieberman Gerald J.: Introduction to Operations Research. McGraw-Hill, New York 1990

[HKet.al.91]
Hiranandani Seema, Kennedy Ken, Koelbel Charles, Kremer Ulrich und Tseng Chau-Wen: An Overview of the Fortran D Programming System. Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, August 1991

[HKTW94]
Hiranandani Seema, Kennedy Ken, Tseng Chau-Wen und Warren Scott: The D Editor: A New Interactive Parallel Programming Tool. Proceedings Supercomputing 94, IEEE Computer Society Press, Los Alamitos 1994

[HPF94]
High Performance Fortran Forum: High Performance Fortran Language Specification Version 1.1. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, November 1994

[HVDG94]
Hurteau G., Van Dongen V. und Gao G.: Overview of EPPP - an Environment for Partable Parallel Programming. In: Proceedings of Supercomputing Symposium 94. Toronto, Canada, Juni 1994

[KenKrem94]
Kennedy Ken und Kremer Ulrich: Automatic Data Layout for High Performance Fortran. Technical Report CRPC-TR94498-2, Center for Research on Parallel Computation, Rice University, Houston, Dezember 1994

[KHRS94]
Kaushik S.D., Huang C.-H., Ramanujam J. und Sadayappan P.: Mulit-Phase Redistribution: A Communication-Efficient Approach to Array Redistribution. Technical Report OSU-CISRC-9/94-52, Department of Computer and Information Science, The Ohio State University, Columbus 1994

[Krem93]
Kremer Ulrich: NP-completeness of Dynamic Remapping. Fourth International Workshop on Compilers for Parallel Computers, Delft University of Technology, Dezember 1993

[LiChen89]
Li Jingke und Chen Marina: Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. Technical Report YALEU/DCS/TR-725, Yale University, November 1989

[MaChap95]
Di Martino B. und Chapman B.: Program Comprehension Techniques to improve Automatic Parallelization. Workshop on Automatic Data Layout and Performance Prediction, Center for Research on Parallel Computing, Rice University, Houston, April 1995

[NVDG94]
Ning Qi, Van Dongen Vincent und Gao Guang R.: Automatic Data and Computation Decomposition for Distributed Memory Machines. Technical Report CRIM-EPPP-94-04-13, Center de Recheche Informatique de Montrèal, Québec, Canada, 1994

[SSPGC95]
Sheffler Thomas J., Schreiber Robert, Pugh William und Chatterjee Siddhartha: Efficient Distribution Analysis via Graph Contraction. Workshop on Automatic Data Layout and Performance Prediction, Center for Research on Parallel Computing, Rice University, Houston, April 1995

[VDBF94]
Van Dongen V., Bonello C. und Freehill C.: High Performance C Language Specification. Technical Report CRIM-EPPP-94/04-12, Centre de Recherche Informatique de Montrèal, Québec, Canada, April 1994

[ZiChap90]
Zima Hans P. und Chapman Barbara M: Supercompilers for Parallel and Vector Computers. ACM Press Frontier Series, Addison-Wesely 1990

[ZiChap92]
Zima Hans P. und Chapman Barbara M.: Compiling for Distributed-Memory Systems. Technical Report ACPC/TR 92-17, Austrian Center for Parallel Computation, Wien, November 1992

ANHANG

Symbolverzeichnis

$\mathcal{E}^A$ ....Menge der Elemente von Array A
δ_R^A ....Aufteilungsfunktion von Array A auf das Prozessorenarray R
$\lambda^A(r)$ ....Distributierungssegment von Array A auf Prozessor r
α_i, β_i ....Koeffizienten von Schleifenindizes in Indexausdrücken
γ, δ ....Konstante Ausdrücke in Indexausdrücken
$expr_d(a)$ ....Indexausdruck von Arrayreferenz a in Dimension d
A_i, B_i ....Dimension i von Array A, B; Alignmentknoten
S_i ....Alignmentset i
$e(A_i, B_j)$.............Alignmentkante zwischen den Knoten A_i und B_j
be........................Kante im bipartiten Graphen
varlev(*e*) ....variation level von Kante *e*
Iter(Do-body) .....Anzahl der Iterationen über einen Schleifenkörper
TR ....True Ratio einer bedingten Anweisung
f ....Frequency
d..........................Dependency
loc.......................Locality
σ ....Startuptime
τ..........................Transfertime
ϕ ....Ausführungshäufigkeit des Kommunikationsstatements
θ ....Anzahl der zu kommunizierenden Daten
ω ....Weight

Abbildungsverzeichnis