Universität Passau
Fakultät für Mathematik und Informatik

Verteilung von allgemeinen Berechnungsinstanzen in automatisch generierten Schleifenprogrammen

Diplomarbeit

Verfasser: Thomas Wondrak

Aufgabensteller: Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung

Betreuer: Peter Faber

Passau, den 9. November 2005

Danksagung

Vorab danke ich Herrn Prof. Christian Lengauer, Ph.D., dafür, dass er meine Arbeit Korrektur las und mir dabei mit Hilfestellungen und Anregungen weiterhalf, wozu er sich trotz seiner Beanspruchung am Lehrstuhl Zeit nahm.
Desweiteren danke ich Herrn Dr. Martin Griebl, der für meine Fragen immer ein offenes Ohr hatte und mich mit wertvollen Ratschlägen wiederholt unterstützte.
Mein besonderer Dank gilt meinem Betreuer Herrn Peter Faber, der sich bereitwillig für lange Diskussionen Zeit nahm, für meine Probleme Verständis aufbrachte und mir mit zahlreichen wichtigen Ratschlägen helfend zur Seite stand.
Zudem bedanke ich mich bei allen Mitgliedern des `LooPo`-Teams für ihre freundliche Unterstützung: Armin Größlinger, Michael Claßen, Phillip Claßen, Michael Seibold, Tilmann Rabl, Tobias Langhammer und Georg Seidel.

Zusammenfassung

Mit der voranschreitenden Entwicklung von paralleler Hardware, wie z.B. der Entwicklung von Prozessoren mit mehreren Ausführungseinheiten, kommt der Parallelisierung von sequenziellen Programmen eine immer größere Bedeutung zu. Ein Modell zur Analyse und Transformation von Schleifen ist das Polyedermodell, das eine automatische Parallelisierung von Schleifensätzen erlaubt. Basierend auf dem Polyedermodell wurde die Methode Loop-Carried Code Placement (LCCP) entwickelt, die mehrfache Berechnungen des gleichen Werts dadurch eliminiert, dass die Berechnung nur einmal durchgeführt und das Ergebnis zur späteren Verwendung zwischengespeichert wird. Diese von den Schleifeniterationen abhängigen Berechnungen werden Berechnungsinstanzen genannt. Da die mehrfach verwendeten Berechnungsinstanzen in Arrays zwischengespeichert werden sollen, benötigen sie eine Platzierung, die durch die in dieser Arbeit vorgestellte Platzierungsmethode berechnet wird. Nach der Transformation des Programms durch LCCP wird HPF-Code erzeugt, der mit einem HPF-Compiler übersetzt wird, um ein ausführbares paralleles Programm zu erhalten. In dieser Arbeit wird dazu der freie HPF-Compiler ADAPTOR verwendet. Die in dieser Arbeit vorgestellte Platzierungsmethode erzeugt für diejenigen Berechnungsinstanzen, die zwischengespeichert werden sollen, neue Platzierungen, so dass ADAPTOR ein effizientes paralleles Programm generieren kann. Dazu wird ein Benchmark für ADAPTOR vorgestellt, der Informationen über das Laufzeitverhalten mehrerer Testfälle mit unterschiedlichen Kommunikationsmustern sammelt. Auf dessen Ergebnisse stützt sich ein Kostenmodell, das bei der Berechnung der Platzierungen Anwendung findet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Schon seit längerem wird versucht, Rechenleistung durch Parallelität z.B. zur Simulation von physikalischen Prozessen zu nutzen. Aufgrund der voranschreitenden Verkleinerung der Strukturen in der Halbleitertechnik gehen seit kurzer Zeit die großen Prozessorhersteller wie Sun, IBM und Intel dazu über, ihre Prozessoren für PCs nicht durch höhere Taktraten und Architekturänderungen, sondern durch Duplikation der Ausführungseinheiten wie z.B. mehrere Prozessorkerne auf einem Chip zu beschleunigen, so dass sich mehrere Operationen gleichzeitig ausführen lassen [Sti05]. Mit dieser Entwicklung kommt der Parallelisierung von sequenziellen Programmen eine immer größere Bedeutung zu, da nur parallele Programme das gesamte Leistungspotential der neuen Prozessoren ausnützen können.
Da die Parallelisierung sequenzieller Programme per Hand mühsam und fehleranfällig ist, sollte sie dem Programmierer durch einen Compiler abgenommen werden, der automatisch die Berechnung und die Daten auf die vorhandenen Prozessoren verteilt und Code zur Kommunikation von nicht lokalen Daten erzeugt. Dabei sollte das durch einen solchen Compiler erzeugte parallele Programm ähnlich effizient laufen, wie eine per Hand parallelisierte Version des Originalprogramms.
Um sequenzielle Fortran-Programme ohne großen Aufwand, also ohne explizit Kommunikation anzugeben, parallelisieren zu können, ist die auf dem Fortran-Standard[1] basierende Sprache High Performance Fortran (HPF) [Hig93, Hig97] als standardisierte Sprachspezifikation für parallele Rechner mit verteiltem Speicher entwickelt worden. Vorgänger von HPF waren Projekte zur Erzeugung paralleler Programme aus sequenziellen Fortran-Programmen wie `Kali` [MvR91], `Fortran D` [CFH$^+$92], das `Paradigm` Projekt [SPB93] und der `Vienna Fotran Compiler` [CMZ92]. Um ein sequenzielles Fortran-Programm zu parallelisieren, versieht der Programmierer sein Progamm mit Direktiven, die die Verteilung der verwendeten Arrays auf die Prozessoren angeben und die Schleifen markieren, die parallel ausgeführt werden sollen. Der HPF-Compiler verteilt die Berechnungen der parallelen Schleifen auf die Prozessoren entsprechend der Verteilung der Arrays, wobei er automatisch Code zur Kommunikation erzeugt, wenn auf nicht lokale Arrayelemente zugegriffen wird.
Da die HPF-Compiler nur begrenzte Analysemöglichkeiten besitzen, ist oft eine Restrukturierung des Programms per Hand nötig, um ein effizientes paralleles Programm zu erhalten. So zeigen z.B. Laufzeitmessungen von HPF-Versionen des NAS Parallel Benchmarks [BHdW$^+$95] gute Performanz und Skalierbarkeit im Vergleich zu einer handcodierten MPI-Version des gleichen Benchmarks. Diese Ergebnisse wurden allerdings erst dadurch erreicht, dass der Quellcode bei der Konversion zu HPF fast vollständig umgeschrieben wurde, was einen fast doppelt so langen Programmcode zur Folge hatte [AJMCY98]. Deshalb sollte ein HPF-Programm vor der Übersetzung durch einen HPF-Compiler so transformiert werden, dass der HPF-Compiler effizienten parallelen Code erzeugen kann.
Ein Modell zur Analyse und Transformation von Schleifen ist das Polyedermodell [GL94, GL95, Gri96], das aus dem von Leslie Lamport entwickelten Polytopmodell [Lam74, Fea92a, Fea92b, Len93] hervorgegangen ist. Das Polyedermodell erlaubt die automatische Parallelisierung von Schleifensätzen, wobei die verschiedenen Iterationen eines n-dimensionalen Schleifensatzes durch Vektoren aus $\mathbb{Z}^n$ interpretiert werden. Die Menge aller durch den Schleifensatz auszuführenden Iterationen wird durch den Polyeder bestimmt, der aus den Grenzen der

[1] Die Version 1.0 der Sprache HPF basiert auf dem Fortran90-Standard [Int91]. Ab der Version 2.0 basiert HPF auf dem jeweils gültigen Fortran-Standard [Int97, Int04].

Schleifen hervorgeht.
Basierend auf dem Polyedermodell ist die Methode Loop-Carried Code Placement (LCCP) [FGL01b, FGL04] entwickelt worden, die mehrfache Berechnungen des gleichen Werts dadurch eliminiert, dass die Berechnung nur einmal durchgeführt und zur späteren Verwendung zwischengespeichert wird, wobei vor allem von den Iterationen abhängige Berechnungen wie z.B. die mehrfache Berechnung der Werte `A(i)+B(i)` für `i` $\in \{1,\ldots,10\}$ berücksichtigt werden. Diese von den Schleifeniterationen abhängigen Berechnungen werden Berechnungsinstanzen genannt.
Da die mehrfach verwendeten Berechnungsinstanzen in Arrays zwischengespeichert werden sollen, benötigen sie eine Platzierung, die durch die in dieser Arbeit vorgestellte Platzierungsmethode berechnet wird. Nach der Transformation des Programms durch LCCP wird wieder HPF-Code erzeugt, der mit einem HPF-Compiler übersetzt wird, um ein auf dem Parallelrechner des Lehrstuhls ausführbares paralleles Programm zu erhalten. In dieser Arbeit wird dafür der freie HPF-Compiler ADAPTOR [BZ94] verwendet. Deshalb erzeugt die hier vorgestellte Platzierungsmethode nur Platzierungen, für die ADAPTOR ein effizientes paralleles Programm generieren kann. Dazu wird in dieser Arbeit ein Benchmark für ADAPTOR entworfen (siehe Abschnitt 6), der Informationen über das Laufzeitverhalten mehrerer Testfälle mit unterschiedlichen Kommunikationsmustern sammelt. Aus den Ergebnissen des Benchmarks wird anschließend ein Kostenmodell abgeleitet, das zur Berechnung der Platzierung verwendet wird.
Die Arbeit gliedert sich in folgende Abschnitte: Abschnitt 2 erläutert kurz als Grundlagen für diese Arbeit die Programmiersprache HPF, den HPF-Compiler ADAPTOR, das Polyedermodell und die Methode LCCP. Abschnitt 3 stellt die Methode zur Berechnung der Platzierungen der Berechnungsinstanzen vor und Abschnitt 4 geht kurz auf die Implementierung dieser Methode im Projekt `LooPo` ein. Anschließend skizziert Abschnitt 5 kurz die Codegenerierung in HPF-Compilern. Abschnitt 6 stellt den Benchmark für den HPF-Compiler ADAPTOR und Abschnitt 7 das daraus hervorgegangene Kostenmodell vor. Abschnitt 8 zeigt kurz einen Schleifensatz aus einer physikalischen Simulation, der mit LCCP transformiert wurde. Abschnitt 9 geht auf zwei weitere Platzierungsmethoden ein, die der in dieser Arbeit vorgestellten Platzierungsmethode ähnlich sind. Die abschließenden Bemerkungen finden sich in Abschnitt 10. Die wichtigsten Notationen für diese Arbeit sind ab Seite 100 aufgelistet.

2 Grundlagen

In diesem Abschnitt werden als grundlegende Voraussetzungen für das Thema dieser Arbeit die Programmiersprache High Performance Fortran, der Compiler ADAPTOR, das Polyedermodell, die Methode LCCP und die Darstellung linearer Relationen kurz erläutert.

2.1 High Performance Fortran

Die Sprache High Performance Fortran (HPF), deren Spezifikation durch den HPF-Standard [Hig93, Hig97] gegeben ist, erweitert die Sprache Fortran90 [Int91] bzw. seit der Version 2.0 den jeweils gültigen Fortran-Standard [Int97, Int04] mit Direktiven zur Verteilung von Arrays über Prozessoren und zur Markierung von parallel auszuführenden Schleifen. HPF ist somit eine datenparallele Programmiersprache, wobei die Verteilung der Berechnungen implizit durch die Verteilung der Arrays gegeben ist.
Eine Einführung in die Sprache bieten das Buch *„Designing and Building Parallel Programs“* [Fos94] von Ian Foster und das *„High Performance Fortran Handbook“* [KLS+94]. Eine ausführliche Übersicht über die Funktionsweise einzelner Komponenten eines HPF-Compilers ist im Buch *„High Performance Compilers for Parallel Computing“* von Michael Wolfe [Wol95] zu finden.

2.1.1 `PROCESSORS`-Direktive

HPF wurde für Topologien konzipiert, bei denen die Prozessoren in einem n_d-dimensionalen Gitter angeordnet sind. Solche Prozessorarrays sind entweder implizit definiert oder werden durch den Benutzer mittels der `PROCESSORS`-Direktive vorgegeben, die ein n_d-dimensionales Prozessorarray mit den Untergrenzen $l_1, \ldots, l_{n_d}$ und den Obergrenzen $u_1, \ldots, u_{n_d}$, wie folgt, definiert:

$$\texttt{!HPF\$ PROCESSORS P(l}_1\texttt{:u}_1\texttt{,...,l}_{n_d}\texttt{:u}_{n_d}\texttt{)}$$

Spezifiziert der Benutzer kein Prozessorarray, dann wird vom HPF-Compiler ein eingenes generiert. Der Compiler ADAPTOR [BZ94, Bra00] z.B. erzeugt bei fehlender `PROCESSORS`-Direktive ein abstraktes Prozessorarray nach folgendem Algorithmus:
Stehen insgesamt n_{cpu} Prozessoren zur Verfügung und soll die Anzahl der Prozessoren pro Dimension für ein n_d-dimensionales Prozessorarray gefunden werden, dann wird folgende Gleichung aufgestellt:

$$u_1 \cdot \ldots \cdot u_{n_d} = n_{cpu} \tag{1}$$

Dabei wird an die einzelen Faktoren folgende Bedingung gestellt:

$$u_1 \leq u_2 \leq \ldots \leq u_{n_d} \tag{2}$$

Dabei wird für die Gleichung (1) unter der Bedingung (2) eine Lösung gesucht, so dass die maximalen Faktoren gefunden werden. Tabelle 1 zeigt die standardmäßigen Prozessortopologien für 1-, 2- und 3-dimensionale Prozessorarrays.
Die Lösung der Gleichung (1) bestimmt das n_d-dimensionale Prozessorarray $\texttt{default}_{n_d}$:

$$\texttt{!HPF\$ PROCESSORS default}_{n_d}\texttt{(1:u}_1\texttt{,...,1:u}_{n_d}\texttt{)}$$

	Dimensionen des Prozessorarrays		
n_{cpu}	1-dim.	2-dim.	3-dim.
5	5	1×5	$1 \times 1 \times 5$
6	6	2×3	$1 \times 2 \times 3$
8	7	2×4	$2 \times 2 \times 2$
12	12	3×4	$2 \times 2 \times 3$
16	16	4×4	$2 \times 2 \times 4$
32	32	4×8	$2 \times 4 \times 4$

Tabelle 1: Standardmäßige Prozessortopologien von ADAPTOR aus [Bra00]

2.1.2 `DISTRIBUTE`-Direktive

Auf die definierten Prozessorarrays lassen sich mittels der `DISTRIBUTE`-Direktive Arrays und Templates verteilen:

`!HPF$ DISTRIBUTE A(dist`$_1$`,...,dist`$_{n_d}$`) [ONTO <processorArray>]`

Für jede Dimension i des n_d-dimensionalen Arrays oder Templates `A` wird durch `dist`$_i$ spezifziert, wie sie auf das Prozessorarray `processorArray` verteilt werden soll. Dazu stehen folgende Deklarationen zur Verfügung:

`*` Die entsprechende Dimension des Arrays wird nicht verteilt.

`BLOCK(b`$_i$`)` In der entsprechenden Dimension wird eine Blockverteilung mit explizit definierter Blockgröße b_i verwendet. Die Elemente des zu verteilenden Arrays in dieser Dimension werden in zusammenhängende Blöcke der Größe b_i unterteilt, wobei jeder Prozessor genau einen Block erhält. Besitzt das Array n_i Elemente in dieser Dimension, dann werden $p_i = \left\lceil \frac{n_i}{b_i} \right\rceil$ Prozessoren in dieser Dimension benötigt, die zur Laufzeit zur Verfügung stehen müssen.

`BLOCK` Hiermit wird in der entsprechenden Dimension eine Blockverteilung angegeben, wobei sich die Blockgröße b_i implizit aus der Anzahl der Elemente des Arrays und der Anzahl der in der Dimension verfügbaren Prozessoren ergibt. Die Blockgröße berechnet sich daher aus

$$b_i = \left\lceil \frac{n_i}{p_i} \right\rceil$$

wobei n_i die Anzahl der Elemente des Arrays in dieser Dimension und p_i die Anzahl der Prozessoren ist, die für diese Dimension bereitgestellt sind.

`CYCLIC(b`$_i$`)` Mit dieser Definition wird in der entsprechenden Dimension eine blockzyklische Verteilung mit Blockgröße b_i angegeben. Jeder Prozessor erhält einen zusammenhängenden Block von b_i Elementen, wobei die Blöcke zyklisch auf die verfügbaren Prozessoren verteilt werden. Sind p_i Prozessoren vorhanden, wird jeder p_i-te Block auf den gleichen Prozessor abgebildet.

`CYCLIC` Diese Definition entspricht `CYCLIC(1)`.

Die durch die `DISTRIBUTE`-Direktive definierbaren Verteilungen lassen sich als Funktion ausdrücken, die jedem Arrayelement k einen Prozessor zuordnet. Dabei ist lb die Untergrenze des Arrays in der betrachteten Dimension, und der erste Prozessor besitzt die Nummer 1.

Blockverteilung — Die Blockverteilung mit Blockgröße b lässt sich durch die Funktion

$$k \mapsto \left\lfloor \frac{k - lb}{b} \right\rfloor + 1$$

modellieren.

Blockzyklische Verteilung — Die blockzyklische Verteilung mit Blockgröße b lässt sich durch die Funktion

$$k \mapsto \left\lfloor \frac{k - lb}{b} \right\rfloor \mod n_{p_j} + 1$$

modellieren, wobei n_{p_j} die Anzahl der Prozessoren in der betrachteten Dimension darstellt.

Wird das Prozessorarray `<processorArray>` weggelassen, dann wählt der Compiler das Prozessorarray `default`$_p$, wobei p die Anzahl der Dimensionen von `A` ist, die mittels `BLOCK` oder `CYCLIC` verteilt wurden.

2.1.3 `TEMPLATE`-Direktive

Anstelle von Arrays können auch Templates verteilt werden, die keinen Speicher benötigen, da sie virtuelle Arrays sind. Ein n_d-dimensionales Template `T` mit den Untergrenzen $l_1, \dots, l_{n_d}$ und den Obergrenzen $u_1, \dots, u_{n_d}$ wird mittels der `TEMPLATE`-Direktive definiert als:

$$\texttt{!HPF\$ TEMPLATE T(l}_1\texttt{:u}_1\texttt{,...,l}_{n_d}\texttt{:u}_{n_d}\texttt{)}$$

Templates werden als Hilfsobjekte verwendet, um komplizierte Verteilungen für Arrays zu definieren. Dazu wird das Template auf ein Prozessorarray verteilt, und anschließend die Arrays im Programm mittels der `ALIGN`-Direktive an dieses Template ausgerichtet.

2.1.4 `ALIGN`-Direktive

Mit Hilfe der `ALIGN`-Direktive kann ein Array an einem bereits verteilten Array oder Template ausgerichtet werden. Der Programmierer spezifiziert, welche Elemente beider Arrays auf dem gleichen Prozessor liegen. Für den Compiler wird dadurch die Analyse erleichtert, für welche Elemente Kommunikationscode erzeugt werden muss, weil sie auf unterschiedlichen Prozessoren liegen. Die `ALIGN`-Direktive hat folgende Syntax:

$$\texttt{!HPF\$ ALIGN Source(d}_1\texttt{,...,d}_q\texttt{) WITH Target(s}_1\texttt{,...,s}_{q'}\texttt{)}$$

Das auszurichtende Array `Source` wird am Array bzw. Template `Target` ausgerichtet, wobei `Source` ein q-dimensionales Array und `Target` ein q'-dimensionales Array bzw. Template ist. Für jede Dimension i des auszurichtenden Arrays ist für $\texttt{d}_i$ folgendes erlaubt:

`*` — Die mit `*` markierte Dimension wird nicht berücksichtigt.

$\texttt{v}_i$ — Die Integervariable $\texttt{v}_i$ wird zur eindeutigen Identifikation dieser Dimension i verwendet, weshalb sie in keiner anderen Dimension j von `Source` mit $j \in \{1, \dots, q\} \setminus \{i\}$ auftreten darf.

Bei jeder Dimension i von `Target` ist für $\texttt{s}_i$ erlaubt:

`*` — Entlang der mit `*` markierten Dimension von `Target` werden die Elemente des Arrays `Source` repliziert.

Ausdruck — Die hier erlaubten Ausdrücke haben im Prinzip folgende Form:

$$a \cdot \mathtt{v}_k + b$$

Dabei sind $\mathtt{v}_k$ eine Variable, die die Dimension k des auszurichtenden Arrays `Source` bezeichnet, a eine ganzzahlige Konstante oder ein Parameter und b ein affiner Ausdruck in den Parametern.

Zusätzlich gilt die Einschränkung, dass eine Variable $\mathtt{v}_k$ in keinem weiteren Ausdruck $\mathtt{s}_l$ von `Target` mit $l \in \{1, \ldots, q'\} \setminus \{i\}$ vorkommen darf. Diese Einschränkungen garantieren eine einfache Repräsentation der Ausrichtung in der entsprechenden Dimension.

Beim Compiler ADAPTOR ist darauf zu achten, dass nur Parameter im Ausdruck vorkommen dürfen, deren Wert zur Compilezeit bekannt ist.

2.1.5 `INDEPENDENT`-Direktive

Mit der `INDEPENDENT`-Direktive markiert der Programmierer eine `DO`-Schleife, deren Iterationen unabhängig voneinander sind, weshalb sie parallel ausgeführt werden können, ohne dabei die Semantik des Programms zu verändern. Der Compiler bestimmt dann, auf welchem Prozessor die jeweiligen Iterationen des Schleifenrumpfes ausgeführt werden. Diese Verteilung erfolgt nach der **Owner-Computes-Rule** [KLS+94].

Definition 2.1 *Die **Owner-Computes-Rule** (OCR) beschreibt folgendes Partitionierungsschema der Berechnungen: Der Prozessor, der das Arrayelement besitzt, in das das Ergebnis der Berechnung abgespeichert werden soll, führt die Berechnung durch.*

Enthält der Rumpf der parallelen Schleife mehrere Schreibzugriffe, dann ist es abhängig vom Compiler, welchen Schreibzugriff er zur Partitionierung der Berechnung auswählt.

2.1.6 Array-Triplet

In Fortran90 wurde zur Angabe von regulären Arraybereichen die **Array-Triplet** Notation [Int97, MR98] eingeführt. Ein Arraybereich eines Arrays lässt sich durch folgende Notation angeben:

```
A(lb:ub:st)
```

Diese Notation spezifiziert jedes `st`-te Element des Arrays `A` zwischen `lb` und `ub` und entspricht folgender Schleife:

```
DO i=lb, ub, st
  ... A(i) ...
END DO
```

Somit lässt sich jeder Arraybereich, der als Array-Triplet angegeben ist, als Schleifensatz darstellen. Die Umkehrung dieser Aussage ist dann nicht gültig, wenn eine innere Schleife von einer äußeren Schleife abhängig ist. So gibt es für folgenden Schleifensatz *keine* Array-Triplet Darstellung:

```
DO i=1, n, 2
   DO j=i, m, 3
      ... A(i,j) ...
   END DO
END DO
```

2.1.7 Beispielprogramm

Zur Demonstration der in dieser Arbeit vorgestellten Platzierungsmethode wurde folgendes Programm konstruiert:

Programm 2.1

```
REAL, DIMENSION (1:N) :: A,B,C
REAL D

!HPF$ TEMPLATE T(1:N)
!HPF$ DISTRIBUTE T(BLOCK)
!HPF$ ALIGN A(i) WITH T(i)
!HPF$ ALIGN B(i) WITH T(i)
!HPF$ ALIGN C(i) WITH T(i)
!HPF$ ALIGN D WITH T(*)

DO i = 2, N
   A(i) = A(i) - ((B(i) * B(i-1)) / (D*D))
END DO
DO i = 1, N-1
   C(i) = C(i) + (B(i+1) * B(i))
END DO
```

Zur Verteilung der Arrays wurde ein eindimensionales Template `T` angelegt, das über die verfügbaren Prozessoren mittels der `DISTRIBUTE`-Direktive blockverteilt wurde. An dieses Template wurden die drei Arrays `A`,`B` und `C` mittels der `ALIGN`-Direktive so ausgerichtet, dass sie die gleiche Verteilung besitzen. Das Skalar wurde auf alle Prozessoren repliziert.
Von den beiden Schleifen im Programm wird an zwei Stellen im Programm in mehreren Iterationen der gleiche Wert berechnet. Zum einen wird die Multiplikation `D*D` mehrfach durchgeführt, zum anderen werden durch die beiden Ausdrücke `B(i) * B(i-1)` und `B(i+1) * B(i)` genau die gleichen Werte berechnet. So wird bei der ersten Iteration beider Schleifen von beiden Schleifen der Wert für `B(2)*B(1)` bestimmt.
Der Fluss der Daten zur Berechnung der Werte von `A` und `C` ist in Abbildung 1 darstellt. Durch die Methode „Loop-Carried Code Placement“ (siehe Abschnitt 2.4) wird für beide Berechnungen ein Hilfsarray eingeführt, in dem die Berechnung gespeichert wird. Zur Codegenerierung muss für beide Hilfsarrays eine Platzierung berechnet werden.

2.2 Der HPF-Compiler ADAPTOR

ADAPTOR [BZ94] (Automatic Data Parallelism Translator) ist ein freier HPF-Compiler, der an dem Institut SCAI der GMD (heute Fraunhofer-Gesellschaft) entwickelt wurde. Er

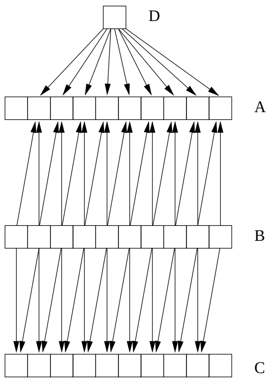


Abbildung 1: Datenfluss für das Beispiel 2.1

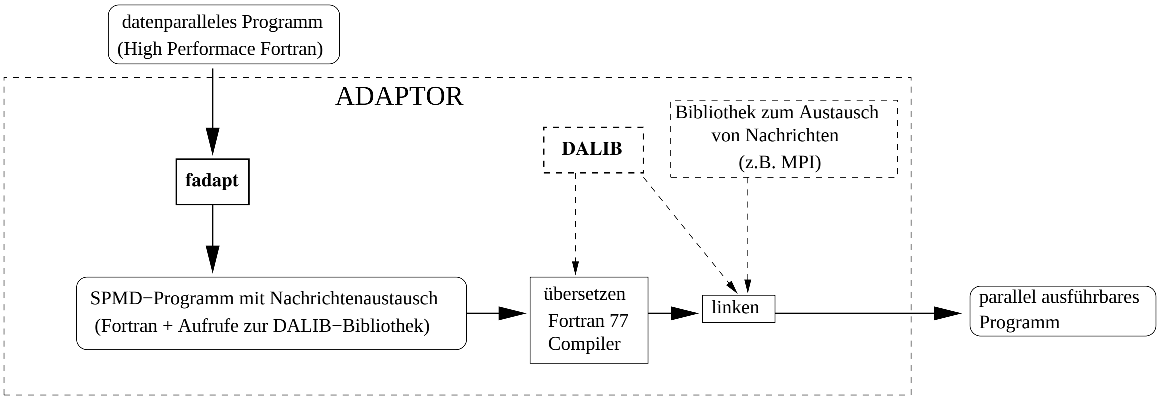


Abbildung 2: Struktur des HPF-Compilers ADAPTOR

übersetzt ein HPF-Programm in ein SPMD-Programm (single program, multiple data), das die Kommunikation durch Nachrichtenaustausch realisiert, so dass es auf Rechnerarchitekturen mit verteiltem Speicher ausgeführt werden kann.
Der interne Aufbau von ADAPTOR ist in Abbildung 2 gezeigt. Der Compiler besteht aus dem Programm `fadapt` und der DALIB (Distributed Array Library). `fadapt` ist ein Programmtransformationssystem, das ein HPF-Programm in ein SPMD-Fortran77-Programm übersetzt, das auf den lokalen Teilen der verteilten Arrays arbeitet und die Kommunikation durch Aufrufe zur DALIB realisiert. Sie enthält als Laufzeitsystem von ADAPTOR Routinen, die die Verteilung der Arrays auf die Prozessoren kontrollieren und die Kommunikation sowie die Pufferverwaltung durchführen.
Der explizite Nachrichtenaustausch wird durch eine Bibliothek zum Austausch von Nachrichten realisiert, die an den Parallelrechner angepasst sein muss. Als Bibliothek kann unter anderem eine Implementierung des MPI-Standards [Mes95] (Message Passing Interface) verwendet werden.

2.3 Polyedermodell

Das Polyedermodell [GL94, GL95, Gri96] das aus dem von Leslie Lamport entwickelten Polytopmodell [Lam74, Fea92a, Fea92b, Len93, Gri04] hervorging, ist ein mathematisches Modell zur Analyse und Transformation eines imperativen Programmfragments, das als einzige

Kontrollstruktur Schleifen und als Datenstrukur Arrays verwendet. In diesem Modell werden die einzelnen Iterationen der Schleifensätze und die sich durch die Arrayzugriffe ergebenden Datenabhängigkeiten zwischen den Schleifeniterationen durch Mengen bzw. Polyeder repräsentiert. Basierend auf dieser Repräsentation kann automatisch eine Transformation gefunden werden, deren Ergebnis eine parallele Version des Programmfragments ist. Diese Transformation wird aus einer Zeitverteilung und einer Raumverteilung konstruiert. Die Zeitverteilung (Schedule) bestimmt, zu welchem Zeitpunkt eine Schleifeniterationen ausgeführt werden kann. Wird durch die Zeitverteilung mehreren Schleifeniteration der gleiche Zeitpunkt zugewiesen, so können diese Schleifeniterationen parallel ausgeführt werden. Die Zeitverteilung berücksichtigt die Abhängigkeiten zwischen den Schleifeniterationen, weshalb garantiert wird, dass die parallele Version semantisch äquivalent zur sequenziellen Version ist. Die Raumverteilung (Placement) gibt für jede Berechnung an, auf welchem Prozessor sie ausgeführt werden soll. Sie berücksichtigt die Eigenschaften der Architektur, auf dem die parallele Version des Programms ausgeführt werden soll. Aus der transformierten Modellrepräsentation wird abschließend wieder der Programmcode erzeugt.
In diesem Abschnitt soll eine kurze Einführung in das Polyedermodell gegeben werden.

2.3.1 Affine Funktionen, homogene Koordinaten und Polyeder

Da das Polyedermodell affine Funktionen und affine Ungleichungssysteme verwendet, soll im Folgenden kurz auf die affinen Funktionen und Ungleichungssysteme und ihre Darstellung in homogenen Koordinaten eingegangen werden.

Definition 2.2 *Eine Funktion $f : \mathbb{Q}^n \to \mathbb{Q}^m$ ist **affin**, wenn gilt:*

$$\forall \mathbf{v} \in \mathbb{Q}^n : \; f(\mathbf{v}) = M_f \cdot \mathbf{v} + \mathbf{b}$$

wobei $\mathbf{b} \in \mathbb{Q}^m$ und $M_f \in \mathbb{Q}^{m \times n}$ die Funktionsmatrix ist.

Definition 2.3 *Sei $\mathbf{a}, \mathbf{v} \in \mathbb{Q}^n, b \in \mathbb{Q}$. Dann ist*

$$\mathbf{a}^{tr} \cdot \mathbf{v} \geq b$$

*eine **affine Ungleichung**, wobei $\mathbf{a}$ der Vektor der Koeffizienten, $\mathbf{v}$ der Vektor der Unbekannten und b der additive Term ist.*

Ein affines Ungleichungssystem, das aus m Ungleichungen $\mathbf{a}_i \cdot \mathbf{v} \geq b_i$ mit $\mathbf{a}_i, \mathbf{v} \in \mathbb{Q}^n$, $b_i \in \mathbb{Q}$ und $i \in \{1, \ldots, m\}$ besteht, lässt sich als Matrix $M \in \mathbb{Q}^{m \times n}$ mit $M = \begin{pmatrix} \mathbf{a}_1^{tr} \\ \vdots \\ \mathbf{a}_m^{tr} \end{pmatrix}$ wie folgt schreiben:

$$M \cdot \mathbf{v} \geq \mathbf{b}$$

Analog lässt sich ein System von m affinen Gleichungen mit

$$M \cdot \mathbf{v} = \mathbf{b}$$

beschreiben.
Um den Umgang mit affinen Funktionen zu erleichten, werden affine Abbildungen in homogenen Koordinaten [Max46, FvDF+94] dargestellt.

Definition 2.4 *Stellt* $\mathbf{v} = (v_1, \ldots, v_m)$ *einen Punkt in einem* m*-dimensionalen Raum dar, dann ist* $\mathbf{v}$ *in* ***homogenen Koordinaten*** *durch den Vektor* $(v_1, \ldots, v_m, m_c)$ *mit* $m_c = 1$ *repräsentiert. Ein* $(m+1)$*-dimensionaler Vektor* $\mathbf{u} = (u_1, \ldots, u_m, u_{m+1})$ *in homogenen Koordinaten entspricht dem Punkt* $(\frac{u_1}{u_{m+1}}, \ldots, \frac{u_m}{u_{m+1}})$ *im* m*-dimensionalen Raum.*

Der Vorteil der Darstellung in homogenen Koordinaten ist, dass sich affine Funktionen als lineare Funktionen repräsentieren lassen. Dabei wird zusätzlich eine neue Variable m_c eingefügt, die immer den Wert 1 besitzt und durch die Funktion nicht geändert wird. Daher lässt sich die Komposition von affinen Funktionen in homogenen Koordinaten als Matrixmultiplikation darstellen.
Das Polyedermodell muss neben Variablen auch mit Strukturparametern umgehen können. Strukturparameter sind Konstanten, deren Wert sich zur Laufzeit des Programms nicht ändert, aber zur Übersetzungszeit nicht bekannt ist. Siehe dazu auch Abschnitt 2.3.3. Entsprechend der Vorgehensweise mit homogenen Koordinaten lassen sich die Parameter ebenfalls in den Vektor mit aufnehmen, so dass ein n-dimensionaler Vektor, der n Variablen beschreibt, um den q-dimensionalen Vektor der Parameter erweitert wird. Dabei gilt:

$$\mathbf{v} = (v_1, \ldots, v_n, m_1, \ldots, m_q)$$

Der letzte Parameter m_q repräsentiert die 1.
Ein Polyeder lässt sich wie folgt definieren:

Definition 2.5 *Eine* ***Hyperebene*** *ist ein* $n-1$ *dimensionaler affiner Unterraum eines* n*-dimensionalen Vektorraums.*

Definition 2.6 *Ein* ***Halbraum*** *umfasst alle Punkte eines* n*-dimensionalen Vektorraums, die auf einer Seite einer Hyperebene liegen und diese Hyperebenen einschließen.*

Definition 2.7 *Ein* ***Polyeder*** *ist der Schnitt von endlich vielen Halbräumen. Ein* ***Polytop*** *ist ein beschränktes Polyeder.*

Da ein Polyeder aus dem Schnitt endlich vieler Halbräume besteht, die jeweils durch eine affine Ungleichung dargestellt werden, kann ein Polyeder durch ein affines Ungleichungssystem repräsentiert werden. Somit enthält das Polyeder alle Punkte, die das Ungleichungssystem erfüllen.

2.3.2 Eigenschaften des zu analysierenden Programmfragments

Bevor auf die Eigenschaften des Programmfragments eingegangen wird, das durch das Polyedermodell dargestellt werden kann, soll eine Klassifikation der im Programmfragment verwendeten Variablen gegeben werden:

Definition 2.8 *Eine Integer-Variable, die als Zähler für eine* `DO`*-Schleife verwendet wird, wird als* ***Index*** *bezeichnet. Die Anzahl der Indizes im Programmfragment ist* n_v*.*

Definition 2.9 *Eine Integer-Variable, deren Wert sich zur Laufzeit des gesamten Programmfragments nicht ändert, wird als* ***Strukturparameter*** *oder kurz als* ***Parameter*** *bezeichnet. Die Anzahl der Parameter, die durch das Programmfragment vorgegeben sind, werden mit* $n_b - 2$ *angegeben.*

Zusätzlich zu den durch das Programmfragment vorgegebenen Strukturparametern, werden zwei weitere Parameter definiert:

- Der Parameter m_c besitzt den Wert 1.
- Der Parameter m_∞ besitzt den Wert ∞.

Im Folgenden werden alle Vektoren immer in homogenen Koordinaten dargestellt, so dass alle Funktionen als lineare Funktionen dargestellt werden können.

Definition 2.10 *Alle Variablen, die weder Indizes noch Parameter sind, werden als Arrays bezeichnet, wobei Skalare als 0-dimensionale Arrays aufgefasst werden.*

Mit dem Polytopmodell lassen sich Programmfragmente analysieren, die folgende Eigenschaften besitzen:

- Als Kontrollstrukturen im Programmfragment sind nur `DO`-Schleifen erlaubt, bei denen die Anzahl der Iterationen zum Start der Schleife feststeht.
- Die Schleifengrenzen der `DO`-Schleifen dürfen nur lineare Ausdrücke in den Indizes und den Parametern sein. Zusätzlich sind als Untergrenzen auch das Maximum und als Obergrenzen das Minimum linearer Ausdrücke möglich.
- Die Schrittweite der Schleifen muss eine Integer-Konstante sein, wobei die Schrittweite ungleich 1 sein kann.
- Als Statements sind entweder Zuweisungen an Arrays oder Aufrufe von nebeneffektfreien Prozeduren und Funktionen erlaubt.
- Die Index-Ausdrücke der Arrays dürfen nur lineare Ausdrücke in den Indizes und den Strukturparametern sein.
- Es gibt keine Operation, die eine Adresse eines Objekts im Speicher zurückliefert.

Das Polyedermodell erweitert das Polytopmodell dadurch, dass zusätzlich Schleifen analysiert werden können, deren Anzahl an Iterationen nicht beim Eintritt in die Schleife bekannt ist, weshalb die Anzahl der Iterationen zur Übersetzungszeit unbeschränkt ist.
Das Polyedermodell bietet folgende Erweiterungen:

- Als Kontrollstrukturen sind `WHILE`-Schleifen und `DO`-Schleifen mit beliebigen Grenzen erlaubt.
- Möglich sind `IF`-Konstrollstrukturen mit beliebigen Bedingungen.

Eine ausführliche Behandlung dieser Erweiterungen findet sich in diversen Arbeiten von Griebl und Lengauer [GL94, GL95, LG95, Gri96]. Im Folgenden soll diese Erweiterung nicht betrachtet werden, da sie für die in dieser Arbeit vorgestellten Methode irrelevant ist.

2.3.3 Modell des Programmfragments

Das Programmfragment wird durch zwei Strukturen im Modell dargestellt, wobei die Iterationen durch ein Polyeder und die Abfolge der Operationen, die auf eine gleiche Speicherzelle zugreifen, durch Abhängigkeiten repräsentiert sind.

Definition 2.11 *Der **Indexraum** $\mathcal{IS}(S)$ eines Statements S, das von m Schleifen umgeben ist, wobei die j-te Schleife die Variable q_j bindet und die Obergrenze u_j und die Untergrenze l_j besitzt, beschreibt folgende Menge:*

$$\mathcal{IS}(S) = \{(q_1, \dots, q_m) \in \mathbb{Z}^m \mid \forall j \in 1, \dots, m : l_j \leq q_j \leq u_j\}$$

Ist das Statement S von keinen Schleifen umgeben, dann setzt man:

$$\mathcal{IS}(S) = \{0\}$$

Da als Ober- und Untergrenzen der Schleifen nur lineare Ausdrücke in den Indizes und den Strukturparametern erlaubt sind, lassen sich die Grenzen als lineare Ungleichungen ausdrücken, weshalb der Indexraum durch ein lineares Ungleichungssystem dargestellt werden kann. Daher lässt sich das Ungleichungssystem als eine $l \times k$ Matrix D_S darstellen, wobei l die Anzahl der affinen Grenzen der Schleifen und k die Anzahl der umgenbenden Schleifen plus die Anzahl der Strukturparameter n_b ist. Also ergibt sich für den Indexraum $\mathcal{IS}(S)$:

$$\mathcal{IS}(S) = \left\{ i \in \mathbb{Z}^k \mid D_S \cdot \mathbf{i} \geq 0 \right\}$$

Dabei ist $\mathbf{i}$ der Vektor der Indizes der umgebenden Schleifen erweitert um den n_b-dimensionalen Vektor der Strukturparameter.

Definition 2.12 *Der **Indexvektor** $\mathbf{i}$ eines Statements S ist ein Element des Indexraums $\mathcal{IS}(S)$. Es gilt also:*

$$\mathbf{i} \in \mathcal{IS}(S)$$

Im Folgenden ist der Indexvektor immer um die Strukturparameter erweitert.

Definition 2.13 *Eine **Operation** $\langle \mathbf{i}, S \rangle$ ist eine Laufzeitinstanz eines Statements S. Sie ist identifziert durch den Namen S des Statements und den Iterationsvektor $\mathbf{i} \in \mathcal{IS}(S)$. Die Menge aller Operationen im Programm wird durch Ω bezeichnet.*

Abhängigkeiten Die zweite Struktur im Modell stellt die Interaktion zwischen den Operationen aus Ω dar, die auf das gleiche Arrayelement zugreifen, und repräsentiert deren Ausführungsreihenfolge im sequenziellen Programmfragment. Diese Reihenfolge muss bei der Transformation erhalten bleiben, damit das transformierte Programmfragment semantisch äquivalent zum ursprünglichen Programmfragment ist. Die Abhängigkeitsanalyse bestimmt basierend auf den Indexräumen und den Arrayzugriffen die Abhängigkeiten.

Definition 2.14 *Der **Datenraum** $\mathcal{D}_{\mathtt{A}}$ eines n-dimensionalen Arrays $\mathtt{A}$ mit den Untergrenzen $(l_1, \dots, l_n)$ und den Obergrenzen $(u_1, \dots, u_n)$ ist wie folgt gegeben:*

$$\mathcal{D}_{\mathtt{A}} = \{l_1, \dots, u_1\} \times \dots \times \{l_n, \dots, u_n\}$$

Skalare besitzen den Datenraum $\{0\}$.

Definition 2.15 *Sei* $\mathtt{A}$ *ein* n*-dimensionals Array und sei*

$$A[\varphi_{\mathtt{A}}(\mathbf{i})]$$

ein Zugriff auf das Array $\mathtt{A}$ *im Programmfragment mit dem Indexvektor* $\mathbf{i}$ *und einer in den Indizes und Strukturparametern linearen Funktion* $\varphi_{\mathtt{A}}$*, da im Polyedermodell alle Indexzugriffe affin sind.*
Dann lässt sich die ***Zugriffsfunktion*** $\varphi_{\mathtt{A}}$ *durch eine Zugriffsmatrix* $M_{\mathtt{A}}$ *der Größe* $n \times d$ *mit*

$$\varphi_{\mathtt{A}}(\mathbf{i}) = M_{\mathtt{A}} \cdot \mathbf{i}$$

darstellen, wobei k *die Dimensionalität des Indexraums von* S *ist.*

Definition 2.16 *Sei* Ω *die Menge aller Operationen in einem Programmfragment und seien* $u, v \in \Omega$ *zwei Operationen mit* $u = \langle \mathbf{i}_u, S_u \rangle$ *und* $v = \langle \mathbf{i}_v, S_v \rangle$*, dann ist die Relation* $<_{seq} \subseteq \Omega \times \Omega$ *definiert durch:*

$$\langle \mathbf{i}_u, S_u \rangle <_{seq} \langle \mathbf{i}_v, S_v \rangle \quad \Leftrightarrow \quad \mathbf{i}_u <_{lex} \mathbf{i}_v \ \vee \ (\mathbf{i}_u = \mathbf{i}_v \ \wedge \ S_u \sqsubseteq S_v)$$

Dabei ist $<_{lex}$ die lexikographische Ordnung und $\sqsubseteq$ die textuelle Ordnung der Statements. Stehen ein Statement S_1 vor S_2 im Programmtext, dann gilt für zwei Statements $S_1 \sqsubseteq S_2$.

Definition 2.17 *Seien* $u, v \in \Omega$ *zwei Operationen mit* $u = \langle \mathbf{i}_u, S_u \rangle$ *und* $v = \langle \mathbf{i}_v, S_v \rangle$*. Dann existiert zwischen* u *und* v *eine* ***Abhängigkeit****, falls folgende zwei Bedingungen erfüllt sind:*

1. $u <_{seq} v$
2. *Ein Array* $\mathtt{A}$ *wird sowohl von* u *mit* $\varphi_{\mathtt{A},u}$ *als auch von* v *mit* $\varphi_{\mathtt{A},v}$ *referenziert, und für die Zugriffsfunktionen gilt:*
$$\varphi_{\mathtt{A},u}(\mathbf{i}_u) = \varphi_{\mathtt{A},v}(\mathbf{i}_v)$$

Sind diese Bedingungen erfüllt, sagt man: v hängt von u ab, geschrieben als $u\delta v$. u wird als Quelle und v als Ziel der Abhängigkeit bezeichnet.
Je nach Art des Zugriffs auf das Array in der Quelle und im Ziel lassen sich vier verschiedene Abhängigkeiten definieren. Diese sind in Abbildung 2 dargestellt.

Ziel (v)	Quelle (u)	
	Schreibzugriff	Lesezugriff
Schreibzugriff	Output	Anti
Lesezugriff	True	Input

Tabelle 2: Die verschiedenen Abhängigkeitstypen für die Abhängigkeit $u\delta v$

Definition 2.18 *Seien* $u, v \in \Omega$*, gelte* $u\delta v$*, da beide Operationen auf das gleiche Arrayelement* C *zugreifen, und gibt es keine weitere Operation* $w \in \Omega$*, die ebenfalls auf das Arrayelement* C *zugreift und für die* $u <_{seq} w <_{seq} v$ *gilt, dann wird die Abhängigkeit* $u\delta v$ *als* ***direkt*** *bezeichnet.*

Definition 2.19 *Eine True-Abhängigkeit, die direkt ist, wird als* ***Flow-Abhängigkeit*** *bezeichnet.*

Es gibt verschiedene Möglichkeiten, die Abhängigkeitsrelation zwischen den einzelnen Operationen durch eine endliche Repräsentation darzustellen. Sind alle Arrayzugriffe und Schleifengrenzen des zu analysierenden Programms durch lineare Ausdrücke darstellbar, dann lässt sich die Abhängigkeitsrelation mittels linearer Abbildungen exakt ausdrücken. Eine solche Abbildung wird h-Transformation genannt und wurde von Paul Feaurtier eingeführt [Fea91].

Definition 2.20 *Seien S_1 und S_2 zwei Statements, sei S_2 von S_1 abhängig, da es einige Operationen v von S_1 gibt mit $v\delta u$, wobei u eine Operation von S_2 ist, und sei $D_h \subseteq \mathcal{IS}(S_2)$. Dann ist das Paar $h = (D_h, f_h)$ eine h-Transformation, wobei für die lineare Abbildung f_h gilt:*

$$f_h : D_h \to \mathcal{IS}(S_1)$$

Die Funktion f_h bildet den Iterationsvektor der Zieloperation einer Abhängigkeit auf den Iterationsvektor einer Quelloperation ab.

2.3.4 Parallelisierung: Vorgehen

Zur Parallelisierung wird eine Transformation konstruiert, die sich aus einer Zeitverteilung (Schedule) und einer Raumverteilung (Placement) zusammensetzt.

Definition 2.21 *Ein **Schedule** θ_S eines Statements ist eine lineare Funktion, die jeder Operation einen ganzzahligen Vektor zuweist, der die logische Zeit darstellt. Die Funktion θ_S kann durch eine $t \times k$ Matrix Θ_S repräsentiert werden, wobei t die Anzahl der Zeitdimensionen und k die Dimensionalität des Indexraums von S ist.*

$$\theta_S(\mathbf{i}) = \Theta_S \cdot \mathbf{i}$$

Damit die Ausführungsreihenfolge abhängiger Operationen erhalten bleibt, wird folgende Kausalitätsbedinung an den Schedule gestellt:

$$\forall u, v \in \Omega : u\delta v \Rightarrow \theta(u) + 1 \leq \theta(v)$$

Definition 2.22 *Ein **Platzierung** (Placement) π_S eines Statements S ist eine in den Indizes und Strukturparametern lineare Funktion, die jede Operation auf einen ganzzahligen Vektor abbildet, der den virtuellen Prozessor darstellt, auf dem die Operation des Statemtents S ausgeführt werden soll. Die Funktion π_S kann als eine $p \times k$ Matrix Π_S repräsentiert werden, wobei p die Anzahl der Prozessordimensionen und k die Dimensionalität des Indexraums von S ist.*

$$\pi_S(\mathbf{i}) = \Pi_S \cdot \mathbf{i}$$

Analog zur Platzierung eines Statements lässt sich auch eine Platzierung für ein Array darstellen:

Definition 2.23 *Ein **Platzierung** $\pi_{\mathtt{A}}$ eines n-dimensionalen Arrays $\mathtt{A}$ ist eine in den Indizes und Strukturparametern lineare Funktion, die jedes Arrayelement auf einen ganzzahligen Vektor abbildet, der den virtuellen Prozessor darstellt, auf dem das Arrayelement gespeichert werden soll.*

$$\pi_{\mathtt{A}}(\mathbf{a}) = \Pi_{\mathtt{A}} \cdot \mathbf{a}$$

Dabei ist $\mathbf{a}$ der $(n+n_b)$-dimensionale Vektor der Indizes des Arrays A, und Π_A die $p \times (n+n_b)$ Matrix.

Wenn eine Platzierung eines Arrays gegeben ist, dann wird der Prozessor, auf den das Arrayelement abgebildet wird, der Besitzer (Owner) genannt. D.h. $\pi_A(\mathbf{a})$ ist der Besitzer von $A[a]$.

Die Dimensionalität des Bildes der Platzierung entspricht der Anzahl der Prozessordimensionen, die vom Parallelrechner zur Verfügung gestellt werden.

Transformationsmatrix Aus dem Schedule und der Platzierung lässt sich die Transformationsmatrix für ein Statement S wie folgt konstruieren:

$$\mathcal{T}_S = \begin{pmatrix} \Theta_S \\ \Pi_S \end{pmatrix}$$

Diese Matrix stellt die Raum-Zeitabbildung dar, die zur Transformation des Indexraums von S verwendet wird. Aus den transformierten Indexräumen wird dann wieder Programmcode erzeugt.

2.4 Loop-Carried Code Placement

Dieser Abschnitt stellt informell die Loop-Carried Code Placement Methode (LCCP) vor [FGL01b, FGL04]. Dabei zeigt Abschnitt 2.4.1 die Darstellung eines Programmfragments, wie sie in dieser Arbeit verwendet wird. Abschnitt 2.4.2 stellt informell die Vorgehensweise von LCCP dar.

2.4.1 Modell

Zur Beschreibung der einzelnen Berechnungen im Programmfragment wird ein präzises Modell der Programmausführung benötigt, das jede vom Prozessor ausgeführte Operation beschreiben und eindeutig identifizieren kann. Die Ausführung soll analog zum Polyedermodell durch Polyeder und Abhängigkeiten dargestellt werden.

Definition 2.24 *$\mathcal{F}$ sei die Menge der Funktionssymbole und enthalte folgende Funktionssymbole:*

- *Namen von Konstanten*
- *benutzerdefinierte Funktionen*
- `+,-,*,/,max,min,CALL,...`*, alle Intrinsic-Funktionen von Fortran*
- *lineare Ausdrücke*
- *Lesezugriff auf ein Array, notiert durch* `read_A`
- *Schreibzugriff auf ein Array, notiert durch* `write_A`
- `=`*: Zuweisungsoperator*

Jedes Funktionssymbol $\odot \in \mathcal{F}$ repräsentiert einen Operator. Die Anzahl der Argumente eines Funktionssymbols $\odot \in \mathcal{F}$ wird durch $\sigma(\odot)$ angegeben und ergibt sich aus Summe der Ein- und Ausgabeargumente $(\sigma_{in}, \sigma_{out})$.

Das Programmfragment lässt sich ohne seine Kontrollstrukturen als ein **Term** τ darstellen, der aus der Menge $\mathcal{F} \cup \{;\}$ konstruiert wird. Dabei bezeichnet das Funktionssymbol $;(\tau_1, \tau_2)$ die Hintereinanderausführung der Terme τ_1 und τ_2.

Beispiel 2.1 *Für das Statement der zweiten Schleife des Beispielprogramms 2.1 in Abschnitt 2.1.7 ergibt sich folgender Term:*

```
=:(+(read_C(i), *(read_B(i+1), read_B(i))), write_C(i))
```

Definition 2.25 *Um jeden Subterm τ' von τ eindeutig identifizieren zu können, wird jedem Subterm τ' durch $\mathrm{Occ}(\tau')$ eine eindeutige Zahl, die **Stelle**, zugeordnet.*
Diese Nummerierung wird so gewählt, dass sie die Ausführungsreihenfolge der Subterme repräsentiert. Wird ein Subterm τ_1 vor dem Subterm τ_2 ausgeführt, dann gilt:

$$\mathrm{Occ}(\tau_1) < \mathrm{Occ}(\tau_2)$$

Definition 2.26 *Sei τ' ein Subterm des durch τ dargestellten Programmfragments, das n_v Schleifen und n_b Parameter besitzt. Sei op die Nummer des zu betrachtenden Arguments von τ', wobei $op = 0$ die Ausführung selbst, also das Resultat der durch τ' durchgeführten Berechnung darstellt. Sei $\mathbf{P}$ der von den τ umgebenden Schleifen gegebene (n_v+n_b)-dimensionale Polyeder, dann ist $\alpha \in \mathbb{Z}^{n_v+n_b+2}$ eine* ***Stelleninstanz****, die folgenden Vektor beschreibt:*

$$\alpha = (\alpha_1, \ldots, \alpha_{n_v+n_b+2})$$

Dabei setzt sich α zusammenaus:

$$\begin{aligned} (\alpha_1, \ldots, \alpha_{n_v}, \alpha_{n_v+3}, \ldots, \alpha_{n_v+n_b+2}) &\in \mathbf{P} \\ \alpha_{n_v+1} &= \mathrm{Occ}(\tau') \\ \alpha_{n_v+2} &= op \end{aligned}$$

Die Projektion einer Stelleninstanz auf die Stelle $\mathrm{Occ}(\tau')$ wird mit $\pi_{\mathrm{Occ}}(\alpha)$ und die Projektion auf den Indexvektor von α mit $\mathrm{Idx}(\alpha)$ bezeichnet.

Definition 2.27 *Die* ***Stelleninstanzmenge*** *$\mathcal{OI}$ bezeichnet die Menge aller Stelleninstanzen α des Programmfragments.*

Da die Stelleninstanz die Indizes aller Schleifen des Programmfragments enthält, wird zur Markierung der Schleifen, die die Stelleninstanz nicht umgeben, folgende Notation eingeführt:

Definition 2.28 *Ist der zu einer Stelleninstanz α gehörige Subterm τ' nicht durch die i-te Schleife des Programmfragments umgeben, dann wird die i-te Komponente von α wie folgt gesetzt:*

$$\alpha_i = \begin{cases} -m_\infty & \textit{wenn } \alpha \textit{ textuell vor der } i\textit{-ten Schleife liegt} \\ m_\infty & \textit{wenn } \alpha \textit{ textuell nach der } i\textit{-ten Schleife liegt} \end{cases}$$

Zusätzlich zu der eingeführten Beschreibung des Programmfragments durch Stelleninstanzen muss der Datenfluss zwischen den einzelnen Subtermen von τ durch Abhängigkeiten modelliert werden. Im Folgenden wird $\alpha_1 \Delta \alpha_2$ als Bezeichung gewählt, wenn die Stelleninstanz α_2 von der Stelleninstanz α_1 abhängig ist. Dabei steht Δ für die verschiedenen Arten von Abhängigkeiten $\Delta^t, \Delta^a, \Delta^o, \Delta^i$, siehe dazu Abschnitt 2.3.3.
Um den Datenfluss in das Programmfragment und aus dem Programmfragment zu modellieren, wird vor und nach dem zu analysierenden Programmfragment für jedes Array ein Schleifensatz eingefügt, der jedes Arrayelement referenziert. Ein solcher Schleifensatz wird im Folgenden als Array-Referenzierungs-Schleife bezeichnet. Für das Array `A(m:n)` wird folgender Schleifensatz erzeugt:

```
DO i=m,n
   A(i) = A(i)
END DO
```

Nun lassen sich die Operationen des Programmfragments in diesem Modell durch den Stelleninstanzgraphen darstellen.

Definition 2.29 *Der* ***Stelleninstanzgraph*** G_{OI} *stellt das Programmfragment durch die Stelleninstanzen und Abhängigkeiten zwischen diesen dar. Die Knoten des Graphen sind die Stelleninstanzen. Zwischen zwei Knoten* α_1, α_2 *gibt es genau dann eine Kante* $(\alpha_1, \alpha_2) \in E$*, wenn* $\alpha_1 \Delta \alpha_2$ *gilt. Also ergibt sich für* $G_{OI} = (V, E)$*:*

$$\begin{aligned} V &= \mathcal{OI} \\ E &= \{\ (v_1, v_2) \mid v_1, v_2 \in V \ \wedge \ v_1 \Delta v_2 \ \} \end{aligned}$$

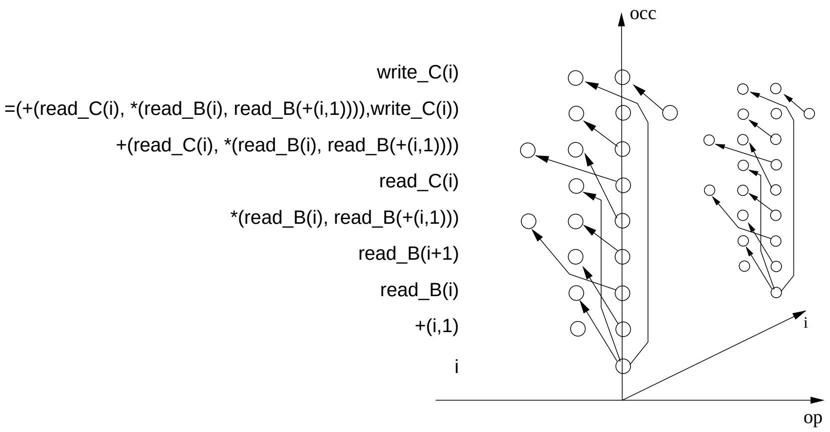


Abbildung 3: Der Stelleninstanzgraph G_{OI} der zweiten Schleife von Programm 2.1

Der Stelleninstanzgraph der zweiten Schleife des Programms 2.1 ist in Abbildung 3 für zwei Iterationen von i in einem dreidimensionalen Koordinatensystem dargestellt, wobei die Stellen (Occ) nach oben und die Argumente (op) nach links aufgetragen sind. Für jede Stelle ist der Subterm des Programmfragments angegeben.
Für die in dieser Arbeit vorgestellte Platzierungsmethode wird eine endliche Repräsentation des Stelleninstanzgraphen benötigt. Dazu werden diejenigen Stelleninstanzen, die die gleiche Stelle beschreiben, zu einer Menge zusammengefasst.

Definition 2.30 *Der* ***reduzierte Stelleninstanzgraph*** G_{ROI} *ist eine endliche Repräsentation des Graphen* G_{OI}*. Die Knoten des Graphen sind Stelleninstanzmengen mit* $O_i \subseteq \mathcal{OI}$*, wobei eine Stelleninstanzmenge* O_i *nur Stelleninstanzen der gleichen Stelle enthält. In* E *gibt es genau dann eine Kante* $v_1 \rightarrow v_2$ *mit* $v_1, v_2 \in V$*, falls zwei Stelleninstanzen* $\alpha_1 \in v_1$ *und* $\alpha_2 \in v_2$ *existieren, für die* $\alpha_1 \Delta \alpha_2$ *gilt. Die Kante besitzt als Markierung die h-Transformation h, die diese Abhängigkeit beschreibt.*
Also ergibt sich für $G_{ROI} = (V, E)$*:*

$$\begin{aligned} V &= \{O_1, \ldots, O_n\} \ \ mit \ \ O_i \subseteq \mathcal{OI} \ \wedge \ (\forall \alpha \in O_i : \forall \alpha' \in O_i : \ \pi_{\text{Occ}}(\alpha) = \pi_{\text{Occ}}(\alpha')) \ \wedge \\ & \qquad\qquad\qquad\qquad\qquad\quad (\forall j \in \{1, \ldots, n\} \backslash \{i\} : \ O_i \cap O_j = \emptyset) \\ E &= \{\ (v_1, v_2, h = (D_h, f_h)) \mid v_1, v_2 \in V \ \wedge \ \exists \alpha_1 \in v_1, \ \exists \alpha_2 \in v_2 : \ \alpha_1 \Delta \alpha_2, \\ & \qquad D_h = \{\hat{\alpha}_2 \mid \hat{\alpha}_2 \in v_2, \ \exists \hat{\alpha}_1 \in v_1 : \ \hat{\alpha}_1 \Delta \hat{\alpha}_2\}, \ f_h(\alpha_2) = \alpha_1\} \end{aligned}$$

Ein reduzierter Stelleninstanzgraph G_{ROI} der beiden Schleifensätze des Programms 2.1 ist in Abbildung 4 gezeigt. Dabei ist für jeden Knoten v die Menge der Indizes $\text{Idx}(O_v)$ angegeben, für die der Knoten ausgeführt wird, wobei die Menge $\mathrm{Idx}(O_v)$ nur auf die Dimensionen projiziert wurde, denen nicht der Wert $\pm m_\infty$ zugeordnet ist. Zusätzlich ist die Stelle $occ = \pi_{Occ}(O_v)$ und das zugehörige Funktionssymbol aus $\mathcal{F}$ angegeben. Die Kanten des Graphen sind als Pfeile im Graphen dargestellt, die als Markierung den Teil der zugehörigen h-Transformation angeben, der die abhängigen Indizes beschreibt. Die Pfeile zeigen von der Quelle einer Abhängigkeit zu ihrem Ziel. Der linke Graph ist der reduzierte Stelleninstanzgraph für die zweite Schleife und der rechte ist der reduzierte Stelleninstanzgraph für die erste Schleife.

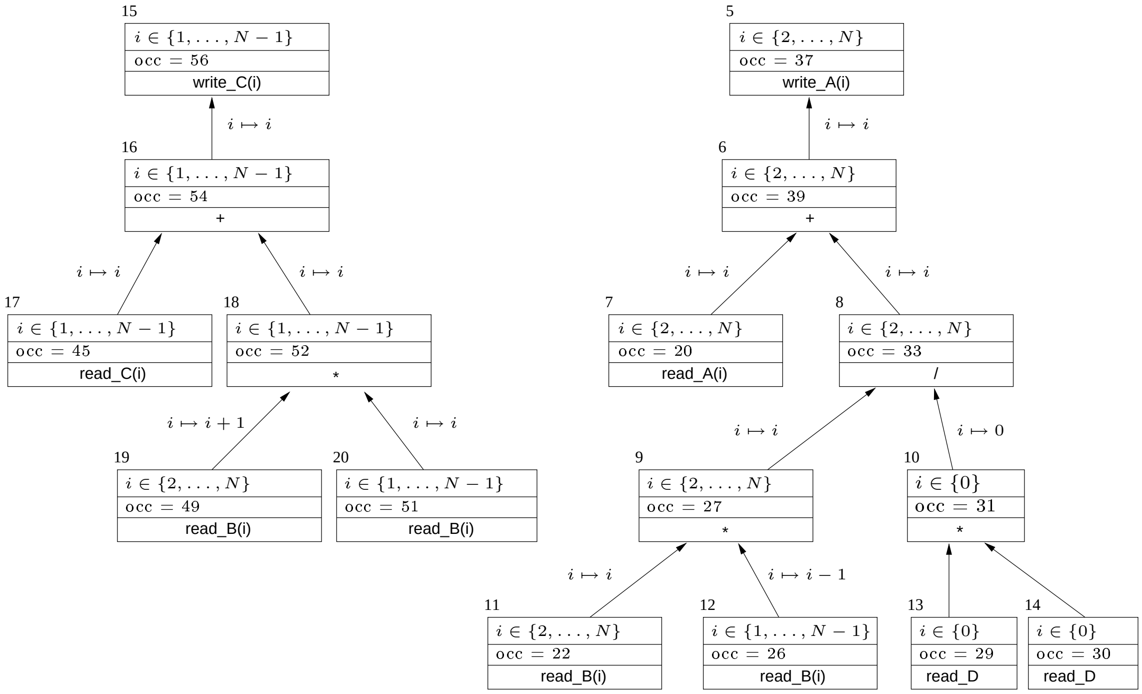


Abbildung 4: Der reduzierte Stelleninstanzgraph G_{ROI} des Programms 2.1

2.4.2 Die Methode

Im Folgenden soll kurz informell die Loop-Carried Code Placement Methode vorgestellt werden [FGL01a, FGL04].

Das Ziel dieser Methode ist es, so viele redundante Berechnungen wie möglich zu eliminieren, um Rechenzeit zu sparen. Dazu werden zuerst im Stelleninstanzgraphen G_{OI} alle Stelleninstanzen, die den gleichen Wert berechnen, zu einer Menge zusammengefasst. Um eine Stelleninstanz zu finden, die diese Menge repräsentiert, kann entweder eine Stelleninstanz aus dieser Menge ausgewählt werden oder eine neue Stelleninstanz erzeugt werden.

Der Stelleninstanzgraph G_{OI} wird danach so umgeschrieben, dass allein die Stelleninstanz im Graphen verbleibt, die als Repräsentantin für diese Berechnung ausgewählt wurde. Der so erzeugte Stelleninstanzgraph wird **kondensierter Stelleninstanzgraph** G_{COI} genannt. Die Stelleninstanz im Graphen, die als Repräsentant für einen mehrfach berechneten Wert gewählt wurde, zeichnet sich im G_{COI} dadurch aus, dass von ihr mehrere Flussabhängigkeiten ausgehen. Diese ergeben sich dadurch, dass der berechnete Wert mehrfach verwendet wird. Solche Stelleninstanzen werden im weiteren Text als **interessant** bezeichnet, da für sie bei der Code-Erzeugung Hilfsarrays angelegt werden müssen, weshalb sie eine Platzierung benötigen.

Definition 2.31 *Eine **interessante Stelleninstanz** ist eine Stelleninstanz aus dem G_{COI}, von der mehr als eine Flussabhängigkeit ausgeht oder die einen Schreibzugriff darstellt.*

Analog zu dieser Definition gibt es im **reduzierten kondensierten Stelleninstanzgraphen** G_{RCOI} interessante Stelleninstanzmengen:
Eine Stelleninstanzmenge v aus dem Graphen G_{RCOI} ist **interessant**, wenn sie eine der folgenden Eigenschaften besitzt:

1. Von v gehen mehrere Flussabhängigkeiten aus, da v eine Menge von berechneten Werten beschreibt, die von unterschiedlichen Stelleninstanzmengen im G_{RCOI} verwendet werden.

2. Von v geht mindestens eine Abhängigkeit aus, wobei die h-Transformation, die diese Abhängigkeit beschreibt, einen Kern besitzt.

3. Der Knoten v repräsentiert einen Schreibzugriff auf ein Array.

In Abbildung 5 ist der reduzierte kondensierte Stelleninstanzgraph des Programms 2.1 graphisch dargestellt. Dabei sind die interessanten Knoten durch dickere Rahmen hervorgehoben. Für sie muss eine Platzierung berechnet werden, da für diese Knoten bei der Codegenerierung Hilfsarrays angelegt werden müssen.

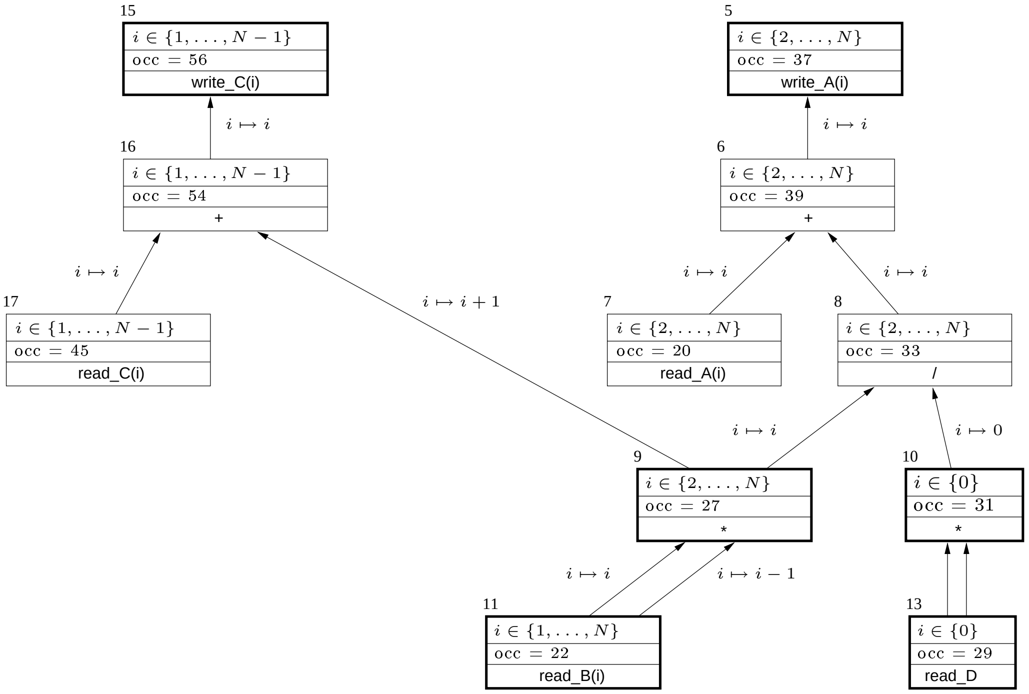


Abbildung 5: Der reduzierte kondensierte Stelleninstanzgraph G_{RCOI} des Programms 2.1; dabei sind die Knoten $v_5, v_9, v_{10}, v_{11}, v_{13}$ und v_{15} interessante Knoten

2.5 Lineare Relationen

Neben linearen Funktionen sind zur Darstellung inverser h-Transformationen oder zur Modellierung von `ALIGN`-Direktiven auch lineare Relationen nötig. In diesem Abschnitt soll kurz

eine Notation einer linearen Relation durch ein Paar von Abbildungen mit zugehörigen Operationen vorgestellt werden.
Eine lineare Relation zwischen zwei Polyedern ist wie folgt definiert:

Definition 2.32 *Eine* ***lineare Relation*** *$\mathcal{R}$ zwischen zwei Polyedern $P_1 = \{x \in \mathbb{Q}^k \mid M_{P_1} \cdot x \geq 0\}$ mit $M_{P_1} \in \mathbb{Q}^{q \times k}$ und $P_2 = \{x \in \mathbb{Q}^{k'} \mid M_{P_2} \cdot x \geq 0\}$ mit $M_{P_2} \in \mathbb{Q}^{q' \times k'}$ ist eine Relation $\mathcal{R} \subseteq \mathbb{Q}^k \times \mathbb{Q}^{k'}$, die sich darstellen lässt als*

$$\mathcal{R} = \left\{ (x,y) \;\middle|\; x \in P_1, y \in P_2 \quad \wedge \quad M_{\mathcal{R}} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = 0 \right\}$$

für eine passende $l \times (k+k')$ Matrix $M_{\mathcal{R}}$.

Um die Menge der Elemente aus P_2 anzugeben, die mit einem Element $x \in P_1$ in Beziehung stehen, wird die Notation $\mathcal{R}(x)$ verwendet, die folgende Menge beschreibt:

$$\mathcal{R}(x) = \{y \mid y \in P_2 \wedge (x,y) \in \mathcal{R}\}$$

Für eine lineare Relation $\mathcal{R}$ gibt es mehrere unterschiedliche Darstellungsformen:

1. Eine lineare Relation $\mathcal{R}$ wie in Definition 2.32 lässt sich durch folgendes Polyeder darstellen:

$$\mathcal{R} = \left\{ (x,y) \;\middle|\; \begin{pmatrix} \multicolumn{2}{c}{M_{\mathcal{R}}} \\ \multicolumn{2}{c}{-M_{\mathcal{R}}} \\ M_{P_1} & 0 \\ 0 & M_{P_2} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \geq 0 \right\}$$

2. Falls nur die Beziehungen zwischen den Punkten aus den Polyedern P_1 und P_2 interessieren, ist die Darstellung einer linearen Relation $\mathcal{R}$ durch ein Paar von linearen Abbildungen möglich.

   Sei $X = \mathbb{Q}^k$ ein Vektorraum mit gleich vielen Dimensionen wie P_1, sei $Y = \mathbb{Q}^{k'}$ ein Vektorraum mit gleich vielen Dimensionen wie P_2 und sei $Z = \mathbb{Q}^l$ ein Vektorraum. Die durch die Matrix $M_{\mathcal{R}}$ in Definition 2.32 beschriebenen Beziehungen zwischen den Punkten der Polyeder P_1 und P_2 lassen sich durch ein Paar von linearen Funktionen $r = (r_L, r_R)$ darstellen, wobei r_L von X in den Zwischenraum Z und r_R von Y in den Zwischenraum Z abbilden, was in Abbildung 6 gezeigt ist. Da r_L und r_R zwei lineare Abbildungen sind, werden sie durch die $l \times k$ Matrix M_{r_L} und die $l \times k'$ Matrix M_{r_R} wie folgt dargestellt:

$$\begin{aligned} r_L(x) &= M_{r_L} \cdot x \\ r_R(y) &= M_{r_R} \cdot y \end{aligned}$$

   Dabei ist $x \in X$ und $y \in Y$. Für ein $x \in X$ beschreibt der Ausdruck

$$(r_R^{-1} \circ r_L)(x)$$

   die Teilmenge von Y, deren Elemente mit x in Relation stehen. Dieser Ausdruck repräsentiert genau dann eine mehrelementige Teilmenge von Y, wenn r_R einen Kern besitzt und daher nicht injektiv ist. Diese Menge lässt sich auch durch den Ausdruck

$$(r_R^g \circ r_L)(x) + \mathrm{Kern}(r_R)$$

$$\begin{tikzcd} X \arrow[r, "r_L"] \arrow[rr, bend right, "r"'] & Z & Y \arrow[l, "r_R"'] \end{tikzcd}$$

Abbildung 6: Darstellung einer linearen Relation durch ein Paar von Abbildungen

beschreiben, wobei $+$ die Mengenaddition $+(x, \{y_1, y_2, \ldots\}) = \{+(x, y_1),\ +(x, y_2), \ldots\}$ und r_R^g eine **generalisierte Inverse** von r_R ist, die einen Punkt aus dem Raum $\mathrm{Kern}(r_R)$ auswählt [Usm87].

Analog zur Matrix $M_{\mathcal{R}}$ aus Definition 2.32, kann auch $r = (r_L, r_R)$ durch eine $l \times (k+k')$ Matrix M_r dargestellt werden, die aus den Matrizen M_{r_L} und M_{r_R} wie folgt konstruiert wird:

$$M_r = (M_{r_L} | - M_{r_R})$$

Diese Matrix kann wie $M_{\mathcal{R}}$ zur Darstellung der durch r beschriebenen linearen Relation $\mathcal{R}_r$ benutzt werden:

$$\mathcal{R}_r = \left\{ (x, y) \;\middle|\; x \in \mathbb{Q}^k, y \in \mathbb{Q}^{k'} \quad \wedge \quad M_r \cdot \begin{pmatrix} x \\ y \end{pmatrix} = 0 \right\}$$

Da in dieser Arbeit die Darstellung der linearen Relation r sowohl durch ein Paar von Abbildungen als auch durch eine Matrix M_r benötigt wird, wird eine Abbildung RelMat definiert, die aus einer linearen Relation $r = (r_L, r_R)$ die Matrix M_r erzeugt:

$$\mathrm{RelMat} : r \mapsto M_r = (M_{r_L} | M_{r_R})$$

Ebenso kann aus einer Matrix M_r das Paar von Abbildungen $r = (r_L, r_R)$ wieder erzeugt werden, da die Matrix M_r die Form $M_r = (M_{r_L} | -M_{r_R})$ besitzt, was durch die Abbildung RelMat^{-1} geschieht:

$$\mathrm{RelMat}^{-1} : M_r \mapsto r = (r_L, r_R)$$

2.5.1 Operationen auf linearen Relationen

Als Operationen für die Darstellung linearer Relationen als Paare von Abbildungen werden die Invertierung, die Komposition und der Test auf Gleichheit definiert.

Invertierung Sei $r = (r_L, r_R)$ eine lineare Relation, dann ist

$$r^{-1} = (r_R, r_L)$$

die Inverse von r.

Komposition zweier linearer Relationen Die lineare Relation, die die Komposition zweier linearer Relationen $r_1 = (r_{1_L}, r_{2_R})$ und $r_2 = (r_{1_L}, r_{2_R})$ repräsentiert, ist wie folgt gegeben:

$$r_{2_R}^{-1} \circ r_{2_L} \circ r_{1_R}^{-1} \circ r_{1_L}$$

Es werden zwei verschiedene Möglichkeiten definiert, die beiden linearen Relationen zu verküpfen, abhängig davon, ob bei der Komposition die Vereinigung oder die Differenz der beiden durch $r_{2_R}^{-1}$ und $r_{1_R}^{-1}$ beschriebenen Mengen gebildet werden soll. Somit ergeben sich zwei verschiedene Kompositionsarten, die wie folgt notiert werden:

$r_2 \circ_+ r_1$ Die Menge der neuen Relation besteht aus der Vereinigung der durch $r_{1_R}^{-1}$ und $r_{1_R}^{-1}$ beschriebenen Mengen.

$r_2 \circ_- r_1$ Die Menge der neuen Relation wird durch die Mengendifferenz gebildet, indem von der durch $r_{1_R}^{-1}$ beschriebenen Menge die durch $r_{2_R}^{-1}$ beschriebene Menge abgezogen wird.

Folgendes Beispiel soll kurz die Funktionsweise der beiden Kompositionsarten darstellen.

Beispiel 2.2 *Seien die beiden linearen Relationen $r_1 = (r_{1_L}, r_{1_R})$ und $r_2 = (r_{2_L}, r_{2_R})$ gegeben durch:*

$$\begin{array}{ll} r_{1_L}(i,j) = (i,0) & r_{1_R}(i',j') = (i',0) \\ r_{2_L}(i,j) = (0,j) & r_{2_R}(i',j') = (0,j') \end{array}$$

Dann beschreiben $r_{1_R}^{-1}$ und $r_{2_R}^{-1}$ folgende Mengen:

$$\begin{array}{rcl} W_{r_1}(i,j) & = & (i,0) + \{(0,x) \mid x \in \mathbb{Q}\} \\ W_{r_2}(i,j) & = & (0,j) + \{(x,0) \mid x \in \mathbb{Q}\} \end{array}$$

Die neue Menge W_+ wird aus der Vereinung und die Menge W_- aus der Mengendifferenz bestimmt. Dann erhält man folgende Mengen:

$$\begin{array}{rcl} W_+(i,j) & = & (0,0) + \{(x,y) \mid x \in \mathbb{Q} \wedge y \in \mathbb{Q}\} \\ W_-(i,j) & = & (i,0) + \{(0,x) \mid x \in \mathbb{Q}\} \end{array}$$

Zur Berechnung der Komposition $r_+ = r_1 \circ_+ r_2$ mit $r_+ = (r_{+_L}, r_{+_R})$ wird die Menge W_+ berücksichtigt, und daher als Ergebnis folgende lineare Relation erzeugt:

$$r_{+_L}(i,j) = (0,0) \qquad r_{+_R}(i',j') = (0,0)$$

Die Komposition $r_- = r_1 \circ_- r_2$ mit $r_- = (r_{-_L}, r_{-_R})$ hingegen berücksichtigt die Menge W_- und erzeugt daher folgende lineare Relation:

$$r_{-_L}(i,j) = (i,0) \qquad r_{-_R}(i',j') = (i',0)$$

Komposition einer linearen Relation mit einer linearen Funktion Um die Notation in dieser Arbeit zu vereinfachen, wird im Folgenden die Komposition einer linearen Relation r und einer linearen Funktion f definiert. Dazu wird die lineare Funktion $f : \mathbb{Q}^q \to \mathbb{Q}^k$ zu einer linearen Relation $r_f = (r_{f_L}, r_{f_R})$ wie folgt konvertiert:

$$\begin{array}{rcl} r_{f_L} & = & f \\ r_{f_R} & = & \mathrm{id}_k \end{array}$$

Dabei ist id_k die Identitätsfunktion mit $id_k : \mathbb{Q}^k \to \mathbb{Q}^k$. Die lineare Relation r_f ist identisch mit f, da für die lineare Relation r_f gilt:

$$r_f(x) = (r_{f_R}^{-1} \circ r_{f_L})(x) = (\mathrm{id}_k^{-1} \circ r_{f_L})(x) = r_{f_L}(x) = f(x)$$

Somit wird die Komposition einer linearen Relation $r = (r_L, r_R)$ mit $r_L : \mathbb{Q}^k \to \mathbb{Q}^l$ und $r_R : \mathbb{Q}^{k'} \to \mathbb{Q}^l$ mit einer linearen Funktion $f : \mathbb{Q}^q \to \mathbb{Q}^k$ wie folgt definiert:

$$r \circ f = r \circ_+ r_f$$

Dabei ist $r_f = (f, \mathrm{id}_k)$ die der Funktion f entsprechende lineare Relation.

Test auf Gleichheit Um Duplikate eliminieren zu können, wird eine Methode benötigt, die die Gleichheit zweier linearer Relationen erkennt. Da für die Darstellung einer linearen Relation durch ein Paar von Abbildungen nur die Beziehungen zwischen den Punkten der Polyeder interessieren, genügt es, für die Bestimmung der Gleichheit zweier linearer Relationen nur Beziehungen zwischen den Punkten zu berücksichtigen, und die Mengen, die durch die Polyeder beschrieben werden, außer Acht zu lassen. Somit sind zwei lineare Relation r_1 und r_2 gleich, wenn sie beide die gleichen Abbildungen besitzen.
Die Gleicheit zweier linearer Relationen $r_1 = (r_{1_L}, r_{1_R})$ und $r_2 = (r_{2_L}, r_{2_R})$ wird durch folgende drei Schritte bestimmt:

1. Für jede der beiden Relationen werden zunächst die Matrizen $M_{r_1} = \mathrm{RelMat}(r_1)$ und $M_{r_2} = \mathrm{RelMat}(r_2)$ erzeugt.

2. Diese beiden Matrizen werden anschließend in reduzierte Zeilenstufenform transformiert, und alle Nullzeilen werden aus den Matrizen entfernt. Durch die Transformation in die reduzierte Zeilenstufenform werden die beiden Matrizen in eine eindeutige Darstellung konvertiert [KM98].

3. Zuletzt werden die transformierten Matrizen verglichen. Sind beide Matrizen gleich, d.h. $M_{r_1} = M_{r_2}$, dann sind auch beide linearen Relationen r_1 und r_2 gleich, da beide linearen Relationen r_1 und r_1 die gleiche Beziehung zwischen den Elementen beschreiben.

3 Platzierungsmethode für allgemeine Berechnungsinstanzen

Im Folgenden wird eine Methode zur Berechnung von Platzierungen vorgestellt, die im Rahmen dieser Arbeit implementiert wurde. Diese Methode bestimmt für ein zu analysierendes Programmfragment, das als reduzierter Stelleninstanzgraph G_{ROI} gegeben ist, für ausgewählte Knoten im G_{ROI} dadurch eine Platzierung, dass nach der Berechnung einer Menge von möglichen Platzierungen diejenige ausgewählt wird, die entsprechend einem frei wählbaren Kostenmodell die beste Kommunikation erzeugt.
Mit der Methode wird für die von LCCP (siehe Abschnitt 2.4) eingeführten Hilfsarrays eine Platzierung berechnet, die es dem zur Codegenerierung eingesetzten HPF-Compiler ermöglicht, eine effiziente Kommunikation zu generieren, um die Zeit, die durch die Einsparung von redundanter Berechnung gewonnen wird, nicht für den Zugriff auf das Hilfsarray aufzubrauchen. Zur Bewertung der Kommunikation wird ein Kostenmodell verwendet, das sowohl den HPF-Compiler als auch den Parallelrechner berücksichtigt.
Das Grundgerüst der Platzierungsmethode zeigt Algorithmus 1, wobei die Bestimmung der Platzierungen in vier Schritten erfolgt. Die als erster Schritt durchgeführte Initialisierung überträgt die Platzierungen der Arrays auf Knoten im G_{ROI} und markiert die Knoten, für die eine neue Platzierung erzeugt werden soll.
Im zweiten Schritt wird für jeden markierten Knoten aus den initialen Platzierungen eine Menge von möglichen Platzierungen generiert. Es folgt im dritten Schritt eine Nachbearbeitung der erzeugten Platzierungen. Im vierten Schritt wird die Kombination der Platzierungen ausgewählt, für die entsprechend dem Kostenmodell die beste Kommunikation erzeugt wird.

Eingabe	Das zu analysierende Programmfragment ist als reduzierter Stelleninstanzgraph G_{ROI} gegeben. Alle verwendeten Arrays besitzen eine als lineare Relation dargestellte Platzierung.
Ausgabe	Die neu berechneten Platzierungen der interessanten Knoten des G_{ROI}.

Algorithmus

1. Markiere die Knoten im G_{ROI}, für die bereits eine initiale Platzierung bekannt ist.
2. Propagiere die initialen Platzierungen entlang der Kanten im Graphen G_{ROI} zu den markierten Knoten, die keine initiale Platzierung besitzen, so dass jedem dieser Knoten mehrere neue Platzierungen zugeordnet sind.
3. Bearbeite die bereits gefundenen Platzierungen nach, indem z.B. nicht durch HPF darstellbare Platzierungen entfernt werden.
4. Berechne für jede mögliche Kombination von Platzierungen die entstehende Kommunikation und bewerte sie entsprechend dem Kostenmodell. Wähle die Kombination von Platzierungen aus, deren Kommunikation die beste Bewertung erhalten hat.

Algorithmus 1: Das Grundgerüst der Methode zur Bestimmung der Platzierung von Stelleninstanzen

Dieser Abschnitt ist entsprechend der vier Schritte in Alogrithmus 1 aufgebaut: Abschnitt 3.1 beschreibt die Darstellung der Verteilung im Modell und die Bestimmung der initialen

Platzierungen. Abschnitt 3.2 stellt die Propagierung der Platzierungen vor und Abschnitt 3.3 beschreibt die Nachbearbeitung der Platzierungen. Abschnitt 3.4 beschreibt die Berechnung der Kommunikation und die Auswahl einer Platzierung. Abschnitt 3.5 fasst die in den vorangegangenen Abschnitten beschriebenen Teilschritte der Platzierungsmethode zu einem Algorithmus zusammen.

3.1 Das Modell für die Verteilung

Ehe auf die Berechnung der Platzierungen eingegangen werden kann, wird zuerst in Abschnitt 3.1.1 die Darstellung der Platzierungen im Modell dargelegt. In Abschnitt 3.1.2 wird die Bestimmung der initialen Platzierungen und in Abschnitt 3.1.3 als Beispiel die Darstellung der initialen Platzierungen der Arrays aus Programm 2.1 gezeigt.

3.1.1 Modell der Verteilung eines Arrays

Zuerst soll die Darstellung der Verteilung der Arrays im Modell aufgezeigt werden. Weil zur Compilezeit im Allgemeinen die Anzahl der zur Verfügung stehenden Prozessoren nicht bekannt ist und es zudem unmöglich ist, eine Verteilung, bei der die Anzahl der Prozessoren unbekannt ist, durch eine lineare Abbildung darzustellen, wird im Modell die genaue Verteilung auf die realen Prozessoren nicht berücksichtigt. Deshalb wird nur die `ALIGN`-Direktive zur Repräsentation der Verteilung eines Arrays gewählt. Dabei wird ohne Beschränkung der Allgemeinheit angenommen, dass jedes Array im Programmfragment durch eine `ALIGN`-Direktive an ein entsprechendes Template ausgerichtet wird, dessen Dimensionen verteilt sind.
Durch diese Form kann man für jede mögliche Prozessorkonfiguration folgende Eigenschaft erhalten: Sind zwei Elemente zwei unterschiedlicher Arrays dem gleichen Template-Element zugeordnet, dann werden sie zur Laufzeit auf dem gleichen Prozessor liegen. Sind sie nicht dem gleichen Template-Element zugeordnet, dann können sie je nach Prozessorkonfigruation auf unterschiedlichen Prozessoren gespeichert sein.
Im Polyedermodell wird üblicherweise eine Platzierung eines Arrays durch eine lineare Funktion dargestellt, die jedem Arrayelement eindeutig einen virtuellen Prozessor zuordnet (siehe Definition 2.23). Also lässt sich mit dieser Modellierung keine Replikation darstellen, da bei der Replikation ein Arrayelement auf mehreren Prozessoren gespeichert ist.
Aus diesem Grund wird eine `ALIGN`-Direktive der Form

```
!HPF$ ALIGN A(...) WITH T(...)
```

durch eine lineare Relation zwischen dem Datenraum $\mathcal{D}_{\mathtt{A}}$ des Arrays `A` und dem Datenraum $\mathcal{D}_{\mathtt{T}}$ des Templates `T` modelliert.
Da die hier vorgestellte Methode zur Berechnung von neuen Platzierungen nur die Beziehung zwischen den Array- und Template-Elementen benötigt, ist die genaue Ausdehnung sowohl des Arrays als auch des Templates irrelevant, weshalb die lineare Relation entsprechend Abschnitt 2.5 durch ein Paar von Abbildungen dargestellt werden kann. Die Platzierung eines n-dimensionalen Arrays `A` auf ein m-dimensionales Template `T` wird durch die folgende lineare Relation beschrieben:

$$\tilde{\Phi}_{\mathtt{A}} = (\tilde{\Phi}_{\mathtt{A}_{OI}}, \tilde{\Phi}_{\mathtt{A}_P})$$

Sie besteht aus der linearen Funktion $\tilde{\Phi}_{\mathtt{A}_{OI}} : \mathbb{Z}^n \to \mathbb{Z}^l$, die die Elemente des Datenraums $\mathcal{D}_{\mathtt{A}}$ des Arrays `A` in einen l-dimensionalen Zwischenraum abbildet, und aus der Funktion $\tilde{\Phi}_{\mathtt{A}_P} : \mathbb{Z}^m \to$

$\mathbb{Z}^l$, die die Elemente des Datenraums des Templates `T` in den l-dimensionalen Zwischenraum abbildet.
Damit die Platzierungen $\tilde{\Phi}$ der Arrays in einen einheitlichen Prozessorraum abbilden, wird ein virtueller Prozessorraum $\mathcal{P}$ definiert, in dem für jede Dimension der Templates eine eigene Dimension reserviert ist. Sind im Programmfragment n Templates definiert, dann lässt sich die Anzahl n_{pDim} der benötigten Dimensionen für die virtuellen Prozessoren wie folgt berechnen:

$$n_{pDim} = \sum_{i=1}^{n} \dim(\mathtt{T}_i) \tag{3}$$

Dabei gibt die Funktion $\dim(\mathtt{T}_i)$ die Anzahl der Dimensionen des i-ten Templates an.
Der Raum der virtuellen Prozessoren $\mathcal{P}$ besitzt daher $n_{pDim} + n_b$ Dimensionen in homogenen Koordinaten. Im Prinzip entspricht der Raum $\mathcal{P}$ dem Kreuzprodukt der Datenräume der verwendeten Templates im Programm.
Damit zu erkennen ist, welches Template die Platzierung $\tilde{\Phi}$ verwendet, setzt $\tilde{\Phi}$ den Wert für die Dimensionen, die den anderen Templates entsprechen, auf den Wert m_∞.

3.1.2 Initiale Platzierung der Stelleninstanzen

Damit jedem Knoten v im Graphen G_{ROI} eine Menge möglicher Platzierungen zugeordnet werden kann, wird die Menge $\mathcal{A}_v$ definiert, die alle für den Knoten v wählbaren Platzierungen enthält. Dabei ist zur Initialisierung diese Menge $\mathcal{A}_v$ leer. Um die einzelnen n Platzierungen aus der Menge $\mathcal{A}_v$ eindeutig identifizieren zu können, wird folgende Notation eingeführt:

$$\mathcal{A}_v = \{\Phi_v^1, \ldots, \Phi_v^n\}$$

Um aus der Menge $\mathcal{A}_v$ eine Platzierung auswählen zu können, wird zusätzlich für den Algorithmus die Funktion sel definiert:

$$\mathrm{sel}_i(\mathcal{A}_v) = \Phi_v^i$$

Damit die Platzierung der Arrays als initiale Platzierung in den Stelleninstanzgraphen G_{ROI} übernommen werden kann, wird jedem Knoten v, der einen Schreibzugriff `write_A` auf ein Array `A` repräsentiert, die initiale Platzierung Φ_v zugeordnet, die sich aus der Platzierung des Arrays `A` wie folgt berechnet:

$$\Phi_v = \tilde{\Phi}_{\mathtt{A}} \circ \varphi$$

Dabei ist φ die Zugriffsfunktion, mit der das Array `A` im Knoten v referenziert wird. Diese Platzierung Φ_v wird in die Menge $\mathcal{A}_v$ eingefügt. Hier ist zu beachten, dass für jeden Knoten, der einen Schreibzugriff auf ein Array repräsentiert, keine weiteren Platzierungen eingefügt werden, weil ein Schreibzugriff auf ein Arrayelement nur auf dem Prozessor ausgeführt werden kann, der auch das entsprechende Arrayelement besitzt. Algorithmus 2 beschreibt die Bestimmung der initialen Platzierungen.

3.1.3 Beispiel

Als Beispiel soll die Platzierung der Arrays des Programms 2.1 durch lineare Relationen dargestellt werden. Da im Programm nur ein eindimensionales Template verwendet wird, ist die Anzahl der benötigten Prozessordimensionen $n_{pDim} = 1$. Zudem besitzt das Programm

`InitPlatzierung`$(G, \tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}, V_{\text{write}}, \mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n})$

Eingabe
- Der reduzierte Stelleninstanzgraph $G = (V, E)$ mit $V = \{v_1, \ldots v_n\}$.
- Die Platzierungen $\tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}$ aller im Programm verwendeten Arrays $\mathtt{A}_1, \ldots, \mathtt{A}_m$ als lineare Relationen.
- Für jeden Knoten $v_i \in V$ die Menge der möglichen Platzierungen $\mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n}$.

Ausgabe
- Die Menge $V_{\text{write}} \subseteq V$ der Knoten aus V, die einen Schreibzugriff auf ein Array darstellen.
- Für jeden Knoten $v_i \in V$ die Menge der möglichen Platzierungen.

Algorithmus

1. Setze für alle Knoten $v_i \in V$ die Menge ihrer möglichen Platzierungen $\mathcal{A}_{v_i} = \emptyset$.
2. Bestimme die Menge der Schreibzugriffe:
$$V_{\text{write}} = \{v \mid v \in V \wedge v \text{ repräsentiert } \mathtt{write\_A}_j \text{ mit } j \in \{1, \ldots, m\}\}$$
3. Für jeden Knoten $v \in V_{\text{write}}$:
   (a) Berechne für den Knoten v, der dem Programmcode $\mathtt{A}(\varphi_{\mathtt{A}}(\mathbf{i}))$ entspricht, die initiale Platzierung Φ_v:
$$\Phi_v = \tilde{\Phi}_{\mathtt{A}} \circ \varphi_{\mathtt{A}}$$
   (b) Setze $\mathcal{A}_v = \{\Phi_v\}$.

Algorithmus 2: Bestimmung der initialen Platzierungen

nur einen Strukturparameter neben m_∞ und m_c, weshalb der Raum der virtuellen Prozessoren $\mathcal{P} = \mathbb{Z}^4$ ist. Aus der `ALIGN`-Direktive des Arrays `A` ergeben sich für die Platzierung $\tilde{\Phi}_{\mathtt{A}} = (\tilde{\Phi}_{\mathtt{A}_{OI}}, \tilde{\Phi}_{\mathtt{A}_P})$ folgende zwei Abbildungen:

$$\begin{aligned} \tilde{\Phi}_{\mathtt{A}_{OI}}(i, N, m_\infty, m_c) &= (i, N, m_\infty, m_c) \\ \tilde{\Phi}_{\mathtt{A}_P}(p_1, N, m_\infty, m_c) &= (p_1, N, m_\infty, m_c) \end{aligned}$$

Die Platzierungen der Arrays `B` und `C` ist mit der von `A` identisch, da sie die gleiche `ALIGN`-Direktive besitzen.
Für die Platzierung $\tilde{\Phi}_{\mathtt{D}}$ des Arrays `D`, das auf dem Template `T` repliziert ist, ergibt sich folgendes Abbildungspaar:

$$\begin{aligned} \tilde{\Phi}_{\mathtt{A}_{OI}}(i, N, m_\infty, m_c) &= (0, N, m_\infty, m_c) \\ \tilde{\Phi}_{\mathtt{A}_P}(p_1, N, m_\infty, m_c) &= (0, N, m_\infty, m_c) \end{aligned}$$

3.2 Erzeugung der möglichen Platzierungen

In diesem Abschnitt soll der erste Schritt der Platzierungsmethode dargelegt werden, mit dem für jeden Knoten die möglichen Platzierungen erzeugt werden. Dazu soll in Abschnitt 3.2.1 mit

einer Motivation begonnen werden, die kurz die Grundidee zeigt, wie mögliche Platzierungen für eine Stelleninstanzmenge bestimmt werden sollen. Im darauffolgenden Abschnitt 3.2.2 wird dargelegt, welche Probleme bei der Propagierung auftreten.
Im abschließenden Abschnitt 3.2.5 werden die in den vorhergehenden Abschnitten 3.2.3 und 3.2.4 beschriebenen Algorithmen zu einem Algorithmus zusammengefasst, der aus den initialen Platzierungen für jeden Knoten die Menge der möglichen Platzierungen bestimmt.

3.2.1 Motivation

Aus einem interessanten Knoten, der keine initale Platzierung besitzt, wird zur Codegenerierung ein Hilfsarray angelegt, das die durch diesen Knoten berechneten Werte zwischenspeichert. Alle Zugriffe auf das Hilfsarray sollen möglichst effizient durchgeführt werden. Dazu sind sowohl die Platzierungen der Arrays, die zur Initialsierung des Hilfsarray benötigt werden, als auch die Platzierung aller Arrays in den Statements zu berücksichtigen, die einen Lesezugriff auf das Hilfsarray besitzen. Folgender Schleifensatz wurde dazu konstruiert, um die zugrunde liegende Idee zu verdeutlichen:

Programm 3.1

```
DO i=1,N
   X(i) = SQRT(C(i))
   DO j=1,M
      Y(j,i) = A(i) * B(i) + SQRT(C(i))
      Z(j,i) = SIN(A(i) * B(i))
   END DO
END DO
```

Der Schleifensatz des Programms 3.1 besitzt drei Berechnungen, die mehrfach verwendet werden. Dies sind die Multiplikation `A(i) * B(i)`, die Berechnung der Quadratwurzel `SQRT(C(i))` und die Berechnung des Sinus `SIN(A(i) * B(i))`.
Soll eine Platzierung für die Multiplikation `A(i) * B(i)` gefunden werden, dann sollte die Platzierung folgender Arrays berücksichtigt werden:

1. Die Arrays `A` und `B`: Sie werden zur Berechnung der Multiplikation benötigt.

2. Die Arrays `Y` und `Z`: Sie verwenden die Werte der Multiplikation.

3. Das Array `C`: In die Berechnung der Werte von `Y` gehen auch die Werte von `SQRT(C(i))` ein. Somit könnte es vorteilhaft sein, auch diese Platzierung zu berücksichtigen.

4. Das Array `X`: Da die Berechnung von `SQRT(C(i))` auch von `X` gelesen wird, sollte die Verteilung von `X` berücksichtigt werden.

Wie in Abbildung 7 des durch die Anwendung von LCCP entstandenen kondensierten Stelleninstanzgraphen ersichtlich ist, repräsentiert der Knoten v_{11} die Multiplikation `A(i) * B(i)`, deren Werte mehrfach verwendet werden. Um die Platzierung des Arrays `A` (Knoten v_{13}) für die Platzierung des Knotens v_{11} zu berücksichtigen, kann die Platzierung von `A` entlang der Kante (v_{13}, v_{11}, h) propagiert werden, wodurch für den Knoten v_{11} eine neue Platzierung erzeugt wird, die folgende Eigenschaft besitzt: Für die Berechnung der Multiplikation wird keine

Kommunikation für den Zugriff auf das Array `A` erzeugt, da das Ergebnis der Multiplikation `A(i) * B(i)` dem gleichen Template-Element wie `A(i)` zugeordnet ist. In diesem Fall wird die Platzierung von der Quelle zum Ziel der Abhängigkeit propagiert.
Wenn die Platzierung des Arrays `Y` (Knoten v_6) zum Knoten v_{11} propagiert werden soll, dann muss diese Propagierung in entgegengesetzter Richtung des Pfades $v_6 \to v_8 \to v_{11}$, also von v_{11} nach v_6 erfolgen. In diesem Fall wird die Platzierung des Ziels zur Quelle der Abhängigkeit propagiert.
In Abbildung 7 geben die gestrichelten Linien die Wege im Graphen G_{ROI} des Programms an, entlang derer die Platzierungen der Arrays zum Knoten v_{11} propagiert werden sollen.

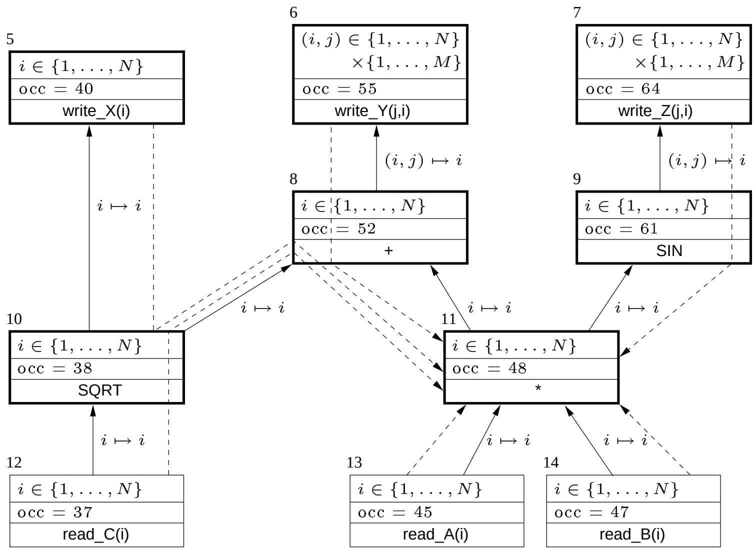


Abbildung 7: Der reduzierte kondensierte Stelleninstanzgraph des Programms aus Beispiel 3.1

3.2.2 Partitionierung

Wird eine Platzierung Φ_v entlang einer Kante $e = (v, v', h)$ propagiert, dann ist zu beachten, dass eine Abhängigkeit zwischen zwei Stelleninstanzmengen nicht unbedingt für alle Elemente dieser Mengen gilt, weshalb für die h-Transformation $h = (D_h, f_h)$ der Kante e folgende zwei Fälle auftreten können:

1. $D_h \subset O_{v'}$: Da die Definitionsmenge D_h der h-Transformation eine echte Teilmenge von $O_{v'}$ ist, gilt f_h nur für einen Teil der Stelleninstanzmenge $O_{v'}$ des Knotens, zu dem die Abhängigkeit h verläuft.

2. $f_h(D_h) \subset O_v$: Weil das Bild der Definitionsmenge D_h der h-Transformation eine echte Teilmenge von O_v ist, gilt f_h nur für einen Teil der Stelleninstanzmenge O_v des Knotens, von dem die Abhängigkeit h ausgeht.

Diese beiden Fälle führen dazu, dass die neue Platzierung $\Phi_{v'}$ des Knotens v', die durch Propagierung der Platzierung Φ_v entlang der Kante e mit

$$\Phi_{v'} = \Phi_v \circ f_h$$

entstanden ist, nicht für die ganze Menge $O_{v'}$ sondern nur für die Teilmenge $D_h \subset O_{v'}$ gilt. Wird diese Einschränkung ignoriert, wird also diese Platzierung $\Phi_{v'}$ für die gesamte Menge $O_{v'}$ verwendet, hat dies zur Folge, dass die Platzierung $\Phi_{v'}(O_{v'})$ in der Regel mehr Template-Elemente referenziert, als vom Template `T` bereitgestellt werden, was wie folgt notiert werden kann:

$$\pi_{\mathtt{T}}(\Phi_{v'}(O_{v'})) \nsubseteq \mathcal{D}_{\mathtt{T}}$$

Dabei ist $\pi_{\mathtt{T}}$ die Projektion des Raums $\mathcal{P}$ auf die Dimensionen, die dem Template `T` zugeordnet sind. Die Möglichkeit, dass eine Platzierung mehr Template-Elemente referenziert, als vom Template bereitgestellt werden, ist in HPF nicht gegeben [Hig93], weshalb $\Phi_{v'}$ keine gültige Platzierung für den Knoten v' ist.

Die gleiche Problematik tritt auf, wenn eine Platzierung $\Phi_{v'}$ vom Ziel v' zur Quelle v entlang der Kante e mit

$$\Phi_v = \Phi_{v'} \circ f_h^{-1}$$

propagiert wird, da auch hier die Platzierung Φ_v nur auf die Teilmenge $f_h(D_h) \subset O_v$ beschränkt ist. Deshalb kann auch hier die neu berechnete Platzierung die Eigenschaft

$$\pi_{\mathtt{T}}(\Phi_v(O_v)) \nsubseteq \mathcal{D}_{\mathtt{T}}$$

besitzen, so dass Φ_v keine gültige Platzierung darstellt.

Als Beispiel für diese Problematik betrachtet man die beiden Kanten e_1, e_2 zwischen den Knoten v_{11} und v_9 im kondensierten reduzierten Stelleninstanzgraphen des Programms 2.1 in Abbildung 5. Die beiden Kanten e_1 und e_2 besitzen folgende Formen:

$$\begin{aligned} e_1 &= (v_{11}, v_9, h_1 = (D_{h_1} = O_9, f_{h_1}(i) = i)) \\ e_2 &= (v_{11}, v_9, h_2 = (D_{h_2} = O_9, f_{h_2}(i) = i - 1)) \end{aligned}$$

Soll eine Platzierung des Knotens v_9 über diese beiden Kanten e_1 und e_2 zum Knoten v_{11} propagiert werden, dann wird für Knoten v_{11} eine Platzierung erzeugt, die außerhalb des Datenraums des Templates `T` liegt.

Da der Knoten v_9 selbst keine initiale Platzierung besitzt, wird eine mögliche Platzierung für v_9 dadurch erzeugt, dass die Platzierung Φ_{v_5} (in Abschnitt 3.1.3 angegeben) entgegengesetzt zum Weg $v_9 \to v_8 \to v_6 \to v_5$, also von $v_5 \to v_9$ propagiert wird. Da die zugehörigen h-Transformationen entlang dem Pfad die Identität darstellen, ergibt sich als mögliche Platzierung des Knotens v_9:

$$\Phi^1_{v_9} = \Phi_{v_5}$$

Die Platzierung $\Phi^1_{v_9}$ richtet die Menge O_{11} der Stelleninstanzen auf das Template `T` mit dem Datenraum $\mathcal{D}_{\mathtt{T}} = \{1, \ldots, N\}$ aus. Wird die Platzierung $\Phi^1_{v_9}$ zum Knoten v_{11} propagiert, dann ergeben sich die beiden Platzierungen $\Phi^1_{v_{11}}$ aus der Propagierung entlang der Kante e_1 und $\Phi^2_{v_{11}}$ aus der Propagierung entlang der Kante e_2, wie folgt:

$$\Phi^1_{v_{11}}(i) = (\Phi_{v_9} \circ f_{h_1}^{-1})(i) = i \tag{4}$$

$$\Phi^2_{v_{11}}(i) = (\Phi_{v_9} \circ f_{h_2}^{-1})(i) = i + 1 \tag{5}$$

Betrachtet man für beide Platzierungen die Menge der Template-Elemente, die für die Platzierung verwendet werden müssen, dann ergeben sich folgende Mengen:

$$\pi_{\mathtt{T}}(\Phi^1_{v_{11}}(O_{11})) = \{2, \ldots, N\} \subseteq \mathcal{D}_{\mathtt{T}} \tag{6}$$

$$\pi_{\mathtt{T}}(\Phi^2_{v_{11}}(O_{11})) = \{3, \ldots, N + 1\} \nsubseteq \mathcal{D}_{\mathtt{T}} \tag{7}$$

Wie aus der Gleichung (7) ersichtlich, stellt die Menge der von $\Phi^2_{v_{11}}$ referenzierten Template-Elemente keine Teilmenge des Datenraums des Templates T dar. Somit kann bei der einfachen Propagierung entlang der Kanten im Graphen eine neue nicht gültige Platzierung erzeugt werden, was zu vermeiden ist.
Die Ursache der Erzeugung nicht gültiger Platzierungen liegt daran, dass das Bild von D_{h_2} unter der h-Transformation f_{h_2} die Indexmenge $\{1,\ldots,N-1\}$ besitzt, weshalb die neue Platzierung $\Phi^2_{v_{11}}$ nur für die Teilmenge $f_{h_2}(D_{h_2})$ von O_{11} gilt, was aus folgender Gleichung ersichtlich ist:

$$\pi_{\mathtt{T}}(\Phi^2_{v_{11}}(f_{h_2}(D_{h_2}))) = \{3,\ldots,N\} \subseteq \mathcal{D}_{\mathtt{T}}$$

Aus diesem Grund muss bei der Propagierung der Platzierung des Knotens v entlang der Kante $e = (v, v', h = (D_h, f_h))$ nach v' gelten, dass $O_{v'} = f_h(D_h)$ ist. Daher wäre es eine gute Idee, durch eine Partitionierung des reduzierterten Stelleninstanzgraphen einen neuen Graphen zu erzeugen, der diese Eigenschaft besitzt.
Dazu wird der Knoten v_{11} so partitioniert, dass für jede Überlappung der Bilder der h-Transformationen eine Partition erzeugt wird, was im Folgenden gezeigt werden soll. Für die Bilder der beiden Abhängigkeiten gilt:

$$\begin{aligned} \mathrm{Idx}(f_{h_1}(D_{h_1})) &= \{2,\ldots,N\} & (8)\\ \mathrm{Idx}(f_{h_2}(D_{h_2})) &= \{1,\ldots,N-1\} & (9) \end{aligned}$$

Partitioniert man die Stelleninstanzmenge O_{11} entsprechend der Überlappungen der beiden Mengen aus Gleichung (8) und (9), dann erhält man folgende drei Partitionen:

$$\mathrm{Idx}(O_{25}) = \{1\} \qquad \mathrm{Idx}(O_{26}) = \{2,\ldots,N-1\} \qquad \mathrm{Idx}(O_{27}) = \{N\}$$

Wird im reduzierten Stelleninstanzgraphen der Knoten v_{11} durch seine Partitionen v_{25}, v_{26} und v_{27} ersetzt und werden die h-Transformationen entsprechend umgeschrieben, dann ergibt sich der partitionierte reduzierte Stelleninstanzgraph, der in Abbildung 8 dargestellt ist.
Die neuen Kanten e_3, e_4, e_5 und e_6, die von den Knoten v_{25}, v_{26} und v_{27} zum Knoten v_9 führen, sind in Tabelle 3 zusammengefasst, wobei für jede Kante sowohl die Definitionsmenge als auch das Bild der Abhängigkeit angegeben sind.

Knoten	Kante	Definitionsmenge von h_i	Bild der Definitionsmenge
v_{25}	$e_3 = (v_{25}, v_9, h_3)$	$\mathrm{Idx}(D_{h_3}) = \{2\}$	$\mathrm{Idx}(f_{h_3}(D_{h_3})) = \{1\}$
v_{26}	$e_4 = (v_{26}, v_9, h_4)$	$\mathrm{Idx}(D_{h_4}) = \{2,\ldots,N-1\}$	$\mathrm{Idx}(f_{h_4}(D_{h_4})) = \{2,\ldots,N-1\}$
	$e_5 = (v_{26}, v_9, h_5)$	$\mathrm{Idx}(D_{h_5}) = \{3,\ldots,N\}$	$\mathrm{Idx}(f_{h_5}(D_{h_5})) = \{2,\ldots,N-1\}$
v_{27}	$e_6 = (v_{27}, v_9, h_6)$	$\mathrm{Idx}(D_{h_6}) = \{N\}$	$\mathrm{Idx}(f_{h_6}(D_{h_6})) = \{N\}$

Tabelle 3: Die h-Transformationen nach der Partitionierung des Knotens v_{11}

Als mögliche Platzierungen für den Knoten v_{26} entstehen die gleichen zwei Platzierungen wie für den Knoten v_{11}, da sich bei der Partitionierung die Funktionsvorschrift der h-Transformationen nicht geändert haben. So besitzt der Knoten v_{26} folgende zwei Platzierungsmöglichkeiten:

$$\begin{aligned} \Phi^1_{v_{26}}(i) &= (\Phi_{v_9} \circ f^{-1}_{h_4})(i) = i & (10)\\ \Phi^2_{v_{26}}(i) &= (\Phi_{v_9} \circ f^{-1}_{h_5})(i) = i+1 & (11) \end{aligned}$$

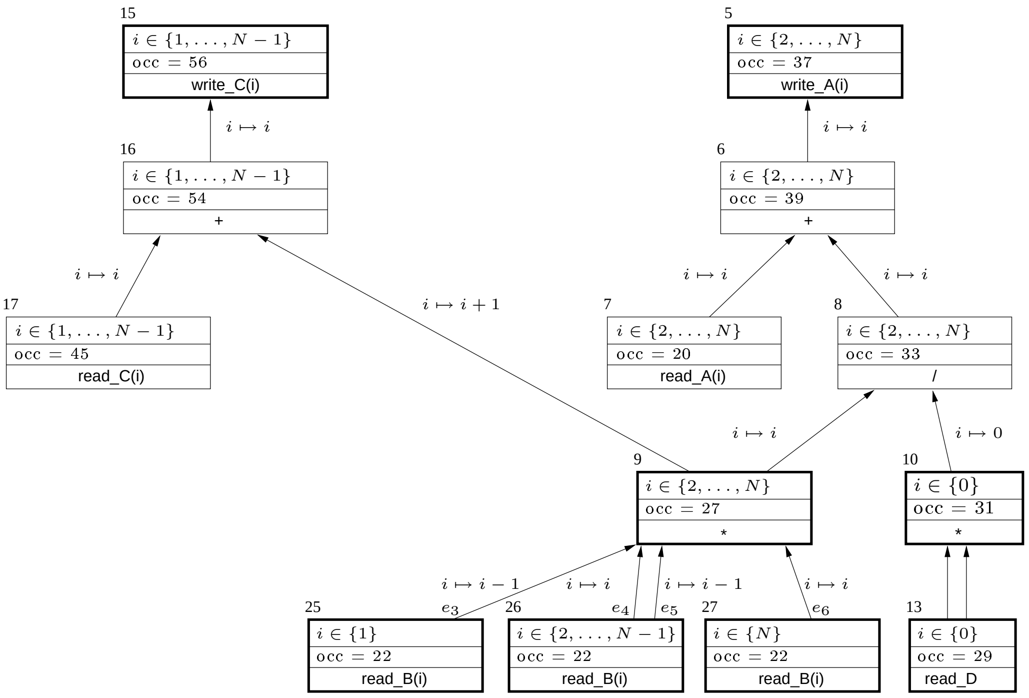


Abbildung 8: Ergebnis der Partitionierung des Knotens v_{11}

Diese beiden Platzierungen referenzieren folgende Template-Elemente:

$$\pi_{\mathtt{T}}(\Phi^1_{v_{26}}(O_{26})) \quad = \quad \{2, \ldots, N-1\} \subseteq \mathcal{D}_{\mathtt{T}} \tag{12}$$

$$\pi_{\mathtt{T}}(\Phi^2_{v_{26}}(O_{26})) \quad = \quad \{3, \ldots, N\} \subseteq \mathcal{D}_{\mathtt{T}} \tag{13}$$

Wie man in den beiden Gleichungen (12) und (13) sehen kann, referenzieren die beiden möglichen Platzierungen der Stelleninstanzmenge O_{26} nur Template-Elemente, die im Datenraum $\mathcal{D}_{\mathtt{T}}$ des Templates $\mathtt{T}$ liegen, so dass beide Platzierungen gültige Platzierungen darstellen.
Wie aus diesem Beispiel zu ersehen ist, garantiert eine entsprechende Partitionierung der Knoten des reduzierten Stelleninstanzgraphen, dass nur gültige Platzierungen erzeugt werden. Es muss also für einen Weg w von v_i nach v_j im Stelleninstanzgraph garantiert werden, dass die neu erzeugte Platzierung für den Knoten v_j auf der gesamten Stelleninstanzmenge O_{v_j} und nicht nur auf einer Teilmenge von O_{v_j} gilt.
Um die Platzierungen entlang beliebiger Wege im Graphen propagieren zu können, wie in Abschnitt 3.2.1 gefordert, wäre es ideal, durch Partitionierung zu erreichen, dass jede Abhängigkeit, die durch die Kante $e = (v, v', h = (D_h, f_h))$ repräsentiert ist, sowohl auf der gesamten Stelleninstanzmenge des Zielknotens als auch auf der gesamten Stelleninstanzmenge des Quellknotens gilt und der reduzierte Stelleninstanzgraph $G_{ROI} = (V, E)$ folgende zwei Eigenschaften besitzt:

$$\forall e \in E, e = (v, v', h = (D_h, f_h)): \qquad O_{v'} = D_h \tag{14}$$

$$\forall e \in E, e = (v, v', h = (D_h, f_h)): \qquad O_v = f_h(D_h) \tag{15}$$

Doch lässt sich ein reduzierter Stelleninstanzgraph in der Regel nicht so partitionieren, dass die beiden Eigenschaften (14) und (15) gleichzeitig erfüllt sind, da Abhängigkeiten auftreten können, die eine solche Partitionierung verhindern, wie folgendes Beispiel zeigt:

Die genaue Betrachtung der eben durchgeführten Partitionierung des Knotens v_{11} in v_{25}, v_{26} und v_{27} lässt erkennen, dass zwar die Quellen der Abhängigkeiten e_3, e_4, e_5 und e_6 mit den Stelleninstanzmengen der Knoten übereinstimmen, von denen die Abhängigkeiten ausgehen, die Ziele aber nicht mehr mit der Stelleninstanzmenge O_9 identisch sind (vgl. Tabelle 3), was der Eigenschaft (14) widerspricht.
Man beachte, dass vor der Partitionierung der reduzierte Stelleninstanzgraph in Abbildung 4 die Eigenschaft (14) besitzt, da die Ziele aller Abhängigkeiten mit den Stelleninstanzmengen der entsprechenden Knoten übereinstimmen und dass nach dieser Partitionierung dadurch die Eigenschaft (14) verloren geht, dass der reduzierte Stelleninstanzgraph die Eigenschaft (15) gewinnt.
Um die Eigenschaft (14) wieder zu erhalten, ist der Knoten v_9 entsprechend der Überlappung der Definitionsmengen der neu entstandenen Abhängigkeiten e_7 und e_8 in folgende drei Mengen zu partitionieren:

$$\mathrm{Idx}(O_{28}) = \{1\} \qquad \mathrm{Idx}(O_{29}) = \{2, \ldots, N-1\} \qquad \mathrm{Idx}(O_{30}) = \{N\}$$

Diese Partitionierung des Knotens v_9 ist als Ausschnitt des reduzierten Stelleninstanzgraphen in Abbildung 9 gezeigt.

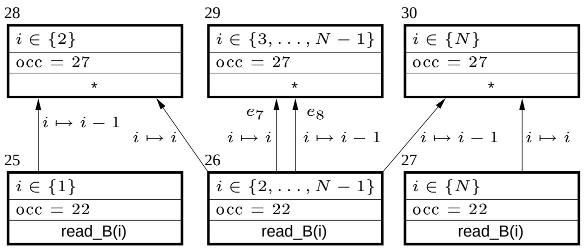


Abbildung 9: Auschnitt aus dem reduzierten Stelleninstanzgraphen

Nach der Partitionierung des Knotens v_9 in die drei Knoten v_{28}, v_{29} und v_{30} besitzt der Graph immer noch nicht die Eigenschaft (14) für den gesamten Graphen, da es von den drei neuen Knoten v_{28}, v_{29} und v_{30} jeweils eine Kante zum Knoten v_8 gibt, die auf unterschiedlichen Partitionen des Knotens v_8 gelten. Das Gleiche trifft auch für den Knoten v_{16} zu. Daher sind auch v_8 und v_{16} zu partitionieren. Diese Partitionierung pflanzt sich entlang der Abhängigkeiten zu den Knoten v_5 und v_{15} fort, um die Eigenschaft (14) im gesamten Graphen zu erhalten.
Betrachtet man nach der eben durchgeführten Partitionierung die beiden Abhängigkeiten $e_7 = (v_{26}, v_{29}, h_7)$ und $e_8 = (v_{26}, v_{29}, h_8)$ zwischen den Knoten v_{29} und v_{26} in Abbildung 9, dann ist zu erkennen, dass die Definitionsmengen der beiden Abhängigkeiten folgende Form haben:

$$\begin{aligned} &\mathrm{Idx}(D_{h_7}) = \{3, \ldots, N-1\} \qquad && \mathrm{Idx}(f_{h_7}(D_{h_7})) = \{3, \ldots, N-1\} \neq \mathrm{Idx}(O_{26}) \\ &\mathrm{Idx}(D_{h_7}) = \{3, \ldots, N-1\} \qquad && \mathrm{Idx}(f_{h_7}(D_{h_7})) = \{2, \ldots, N-2\} \neq \mathrm{Idx}(O_{26}) \end{aligned}$$

Wie man aus den Bildern der Abhängigkeiten erkennen kann, gelten diese nicht mehr auf der gesamten Stelleninstanzmenge des Knotens v_{26}, der die Quelle der beiden Abhängigkeiten darstellt. Daher ist die Eigenschaft (15), die durch die Partitionierung des Knotens v_{11} zu Beginn erreicht wurde, wieder zerstört worden. Somit ist wieder die gleiche Situtation wie vor der Partitionierung eingetreten: Der reduzierte Stelleninstanzgraph besitzt wieder die Eigenschaft (14) aber nicht die Eigenschaft (15).

Auch nach mehrfacher Partitionierung des reduzierten Stelleninstanzgraphen kann der Graph nicht in eine Form transformiert werden, die beide Eigenschaften (14) und (15) gleichzeitig hat, da immer die eine Eigenschaft verloren geht, wenn man die andere durch Partitionierung gewinnt.
Wie aus obigem Beispiel ersichtlich ist, kann im Allgemeinen ein reduzierter Stelleninstanzgraph auch durch mehrfache Partitionierung nicht so partitioniert werden, dass er die beiden Eigenschaften (14) und (15) gleichzeitig besitzt.
Da der reduzierte Stelleninstanzgraph des obigen Beispiels keine Zyklen enthält, ist es durch Partitionierung möglich, dass der Graph nach der Partitionierung eine der beiden Eingeschaften (14) oder (15) besitzt. In einem Graphen mit Zyklen kann durch Partitionierung nicht einmal eine der beiden Eigenschaften erreicht werden.
Aus diesen beiden Gründen wird eine einfachere Methode zur Propagierung gewählt, die zur Berechnung der möglichen Platzierungen eines Knotens weniger initiale Platzierungen berücksichtigt, als in Abschnitt 3.2.1 vorgeschlagen wurde.
Dabei wird wie folgt vorgegangen:

1. Aus dem reduzierten Stelleninstanzgraphen wird eine Version ohne Zyklen erzeugt, indem nur die Flussabhängigkeiten betrachtet werden und von allen Knoten, die einen Schreibzugriff darstellen, die ausgehenden oder die eingehenden Kanten entfernt werden. Da der reduzierte Stelleninstanzgraph nur die Flussabhängigkeiten berücksichtigt, besitzt ein Zyklus immer einen Knoten, der einen Schreibzugriff darstellt, so dass das angegebene Entfernen der Kanten einen zyklenfreien Graph erzeugt.

2. Der zyklenfreie Graph wird so partitioniert, dass der gesamte Graph entweder die Eigenschaft (14) oder die Eigenschaft (15) besitzt.

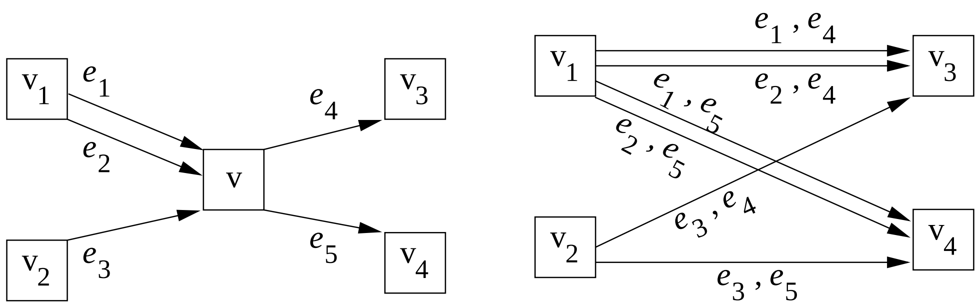
3. Nun kann enlang der Kanten propagiert werden. Besitzt der Graph die Eigenschaft (14), dann werden die initialen Platzierungen auf Wegen in Richtung Quelle zum Ziel propagiert. Besitzt der Graph die Eigenschaft (15), dann werden die initialen Platzierungen auf Wegen in Richtung Ziel zur Quelle propagiert.

3.2.3 Kompaktifizierung

Da lediglich für die interessanten Knoten bei der Codegenerierung Arrays erzeugt werden, benötigen nur sie eine Platzierung. Deshalb genügt es, nur für diese Knoten mögliche Platzierungen zu berechnen. Dies reduziert den Aufwand, da weniger Platzierungen zu berechnen und deshalb weniger Kombinationen von Platzierungen durch das Kostenmodell zu bewerten sind.
Um nur interessante Knoten zu berücksichtigen, wird der reduzierte Stelleninstanzgraph G_{ROI} des Programms in den kompaktifizierten Graphen $G_{ROI_{comp}}$ überführt, der dann nur noch die interessanten Knoten enthält. Da die Abhängigkeiten zwischen den interessanten Knoten erhalten bleiben müssen, ist jeder Weg über einen nicht interessanten Knoten im kompaktifizierten Graphen zu berücksichtigen.
Betrachtet man den Ausschnitt eines Graphen in Abbildung 10(a), dann sind folgende Wege im Graphen möglich:

$$\begin{array}{ll} v_1 \xrightarrow{e_1} v \xrightarrow{e_4} v_3 & \qquad v_1 \xrightarrow{e_1} v \xrightarrow{e_5} v_4 \\ v_1 \xrightarrow{e_2} v \xrightarrow{e_4} v_3 & \qquad v_1 \xrightarrow{e_2} v \xrightarrow{e_5} v_4 \\ v_2 \xrightarrow{e_3} v \xrightarrow{e_4} v_3 & \qquad v_2 \xrightarrow{e_3} v \xrightarrow{e_5} v_4 \end{array}$$

Wird z.B. der Knoten v entfernt, dann ist für jeden dieser Wege eine neue Kante in den Graphen einzufügen. Die sich ergebenen neuen Kanten sind in Abbildung 10(b) veranschaulicht, wobei die Markierung e_i, e_j angibt, dass die entsprechende Kante aus dem Weg $\xrightarrow{e_i} v \xrightarrow{e_j}$ entstanden ist. Somit sind nach der Entfernung des Knotens v alle möglichen Wege über diesen



(a) Graph vor Kompaktifizierung (b) Graph nach Kompaktifizierung

Abbildung 10: Ausschnitt eines kompaktifizierten Graphen

Knoten im kompaktifizierten Graphen erhalten geblieben.
Aus diesem Grund wird für jeden Weg der Form $v' \rightarrow v \rightarrow v''$ über einen nicht interessanten Knoten v mit $e_{\text{in}} = (v', v, h_{\text{in}})$ und $e_{\text{out}} = (v, v'', h_{\text{out}})$ zu einer neuen Kante $v' \rightarrow v''$ transformiert, die die Abhängigkeit durch die Funktion f beschreibt, die wie folgt bestimmt wird:

$$f = f_{h_{\text{in}}} \circ f_{h_{\text{out}}}$$

Da nach der Kompaktifizierung die Definitionsmengen der h-Transformationen nicht mehr benötigt werden, brauchen die Definitionsmengen bei der Komposition nicht berücksichtigt werden, und so besteht im kompaktifizierten Graphen die Markierung einer Kante nur noch aus der Funktionsvorschrift ohne die zugehörige Definitionsmenge.
Algorithmus 3 beschreibt die Erzeugung eines kompaktifizierten Graphen $G_{ROI_{comp}}$ aus einem reduzierten Stelleninstanzgraphen G_{ROI}. Dabei wird jeder Weg über einen nicht interessanten Knoten dadurch beibehalten, dass aus jeder Kombination ein- und ausgehender Kanten eines nicht interessanten Knotens im G_{ROI} eine neue Kante in den kompaktifizierten Graphen eingefügt wird.
Als Beispiel der Kompaktifizierung dient der partitionierte, reduzierte Stelleninstanzgraph des Programms 2.1 aus Abbildung 8. Die kompaktifizierte Version des Graphen ist in Abbildung 11 gezeigt. Dabei sind alle nicht interessanten Knoten entfernt und die Wege über sie als neue Kanten in den kompaktifizierten Graphen übernommen.

3.2.4 Die Propagierung

Algorithmus 4 beschreibt die Propagierung der initialen Platzierungen zu den interessanten Knoten ohne initiale Platzierung. Dabei wird vorausgesetzt, dass der gesamte Graph bereits so partitioniert ist, dass die Propagierung in eine Richtung durchgeführt werden kann, so dass nur gültige Platzierungen erzeugt werden. Die Richtung der Propagierung ist durch das Argument *Richtung* gegeben und kann folgende zwei Werte annehmen:

`Kompaktifizieren`(G, G_{comp})

Eingabe — Der reduzierte Stelleninstanzgraph $G = (V, E)$.

Ausgabe — Der kompaktifizierte Stelleninstanzgraph $G_{comp} = (V_{comp}, E_{comp})$, wobei jede Kante $e \in E_{comp}$ mit $e = (v, v', f)$ als Markierung eine lineare Funktion f besitzt.

Alogrithmus

1. Setze $V_{comp} = V$ und $E_{comp} = \emptyset$.
2. Erzeuge für jede Kante $e \in E$ mit $e = (v, v', h = (D_h, f_h))$ die neue Kante $e' = (v, v', f_h)$ und füge sie in E_{comp} ein.
3. Bestimme für jeden nicht interessanten Knoten $v \in V_{comp}$ die Menge $\text{in}(v) \subseteq E_{comp}$ der in v eingehenden Kanten und die Menge $\text{out}(v) \subseteq E_{comp}$ der aus v ausgehenden Kanten.
   (a) Berechne für jede Kombination der Kanten aus $e_{\text{in}} \in \text{in}(v)$ und $e_{\text{out}} \in \text{out}(v)$ mit $e_{\text{in}} = (v_1, v, f_{\text{in}})$ und $e_{\text{out}} = (v, v_2, f_{\text{out}})$ die Abbildung

$$f = f_{\text{in}} \circ f_{\text{out}}$$

   und konstruiere daraus die neue Kante $\hat{e}$, wie folgt:

$$\hat{e} = (v_1, v_2, f)$$

   Füge $\hat{e}$ in die Menge der Kanten E_{comp} ein.
   (b) Entferne die Mengen $\text{in}(v)$ und $\text{out}(v)$ aus E_{comp}.
   (c) Entferne v aus der Menge V_{comp}.

Algorithmus 3: Kompaktifizierung des reduzierten Stelleninstanzgraphen G_{ROI}

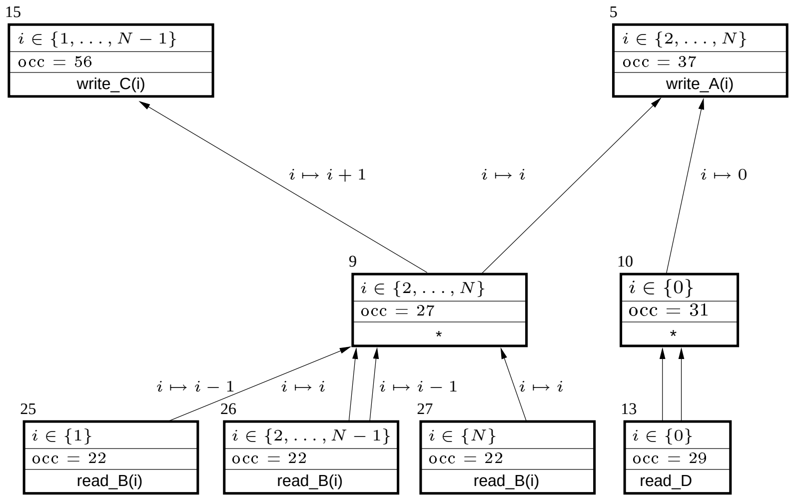


Abbildung 11: Der kompaktifizierte, partitionierte, reduzierte Stelleninstanzgraph $G_{ROI_{comp}}$ des Programms 2.1

`Propagierung`$(G, V_{\text{write}}, \textit{Richtung}, \Phi_{v_1}, \ldots, \Phi_{v_n}))$

Eingabe

- Der reduzierte Stelleninstanzgraph G mit $V = \{v_1, \ldots, v_n\}$.
- Für jeden Knoten $v_i \in V$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{v_i}$.
- Der Parameter *Richtung*, der einen der beiden Werte Quelle→Ziel und Ziel→Quelle annehmen kann.

Ausgabe

- Für jeden interessanten Knoten $v_i \in V$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{v_i}$.

Algorithmus

1. Führe `Kompaktifizieren`(G, G_{comp}) (Algorithmus 3) aus, um den kompaktifizierten Graphen von G zu erhalten.
2. Führe `TransitiveHülle`(G_{comp}, G') (Algorithmus 5) aus, um die transitive Hülle G' des kompaktifizierten Graphen G_{comp} zu erzeugen. Dabei ist $G' = (V', E')$ mit $V' = \{v'_1, \ldots, v'_m\}$.
3. Propagiere für jeden Knoten $v_i \in V_{\text{write}}$ die Platzierungen $\Phi^i_{v_i} \in \mathcal{A}_{v_i}$ zu allen Knoten $v'_j \in V' - V_{\text{write}}$ durch folgende Vorschrift:

Ist *Richtung* = Quelle→Ziel gesetzt, dann
erzeuge für alle Kanten $e \in E'$ mit $e = (v_i, v_j, f)$ die möglichen Platzierungen für den Knoten v_j wie folgt:
$\Phi_{v_j} = \Phi_{v_i} \circ f$
Füge Φ_{v_j} in $\mathcal{A}_{v_j}$ ein.
Ist *Richtung* = Ziel→Quelle gesetzt, dann
erzeuge für alle Kanten $e \in E'$ mit $e = (v_j, v_i, f)$ die möglichen Platzierungen für den Knoten v_j wie folgt:
$\Phi_{v_j} = \Phi_{v_i} \circ f^{-1}$
Füge Φ_{v_j} in $\mathcal{A}_{v_j}$ ein.

Algorithmus 4: Der Algorithmus zur Bestimmung der möglichen Platzierungen für interessante Stelleninstanzen

Quelle→Ziel Besitzt das Argument *Richtung* diesen Wert, dann werden die initialen Platzierungen in Richtung Quelle zum Ziel propagiert. Um dabei nur gültige Platzierungen zu erzeugen, muss der gesamte reduzierte Stelleninstanzgraph die Eigenschaft (14) aus Abschnitt 3.2.2 besitzen.

Ziel→Quelle Nimmt das Argument *Richtung* diesen Wert an, dann werden die initialen Platzierungen in Richtung Ziel zur Quelle propagiert. Dabei muss der gesamte Graph die Eigenschaft (15) aus Abschnitt 3.2.2 besitzen, damit nur gültige Platzierungen erzeugt werden.

Bevor die initialen Platzierungen zu den interessanten Knoten ohne Platzierungen propagiert werden können, wird zuerst der Graph kompaktifiziert, da nur die interessanten Knoten, wie in Abschnitt 3.2.3 ausgeführt, eine Platzierung erhalten. Durch diese Verkleinerung des Graphen wird der Aufwand der nachfolgenden Berechnung der transitiven Hülle verringert. Anschließend wird auf dem kompaktifizierten Graphen $G_{comp} = (V_{comp}, E_{comp})$ die transitive Hülle entsprechend Algorithmus 5 gebildet, der dem Algorithmus von Floyd-Warshall entspricht [Sed03]. Dadurch wird für jeden Weg w zwischen zwei Knoten v und v' im kompaktifizierten Graphen eine Kante $e = (v, v', f)$ erzeugt, die diesem Weg entspricht. Soll anschließend die initiale Platzierung Φ_v vom Knoten v zu allen anderen interessanten Knoten ohne initiale Platzierung über alle im Graphen möglichen Wege gegangen werden, muss die Platzierung Φ_v nur entlang aller Kanten propagiert werden, die den Knoten v mit den anderen Knoten verbinden. Im dritten Schritt des Algorithmus 4 wird die Propagierung dann abhängig vom Argument *Richtung* durchgeführt.

`TransitiveHülle`(G, G')

Eingabe Der kompaktifizierte reduzierte Stelleninstanzgraph $G = (V, E)$.

Ausgabe Die transitive Hülle $G' = (V', E')$ des Graphen G.

Algorithmus

Setze $V' = V$ und $E' = E$.
Für jeden Knoten $v_1 \in V'$:
    Für jeden Knoten $v_2 \in V'$:
        Für jeden Knoten $v_3 \in V'$:
            Für alle Kanten $e_1, e_2 \in E'$ mit $e_1 = (v_1, v_2, f_1)$ und $e_2 = (v_2, v_3, f_2)$:
                Bestimme die neue Kante
                $e = (v_1, v_3, f = f_1 \circ f_2)$
                und füge sie in die Menge E' ein.

Algorithmus 5: Der Algorithmus der transitiven Hülle

Nachdem der Algorithmus zur Propagierung der Platzierungen in eine Richtung gezeigt wurde, folgt die Beschreibung des Algorithmus 6, der für eine Richtung die Partitionierung und anschließend die Propagierung der Platzierungen durchführt.
Wie bereits in Abschnitt 3.2.2 erwähnt, darf der reduzierte Stelleninstanzgraph G sowohl zur Partitionierung als auch zur darauf folgenden Propagierung keine Zyklen enthalten. Um dies

zu erreichen, werden abhängig vom Wert des Arguments *Richtung* folgende Kanten aus dem Graphen entfernt:

Quelle→Ziel Es werden alle Kanten aus dem Graphen entfernt, die in einen Knoten v eingehen, der einen Schreibzugriff darstellt.

Ziel→Quelle Es werden alle Kanten aus dem Graphen entfernt, die von einem Knoten v ausgehen, der einen Schreibzugriff darstellt.

Die Kanten müssen am Ende des Algorithmus wieder in den partitionierten Graphen eingefügt werden, so dass diese Information nicht verloren geht. Sonst wäre es nicht möglich, den Algorithmus mehrere Male hintereinander mit verschiedenen Werten des Arguments *Richtung* auf einen reduzierten Stelleninstanzgraphen anzuwenden.
Nachdem die Kanten aus dem Graphen entfernt wurden, enthält er keinen Zyklus mehr, so dass er entsprechend dem Argument *Richtung* partitioniert werden kann. Danach wird entsprechend Algorithmus 4 die initialen Platzierungen zu den Knoten ohne initiale Platzierung propagiert. Dadurch, dass vor der Propagierung der Platzierungen die Knoten des Graphen partitioniert werden, wird garantiert, dass nur gültige Platzierungen erzeugt werden.

3.2.5 Algorithmus zur Berechnung aller möglichen Platzierungen

Nachdem im vorhergehenden Abschnitt die Propagierung der Platzierung in eine Richtung beschrieben wurde, kann abschließend Algorithmus 7 angegeben werden, der durch Propagierung die möglichen Platzierungen für einen interessanten Knoten ohne initiale Platzierung erzeugt. Dazu werden nach der Erzeugung der initialen Platzierungen durch Algorithmus 2 zuerst die initialen Platzierungen in Richtung Quelle zum Ziel propagiert, wobei der Graph entsprechend partitioniert wird. Anschließend werden die initialen Platzierungen im partitionierten Graphen in Richtung Ziel zur Quelle propagiert.
Am Schluss wird noch der kompaktifizierte Graph erzeugt, da dieser im nachfolgenden Schritt der Platzierungsmethode zur Berechnung der Kommunikation benötigt wird.
Dieser Algorithmus gibt einen partitionierten und kompaktifizierten Graphen $G = (V, E)$ aus, wobei jedem Knoten $v \in V$ die Menge $\mathcal{A}_v$ zugeordnet ist, die seine möglichen Platzierungen enthält.

3.3 Nachbearbeitung der möglichen Platzierungen

Nachdem die möglichen Platzierungen durch Propagierung entstanden sind, können sie pro Knoten v aus $\mathcal{A}_v$ weiter bearbeitet werden. Als eine mögliche Nachbearbeitung, die aber in der Arbeit nicht weiter berücksichtigt wird, können die Platzierungen aus der Menge $\mathcal{A}_v$ zu neuen Platzierungen kombiniert werden (Abschnitt 3.3.1). Da zur Codegenerierung HPF-Code erzeugt wird und da durch die Propagierung entlang der Kanten kompliziertere, nicht in HPF darstellbare Platzierungen entstehen können, wird als Nachbearbeitung überprüft, ob alle möglichen Platzierung in HPF darstellbar sind (Abschnitt 3.3.2).

3.3.1 Kombination von Platzierungen

Die grundlegende Idee der Kombination von Platzierungen ist, aus zwei für einen Knoten v möglichen Platzierungen Φ_v^1 und Φ_v^2, die über zwei unterschiedliche Wege w_1 und w_2 durch

`Partition&Propagierung`$(G, V_{\text{write}}, \textit{Richtung}, \Phi_{v_1}, \ldots, \Phi_{v_n}, G'', \Phi_{v'_1}, \ldots, \Phi_{v'_m})$

Eingabe
- Der reduzierte Stelleninstanzgraph $G = (V, E)$. mit $V = \{v_1, \ldots, v_n\}$.
- Für jeden Knoten $v_i \in V$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{v_i}$.
- Der Parameter *Richtung*, der einen der beiden Werte Quelle→Ziel und Ziel→Quelle annehmen kann.
- Die Menge V_{write} der Knoten aus V, die einen Schreibzugriff darstellen.

Ausgabe
- Der partitionierte reduzierte Stelleninstanzgraph $G' = (V', E')$.
- Für jeden Knoten $v'_i \in V'$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{v_i}$.

Algorithmus

1. Bestimme die Menge der zu entfernenden Kanten:

$$\hat{E} = \begin{cases} \bigcup_{v \in V_{\text{write}}} \text{in}(v) & \text{falls } \textit{Richtung} = \text{Quelle}\rightarrow\text{Ziel} \\ \bigcup_{v \in V_{\text{write}}} \text{out}(v) & \text{falls } \textit{Richtung} = \text{Ziel}\rightarrow\text{Quelle} \end{cases}$$

   Bestimme die Menge $E' = E - \hat{E}$, setze $V' = V$ und erzeuge daraus den Graphen $G' = (V', E')$.

2. Führe `Partition`$(G', V_{\text{write}}, \textit{Richtung}, \Phi_{v_1}, \ldots, \Phi_{v_n}, G'', \Phi_{v''_1}, \ldots, \Phi_{v''_m})$ aus, um den Graphen G' entsprechend des Wertes von *Richtung* partitionieren. Dabei ist $G'' = (V'', E'')$ mit $V'' = \{v''_1 \ldots v''_m\}$ und $\Phi_{v''_i}$ die Menge der möglichen Platzierungen von v''_i. Diese Partitionierung erfolgt wie in Abschnitt 3.2.2 beschrieben, wobei bei einer Partitionierung eines Knotens alle möglichen Platzierungen zu allen Partitionen kopiert werden.

3. Führe `Propagierung`$(G'', V_{\text{write}}, \textit{Richtung}, \Phi_{v''_1}, \ldots, \Phi_{v''_m})$ (Algorithmus 4) aus, um die initialen Platzierungen entlang der möglichen Wege im partitionierten Graphen G'' zu propagieren.

4. Füge die entfernten Kanten aus $\hat{E}$ wieder in den partitionierten Graphen G' ein:
   Für jede Kante $e \in \hat{E}$ mit $e = (v, v', h = (D_h, f_h))$:
   (a) Bestimme die Knoten $v_1, \ldots, v_q \in V''$, für die gilt:
   $$O_{v_i} \cap f_h(D_h) \neq \emptyset$$
   (b) Bestimme die Knoten $v'_1, \ldots, v'_{q'} \in V''$, für die gilt:
   $$O_{v_j} \cap D_h \neq \emptyset$$
   (c) Setze $U = \{1, \ldots, q\} \times \{1, \ldots, q'\}$
   (d) Für jedes $u \in U$ mit $u = (u_1, u_2)$:
      i. Berechne die Menge $D'_h = f^{-1}(O_{v_{u_1}} \cap O_{v_{u_2}})$.
      ii. Ist $D'_h \neq \emptyset$, dann erzeuge die Kante $e = (v_{u_1}, v_{u_2}, h' = (D'_h, f_h))$ und füge e in E'' ein.

Algorithmus 6: Propagierung der Platzierungen in eine Richtung

`Berechnungen`$(G_{ROI}, \tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}, \hat{G}, \mathcal{A}_{\hat{v}_1}, \ldots, \mathcal{A}_{\hat{v}_q})$

Eingabe
- Der reduzierte Stelleninstanzgraph $G_{ROI} = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$.
- Die Platzierungen $\tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}$ aller im Programm verwendeten Arrays $\mathtt{A}_1, \ldots, \mathtt{A}_m$ als lineare Relationen.

Ausgabe
- Der partitionierte und kompaktifizierte Stelleninstanzgraph $\hat{G} = (\hat{V}, \hat{E})$ mit $\hat{V} = \{\hat{v}_1, \ldots, \hat{v}_q\}$.
- Für jeden Knoten $\hat{v}_i \in \hat{V}$ die Menge $\mathcal{A}_{v_i}$ der möglichen Platzierungen.

Algorithmus

1. Führe `InitPlatzierung`$(G_{ROI}, \tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}, V_{\text{write}}, \mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n})$ (Algorithmus 2) aus, um die Menge der V_{write} der einen Schreibzugriff darstellenden Knoten des Graphen und für jeden interessanten Knoten $v_i \in V$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{\hat{v}_i}$ zu bestimmen.
2. Propagiere die initialen Platzierungen entlang der Abhängigkeiten in Richtung Quelle zum Ziel der Abhängigkeiten:

   Führe
   `Partition&Propagierung`$(G_{ROI}, V_{\text{write}}, \text{Quelle}{\to}\text{Ziel}, \Phi_{v_1}, \ldots, \Phi_{v_n}, G', \Phi_{v'_1}, \ldots, \Phi_{v'_{n'}})$ (Algorithmus 6) aus, um den partitionierten Graphen $G' = (V', E')$ mit $V' = \{v'_1, \ldots, v'_{n'}\}$ und die Menge der möglichen Platzierungen für den partitionierten Graphen zu erhalten.
3. Propagiere die initialen Platzierungen entlang der Abhängigkeiten in Richtung Ziel zur Quelle einer Abhängigkeit:

   Führe
   `Partition&Propagierung`$(G', V_{\text{write}}, \text{Ziel}{\to}\text{Quelle}, \Phi_{v'_1}, \ldots, \Phi_{v'_{n'}}, G'', \Phi_{v''_1}, \ldots, \Phi_{v''_{n''}})$ (Algorithmus 6) aus, um den partitionierten Graphen $G'' = (V'', E'')$ mit $V'' = \{v''_1, \ldots, v''_{n''}\}$ und die Menge der möglichen Platzierungen für den partitionierten Graphen zu erhalten.
4. Führe `Kompaktifizieren`$(G'', \hat{G})$, um $\hat{G}$ als kompaktifizierten Graphen zu erhalten.

Algorithmus 7: Der Algorithmus zur Bestimmung der möglichen Platzierungen für interessante Stelleninstanzen

Propagierung erzeugt wurden, eine Platzierung Φ_v^c durch Kombination aus Φ_v^1 und Φ_v^2 zu erzeugen, so dass Φ_v^c die Stelleninstanzmenge des Knotens v so platziert ist, dass die beiden Zugriffe über die Wege w_1 und w_2 lokal durchgeführt werden können. Dazu wird der Unterschied zwischen Φ_v^1 und Φ_v^2 „in den Kern der neuen Platzierung Φ_v^c gelegt“, d.h. die Stelleninstanzen, die durch Φ_v^1 und Φ_v^2 unterschiedlich verteilt sind, werden repliziert gespeichert und der Zugriff kann somit lokal erfolgen.
Man beachte, dass es nur Sinn macht, die Kombination für zwei Platzierungen zu bestimmen, die das gleiche Template verwenden, da andernfalls Φ_v^c die Replikation des Knotens v darstellt, weil es keine Gemeinsamkeiten der beiden Platzierungen gibt.
Die Kombination $\mathrm{Comb}(r_1, r_2)$ der beiden linearen Relationen r_1 und r_2 ist die lineare Hülle von $r_1 \cup r_2$.
Für eine Menge $\mathcal{A}_v$ der möglichen Platzierungen eines Knotens v wird die Kombination aller Platzierungen, die das gleiche Template besitzen, aus der Menge $\mathcal{A}_v$ berechnet und in $\mathcal{A}_v$ eingefügt. Dies wird solange wiederholt, bis keine neue Platzierungen durch Kombination erzeugt werden. Die Berechnung aller Kombinationen der für einen Knoten möglichen Platzierungen ist in Algorithmus 8 gezeigt.

`Kombination`$(\mathcal{A}_v)$

Eingabe Die Menge der möglichen Platzierungen $\mathcal{A}_v$ eines Knotens v.

Ausgabe Die Menge der möglichen Platzierungen $\mathcal{A}_v$.

Algorithmus

1. Setze die Menge der kombinierten Platzierungen $W = \emptyset$.
2. Sei $n = |\mathcal{A}_v|$ und erzeuge die Menge $U = \mathfrak{P}(\{1, \ldots, n\})$.
3. Für jede Menge $u \in U$ mit $u = \{u_1, \ldots, u_m\}$:
   (a) Wähle die Platzierungen $\Phi_v^{u_1} = \mathrm{sel}_{u_1}(\mathcal{A}_v), \ldots, \Phi_v^{u_m} = \mathrm{sel}_{u_m}(\mathcal{A}_v)$ aus.
   (b) Bilde die neue Platzierung Φ'_v wie folgt:
   $$\Phi'_v = \mathrm{Comb}(\Phi_v^{u_1}, \mathrm{Comb}(\ldots, \mathrm{Comb}(\Phi_v^{u_{m-1}}, \Phi_v^{u_m}) \ldots))$$
   (c) Füge Φ'_v in W ein.
4. Füge alle neu kombinierten Platzierungen aus der Menge W in $\mathcal{A}_v$ ein.

Algorithmus 8: Algorithmus zur Erzeugung aller möglicher Kombinationen von Platzierungen

3.3.2 Erzeugung von HPF-konformen Platzierungen

Durch die Propagierung können Platzierungen entstanden sein, die nicht durch in HPF gültige `ALIGN`-Direktiven dargestellt werden können, da sie kompliziertere Platzierungen beschreiben, als in HPF darstellbar sind.

Beispiel 3.1 *So kann es z.B. im reduzierten Stelleninstanzgraphen eine Kante geben, deren h-Transformation durch die Funktion $f_h(i,j) = (i,i)$ beschrieben wird. Propagiert man eine Platzierung $\Phi = (\Phi_{OI}, \Phi_P)$ mit*

$$\Phi_{OI}(i,j,m_\infty,m_c) = (i,j,m_\infty,m_c) \qquad \Phi_P(p_1,p_2,m_\infty,m_c) = (p_1,p_2,m_\infty,m_c)$$

entlang dieser Kante mit $\Phi' = \Phi \circ f_h$, dann erhält man die neue Platzierung $\Phi' = (\Phi'_{OI}, \Phi'_P)$, die aus folgenden beiden Abbildungen besteht:

$$\Phi'_{OI}(i,j,m_\infty,m_c) = (i,i,m_\infty,m_c) \qquad \Phi'_P(p_1,p_2,m_\infty,m_c) = (p_1,p_2,m_\infty,m_c)$$

Platziert Φ' die Elemente eines Arrays `A` auf ein Template `T`, dann enspricht diese Platzierung folgender `ALIGN`-Direktive:

```
!HPF$ ALIGN A(i,j) WITH T(i,i)
```

Diese `ALIGN`-Direktive ist aber nicht gültig, da die Variable `i` nicht in mehreren Dimensionen des Templates auftreten darf und somit die Direktive den Spezifikationen des HPF-Standards widerspricht [Hig97] (siehe dazu Abschnitt 2.1.4).

Entsprechend dem HPF-Standard lassen sich durch eine `ALIGN`-Direktive nur Platzierungen beschreiben, die folgende Form besitzen:

1. Eine Dimension des auszurichtenden Arrays wird nicht für mehrere Dimensionen des Templates verwendet.

2. Mehrere Dimensionen des auszurichtenden Arrays werden nicht zur Beschreibung der Ausrichtung für eine Dimension des Templates verwendet.

Daraus lässt sich ableiten, wie für eine Platzierung Φ zu erkennen ist, ob sie den Anforderungen des HPF-Standards entspricht:

Satz 3.1 *Sei $\Phi = (\Phi_{OI}, \Phi_P)$ eine Platzierung, sei die Matrix $M_\Phi = \mathrm{RelMat}(\Phi)$ die zugehörige Matrix und sei M_Φ in reduzierter Zeilen-Stufen-Form. Dann ist Φ eine HPF-konforme Platzierung, wenn für die Zeilen in M folgendes gilt*[2]*:*

1. *Es gibt keine Zeile, in der mehr als eine Index-Dimension verwendet wird.*

2. *Es gibt keine Zeile, in der mehr als eine Prozessor-Dimension verwendet wird.*

3. *Es gibt nicht mehrere Zeilen, die dieselbe Index-Dimension verwenden.*

4. *Es gibt nicht mehrere Zeilen, die dieselbe Prozessor-Dimension verwenden.*

[2]Für den HPF-Compiler ADAPTOR ließe sich die Definition einer HPF-konformen Platzierung noch weiter einschränken: Eine Zeile darf keine Linearkombination von Strukturparametern besitzen, da ADAPTOR dies nicht analysieren kann.

Beweis Entsprechend Abschnitt 2.1.4 hat die `ALIGN`-Direktive folgende Syntax:

`!HPF$ ALIGN Source(`$\mathtt{d}_1,\ldots,\mathtt{d}_q$`) WITH Target(`$\mathtt{s}_1,\ldots,\mathtt{s}_{q'}$`)`

Dabei ist für jede Dimension $\mathtt{s}_i$ von `Target` ist erlaubt:

- $\mathtt{s}_i$ kann einen Ausdruck darstellen, der im Prinzip folgende Form besitzt:

$$a \cdot \mathtt{d}_k + b$$

  Dabei sind $\mathtt{d}_k$ eine Variable, die die Dimension k des auszurichtenden Arrays `Source` bezeichnet, a eine ganzzahlige Konstante oder ein Parameter und b ein affiner Ausdruck in den Parametern.
  Da die Index-Dimensionen aus M den Variablen $\mathtt{d}_1,\ldots,\mathtt{d}_q$ entsprechen, darf eine Zeile höchstens genau eine Index-Dimension verwenden. Sobald eine Zeile mehr als eine Index-Dimension verwendet, ist sie daher nicht HPF-konform.
  Außerdem darf nach dieser Definition kein $\mathtt{s}_i$ von $\mathtt{s}_j$ mit $j \neq i$ und $i, j \in \{1, \ldots, q'\}$ abhängen, so dass auch Zeilen illegal sind, die mehr als eine Prozessordimension verwenden.

- Zusätzlich gilt die Einschränkung, dass eine Variable $\mathtt{d}_k$ in keinem weiteren Ausdruck $\mathtt{s}_l$ von `Target` mit $l \in \{1, \ldots, q'\} \setminus \{i\}$ vorkommen darf.
  Diese Forderung entspricht der Bedingung an die Matrix M, dass sie nicht mehrere Zeilen enthalten darf, die dieselbe Index-Dimension verwenden. Ebenso darf die Matrix M keine Zeilen enthalten, die dieselbe Prozessor-Dimension mehrfach verwendet.

□

Um zu gewährleisten, dass die Menge $\mathcal{A}_v$ der möglichen Platzierungen eines Knotens v nur HPF-konforme Platzierungen enthält, werden aus den Platzierungen $\Phi_v^i \in \mathcal{A}_v$, die nicht HPF-konform sind, neue HPF-konforme Platzierungen erzeugt. Danach wird Φ_v^i aus der Menge $\mathcal{A}_v$ entfernt. Um aus einer Platzierung Φ_v^i HPF-konforme Platzierungen zu erzeugen, wird aus der Matrix $M_{\Phi_v^i}$ in Satz 3.1 jede Kombination an Zeilen, die nicht den Eigenschaften von Satz 3.1 entsprechen, jeweils durch eine Null-Zeile ersetzt. Die so entstehenden Matrizen werden darauf überprüft, ob sie HPF-konform sind. Ist dies der Fall, werden aus ihnen neue Platzierungen erzeugt, die in $\mathcal{A}_v$ eingefügt werden. Dadurch entstehen neue HPF-konforme Platzierungen aus Φ_v^i, die die Stelleninstanzen entlang derjenigen Template-Dimensionen replizieren, die von Zeilen verwendet wurden, die Φ_v^i zu einer nicht HPF-konformen Platzierung machen. Für eine neue HPF-konforme Platzierung $\hat{\Phi}_v^i$ gilt dann:

$$\forall \alpha \in O_v : \ \hat{\Phi}_v^i(\alpha) \supseteq \Phi_v^i(\alpha)$$

Algorithmus 9 transformiert alle Platzierungen aus der Menge $\mathcal{A}_v$ für einen Knoten v in HPF-konforme Platzierungen.

Beispiel 3.2 *Die Platzierung Φ' aus Beispiel 3.1 lässt sich durch folgende Matrix darstellen:*

$$M_{\Phi'} = \begin{array}{c} \begin{array}{cccccccc} i & j & m_\infty & m_c & p_1 & p_2 & m_\infty & m_c \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{16}$$

`HPFKonform`($\mathcal{A}_v$)

Eingabe Die Menge der möglichen Platzierungen $\mathcal{A}_v$ für einen Knoten v.

Ausgabe Die Menge der möglichen Platzierungen $\mathcal{A}_v$. ($\mathcal{A}_v$ enthält nur HPF-konforme Platzierungen.)

Algorithmus

1. Für jede Platzierung $\Phi_v^i \in \mathcal{A}_v$, die entsprechend Satz 3.1 nicht HPF-konform ist:
   (a) Erzeuge die Matrix $M_{\Phi_v^i} = \mathrm{RelMat}(\Phi_v^i)$.
   (b) Setze die Menge $W = \emptyset$, die die HPF-konformen Matrizen aufnehmen soll.
   (c) Bestimme die Menge H der Zeilen, die entsprechend Satz 3.1 in der Matrix $M_{\Phi_v^i}$ nicht HPF-konform sind.
   (d) Setze $U = \mathfrak{P}(H) - \{\emptyset\}$.
   (e) Bestimme für jede Menge $R \in U$:
      i. Setze $M'_v = M_{\Phi_v^i}$.
      ii. Ersetze alle Zeilen, die durch die Elemente von R referenziert werden, in M'_v durch die Nullzeile.
      iii. Füge M'_v in W ein.
2. Überprüfe für jede Matrix $M' \in W$, ob sie entsprechend Satz 3.1 HPF-konform ist. Ist dies nicht der Fall, dann entferne sie aus W.
3. Erzeuge aus allen Matrizen $M' \in W$ wieder eine Platzierung $\Phi_v = \mathrm{RelMat}^{-1}(M')$ und füge Φ_v in $\mathcal{A}_v$ ein.

Algorithmus 9: Algorithmus zur Erzeugung von HPF-konformen Platzierungen für einen Knoten v

Die Platzierung Φ' ist entsprechend Satz 3.1 nicht HPF-konform, da beide virtuellen Prozessordimensionen p_1 und p_2 auf die gleiche Variablen-Dimension i zugreifen. Entfernt man die erste Zeile, wie in Gleichung (17), oder entfernt man die zweite Zeile, wie in Gleichung (18), dann enthält man zwei neue Platzierungen, die HPF-konform sind:

$$M_{\Phi''} = \begin{array}{c} \begin{array}{cccccccc} i & j & m_\infty & m_c & p_1 & p_2 & m_\infty & m_c \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{17}$$

$$M_{\Phi'''} = \begin{array}{c} \begin{array}{cccccccc} i & j & m_\infty & m_c & p_1 & p_2 & m_\infty & m_c \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{18}$$

Aus diesen beiden Matrizen werden die Platzierungen Φ'' und Φ''' durch

$$\begin{aligned} \Phi'' &= \mathrm{RelMat}^{-1}(M_{\Phi''}) \\ \Phi''' &= \mathrm{RelMat}^{-1}(M_{\Phi'''}) \end{aligned}$$

bestimmt und durch folgende Funktionen beschrieben:

$$\begin{aligned} \Phi''_{OI}(i,j,m_\infty,m_c) &= (0,i,m_\infty,m_c) & \Phi''_{P}(p_1,p_2,m_\infty,m_c) &= (0,p_2,m_\infty,m_c) \\ \Phi'''_{OI}(i,j,m_\infty,m_c) &= (i,0,m_\infty,m_c) & \Phi'''_{P}(p_1,p_2,m_\infty,m_c) &= (p_1,0,m_\infty,m_c) \end{aligned}$$

Die Platzierung Φ'' lässt sich durch die `ALIGN`-Direktive

```
!HPF$ ALIGN A(i,j) WITH T(*,i)
```

und die Platzierung Φ''' durch die `ALIGN`-Direktive

```
!HPF$ ALIGN A(i,j) WITH T(i,*)
```

darstellen. Wie man erkennen kann, werden aus der Platzierung Φ' aus Beispiel 3.1 dadurch zwei neue HPF-konforme Platzierungen erzeugt, indem die durch Φ' platzierten Stelleninstanzen einmal in der ersten Dimension und einmal in der zweiten Dimension des Templates repliziert gespeichert werden.

3.4 Auswahl der Platzierung

Nachdem in den beiden vorhergehenden Abschnitten beschrieben wurde, wie für jeden Knoten im reduzierten Stelleninstanzgraphen die Menge der möglichen Platzierungen erzeugt wird, kann in diesem Abschnitt die Auswahl einer Platzierung entsprechend einem Kostenmodell gezeigt werden.

Dazu soll zuerst in Abschnitt 3.4.1 dargelegt werden, wie für eine mögliche Wahl von Platzierungen die Kommunikation berechnet wird. Im darauffolgenden Abschnitt 3.4.2 wird dann der Algorithmus vorgestellt, der für jede Kombination von Platzierungen die Kommunikation berechnet, diese entsprechend einem Kostenmodell bewertet und die beste Kombination von Platzierungen auswählt.

3.4.1 Bestimmung der Kommunikation

Nachdem für jeden Knoten v im kompaktifizierten reduzierten Stelleninstanzgraphen $G_{ROI_{comp}}$ die möglichen Platzierungen $\mathcal{A}_v$ bestimmt sind, wird für eine Wahl von möglichen Platzierungen die sich ergebende Kommunikationsstruktur berechnet. Dazu werden die durch die Kanten im $G_{ROI_{comp}}$ beschriebenen Abhängigkeiten zwischen den Stelleninstanzen zu Abhängigkeiten zwischen den Template-Elementen umgeschrieben. Diese neuen Abhängigkeiten setzen Punkte des Raumes der virtuellen Prozessoren $\mathcal{P}$ in Beziehung zueinander und modellieren die durch den HPF-Compiler erzeugte Kommunikation.
Für eine h-Transformation $h = (D_h, f_h)$ der Kante $e = (v, v', h)$ und für die Platzierungen Φ_v und $\Phi_{v'}$ lässt sich die zugehörige Kommunikationsrelation $h_{comm} = (h_{comm_L}, h_{comm_R})$, wie folgt, darstellen:

$$h_{comm} \quad = \quad \Phi_v \circ_- f_h \circ_+ \Phi_{v'}^{-1} \tag{19}$$

Die lineare Relation $h_{comm}(\delta_{v'})$ gibt für ein Template-Element $\delta_{v'}$ die Menge derjenigen Template-Elemente an, von denen die Werte zur Berechnung der auf $\delta_{v'}$ liegenden Stelleninstanzen gelesen werden müssen. Dabei kann sowohl Φ_v als auch $\Phi_{v'}$ eine Menge darstellen. Für eine Stelleninstanz $\alpha_{v'}$ des Knotens v' gibt $\alpha_v = f_h(\alpha_{v'})$ die Stelleninstanz an, von der $\alpha_{v'}$ lesen muss. Die Menge $\Phi_v(\alpha_v)$ repräsentiert die Template-Elemente, von denen die Stelleninstanz $\alpha_{v'}$ lesen kann. Da alle Template-Elemente dieser Menge den gleichen Wert speichern, lässt sich aus der Menge ein Template-Element so auswählen, dass beim Lesezugriff auf den Wert der Stelleninstanz α_v möglichst wenig Kommunikation auftritt. Um das zu erreichen, wird ein Template-Element gewählt, dessen Koordinaten in möglichst vielen Dimensionen mit den Koordinaten des Template-Elementes, auf dem $\alpha_{v'}$ gespeichert ist, übereinstimmen. Diese Auswahl erfolgt in der Berechnung durch den Operator $\circ_-$ in $\Phi_v \circ_- f_h(\alpha_{v'})$ dadurch, dass für diejenigen Dimensionen, die diesbezüglich zur Verfügung stehen, die Identität gewählt wird.
Sind auf einem Template-Element $\delta_{v'}$ mehrere Stelleninstanzen der Stelleninstanzmenge $O_{v'}$ gespeichert, so dass $\Phi_{v'}(\delta_{v'})$ eine Menge beschreibt, dann kann aus dieser Menge kein Element ausgewählt werden, sondern es muss die gesamte Menge betrachtet werden, da diese Menge das Ziel der Abhängigkeit h darstellt und daher alle Stelleninstanzen ausgeführt werden müssen.

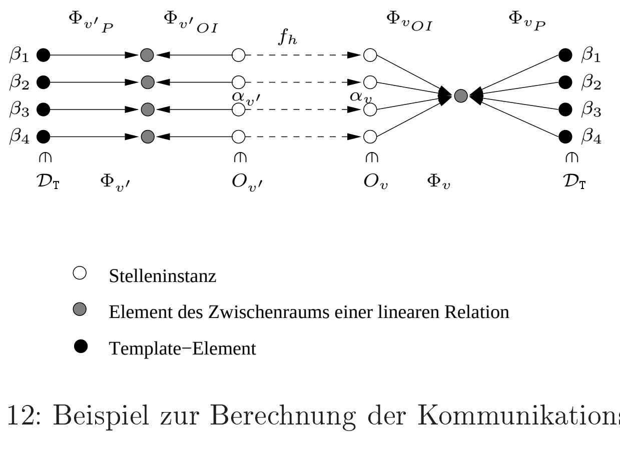


Abbildung 12: Beispiel zur Berechnung der Kommunikationsrelation

Die Abbildung 12 zeigt als Beispiel die Abhängigkeiten zwischen den Stelleninstanzen zweier Knoten v, v' im reduzierten Stelleninstanzgraph für die Berechnung einer Kommunikationsrelation. Die Stelleninstanzmengen O_v und $O_{v'}$ der Knoten v, v' enthalten jeweils vier Stelleninstanzen, wobei die Stelleninstanzen aus O_v auf die Template-Elemente aus $\mathcal{D}_T = \{\beta_1, \ldots, \beta_4\}$

verteilt und die Stelleninstanzen aus $O_{v'}$ repliziert gespeichert sind. Durch die h-Transformation wird jeder Stelleninstanz $\alpha_{v'}$ genau eine Stelleninstanz α_v aus O_v zugeordnet.
Soll z.B. der Wert einer Stelleninstanz $\alpha_{v'}$, die auf dem Template-Element β_3 liegt, berechnet werden, ist der Wert der Stelleninstanz α_v zu lesen, der auf allen Template-Elementen verfügbar ist. Dabei braucht der Wert nicht von allen sondern nur von einem Template-Element gelesen zu werden. Dazu wird das Template-Element β_3 ausgewählt, so dass sowohl α_v als auch $\alpha_{v'}$ demselben Template-Element zugeordnet sind, weshalb keine Kommunikation erzeugt wird.
Um die Kommunikation für eine mögliche Platzierung der einzelnen Knoten im kompaktifizierten, reduzierten Stelleninstanzgraphen zu bestimmen, wird Algorithmus 10 verwendet. Er erzeugt den Kommunikationsgraphen G_{comm} aus $G_{ROI_{comp}}$, indem er für jede Kante in $G_{ROI_{comp}}$ die Kommunikationrelation entsprechend der Gleichung (19) berechnet.

`ErzKommGraph`$(G, \Phi_{v_1}, \ldots, \Phi_{v_n}, G_{comm})$

Eingabe
- Der kompaktifizierte reduzierte Stelleninstanzgraph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$.
- Für jeden Knoten $v_i \in V$ eine ausgewählte Platzierung Φ_{v_i}

Ausgabe
- Der Kommunikationsgraph $G_{comm} = (V_{comm}, E_{comm})$.

Alogrithmus

1. Setze $V_{comm} = V_{comp}$ und $E_{comm} = \emptyset$.
2. Für jede Kante $e \in E_{comp}$ mit $e = (v, v', f)$ berechne die lineare Relation $h_{comm} = (h_{comm_L}, h_{comm_R})$ mit der Vorschrift:

$$h_{comm} = \Phi_v \circ_- f \circ_+ \Phi_{v'}^{-1}$$

   Füge die neue Kante $e' = (v, v', h_{comm})$ in E_{comm} ein.

Algorithmus 10: Berechnung des Kommunikationsgraphen G_{comm}

Der Kommunikationsgraph G_{comm} repräsentiert also die vom HPF-Compiler erzeugte Kommmunikation und stellt folgende Informationen zur Verfügung: Für jede Flussabhängigkeit im kompaktifizierten reduzierten Stelleninstanzgraphen enthält der Kommunikationsgraph G_{comm} eine Kante, die als Markierung für die entsprechende Flussabhängigkeit die Kommunikationsrelation h_{comm} besitzt, die die Beziehungen zwischen den Template-Elementen beschreibt.
Basierend auf der Beziehungen der Template-Elemente, die durch eine Kommunikationsrelation h_{comm} beschrieben wird, lässt sich entscheiden, ob der Zugriff lokal ist oder ob Kommunikation benötigt wird. Ist die Verteiltung genauer spezifiziert, z.B. dass alle Template-Dimensionen blockverteilt sind, dann kann anhand der Form der Kommunikationsrelation zusätzlich bestimmt werden, welches Kommunikationsmuster sie beschreibt.
Somit enthält der Kommunikationsgraph G_{comm} alle zur Bewertung der Kommunikation nötigen Informationen. Ein Kostenmodell spezifiziert daher zur Bewertung die Kostenfunktion CostFunc, die die folgende Signatur besitzt:

$$\text{CostFunc} : G_{comm} \to \mathbb{N}$$

3.4.2 Auswahl einer möglichen Platzierung

Nachdem der Algorithmus zur Berechnung der Kommunikation für eine Wahl von Platzierungen angegeben ist, kann der Algorithmus für die Auswahl der Platzierung dargestellt werden. Algorithmus 11 berechnet für jede Kombination von Platzierungen den Kommunikationsgraphen und bewertet ihn durch die Funktion CostFunc, die als Argument übergeben wird. Zum Schluss werden die Platzierungen ausgegeben, die entsprechend der Kostenfunktion CostFunc die beste Kommunikation erzeugt haben.

`Bewertung`$(G, \mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n}, \text{CostFunc}, \Phi_{v_1}, \ldots, \Phi_{v_n})$

Eingabe
- Der kompaktifizierte reduzierte Stelleninstanzgraph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$.
- Für jeden Knoten $v_i \in V$ die Menge seiner möglichen Platzierungen $\mathcal{A}_{v_i}$.
- Eine Kostenfunktion $\text{CostFunc} : G_{comm} \to \mathbb{N}$.

Ausgabe
- Für jeden Knoten $v_i \in V$ seine ausgewählte Platzierung Φ_{v_i}.

Algorithmus

1. Setze die Variable $cost_{best} = \infty$.
2. Bestimme die Anzahl m_{v_i} der möglichen Platzierungen des Knotens v_i für alle $i \in \{1, \ldots, n\}$.
3. Bestimme die Menge der Kombinationen der möglichen Platzierungen:

$$\mathcal{K} = \{1, \ldots, m_{v_1}\} \times \ldots \times \{1, \ldots, m_{v_n}\}$$

   Für ein $u \in \mathcal{K}$ und ein $v_i \in V$ sei $\pi_{v_i}(u)$ die Projektion auf die i-te Koordinate, die dem Knoten v_i zugeordnet ist.
4. Setze den n-dimensionalen Vektor $\varrho \in \mathcal{K}$ auf $\varrho = (1, \ldots, 1)$.
5. Für jedes $u \in \mathcal{K}$:
   (a) Wähle für jeden Knoten $v_i \in V$ mit $\Phi_{v_i} = \text{sel}_{\pi_{v_i}(u)}(\mathcal{A}_{v_i})$ eine Platzierung aus.
   (b) Führe `ErzKommGraph`$(G, \Phi_{v_1}, \ldots, \Phi_{v_n}, G_{comm})$ (Algorithmus 10) aus, um für die aktuelle Wahl an Platzierungen den Kommunikationsgraphen G_{comm} zu berechnen.
   (c) Betsimme die Kosten $cost$ mit $cost = \text{CostFunc}(G_{comm})$.
   (d) Ist $cost_{best} > cost$, dann setze $\varrho = u$ und $cost_{best} = cost$.
6. Für jeden Knoten $v_i \in V$ wähle die Platzierung $\Phi_{v_i} = \text{sel}_{\pi_{v_i}}(\varrho)(\mathcal{A}_v)$ aus.

Algorithmus 11: Auswahl der bezüglich der Kostenfunktion CostFunc besten Platzierung für jeden Knoten des Graphen G

3.5 Der Platzierungs-Algorithmus

Abschließend wird Algorithmus 12 vorgestellt, der für ein als reduzierter Stelleninstanzgraph gegebenes Programmfragment die bezüglich einer Kostenfunktion CostFunc beste Platzierung für die interessanten Knoten des reduzierten Stelleninstanzgraphen auswählt. Dabei wird analog zum Algorithmus 1 vorgegangen. Die Kostenfunktion CostFunc wird von außen vorgegeben, so dass der Algorithmus unabhängig von dem der Kostenfunktion CostFunc zugrundeliegenden Kostenmodell ist.
Algorithmus 12 gibt den partitionierten und kompaktifizierten Graphen G aus, wobei für jeden Knoten von G eine Platzierung ausgewählt ist.

`OIAllocator`$(G_{ROI}, \tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}, \text{CostFunc}, G, \Phi_{v_1}, \ldots, \Phi_{v_n})$

Eingabe

- Der reduzierte Stelleninstanzgraph G_{ROI} des zu analysierenden Programmfragments.
- Die Platzierungen $\tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}$ aller im Programm verwendeten Arrays $\mathtt{A}_1, \ldots, \mathtt{A}_m$ als lineare Relationen.
- Eine Kostenfunktion CostFunc mit $\text{CostFunc} : G_{comm} \to \mathbb{N}$.

Ausgabe

- Der partitionierte und kompaktifizierte reduzierte Stelleninstanzgraph $G = (V, E)$.
- Die Platzierungen $\Phi_{v_1}, \ldots, \Phi_{v_n}$ der interessanten Knoten aus $V = \{v_1, \ldots, v_n\}$.

Algorithmus

1. Führe `Berechnungen`$(G_{ROI}, \tilde{\Phi}_{\mathtt{A}_1}, \ldots, \tilde{\Phi}_{\mathtt{A}_m}, G, \mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n})$ (Algorithmus 7) aus, um den partitionierten und kompaktifizierten Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ und für jeden Knoten $v_i \in V$ die Menge der möglichen Platzierungen $\mathcal{A}_{v_i}$ zu bestimmen.
2. Nachbearbeitung der bereits gefundenen Platzierungen:
   (a) (optional) Führe `Kombination`$(\mathcal{A}_{v_i})$ (Alogrithmus 8) für jeden Knoten $v_i \in V$ aus, um für jede Menge $\mathcal{A}_{v_i}$ neue Platzierungen dadurch zu erzeugen, dass alle Platzierungen aus $\mathcal{A}_{v_i}$ miteinander kombiniert werden.
   (b) Führe `HPFKonform`$(\mathcal{A}_{v_i})$ (Algorithmus 9) für jeden Knoten $v_i \in V$ aus, so dass für jeden Knoten v_i die Menge $\mathcal{A}_{v_i}$ nur HPF konforme Platzierungen enthält.
3. Führe `Bewertung`$(G, \mathcal{A}_{v_1}, \ldots, \mathcal{A}_{v_n}, \text{CostFunc}, \Phi_{v_1}, \ldots, \Phi_{v_n})$ (Algorithmus 11) aus, um aus allen möglichen Kombinationen von Platzierungen eine Kombination auszuwählen, die entsprechend der Kostenfunktion CostFunc die beste Kommunikation erzeugen.

Algorithmus 12: Methode zur Berechnung von Platzierungen für allgemeine Berechnungsinstanzen

4 Implementierung

Die in Abschnitt 3 beschriebene Platzierungsmethode wurde als Komponente des Projekts `LooPo` [GL96] entwickelt und in ein Skript integriert, das die Schleifensätze des übergebenen Programms transformiert, das Resultat in das Originalprogramm wieder einfügt und anschließend das transformierte HPF-Programm durch den HPF-Compiler ADAPTOR übersetzt. Die Implementierung besteht aus folgenden Programmen:

`basicallocs` Dieses Programm liest ein in HPF geschriebenes Programm ein und extrahiert diejenigen Schleifensätze, die von `LooPo` transformiert werden sollen. Für jeden Schleifensatz analysiert es die `ALIGN`-Direktiven der verwendeten Arrays und gibt die so beschriebenen Platzierungen als lineare Relationen aus.

`oiallocator` Dieses Programm führt die eigentliche Berechnung der Platzierung durch, wobei es den bereits partitionierten reduzierten Stelleninstanzgraphen des zu analysierenden Programmfragmentes einliest, der von der Implementierung der Methode LCCP generiert wurde.

`genAlign` Dieses Programm erzeugt anschließend an die Codegenerierung die `ALIGN`-Direktiven für die neu eingeführten Hilfsarrays.

4.1 Programm `basicallocs`

Das Programm `basicallocs` extrahiert aus einem gegebenen HPF-Programm die durch `LooPo` transformierbaren Schleifensätze, und die zugehörigen Platzierungen der Arrays und bereitet diese Informationen so auf, dass sie durch die von `LooPo` bereitgestellten Programme bearbeitet werden können.
Da die Implementierung eines vollwertigen Parsers für HPF-Programme sehr zeitaufwändig ist, wurde nur ein einfaches Programm entwickelt, das zeilenweise das eingelesene Programm analysiert und anhand von Schlüsselwörtern die gewünschte Information extrahiert. Dabei fehlen allerdings Möglichkeiten, ein HPF-Programm genau zu analyiseren, um z.B. die durch `LooPo` transformierbaren Schleifensätze oder die im Schleifensatz verwendeten Strukturparameter automatisch zu erkennen. Um diese nötigen Informationen zu erhalten, unterstützt `basicallocs` Direktiven, die im HPF-Programm vor den jeweiligen Schleifensätzen stehen. Tabelle 4 listet die möglichen Direktiven auf.
Die Analyse des eingelesenen HPF-Programms erfolgt pro Programmblock, wobei ein Programmblock durch

- `SUBROUTINE ... END SUBROUTINE`,
- `FUNCTION ... END FUNCTION` oder
- `PROGRAM ... END PROGRAM`

begrenzt ist. Nachdem für einen Programmblock zuerst alle in ihm definierten Arrays analysiert und ihre Platzierung als lineare Relation gespeichert worden sind, wird für jeden zu analysierenden Schleifensatz im aktuellen Programmblock ein Programmfragment für `LooPo` erzeugt, das aus folgenden Informationen besteht:

`!LOOPO$ CONSTANT <cList>`	Mit dieser Direktive wird durch `<cList>` die Liste der Strukturparameter angegeben, die im folgenden Schleifensatz verwendet werden.
`!LOOPO$ DEF <definition>`	Diese Direktive kopiert den in `<definition>` angegebenen Text direkt in die Eingabe für `LooPo`, die für den nachfolgenden Schleifensatz erzeugt wird, wodurch die Möglichkeit der Eingabe von Definitionen der verwendeten Funktionen entsteht.
`!LOOPO$ SCALARS <scalars>`	Diese Direktive gibt mit `scalars` die Liste der Skalare an, die in der nachfolgenden Schleife referenziert werden.
`!LOOPO$ IGNORELOOP`	Der dieser Direktive nachfolgende Schleifensatz wird nicht transformiert.

`!LOOPO$ BEGIN`
`<code>`
`!LOOPO$ END`
	Der zwischen diesen beiden Direktiven stehende Programmcode wird als ein neues zu analysierendes Programmfragment betrachtet, wodurch mehrere aufeinander folgende Schleifensätze als ein neues Programmfragment markiert werden können.

Tabelle 4: Mögliche Direktiven für das Programm `basicallocs`

- Der Quelltext des aktuellen Schleifensatzes aus dem aktuellen Programmblock wird als Eingabe für den Parser von `LooPo` erzeugt, wobei für jedes Array je ein Schleifensatz, der den Datenraum des Arrays aufzählt, vor und nach dem Quelltext des aktuellen Schleifensatzes eingefügt wird. Dabei sind alle Arrays mit dem gleichen Datenraum in einer Schleife zusammengefasst, um die Anzahl von Dimensionen im Modell zu vermindern.
- Die Platzierungen aller im aktuellen Programmblock definierten Arrays werden als lineare Relationen gespeichert.
- Für jedes Array werden der Datenraum und der Name des Templates, an den dieses Array ausgerichtet ist, gespeichert.
- Außerdem wird die Zuordnung der Templates zu den virtuellen Prozessordimensionen vermerkt.

Nachdem für alle zu transformierenden Schleifensätze die zugehörigen Programmfragmente erzeugt wurden, wird zusätzlich das Original-Programm ausgegeben, wobei diejenigen Stellen im Programmtext markiert sind, an denen die transformierten Schleifensätze einzufügen sind, so dass nach der Transformation das Programm wieder zusammengefügt werden kann.
Um die oben angegebenen Informationen aus dem HPF-Programm extrahieren zu können, muss es folgende Form besitzen:

- Entsprechend Abschnitt 3.1.1 ist gefordert, dass alle Arrays durch `ALIGN`-Direktiven an Templates ausgerichtet sind.
- Aufgrund der eingeschränkten Analysemöglichkeiten von `basicallocs`, muss jedem `END` der Name desjenigen Konstrukts folgen, dessen Gültigkeitsbereich das `END` abschließt.

- Die Fortran-Notation der Schleifen mit einem Label

  DO < *label* > i=l,u

  wird nicht unterstützt.

4.1.1 Einlesen der Platzierung

Um aus den ALIGN-Direktiven der in einem Programmblock definierten Arrays die Platzierung als lineare Relationen zu erzeugen, werden die Anzahl n_{pDim} der benötigten virtuellen Prozessordimensionen wie in Abschnitt 3.1.1 beschrieben aus der Summe der Dimensionen aller im Programmfragment verwendeten Templates und der Zuordnung der virtuellen Prozessordimensionen zu den im Programmfragment auftretenden Templates bestimmt.
Anschließend wird für jedes Array die Platzierung als lineare Relation nach folgendem Algorithmus erzeugt:
Im Allgemeinen lässt sich eine ALIGN-Direktive, die ein m-dimensionales Array A an ein n-dimensionales Template T ausrichtet, wie folgt, schreiben (vgl. Abschnitt 2.1.1):

```
!HPF$ ALIGN A(d_1,...,d_m) WITH T(s_1,...,s_n)
```

Dabei ist $\mathtt{d}_i$ eine Variable, die die Dimension i des Arrays A angibt. Zur Erzeugung der linearen Relation wird der Ausdruck des Ziels der ALIGN-Direktive ausgewertet, der die Zuordnung der Arrayelemente auf die Template-Elemente darstellt. Damit sich dieser Ausdruck durch eine in den Arraydimensionen $\mathtt{d}_1,\ldots,\mathtt{d}_m$ und in den Strukturparametern lineare Funktion darstellen lässt, werden diejenigen $\mathtt{s}_k$, die das Zeichen * repräsentieren, auf m_∞ gesetzt, so dass alle Dimensionen des Templates, auf denen repliziert werden soll, mit m_∞ markiert sind.
Nun lässt sich $\mathtt{T}(\mathtt{s}_1,\ldots,\mathtt{s}_n)$ durch die Zugriffsfunktion φ mit $\mathtt{T}(\varphi(\mathbf{i}))$ darstellen, wobei $\varphi : \mathbb{Q}^{m+n_b} \to \mathbb{Q}^{n+n_b}$ eine in den m Indizes $\mathtt{d}_1,\ldots,\mathtt{d}_m$ und in den n_b Strukturparameter lineare Funktion ist, die vom Datenraum des Arrays A in den Datenraum des Templates T abbildet. Die Reihenfolge der Dimensionen für die Variablen entspricht der Reihenfolge der Variablen $\mathtt{d}_1,\ldots,\mathtt{d}_m$.
Um die Platzierung $\tilde{\Phi}_\mathtt{A}$ des Arrays A zu erzeugen, wird aus der Matrix M_φ der Abbildung φ und der $(m+n_b)\times(m+n_b)$ Einheitsmatrix M', die Matrix $M_{\tilde{\Phi}}$, wie folgt, aufgestellt:

$$M = (M_\varphi | - M')$$

Um die Replikation korrekt darzustellen, werden in M alle Zeilen durch die Null-Zeile ersetzt, die nur den Eintrag 1 in derjenigen Spalte besitzen, die den Parameter m_∞ darstellt.
Abschließend wird aus M die lineare Relation $\tilde{\Phi}$ durch

$$\tilde{\Phi} = \mathrm{RelMat}^{-1}(M)$$

erzeugt.

4.2 Programm oiallocator

Das Programm `oiallocator` führt die eigentliche Platzierungsmethode aus. Als Darstellung des zu analysierenden Programmfragments liest es den vom Programm `lccp` - der Implementierung der Methode LCCP - ausgegebenen reduzierten Stelleninstanzgraphen, der bereits durch `lccp` partitioniert wurde und der nur die Fluss-Abhängigkeiten enthält, so dass bei der Berechnung der möglichen Platzierungen kein Zyklus im Graphen auftreten kann.

4.2.1 Repräsentation einer linearen Relation

Die lineare Relation $r = (r_L, r_R)$ wird in der Klasse `virtualMapping` gespeichert, die die beiden Funktionen r_L und r_R als zwei Attribute der Klasse `scopedFunction` repräsentiert. Die Klasse `scopedFunction` enthält die Abbildung und deren Definitionsmenge als Polyeder, der leer sein kann und nur für den Test auf Gleichheit zweier linearer Relationen (siehe Abschnitt 2.5.1) berücksichtigt wird. Die Abbildung selbst wird in der Klasse `AffineFunction` als Matrix gespeichert, die neben den Dimensionen aller Strukturparameter nur die in der Abbildung verwendeten Dimensionen der Variablen enthält.
Diese Vorgehensweise wurde gewählt, da die Matrizen, die alle Variablen-Dimensionen enthalten, nur dünn besetzt sind und daher viele Null-Werte gespeichert werden, die nie einen anderen Wert annehmen und keine Information tragen.
Als Beispiel soll kurz die Anzahl der im Programm 3.1 aus Abschnitt 3.2.1 verwendeten Indizes gezeigt werden. In Tabelle 5 sind die benötigten Indizes aufgelistet. Es werden 4 Strukturpa-

Array-Referenzierungs-Schleife vor Programmfrag.	2-dim. Array 1-dim. Array	2 Indizes 1 Index
Array-Referenzierungs-Schleife nach Programmfrag.	2-dim. Array 1-dim. Array	2 Indizes 1 Index
Schleifen im Programmfrag.	2 Schleifen	2 Indizes
Zusätzliche Dimensionen	für *op* und *occ*	2 Indizes
Alle Indizes		10 Indizes

Tabelle 5: Die Anzahl der Indizes des Programms 3.1 aus Abschnitt 3.2.1

rameter verwendet, die sich aus den Parametern `M,N` und m_∞, m_c zusammensetzen. Damit wird eine lineare Funktion, die alle Dimensionen berücksichtigt durch eine $(10+4) \times (10+4)$ Matrix beschrieben. Da das Programmfragment Schleifensätze mit maximal 2 Schleifen besitzt, werden z.B. für die h-Transformationen maximal Funktionen benötigt, die 3 Indizes (2 Dimensionen für die Schleifen plus die Dimension für *occ*) verwenden, so dass sie durch eine $(3+4) \times (3+4)$ Matrix dargestellt werden kann.
Die gesamte Beschreibung der Dimensionen wird durch die Klasse `descriptionOfDims` verwaltet. Diese Klasse besitzt Funktionen, um Matrizen von dem von `LooPo` vorgegebenen Format in das interne Format und umgekehrt zu wandeln. Außerdem bietet sie Methoden, einen transparenten Zugriff auf einzelne Dimensionen zu liefern.
Um z.B. zu erkennen, welche Dimensionen eine h-Transformation benötigt, werden die Indexräume der Quelle und des Ziels der Abhängigkeit betrachtet und es werden nur die Dimensionen gewählt, die nicht auf $\pm m_\infty$ gesetzt sind.
Die Implementierung des Tests auf Gleichheit zweier Relationen weicht dadurch von der Beschreibung aus Abschnitt 2.5.1 ab, dass in die beiden Matrizen

$$\begin{aligned} M_{r_1} &= \mathrm{RelMat}(r_1) \\ M_{r_2} &= \mathrm{RelMat}(r_2) \end{aligned}$$

zusätzlich alle Gleichungen aus den Definitionsmengen von ${r_1}_L$, ${r_1}_R$ und ${r_2}_L$, ${r_2}_R$ eingefügt werden. Damit werden die Gleichungen aus den Polyedern berücksichtigt, die nicht in den Abbildungen ${r_1}_L$, ${r_1}_R$ und ${r_2}_L$, ${r_2}_R$ dargestellt sind.

4.2.2 Repräsentation des reduzierten Stelleninstanzgraphen

Die Klasse `OIDepGraph` repräsentiert einen reduzierten Stelleninstanzgraphen und enthält die Methoden zur Kompaktifizierung, Invertierung, Berechnung der transitiven Hülle und zur Abfrage spezieller Knoteninformationen. Dabei werden die Kanten im Graphen als lineare Relationen gespeichert, um auch inverse h-Transformationen darstellen zu können, weshalb für alle Berechnungen die Implementierungen der Operatoren $\circ_+$ und $\circ_-$ verwendet werden.
Die Klasse `variableAllocation` repräsentiert die Menge $\mathcal{A}_v$ der für den Knoten v möglichen Platzierungen und enthält Methoden zur Erstellung von HPF-konformen Platzierungen und zum direkten Zugriff auf einzelne Platzierungen.

4.2.3 Initialisierung der Platzierungen

Die Initialisierung der Platzierungen erfolgt in der Methode `oiallocator::makePermVect`. Dabei werden die Platzierungen der Arrays den Knoten des Stelleninstanzgraphen zugewiesen, die einen Schreibzugriff auf ein Array darstellen. Die Platzierungen von Skalaren und beliebigen affinen Ausdrücken werden gesondert behandelt, da Skalare in HPF-Programmen im Allgemeinen auf allen Prozessoren repliziert gespeichert sind, falls keine explizite Platzierung vorgegeben ist [Hig97]. Auch die von Schleifenvariablen unabhängigen affinen Ausdrücke sind repliziert. In der Implementierung werden alle affinen Ausdrücke als repliziert gespeichert.
Die Darstellung der Replikation auf alle Prozessoren kann in der Implementierung nicht durch eine einzige Platzierung ausgedrückt werden, da die Replikation auf alle Prozessoren bedeutet, dass eine Stelleninstanzmenge auf alle virtuellen Prozessordimensionen verteilt ist. Die Implementierung erlaubt aber nur, dass eine Platzierung eine Stelleninstanzmenge lediglich auf denjenigen Teil der virtuellen Prozessordimensionen ausrichtet, der genau ein Template beschreibt.
Gibt man die Replikation eines Knotens auf alle Prozessoren explizit dadurch an, dass er für jedes im Programmfragment verwendete Template eine Platzierung besitzt, die den Knoten auf das entsprechende Template repliziert, dann kann bei der Bewertung der Kommunikationsrelation für einen auf alle Prozessoren replizierten Knoten nicht immer erkannt werden, dass der Zugriff lokal ist.
Um dieses Problem zu umgehen werden in der Implementierung standardmäßig diejenigen Knoten, die Skalare oder affine Ausdrücke darstellen, als nicht interessant markiert, so dass ihre Platzierungen nicht bestimmt werden müssen. Dies ist möglich, da die Propagierung der Platizerungen der Knoten, die nur Skalare oder affine Ausdrücke darstellen, keine weiteren interessanten Platzierungen erzeugt und der Zugriff auf sie bei der Bewertung irrelevant ist, da auf sie immer lokal zugegriffen werden kann, egal welche Platzierung die anderen Knoten besitzen.

4.2.4 Erzeugung aller möglicher Platzierungen

Die Erzeugung aller möglicher Platzierungen erfolgt entsprechend Algorithmus 7 in Abschnitt 3.2.5, wobei jedoch die Partitionierung nicht durchgeführt wird, da der Graph bereits sowohl in Richtung Quelle zum Ziel als auch von dem Ziel zur Quelle von lccp partitioniert worden ist. Bei der Implementierung wurde darauf geachtet, dass sowohl der Graph zwischen zwei Knoten nicht mehrere Kanten mit gleichem Gewicht als auch die Menge der möglichen Platzierungen eines Knotens keine gleichen Platzierungen mehrfach enthalten. Dies ist daher nötig, um die Laufzeit des Algorithmus zur Berechnung des Kommunikationsgraphen und zur Auswahl der

Platzierung zu verkürzen. Das Erkennen der Duplikate wird durch den Test auf Gleichheit zweier linearer Relationen realisiert, wie er für die Implementierung von linearen Abbildungen in Abschnitt 4.2.1 beschrieben ist.

4.2.5 Nachbearbeitung der möglichen Platzierungen

Als Nachbearbeitung der möglichen Platzierungen wurde nur die Erzeugung von HPF-konformen Platzierungen entsprechend Algorithmus 9 aus Abschnitt 3.3.2 implementiert. Die Erzeugung aller möglichen Kombinationen von Platzierungen wurde nicht implementiert, da dies zu viele Möglichkeiten für einen Knoten erzeugt und somit die Auswahl der Platzierungen zuviel Laufzeit in Anspruch nehmen würde.

4.2.6 Bewertung der Platzierungen

Zur Bewertung der Platzierungen werden für alle möglichen Kombinationen der Platzierung für die Knoten im kompaktifizierten, reduzierten Stelleninstanzgraphen die Kommunikationsrelationen berechnet. Dabei wird durch den Iterator `permutationVectorIterator` jede Permutation aufgezählt, für die der Kommunikationsgraph als `OIDepGraph` durch die Funktion `oiallocator::transformDeps` erzeugt wird.
Die Bewertung des Kommunikationsgraphen erfolgt durch die freie Funktion `evaluate`, die der in Abschnitt 3.4.1 definierten Kostenfunktion CostFunc : $G_{comm} \rightarrow \mathbb{N}$ entspricht. Als Fallbeispiel wurde die Kostenfunktion $\text{CostFunc}_{\texttt{ADAPTOR}}$ (Algorithmus 14) implementiert, der das Kostenmodell in Abschnitt 7.3 beschriebene zugrunde liegt.

4.3 Programm `genAlign`

Das Programm `genAlign` erzeugt für alle neu eingeführten Hilfsarrays eine `ALIGN`-Direktive. Ist ein n-dimensionales Array `A`, das für den Knoten v im reduzierten Stelleninstanzgraphen angelegt wird, auf ein m-dimensionales Template `T` mittels der Platzierung Φ_v verteilt, dann wird folgende `ALIGN`-Direktive erzeugt:

$\texttt{ALIGN A(i}_1\texttt{,...,i}_n\texttt{) WITH T(}p_1\texttt{,...,}p_m\texttt{)}$

Dabei sind $\texttt{i}_1,\ldots,\texttt{i}_n$ Variable, die die Dimensionen des Arrays `A` identifizieren. Für jede Dimension l kann p_l folgende Werte annehmen:

- Es wird $p_l = \texttt{i}_k$ gesetzt, wenn die k-te Dimension des Arrays `A` auf die l-te Dimension des Templates `T` verteilt ist. Es ist zu beachten, dass hier nicht die genaue Zuordnung der Arrayelemente auf die Template-Elemente durch $p_l = a \cdot \texttt{i}_k + q$ berücksichtigt wird (vgl. Abschnitt 2.1.4), da diese Zuordnung bereits bei der Definition des Arrays zur Codegenerierung berücksichtigt wurde.
- Es wird $p_l = \texttt{*}$ gesetzt, wenn das Array entlang der l-ten Dimension repliziert ist.

Als Beispiel wird eine 2-dimensionale Stelleninstanzmenge, für die das 2-dimensionale Array `A` angelegt wurde, durch die Platzierung $\Phi_v = (\Phi_{v_{OI}}, \Phi_{v_P})$ auf ein 2-dimensionales Template `T` wie folgt ausgerichtet:

$$\begin{aligned} \Phi_{v_{OI}}(i_1, i_2, m_\infty, m_c) &= (0, i_1 + 3, m_\infty, m_c) \\ \Phi_{v_P}(p_1, p_2, m_\infty, m_c) &= (0, p_2, m_\infty, m_c) \end{aligned}$$

Dabei wird die zweite Dimension $\mathtt{i}_2$ des Arrays entlang der ersten Dimension p_1 des Templates repliziert und die erste Dimension $\mathtt{i}_1$ des Arrays auf die zweite Dimension p_2 des Templates ausgerichtet, wobei die Zuordnung Arrayelement zu Template-Element durch $i \mapsto i + 3$ beschrieben wird. Aus der angegebenen Platzierung Φ_v wird folgende `ALIGN`-Direktive erzeugt:

```
ALIGN A(i1,i2) WITH T(*,i1)
```

5 Codegenerierung in HPF-Compilern

Ehe ein geeignetes Kostenmodell zur Bewertung der Platzierungen erstellt werden kann, muss zuerst auf die Arbeitsweise eines HPF-Compilers eingegangen werden. Da die hierfür eingesetzten Methoden zur Analyse der Kommunikation und zur Codegenerierung für die Kommunikation entscheidend die Performanz des Programms beeinflussen, sind sie im Kostenmodell zu berücksichtigen.

5.1 Analyse der Kommunikation und Darstellung im Compiler

Ein HPF-Compiler liest ein HPF-Programm ein, in dem die Datenverteilung der Arrays durch Direktiven, wie `ALIGN` und `DISTRIBUTE`, gegeben ist und die Schleifen markiert sind, die parallel auszuführen sind. Diese Direktiven benutzt der HPF-Compiler, um die Berechnungen auf die verschiedenen Prozessoren zu verteilen und um Code zu erzeugen, der die Kommunikation und Synchronisation übernimmt. Die Kommunikation kann für unterschiedliche Architekturmodelle erzeugt werden. Für Systeme mit gemeinsamem Speicher gestaltet sich die Codegenerierung einfach, da keine explizite Kommunikation benötigt wird. Dagegen wird für Systeme mit verteiltem Speicher, der Mehrzahl der heutigen Parallelrechner, meist expliziter Nachrichtenaustausch durch eine Implementierung des MPI-Standards [Mes95] verwendet, da viele Parallelrechner eine speziell auf sie abgestimmte Implementierung des MPI-Standards besitzen. Eine Auswahl von MPI-Implementierungen ist in Tabelle 6 zu finden [Uni05].
Für die in dieser Arbeit vorgestellte Methode ist hauptsächlich die Codegenerierung von parallelen Schleifen für verteilte Systeme mit explizitem Nachrichtenaustausch wichtig, weshalb auf die Analyse und Codegenerierung von parallelen `DO`-Schleifen eingegangen werden soll. Für einen parallelen Schleifensatz muss der Compiler folgende Schritte durchführen:

1. Die Iterationen des Schleifensatzes müssen auf die Prozessoren verteilt werden.

2. Für jedes Statement im Schleifensatz sind die Elemente zu bestimmen, die zum Ausführungsort der Iteration transportiert werden müssen, wobei für jedes Element der Kommunikationspartner zu ermitteln ist.

Zuerst bestimmt der Compiler für jedes verteilte Array `A`, das in dem Schleifensatz referenziert wird, seine „Owner"-Menge:

$$O_{\mathtt{A}}(p)$$

Diese Menge gibt für den Prozessor p an, welche Elemente von Array `A` er besitzt. Aus dieser Menge kann anhand der Zugriffsfunktionen der im parallelen Schleifensatz verwendeten Arrays entsprechend der Owner-Computes-Rule sowohl die Menge der Iterationen $C(p)$, die lokal auf Prozessor p ausgeführt werden, als auch die zu kommunizierenden Elemente $S(p, p')$, die von Prozessor p zu Prozessor p' zu schicken sind, bestimmt werden.
Die Eigenschaften der Zugriffsfunktionen bestimmen die Kommunikationsstruktur. Man spricht von einer irregulären Kommunikation, wenn die Zugriffsfunktionen keine affinen Funktionen sind, da sich deswegen die Mengen $C(p)$ und $S(p, p')$ nur zur Laufzeit bestimmen lassen. Zur Ausführung der Kommunikation wird hierfür meist das Inspector/Executor-Schema [Wol95] verwendet. Zur Laufzeit wird die Kommunikation in zwei Phasen durchgeführt:

1. Die erste Phase, die Inspector-Phase, analysiert, wie die benötigten Daten über die Prozessoren verteilt sind, und bestimmt aus dieser Information die Kommunikation und den Ausführungsort der Berechnung.

Bibliothek	Version	Platform	Firma
Alpha Data MPI	1.0	Alpha Data Systems z.B. (AD 164)	Alpha Data Parallel Systems ltd.
Appleseed	12.04.2001	MacOS 8.1 or later, IP	U. C. Los Angeles
ChaMPlon/Pro	1.0	z.B. TCP, SMP	MPI Software Technologie, Inc.
CHIMP/MPI	2.1.1c	SPARC, SGI, HP, IBM RS/6000	Edinburgh Parallel Computing Centre (EPCC)
CRI/EPCC	1.7a	Cray T3D/T3E	Cray Research Inc via EPCC
DISI	Juli 2000	Intel, Ethernet	Universita di Genova, Italy
Hitachi MPI	03-01-c	SR8000	Hitachi
HP MPI	1.7	HP-UX 11.0 or later	Hewlett-Packard, Co
IBM, AIX-MPI Library	2.1.0	RISC System/6000	IBM Corporation
LAM/MPI	7.0.2	POSIX	Indiana University
MP-MPICH	1.2.0	TCP/IP, SCI Interconnet	RWTH: Scalable Computing, Aachen
MPI for UNICOS	1.4	UNICOS 9.3	SGI
MPI-FM	2.1	Intel x86, Myrinet	University of Illinois, Concurrect Systems Architecture Group
MPI/Pro	1.6.5	TCP, InfiniBand, Myrinet	MPI Software Technology
MPI/SX	6.4	all SX series systems	NEC Corporation
MPIAP	1.6	Fujitsu AP1000	Australian National University - CAP Research Program
MPICH	1.2.4	z.B. TCP, Cray T3D	Argonne National Libary
MPICH/NT	0.8b	Intel x86 Windows NT	Mississippi State University
OS/390	Part of OS/390 V2R9	OS 390 Unix System Services	IBM Corporation
Paragon	R1.4	Paragon	Intel Corporation
Race-MPI	3.3	z.B. MCOS, Sun OS	Hughes Aircraft Co
ScaMPI	1.10.2	z.B. RedHat, Solaris	Scali AS
SGI Message Passing Toolkit	1.7	IRIX/MIPS, Linux/Altix	Computer Systems Business Unit, SGI
Sun MPI	Juli 2000	All Solaris/UltraSPARC	Sun Microsystems
T.MPI	Juli 2000	Telmat TN110, TN300 series	Telmat Multinode
TransMPI	1.0-C	all Transputers 32bits	PERIASTRON
WMPI	1.5.6	Windows NT und XP	Critical Software, SA
WMPI II	2.3.0	z.B. Windows NT, Debian	Critical Software, SA

Tabelle 6: Verschiedene MPI-Implementierungen (der Internet-Seite der MPI-Implementierung LAM entnommen [Uni05])

2. In der zweiten Phase, der Executor-Phase, werden Kommunikation und Berechnung durchgeführt.

Wenn die Zugriffsfunktionen affin sind, spricht man von einer regulären Kommunikation, da sich $C(p)$ und $S(p, p')$ zur Übersetzungszeit bestimmen lassen. Coelho, Germain und Pazat beschreiben eine Methode, wie diese Mengen berechnet werden können [CGP96], die im folgenden Beispiel für eine parallele Schleife informell demonstriert wird:

Beispiel 5.1 *Sei folgende parallele Schleife gegeben:*

```
!HPF$ INDEPENDENT
      DO i=1,N
S:       A(f(i)) = B(g_1(i)) + B(g_2(i))
      END DO
```

Die Elemente von `A` *und* `B`*, die auf Prozessor p liegen, lassen sich durch folgende Mengen beschreiben:*

$$
\begin{aligned}
O_{\mathtt{A}}(p) &= \{a \mid \mathtt{A}(a) \text{ liegt auf Prozessor } p\} \\
O_{\mathtt{B}}(p) &= \{b \mid \mathtt{B}(b) \text{ liegt auf Prozessor } p\}
\end{aligned}
$$

Entsprechend OCR werden auf dem Prozessor p nur die Iterationen ausgeführt, deren Ergebnisse in die Arrayelemente $O_{\mathtt{A}}(p)$ geschrieben werden. Diese Iterationen bilden die Menge:

$$C_S(p) = \{i \in \mathcal{IS}(S) \mid f(i) \in O_{\mathtt{A}}(p)\}$$

Um die Kommunikation zu bestimmen, muss für Prozessor p die Menge der Elemente von B, die er zur Berechnung benötigt, ermittelt werden:

$$V_{\mathtt{B}}(p) = \{b \mid \exists i \in C_S(p) : b = g_1(i) \lor b = g_2(i)\}$$

Die Kommunikation für den Schleifensatz lässt sich nun mittels des Durchschnitts der Mengen $O_{\mathtt{B}}$ und $V_{\mathtt{B}}$ darstellen. Prozessor p schickt die Elemente von `B`*, die in der Menge $S_{\mathtt{B}}(p, p')$ enthalten sind, an Prozessor p':*

$$S_{\mathtt{B}}(p, p') = O_B(p) \cap V_{\mathtt{B}}(p')$$

Aus diesen Mengen lässt sich dann das SPMD-Programm erzeugen. Mit dem generierten Code kann z.B. zuerst die Kommunikation und danach lokal die Berechnung durchgeführt werden, wie im folgenden Programm gezeigt ist:

```
send S(p,*)        ! Sende die Elemente von B an alle Prozessoren p',
                   ! für die S_B(p,p') ≠ ∅ gilt.
receive S(*,p)     ! Empfange alle benötigten Elemente von B von den
                   ! Prozessoren p', für die S_B(p,p') ≠ ∅ gilt.
                   ! Die empfangenen Daten sind im Array ReceiveB.
DO i in C(p)
   tmp1 =          ! Gilt g_1(i) ∈ O_B(p), dann Wert aus B(g_1(i)) holen,
                   ! sonst aus Puffer ReceiveB holen.
   tmp2 =          ! Gilt g_2(i) ∈ O_B(p), dann Wert aus B(g_2(i)) holen,
                   ! sonst aus Puffer ReceiveB holen.
   A(f(i)) = tmp1 + tmp2
END DO
```

Wie man an diesem Beispiel sehen kann, eignen sich die Mengen $O_A(p), C_S(p)$ und $S(p,p')$ zur Modellierung der Kommunikation und der Partitionierung der Berechnungen. Zur Repräsentation dieser Mengen im HPF-Compiler gibt es verschiedene Datenstrukturen, die sich sowohl in der Genauigkeit der Darstellung als auch in der Komplexität der Operationen unterscheiden. Dafür eignen sich z.B. Arraysektionen oder Polyeder. Die gewählte Datenstruktur sollte folgende Eigenschaften besitzen:

- leichte Implementierung
- Modellierbarkeit aller möglichen Mengen
- Abschluss unter allgemeinen Operatoren, wie Vereinigung, Durchschnitt und Transformation durch affine Abbildungen
- möglichst einfache Erzeugung von Code aus der Darstellung der Mengen
- möglichst effizienter resultierender Code

Arraysektionen Eine Arraysektion ist ein Teil eines Arrays, das durch Array-Triplets beschrieben werden kann, die bereits in Abschnitt 2.1.6 eingeführt wurden. Diese einfache Technik wurde schnell in kommerzielle Compiler übernommen. Auch der Compiler ADAPTOR [Bra00] verwendet Arraysektionen zur Darstellung der zu kommunizierenden Elemente. Die in HPF erlaubten Verteilungen lassen sich als Arraysektionen darstellen, wie folgendes Beispiel zeigt:

Beispiel 5.2 *Die Verteilung für die beiden Arrays `A(0:N)` und `B(0:N)` in Beispiel 5.1 sind gegeben durch:*

```
!HPF$ DISTRIBUTE A(BLOCK(k))
!HPF$ DISTRIBUTE B(CYCLIC)
```

Wenn beide Arrays auf n_{cpu} Prozessoren verteilt werden, dann besitzt der Prozessor p mit $0 \leq p < n_{cpu}$ die Arraysektionen:

$$\begin{aligned} O_A(p) &= [p \cdot k \ : \ \min(p \cdot k + k - 1, n) \ : \ 1] \\ O_B(p) &= [p \ : \ n \ : \ n_{cpu}] \end{aligned}$$

Da die Arraysektionen unter Durchschnitt und affiner Transformation abgeschlossen sind, lassen sich auch die Mengen $C_S(p)$ und $S(p,p')$ durch Arraysektionen darstellen. Die Implementierung dieser Operationen gestaltet sich einfach, und auch das Erzeugen eines Schleifensatzes aus einer Arraysektion ist leicht, wie in Abschnitt 2.1.6 gezeigt. So relativ einfach die Handhabung von Arraysektionen ist, so besitzt sie doch einen entscheidenden Nachteil: Es können nur Arraybereiche beschrieben werden, die rechteckige Form besitzen. Andere Arraybereiche, die z.B. dreieckige Form besitzen, sind nicht mit Arraysektionen darstellbar, da die einzelnen Array-Triplets nicht voneinander abhängig sein dürfen.

Polyeder Eine genaue Darstellung der Owner- und Kommunikationsmengen sind Polyeder, die in Abschnitt 2.3 eingeführt wurden. Zwar lassen sich hiermit Arraybereiche beliebiger Form darstellen, jedoch ist die Implementierung komplexer, da Operationen auf Polyedern schwieriger zu implementieren sind und die Erzeugung von effizientem Code, der ein Polyeder aufzählt, kompliziert ist [GLW98, QRW00]. Daher wurde diese Darstellungsform nur in sehr wenige HPF-Compiler integriert, wie z.B. in den dHPF-Compiler [AMC98, AMC01].

5.2 Codeerzeugung

Im Allgemeinen ist bei der Erzeugung von Kommunikationscode zu beachten:

- Eine Kommunikation kann um Größenordnungen langsamer sein als die Berechnung, abhängig vom verwendeten Kommunikationsnetz.
- Die Kosten zur Bereitstellung der Daten und das Initiieren der Kommunikation (startup-Zeit), sind meist höher als die Zeit für die Übertragung eines Arrayelementes.

Daher muss es Ziel der Optimierung der Kommunikation sein, möglichst wenige Nachrichten zu erzeugen. Man unterscheidet zwischen globaler und lokaler Optimierung. Die Optimierung auf globaler Ebene analysiert die Kommunikation über den gesamten Code auf intra- und interprozeduraler Ebene. Hier kann z.B. erkannt werden, dass zwei parallele Schleifen die gleiche Kommunikation benötigen, so dass die Daten nicht mehrfach übertragen werden müssen. Die Optimierung auf lokaler Ebene entfernt redundante Nachrichten und fasst mehrere Nachrichten zwischen zwei gleichen Kommunikationspartnern zu einer zusammen. Die beiden wichtigsten Formen der Optimierung auf lokaler Ebene sind:

Message Vectorisation: Sind für ein Array mehrere Elemente zwischen zwei Prozessoren auszutauschen, dann werden diese Elemente in eine Nachricht gepackt, anstatt für jedes Element eine eigene Nachricht zu verschicken. Damit kann die Kommunikation aus der Schleife herausgezogen werden, wodurch die Kommunikationsfrequenz reduziert wird. Um die Vektorisierung der Kommunikation als Optimierung durchführen zu können, hat der Compiler die Menge der Daten zu bestimmen, die zwischen jedem Paar von Prozessoren auszutauschen sind.

Message Coalescing: Müssen Arrayelemente mehrerer unteschiedlicher Arrays zwischen zwei Prozessoren ausgetauscht werden, dann wird die gesamte Kommunikation als eine Nachricht verschickt, anstatt für jedes Array eine eigene Nachricht zu versenden. Diese Zusammenfassung benötigt die Vereinigung von Kommunikationsmengen und mitunter komplizierte Techniken, um bei der Berechnung effizient auf die im Puffer liegenden Daten zugreifen zu können.

5.3 Codegenerierung durch ADAPTOR

Wie bereits in Abschnitt 2.2 beschrieben, erzeugt ADAPTOR ein Fortran77-Programm aus dem HPF-Quellcode, in dem die Kommunikation durch Aufrufe der DALIB [Bra00] realisiert sind. Für alle wichtigen Objekte im Programm, wie Arrays, Templates, Prozessorarrays, Arraysektionen und `ALIGN`-Direktiven, werden Deskriptoren angelegt, die alle nötigen Informationen über diese Objekte enthalten. So speichert ein Deskriptor eines Arrays dessen Größe und Verteilung auf die Prozessoren. Die Arraysektionen werden intern mittels Array-Triplets realisiert, wobei z.B. der Schnitt von Arraysektionen und die Berechnung von Owner-Mengen für Arrays implementiert sind.

5.3.1 DALIB-Funktionen

Bevor auf die Codegenerierung von ADAPTOR mit einem konkreten Beispiel eingegangen werden kann, sollen zuerst die wichtigsten Aufrufe der DALIB angesprochen werden, die zur

Berechnung lokaler Grenzen, zur Definition von Arraysektionen und zur Durchführung der Kommunikation dienen.

Bestimmung der lokalen Grenzen Zur Berechnung der lokalen Unter- und Obergrenzen im Programm wird die Prozedur `DALIB_array_my_slice` aufgerufen. Sie besitzt folgende Argumente:

```
DALIB_array_my_slice(DSPArray, dimension, globUntergrenze, globObergrenze,
                     lokUntergrenze, lokObergrenze)
```

Diese Prozedur berechnet aus der Verteilung der Dimension `dimension` des durch den Deskriptor `DSPArray` beschriebenen Arrays und aus den globalen Unter- und Obergrenzen die lokalen Grenzen: `lokUntergrenze` und `lokObergrenze`. Dabei geben `globUntergrenze` und `globObergrenze` den Bereich des Arrays an, der im Schleifensatz des HPF-Programms referenziert wird.

Definition einer Arraysektion Die Prozedur `DALIB_section_create` erzeugt eine Beschreibung einer Sektion eines Arrays als Array-Triplet. Die Prozedur hat für eine 2-dimensionale Arraysektion folgende Argumente:

$\texttt{DALIB\_section\_create(DSPSection, DSPArray, typ}_1\texttt{, lb}_1\texttt{, up}_1\texttt{, st}_1\texttt{,}$
$\qquad\qquad\qquad\qquad\qquad\qquad\texttt{typ}_2\texttt{, lb}_2\texttt{, up}_2\texttt{, st}_2\texttt{)}$

Mit dieser Prozedur wird eine Beschreibung `DSPSection` des durch den Deskriptor `DSPArray` repräsentierten Arrays für eine 2-dimensionale Arraysektion erzeugt, die sich folgendermaßen in Array-Triplet-Notation darstellen lässt:

$\texttt{Array(lb}_1\texttt{:up}_1\texttt{:step}_1\texttt{, lb}_2\texttt{:up}_2\texttt{:step}_2\texttt{)}$

Für jede Dimension i gibt $\texttt{typ}_i$ an, ob die hier definierte Arraysektion nur einen Punkt ($\texttt{typ}_i = 0$) oder mehrere Punkte ($\texttt{typ}_i = 1$) umfasst.

Durchführung einer Kommunikation Die Prozedur `DALIB_assign`, die die Kommunikation durchführt, besitzt folgende Argumente:

```
DALIB_assign(DSPZiel, DSPQuelle)
```

Diese Prozedur kopiert die Elemente der Arraysektion, die durch den Deskriptor `DSPQuelle` beschrieben ist, in die Arraysektion, die durch den Deskriptor `DSPZiel` definiert ist, wobei nichtlokale Arrays von dem entsprechenden Prozessor angefordert werden. Als Vereinfachung können auch Arraydeskriptoren als Argumente der Prozedur verwendet werden, damit keine Arraysektionen angelegt werden müssen, die das gesamte Array beschreiben.

5.3.2 Beispiel Codegenerierung

Im Folgenden soll anhand einiger Variationen des Programms 5.1 auf die Codegenerierung für parallele Schleifen von ADAPTOR eingegangen werden, wobei der von ADAPTOR generierte Code an manchen Stellen zur leichteren Lesbarkeit modifiziert wurde. Dabei wurde sowohl die Linearisierung der 2-dimensionalen Arrays rückgängig gemacht, als auch die dynamisch allozierten temporären Arrays als statische Arrays mit der entsprechenden Verteilung deklariert.

Ohne Kommunikation

Programm 5.1

```
!HPF$ TEMPLATE T(1:N,1:N)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK)
!HPF$ ALIGN ALIGN A(i,j) WITH T(i,j)
!HPF$ ALIGN B(i,j) WITH T(i,j)
REAL, DIMENSION(1:N,1:N):: A,B

!HPF$ INDEPENDENT
DO j = 10,N
   !HPF$ INDEPENDENT
   DO i = 10,N
      A(i,j) = B(i,j) * B(i,j)
   END DO
END DO
```

Im Programm 5.1 sind zwei Arrays `A` und `B` über ein 2-dimensionales Prozessorfeld verteilt. Aufgrund der angegebenen Verteilung gilt, dass die Arrayelemente `A(i,j)` und `B(i,j)` auf dem gleichen Prozessor liegen.
ADAPTOR generiert folgenden Code:

```
CALL DALIB_array_my_slice(T_DSP,2,10,N,T_START2,T_STOP2)
CALL DALIB_array_my_slice(T_DSP,1,10,N,T_START1,T_STOP1)
DO J=T_START2,T_STOP2
   DO I=T_START1,T_STOP1
      A(I,J) = B(I,J)*B(I,J)
   END DO
END DO
```

Da keine Kommunikation erzeugt werden muss, genügt es, den Schleifensatz nur mit neuen Grenzen zu versehen. Die lokalen Grenzen der äußeren Schleife mit der Zählvariable `j` werden vom ersten Aufruf der Prozedur `DALIB_array_my_slice` aus den Grenzen der äußeren Schleife des Quellprogramms und aus der Verteilung der zweiten Dimension des Templates `T` berechnet, da das Array `A` an das Template `T` ausgerichetet ist und `j` im Programm 5.1 die zweite Dimension von `A` indiziert. Ebenso werden die Grenzen der zweiten Schleife durch den zweiten Aufruf von `DALIB_array_my_slice` bestimmt.

Einfache Kommunikation Im Programm 5.2 ist das Berechnungsstatement so modifiziert, dass nicht auf `B(i,j)` sondern auf `B(i-8,j)` und `B(i-7,j)` zugegriffen wird, so dass Kommunikation erzeugt werden muss.

Programm 5.2

```
!HPF$ INDEPENDENT
DO j = 10,N
   !HPF$ INDEPENDENT
   DO i = 10,N
```

```
      A(i,j) = B(i-8,j) * B(i-7,j)
   END DO
END DO
```

Der Compiler ADAPTOR legt zwei neue temporäre Arrays an:

```
REAL, DIMENSION(10:N,10:N):: TMP0,TMP1
```

Sie enthalten die kommunizierten Elemente und werden wie folgt verteilt:

```
!HPF$ ALIGN TMP0(i,j) WITH T(i,j)
!HPF$ ALIGN TMP1(i,j) WITH T(i,j)
```

Für die Schleife im Programm 5.2 wird folgender Code erzeugt:

```
! Kommunikation für TMP0(i,j) = B(i-8,j)
CALL DALIB_section_create(ISEC_DSP2,B_DSP,1,10-8,N-8,1,1,10,N,1)
CALL DALIB_assign(TMP0_DSP,ISEC_DSP2)
...
! Kommunikation für TMP1(i,j) = B(i-7,j)
CALL DALIB_section_create(ISEC_DSP2,B_DSP,1, 10-7,N-7,1,1,10,N,1)
CALL DALIB_assign(TMP1_DSP,ISEC_DSP2)
...
CALL DALIB_array_my_slice(T_DSP,2,10,N,T_START2,T_STOP2)
CALL DALIB_array_my_slice(T_DSP,1,10,N,T_START1,T_STOP1)
DO J=T_START2,T_STOP2
   DO I=T_START1,T_STOP1
      A(I,J) = TMP0(I,J)*TMP1(I,J)
   END DO
END DO
```

Die Kommunikation für `B(i-8,j)` wird in zwei Schritten durchgeführt. Zuerst wird mit dem Aufruf von `DALIB_section_create` die Arraysektion von `B` definiert, die in das Array `TMP0` kopiert werden soll. Diese Arraysektion lässt sich in Array-Triplet-Notation als `B(10-8:N-8:1, 10:N:1)` schreiben. Dann führt die Prozedur `DALIB_assign` die Kommunikation durch. Diese Zuweisung lässt sich in Fortran90-Notation schreiben als:

```
TMP(10:N:1, 10:N:1) = B(10-8:N-8:1,10:N:1)
```

Analog wird die Kommunikation für das temporäre Array `TMP1` durchgeführt.
Die Berechnung der lokalen Grenzen erfolgt genau so, wie bereits für das Programm 5.3 gezeigt wurde. In der Schleife wurden die Zugriffe `B(i-8,j)` durch `TMP0(i,j)` und `B(i-7,j)` durch `TMP1(i,j)` ersetzt.

Dreieck-Kommunikation Im folgenden Programm ist die obere Grenze der zweiten Schleife im Programm 5.2 durch `j` ersetzt. Somit zählt der Schleifensatz ein Dreieck auf, das für `N` = 15 in Abbildung 13 graphisch dargestellt ist.

Programm 5.3

```
!HPF$ INDEPENDENT
DO j = 10,N
   !HPF$ INDEPENDENT
   DO i = 10,j
      A(i,j) = B(i-8,j) * B(i,j-7)
   END DO
END DO
```

ADAPTOR erzeugt wie oben zwei neue temporäre Arrays, die die Elemente von `B(i-8,j)` und `B(i-7,j)` enthalten:

```
! Kommunikation für TMP0(i,j) = B(i-8,j)
DO J=10,N
   CALL DALIB_section_create(ISEC_DSP1,TMP0_DSP,1, 10,J,1,0,J,0,0)
   CALL DALIB_section_create(ISEC_DSP2,B_DSP,1, 10-8,J-8,1,0,J,0,0)
   CALL DALIB_assign(ISEC_DSP1,ISEC_DSP2)
   ...
END DO

! Kommunikation für TMP1(i,j) = B(i-7,j)
! Analog zur Kommunikation für TMP0
...
CALL DALIB_array_my_slice(T_DSP,2,10,N,T_START2,T_STOP2)
DO J=T_START2,T_STOP2
   CALL DALIB_array_my_slice (T_DSP,1,10,J,T_START1,T_STOP1)
   DO I=T_START1,T_STOP1
      A(I,J) = TMP0(I,J)*TMP1(I,J)
   END DO
END DO
```

Die Menge der zu kommunizierenden Elemente kann jedoch nicht durch eine Arraysektion dargestellt werden, da ein nicht rechteckiger Bereich des Arrays `B` zu kommunizieren ist. Daher partitioniert ADAPTOR die Menge in $N - 10$ Arraysektionen, die durch eine Schleife aufgezählt werden müssen, wie man in folgendem Code sehen kann. Für `N` $= 15$ sind diese Arraysektionen als Rechtecke um die einzelnen Indexpunkte in Abbildung 13 graphisch dargestellt.

```
! Kommunikation für TMP0(i,j) = B(i-8,j)
DO J=10,N
   CALL DALIB_section_create(ISEC_DSP1,TMP0_DSP,1, 10,J,1,0,J,0,0)
   CALL DALIB_section_create(ISEC_DSP2,B_DSP,1, 10-8,J-8,1,0,J,0,0)
   CALL DALIB_assign(ISEC_DSP1,ISEC_DSP2)
   ...
END DO

! Kommunikation für TMP1(i,j) = B(i-7,j)
! Analog zur Kommunikation für TMP0
```

```
...
CALL DALIB_array_my_slice(T_DSP,2,10,N,T_START2,T_STOP2)
DO J=T_START2,T_STOP2
   CALL DALIB_array_my_slice (T_DSP,1,10,J,T_START1,T_STOP1)
   DO I=T_START1,T_STOP1
      A(I,J) = TMP0(I,J)*TMP1(I,J)
   END DO
END DO
```

Die Kommunikation für `TMP0` wird durch die erste Schleife in obiger Ausgabe von ADAPTOR durchgeführt. Für jedes $j \in \{10, \ldots, N\}$ erzeugt `DALIB_section_create` die Arraysektionen `TMP0(10:j:1, j)` und `B(10-8:j-8:1, j)`, für die dann `DALIB_assign` die Zuweisung durchführt, die sich wie folgt in Fortran90 Notation schreiben lässt:

```
TMP0(10:j:1, j) = B(10-8:j-8:1, j)
```

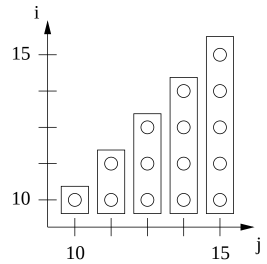


Abbildung 13: Der Indexraum mit den Arraysektionen, der vom Schleifensatz aus Beispiel 5.3 aufgezählt wird

Die Berechnung der lokalen Grenzen der inneren Schleife ist für jede Iteration der äußeren Schleife zu wiederholen, da die innere Schleife von der äußeren abhängt.

5.4 Schlussfolgerungen

Soll ein Kostenmodell für einen HPF-Compiler entworfen werden, genügt es nicht, nur das Kommunikationsvolumen als dominierende Kosten einer Kommunikation zu betrachten, da mehrere zusätzliche Faktoren die Kosten beeinflussen, die unabhängig vom Kommunikationsvolumen sind.
Ein wichtiger Faktor für die Kommunikationskosten ist die Fähigkeit der Datenstrukturen im Compiler, die Mengen der zu kommunizierenden Elemente geschlossen darstellen zu können. Wie bereits in Abschnitt 5.1 beschrieben, können einige Mengen nicht durch Arraysektionen dargestellt werden. Dies ist bei Programm 5.3 zu sehen. Daher können zwei Programme, die das gleiche Kommunikationsvolumen haben, entsprechend der verwendeten Kommunikationsstruktur unterschiedlich lange Laufzeiten besitzen. Auch Optimierungen, wie das Zusammenfassen von Kommunikation, sind compilerabhängig.
Daher ist es sinnvoll, einen Benchmark zu entwickeln, der verschiedene Klassen an Kommunikation für den HPF-Compiler bewertet. Aus dem Ergebnis des Benchmarks werden die Werte für das Kostenmodell gewonnen.

6 Benchmark

Um den Benchmark für einen HPF-Compiler vorzustellen, werden in Abschnitt 6.1 die Konzeption und die verschiedenen Testfälle des Benchmarks und in Abschnitt 6.2 die für jeden Testfall gemessenen Parameter dargestellt. Anschließend wird in Abschnitt 6.3 die Auswertung der Messergebnisse des Benchmarks für den HPF-Compiler ADAPTOR und den hpcLine-Parallelrechner des Lehrstuhls für Programmierung an der Universität Passau gezeigt.

6.1 Konzeption

Da die Kosten für eine Kommunikation nicht nur von der Menge der zu kommunizierenden Elemente, sondern auch von dem Kommunikationsmuster abhängen, wie in Abschnitt 5.1 gezeigt, wurden Testfälle für verschiedene Kommunikationsmuster entwickelt, deren Laufzeit für verschiedene Arraygrößen und für eine unterschiedliche Anzahl von Prozessoren gemessen wurden.
Grundsätzlich entspricht ein Testfall einem n_T-dimensionalen Schleifensatz, dessen Rumpf aus folgender Zuweisung besteht:

$$\mathtt{T}(\mathbf{i}) = \mathtt{S}(f(\mathbf{i}))$$

Dabei ist `T` ein n_T-dimensionales Array, `S` ein n_S-dimensionales Array mit der Zugriffsfunktion $f : \mathbb{Z}^{n_T} \rightarrow \mathbb{Z}^{n_S}$ und $\mathbf{i}$ der durch den Schleifensatz gegeben Indexvektor. Pro Dimension d enthalten die Arrays N_d Elemente. Die k-te Schleife des Schleifensatzes hat folgende Form:

$\mathtt{DO\ i}_k\mathtt{\ =\ 1\ +\ shift}_k\mathtt{,\ N}_k$

Der Parameter $\mathtt{shift}_k$ schränkt die zu durchlaufenden Elemente in der Dimension k ein. Er wird z.B. für den Testfall $\mathtt{shift}_{1D}$ benötigt, weil dadurch, dass man $\mathtt{shift}_1\ \mathtt{=}\ \mathtt{3}$ setzt, für die Zuweisung $\mathtt{T(i}_1\mathtt{)\ =\ S(i}_1\mathtt{-3)}$ garantiert wird, dass $\mathtt{i}_1\mathtt{-3}$ keine Werte außerhalb des Datenraums von `S` referenziert. Damit für eine Arraygröße und eine Iterationsmenge unterschiedliche Kommunikationsmuster vergleichbar sind, wird der Parameter $\mathtt{shift}_k$ bei allen Testfällen verwendet.
Der Benchmark beschränkt sich auf Messungen für 1- und 2-dimensionale Arrays. Die verwendeten Arraygrößen und die Werte des Parameters `shift` sind für die 1- und 2-dimensionalen Testfälle in Tabelle 7 aufgelistet.

	Arraygröße
1-dim.	$N_1 \in \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536\}$
	$shift_1 \in \{0, 1, 5, 128, 1024\}$
2-dim	$N_1 = N_2$ mit
	$N_1 \in \{128, 256, 512, 1024, 2048\}$
	$shift_1 = 0$ und
	$shift_2 \in \{0, 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 704, 768, 832, 896, 960, 1024\}$

Tabelle 7: Die verschiedenen Arraygrößen und Werte des Parameters `shift` für 1- und 2-dimensionale Testfälle

Die Testfälle lassen sich in vier Gruppen gliedern:

Gruppe 1 Hier sind die Testfälle zusammengefasst, bei denen T und S gleich groß sind und somit $n_T = n_S$ gilt. Diese Testfälle, die sich durch eine Variation der Zugriffsfunktion f ergeben, werden in Abschnitt 6.1.1 aufgelistet, ihre Auswertung ist in Abschnitt 6.3.2 zu finden.

Gruppe 2 Die zweite Gruppe enthält alle die Testfälle, bei denen T und S von unterschiedlicher Dimensionalität sind, so dass die Kommunikation zwischen zwei unterschiedlich verteilten Arrays stattfindet. Eine genaue Beschreibung findet sich in Abschnitt 6.1.2, die Auswertung in Abschnitt 6.3.3.

Gruppe 3 Die dritte Gruppe enthält diejenigen Testfälle, die die Laufzeit der Berechnungen wie + oder * messen. Die Auflistung der Testfälle erfolgt in Abschnitt 6.1.3, die Auswertung in Abschnitt 6.3.4.

Gruppe 4 Die letzte Gruppe enthält nur einen Testfall, um die Laufzeit nicht rechteckiger Iterationsräume zu bestimmen. Dazu wurde ein Testfall aus der ersten Gruppe so umgeschrieben, dass sein rechteckiger Iterationsraum in zwei Schleifen aufgeteilt wurde, die jeweils das untere bzw. das obere Dreieck aufzählen. Eine Beschreibung dieses Testfalls findet sich in Abschnitt 6.1.4, die zugehörige Auswertung in Abschnitt 6.3.5.

6.1.1 Gruppe 1

Die erste Gruppe, die die Kommunikation zwischen zwei gleichverteilten Arrays bewertet, besteht aus Testfällen für 1-dimensionale Arrays ($n_T = 1$) und 2-dimensionale Arrays ($n_T = 2$). Für diese ist der Code der Schleifensätze in Abbildung 14 gezeigt.

```
! 2-dimensionale Testfälle:
INTEGER, DIMENSION (1:N1,1:N2) :: S, T
!HPF$ DISTRIBUTE S(BLOCK,BLOCK)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK)
DO i1=1+shift1,N1
    DO i2=1+shift2,N2
       T(i2,i2) = S(f(i2,i1))
    END DO
END DO
```

```
! 1-dimensionale Testfälle:
INTEGER, DIMENSION (1:N1) :: S, T
!HPF$ DISTRIBUTE S(BLOCK)
!HPF$ DISTRIBUTE T(BLOCK)
DO  i1=1+shift1,N1
    T(i1) = S(f(i1))
END DO
```

Abbildung 14: Die Schleifensätze der Testfälle aus der ersten Gruppe

Durch die Variation der Zugriffsfunktion $\texttt{f(i}_2\texttt{,i}_1\texttt{)}$ bzw. $\texttt{f(i}_1\texttt{)}$ von S werden verschiedene Kommunikationsmuster erzeugt. Die Testfälle mit ihren zugehörigen Zugriffsfunktionen sind in Tabelle 8 aufgelistet.
Die Testfälle $\texttt{local}_{1D}$ bzw. $\texttt{local}_{2D}$ erzeugen keine Kommunikation, da sie die Daten nur lokal kopieren. Dagegen stellen die Testfälle $\texttt{shift}_{1D}$ bzw. $\texttt{shift}_{2D}$ ein Kommunikationsmuster dar, bei dem die Werte in einer Dimension um eine Konstante verschoben werden. Der Testfall $\texttt{perm}_{2D}$ beschreibt als Kommunikationsmuster die Permutation des Arrays S. (Im zweidimensionalen Fall ist als Permutation nur die Transposition des Arrays S möglich.) Die letzten

Testfall	2-dimensional	1-dimensional
local	$f_{\texttt{local}_{2D}}(i,j) = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \cdot \begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}0\\ 0\end{pmatrix}$	$f_{\texttt{local}_{1D}}(i) = i$
shift	$f_{\texttt{shift}_{2D}}(i,j) = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \cdot \begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}\texttt{shift}_2\\ 0\end{pmatrix}$	$f_{\texttt{shift}_{1D}}(i) = i + \texttt{shift}_1$
$\texttt{perm}_{2D}$	$f_{\texttt{perm}_{2D}}(i,j) = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \cdot \begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}0\\ 0\end{pmatrix}$	-
broadCast	$f_{\texttt{broadCast}}(i,j) = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} \cdot \begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}\texttt{shift}_2\\ 1\end{pmatrix}$	-
general	$f_{\texttt{general}_{2D}}(i,j) = lookUpTable(i,j)$	$f_{\texttt{general}_{1D}}(i) = lookUpTable(i)$

Tabelle 8: Die verschiedenen Klassen der Zugriffsfunktion für 1- und 2-dimensionale Testfälle

beiden Testfälle $\texttt{general}_{1D}$ bzw. $\texttt{general}_{2D}$ messen dadurch die Berechnung der Kommunikation zur Laufzeit, dass die Zugriffsfunktion durch indirekte Adressierung über die Tabelle *lookUpTable* realisiert ist. Dabei enspricht das Kommunikationsmuster dem Testfall $\texttt{shift}_{1D}$ bzw. $\texttt{shift}_{2D}$.

6.1.2 Gruppe 2

Die zweite Gruppe umfasst die Testfälle zur Bewertung der Kommunikation zwischen zwei unterschiedlich verteilten Arrays. Dazu werden Elemente eines 2-dimensionalen Arrays in ein 1-dimensionales Array und Elemente eines 1-dimensionalen Arrays in ein 2-dimensionales Array kopiert. Der Code für die beiden Versionen der Schleifensätze ist in Abbildung 15 angegeben.

```
! Kommunikation 1D-Array -> 2D-Array
DO  i1=1+shift1, N1
    T(f(i1)) = S(i1)
END DO
```

```
! Kommunikation 2D-Array -> 1D-Array
DO  i1=1+shift1, N1
    T(i1) = S(f(i1))
END DO
```

Abbildung 15: Die Schleifensätze der Testfälle aus der zweiten Gruppe

Als Zugriffsfunktionen wurden zum einen die Funktion

$$f_1(i) = \begin{pmatrix}i\\ 1\end{pmatrix}$$

gewählt, die auf die erste Zeile des 2-dimensionalen Arrays zugreift, und zum anderen die Funktion

$$f_{diag}(i) = \begin{pmatrix}i\\ i\end{pmatrix}$$

die auf die Diagonale des 2-dimensionalen Arrays zugreift. Daraus ergeben sich folgende Testfälle:

	1D-Array -> 2D-Array	2D-Array -> 1D-Array
$f_1(i)$	$\texttt{copy1D2D}_1$	$\texttt{copy2D1D}_1$
$f_{diag}(i)$	$\texttt{copy1D2D}_{diag}$	$\texttt{copy2D1D}_{diag}$

6.1.3 Gruppe 3

Diese Gruppe fasst diejenigen Testfälle zusammen, die die Laufzeit für die binären Grundrechenoperationen vom Datentyp `REAL` messen. Dabei kommt der 2-dimensionale Schleifensatz der ersten Gruppe aus Abbildung 14 zur Anwendung. In diesem Schleifensatz sind für die Testfälle die entsprechenden Statements jeweils als Rumpf eingesetzt, die in Tabelle 9 aufgelistet sind. Die Messung dieser Testfälle erfolgte auf einem Prozessor.

Testfall	Statement
`calcPlus`	`T(i,j) = S(i,j) + 5.87328344`
`calcMinus`	`T(i,j) = S(i,j) - 5.87328344`
`calcMult`	`T(i,j) = S(i,j) * 5.87328344`
`calcDiv`	`T(i,j) = S(i,j) / 5.87328344`
`calcExp`	`T(i,j) = S(i,j) ** 5.87328344`

Tabelle 9: Testfälle der 3. Gruppe

6.1.4 Gruppe 4

Die letzte Gruppe besteht nur aus dem Testfall `dreieckGrenzen`, bei dem der rechteckige Iterationsraum des Testfalls $\texttt{shift}_{2D}$ auf zwei Schleifensätze aufgeteilt wurde, die beide einen dreieckigen Bereich aufzählen. Damit kann bewertet werden, wie gut der Compiler mit komplexeren Schleifengrenzen und Indexausdrücken umgehen kann. Die Abbildung 16 zeigt den Code für diesen Testfall.

```
DO i1 = 1 + shift1, N1
   DO i2 = 1 + shift2, i1-1
      T(i2, i1) = S(i2 + shift2, i1 + shift1)
   END DO
END DO
DO i1=1+shift1,N1
   DO i2 = MAX(i1, 1+shift2), N2
      T(i2, i1) = S(i2 + shift2, i1 + shift1)
   END DO
END DO
```

Abbildung 16: Schleife für den Testfall `dreieckGrenzen`

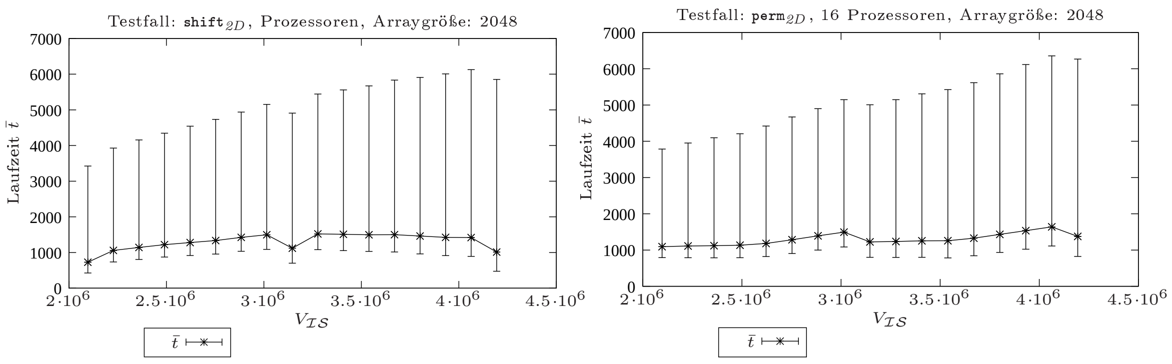
6.2 Messverfahren

Für jede Kombination von der in Tabelle 7 angegebenen Arraygröße und von dem Wert Parameter `shift` wurde die Laufzeit in Millisekunden und das Kommunikationsvolumen gemessen.

Um bei der Zeitmessung Störungen z.B. durch das Betriebssystem eliminieren zu können, wurde die Messung 10-mal wiederholt. Als Messwerte für die Laufzeit wurden das Maximum $t_{\max}$, das Minimum $t_{\min}$ und der Mittelwert $\bar{t}$ dieser 10 Messungen gespeichert.
Zur Charakterisierung der Kommmunikation wurde neben dem Volumen des Indexraums $V_{\mathcal{IS}}$ die Gesamtanzahl der kommunizierten Arrayelemente V_{comm}, die maximale Anzahl der Elemente V_{comm,p^*}, die ein Prozessor verschickt hat, und die maximale Anzahl der Elemente $V_{comm,p,p'}$, die zwischen zwei Prozessoren ausgetauscht wurden, gemessen. Um die Verteilung des Arrays beurteilen zu können, wurde zusätzlich die Blockgröße bestimmt.

6.3 Messungen mit ADAPTOR 7.1 mit SCI-Netz

Der Benchmark wurde auf dem hpcLine-Parallelrechner des Lehrstuhls für Programmierung an der Universität Passau durchgeführt. Dieser Parallelrechner ist ein Linux-Cluster bestehend aus 32 Dual-Pentium III 1.0 GHz-Prozessor-Knoten mit 512 MB Arbeitsspeicher pro Knoten. Als Kommunikationsnetz zwischen den Knoten stehen ein SCI-Netz, das die Knoten in einem 2-dimensionalen Torus verbindet, und ein schnelles Ethernet mit Switch zur Verfügung.
Der Benchmark für den HPF-Compiler ADAPTOR 7.1 wurde unter Verwendung des SCI-Netzes durchgeführt, wobei der durch ADAPTOR erzeugte Fotran77-Code mit dem Compiler `g77` übersetzt wurde, bei dem keine Optimierungen eingeschaltet waren. Als MPI-Bibliothek wurde die Implementierung von ScaMPI verwendet.
Bei der Durchführung des Benchmarks traten keine Schwierigkeiten auf, außer dass die Testfälle `general`$_{2D}$ und `dreieckGrenzen` hohe Laufzeiten aufwiesen und daher vor dem regulären Ende abgebrochen werden mussten.
Zur Kontrolle der Messungen wurde für jede Arraygröße der Mittelwert der Laufzeit $\bar{t}$ mit der Abweichung von $t_{\min}$ und $t_{\max}$ gegenüber $V_{\mathcal{IS}}$ aufgetragen. In Abbildung 17 sind die Graphen für die Testfälle `shift`$_{2D}$ und `perm`$_{2D}$, die für die Arraygröße 2048×2048 auf 16 Prozessoren durchgeführt wurden, dargestellt.



(a) Das Laufzeitverhalten des Testfalls `shift`$_{2D}$ (b) Das Laufzeitverhalten des Testfalls `perm`$_{2D}$

Abbildung 17: Das Laufzeitverhalten der Testfälle `shift`$_{2D}$ und `perm`$_{2D}$ mit Arraygröße 2048×2048 auf 16 Prozessoren

Dabei fällt die große Abweichung zwischen dem Minimum und dem Maximum auf, wobei der Mittelwert nahe am Minimum liegt. Das lässt darauf schließen, dass die Ergebnisse der Lauf-

zeitmessungen nicht über den gesamten Bereich verstreut sind, sondern ein einziger Messwert stark vom minimalen Werten abweicht und die restlichen Messwerte nahe beim minimalen Wert liegen. Dieser „Ausreißer“ kann z.B. durch eine Aktion des Betriebssystems verursacht worden sein.
Weil die einzelnen Messwerte der 10 Messungen nicht vorliegen, sondern nur das daraus bestimmte $t_{\max}, t_{\min}$ und $\bar{t}$ bekannt ist, soll im Folgenden eine Methode gezeigt werden, mit der überprüft werden kann, ob in den Mittelwert nur ein einziger „Ausreißer“ eingegangen ist.
Sei $a_1, \ldots, a_n$ eine Messreihe aus n Messungen. Der Mittelwert $\bar{a}$, das Maximum $a_{\max}$ und das Minimum $a_{\min}$ der Messreihe ergeben sich wie folgt:

$$\begin{aligned}\bar{a} &= \frac{1}{n}\sum_{i=1}^{n} a_i \\ a_{\max} &= \max\{a_1, \ldots, a_n\} \\ a_{\min} &= \min\{a_1, \ldots, a_n\}\end{aligned}$$

Die Messungen $a_1, \ldots, a_n$ können so geordnet werden, dass $a_1 \leq \ldots \leq a_n$ gilt und somit der letzte Wert a_n das Maximum der Messreihe ist, ohne dass sich $\bar{a}, a_{\max}$ und $a_{\min}$ ändern.
Nimmt man an, dass bis auf das Maximum a_n alle anderen Werte nahe beim Minimum $a_{\min}$ liegen, dann kann man a_n aus der Messreihe entfernen, so dass für die restlichen Messwerte $a_1, \ldots, a_{n-1}$ gilt:

$$\frac{1}{n-1}\sum_{i=1}^{n-1} a_i \quad \approx \quad \max\{a_1, \ldots, a_{n-1}\} \quad \approx \quad \min\{a_1, \ldots, a_{n-1}\}$$

Um zu zeigen, dass eine Messreihe, von der nur $a_{\min}$, $a_{\max}$ und $\bar{a}$ bekannt ist, nur einen einzigen „Ausreißer“ enthält, kann ein Mittelwert $\hat{a}$ berechnet werden, der diese Eigenschaft besitzt. Gilt

$$\bar{a} \approx \hat{a}$$

für die beiden Mittelwerte, dann besitzt die Messreihe nur einen „Ausreißer“.
Zur Bestimmung von $\hat{a}$ kann man von der Definition zur Berechnung eines Mittelwerts ausgehen, wobei sich jeder Messwert a_i durch $a_i = a_{\min} + \varepsilon_i$ ausdrücken lässt, so dass gilt:

$$\hat{a} = \frac{1}{n}\sum_{i=1}^{n}(a_{\min} + \varepsilon_i) \tag{20}$$

Durch Umordnung der Summanden lässt sich die Gleichung (20) wie folgt schreiben:

$$\begin{aligned}\hat{a} &= \frac{1}{n}\left(\sum_{i=1}^{n} a_{\min} + \sum_{i=1}^{n} \varepsilon_i\right) \\ \hat{a} &= \frac{1}{n}\left(n \cdot a_{\min} + \sum_{i=1}^{n} \varepsilon_i\right) \\ \hat{a} &= a_{\min} + \frac{1}{n}\sum_{i=1}^{n} \varepsilon_i\end{aligned}$$

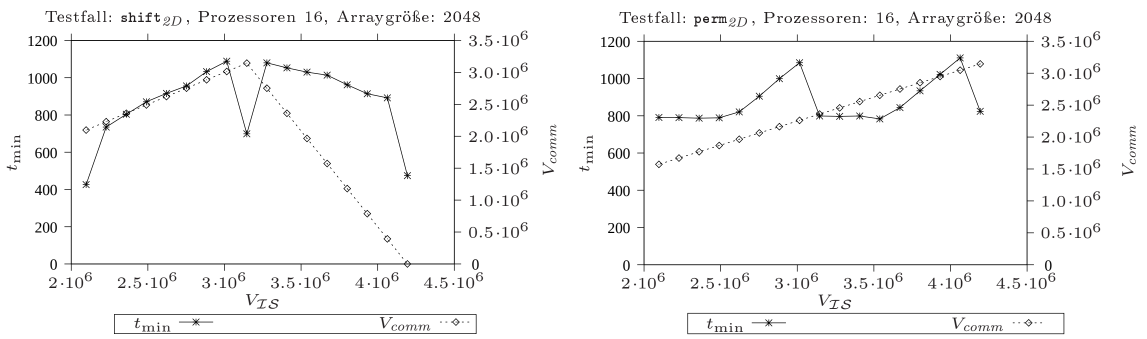
Da aufgrund der Sortierung der Messwerte $a_n = a_{\max} = a_{\min} + \varepsilon_{\max}$ gilt, lässt sich die Summe wie folgt aufspalten:

$$\hat{a} \quad = \quad a_{\min} + \frac{1}{n}\left(\sum_{i=1}^{n-1} \varepsilon_i + \varepsilon_{\max}\right)$$

Weil nach obiger Annahme alle anderen Messwerte nahe bei $a_{\min}$ liegen, können alle ε_i mit $i \in \{1, \ldots, n-1\}$ gleich Null gesetzt werden, so dass sich die Summe wie folgt vereinfachen lässt:

$$\hat{a} \quad = \quad a_{\min} + \frac{1}{n} \cdot \varepsilon_{\max}$$

Daher hat der Mittelwert $\hat{a}$ die Eigenschaft, dass in der Messreihe alle bis auf einen einzigen Messwert dem Minimum entsprechen.
Nachdem für alle Messwerte des Benchmarks $\hat{a}$ bestimmt und mit dem gespeicherten Mittelwert $t_{\min}$ verglichen wurden, fand sich keine Messung eines Testfalls, deren Abweichung größer als 10% war. Aus diesem Grund wird zur weiteren Auswertung als Messwert für die Laufzeit eines Testfalles das ausgegebene Minimum $t_{\min}$ verwendet.
Als nächstes sollte überprüft werden, ob die Laufzeiten der Testfälle mit dem Kommunikationsvolumen korrelieren. Dazu wurde in einem Graphen sowohl die Laufzeit (linke Achse) als auch V_{comm} (rechte Achse) über $V_{\mathcal{IS}}$ aufgetragen. In Abbildung 18 ist dies beispielhaft für die Testfälle `shift`$_{2D}$ (Abbildung 18(a)) und `perm`$_{2D}$ (Abbildung 18(b)) mit Arraygröße 2048×2048 gezeigt.

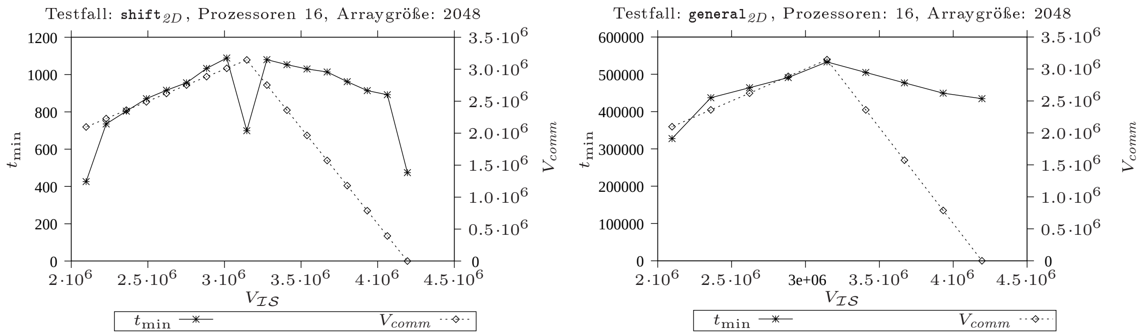


(a) (b)

Abbildung 18: Vergleich der Testfälle `shift`$_{2D}$ und `perm`$_{2D}$

Beim Testfall `shift`$_{2D}$ korreliert zwar bei 16 Prozessoren die Laufzeit anfangs mit dem Ansteigen des Kommunikationsvolumens, sobald aber das Kommunikationsvolumen wieder abnimmt, vermindert sich die Laufzeit nicht in demselben Maße, wie entsprechend der Reduzierung der Kommunikation zu erwarten wäre. Somit tritt bei dieser Messung eine schwache Korrelation zwischen Laufzeit und Kommunikation auf. Betrachtet man anschließend die Messwerte des Testfalls auf nur 8 und dann auf nur 4 Prozessoren, lässt sich auch hier keine direkte Korrelation zwischen Laufzeit und Kommunikation finden.
Beim Testfall `perm`$_{2D}$ in Abbildung 18(b) zeigt sich bei 16 Prozessoren die Laufzeit unabhänigig vom Kommunikationsvolumen, da das Kommunikationsvolumen zwar ansteigt, aber

die Laufzeit im Mittel nicht zunimmt. Wird der Testfall auf 8 und dann auf 4 Prozessoren gemessen, ist eine geringe Abhängigkeit zwischen Kommunikationsvolumen und Laufzeit zu erkennen.
Wird die Laufzeit des Testfalls $\texttt{shift}_{2D}$ (Abbildung 19(a)) mit der des Testfalls $\texttt{general}_{2D}$ (Abbildung 19(b)) verglichen, dann ist eine große Ähnlichkeit der Form der beiden Graphen erkennbar. Allerdings ist die Skalierung der Laufzeit sehr unterschliedlich, da die Laufzeit des Testfalls $\texttt{general}_{2D}$ ca. um den Faktor 500 höher ist, obwohl das Volumen und das Muster der Kommunikation bei beiden Testfällen gleich sind.



(a) Laufzeitverhalten des Testfalls $\texttt{shift}_{2D}$

(b) Laufzeitverhalten des Testfalls $\texttt{general}_{2D}$

Abbildung 19: Vergleich der Testfälle $\texttt{shift}_{2D}$ und $\texttt{general}_{2D}$

6.3.1 Bewertung der Messungen

Da im Modell weder die Anzahl der Prozessoren noch die Arraygröße notwendigerweise bekannt sind, soll für dieses Kostenmodell der Unterschied zwischen zwei Kommunikationsmustern unabhängig vom Kommunikationsvolumen bestimmt werden. Dies ist vertretbar, da sich bei den Messungen für das SCI-Netz keine starke Abhängigkeit zwischen Kommunikationsvolumen und Laufzeit gezeigt hat. Wie die Messungen des Testfalls $\texttt{shift}_{2D}$ und $\texttt{general}_{2D}$ gezeigt haben, gibt es aber eine starke Abhängigkeit der Laufzeit vom Kommunikationsmuster. Um ein Maß für den Unterschied zweier Kommunikationsmuster zu erhalten, lässt sich für jedes Muster M ein Faktor F_M bestimmen, der angibt, um wieviel teuerer das Kommunikationsmuster M gegenüber dem lokalen Zugriff ist. Es gilt also für die Kosten $cost_M$ eines Kommunikationsmusters M:

$$cost_M = F_M \cdot cost_{\texttt{local}}$$

Dieser Faktor soll für die in diesem Benchmark gemessenen Testfälle bestimmt werden.
Dazu kann wie folgt vorgegangen werden: Für einen Testfall $test_i$ des Benchmarks wird für jede verwendete Arraygröße N und Anzahl der Prozessoren p die Laufzeit als die Funktion $time_{test_i,N,p} : V_{\mathcal{IS}} \rightarrow t_{\min}$ dargestellt. Der Faktor für diese Messung berechnet sich aus:

$$F_{test_i,N,p} = \frac{\int time_{test_i,N,p}}{\int time_{\texttt{local},N,p}}$$

Da $time_{test_i,N,p}$ keine stetige Funktion ist, sondern nur eine Funktion, die sich aus den einzelnen Messpunkten ergibt, ist eine nummerische Integrationsmethode anzuwenden. In vorliegendem Fall wurde auf die Trapez-Methode zurückgegriffen.

Der Faktor F_{test} für den Testfall $test_i$ ergibt sich aus dem Mittelwert aller Faktoren $F_{test_i,N,p}$.

6.3.2 Bewertung der Gruppe 1

Bei der Analyse des Laufzeitverhaltens tritt, wie bereits in Abbildung 18 gezeigt, keine direkte Abhängigkeit der Laufzeit vom Kommunikationsvolumen für die gemessenen Arraygrößen auf. Daher genügt es, die Faktoren entsprechend der Methode in Abschnitt 6.3.1 zu bestimmen. Diese Faktoren sind für die 1-dimensionalen Testfälle in Tabelle 11 und für die 2-dimensionalen Testfälle in Tabelle 10 für die Arraygrößen $512 \times 512, 1024 \times 1024$ und 2048×2048 aufgelistet.

Testfall	4 Prozessoren			8 Prozessoren			16 Prozessoren		
	512	1024	2048	512	1024	2048	512	1024	2048
`shift`$_{2D}$	2	2	2	2	2	2	3	2	3
`perm`$_{2D}$	2	3	3	3	4	5	4	3	3
`general`$_{2D}$	320	461	477	513	697	759	908	1204	1611
`broadCast`	1	1	1	1	1	1	1	1	1

Tabelle 10: Faktoren der 2-dimensionalen Testfälle der ersten Gruppe

Arraygröße	4 Prozessoren		8 Prozessoren		16 Prozessoren	
	`shift`$_{1D}$	`general`$_{1D}$	`shift`$_{1D}$	`general`$_{1D}$	`shift`$_{1D}$	`general`$_{1D}$
2048	13	551	18	870	25	1528
4096	11	717	15	1076	20	1649
8192	12	529	12	1048	15	1689
16384	13	419	13	850	13	1671
32768	13	369	13	738	13	1490
65536	10	259	13	691	13	1383

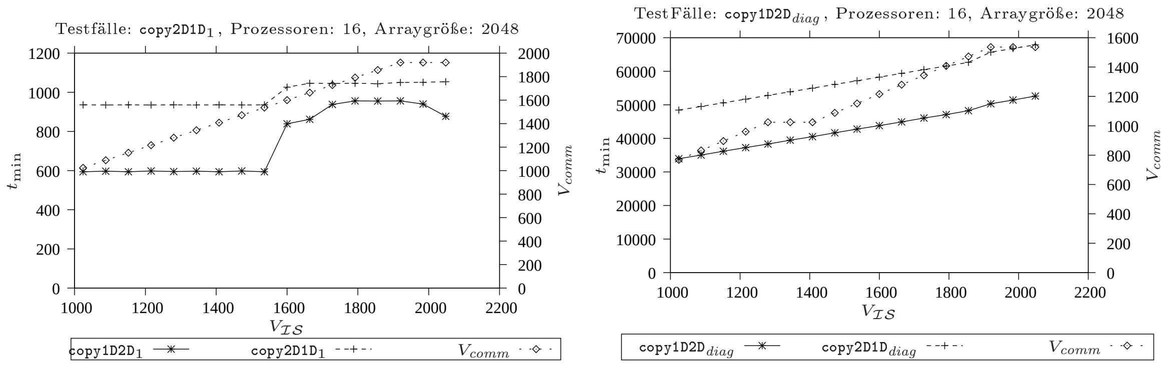
Tabelle 11: Faktoren der 1-dimensionalen Testfälle der ersten Gruppe

Dabei ist zu bemerken, dass der Testfall `broadCast` immer genau so schnell wie das lokale Kopieren ist, obwohl er ein ähnlich großes Kommunikationsvolumen V_{comm} wie der Testfall `shift`$_{2D}$ besitzt.

6.3.3 Bewertung der Gruppe 2

Die Laufzeitmessung auf 8 Prozessoren für die Testfälle dieser Gruppe ist in Abbildung 20 graphisch dargestellt. Vergleicht man `copy2D1D`$_1$ und `copy1D2D`$_1$, dann unterscheiden sich die Laufzeiten beider Testfälle nicht bei 4 oder 8 sondern erst bei 16 Prozessoren. Es ist zu vermuten, dass bei 16 Prozessoren der Unterschied der Laufzeit durch die große Anzahl von kleinen Nachrichten verursacht wird, da dieser Unterschied bei 4 oder 8 Prozessoren nicht auftritt. Das Gleiche gilt auch für `copy2D1D`$_{diag}$ und `copy1D2D`$_{diag}$. Aus diesem Grund ist es legitim, im Allgemeinen davon auszugehen, dass das Kopieren von einem 1-dimensionalen in ein 2-dimensionales Array und von einem 2-dimensionalen in ein 1-dimensionales Array gleich schnell ist.

Vergleicht man den Zugriff auf die Diagonale `copy2D1D`$_{diag}$ mit dem Zugriff auf die erste Zeile `copy2D1D`$_1$, dann ist der Zugriff auf die Diagonale zwei Größenordnungen langsamer, weshalb Zugriffsfunktionen wie f_{diag} unbedingt zu vermeiden sind.



(a) (b)

Abbildung 20: Laufzeitverhalten der Testfälle der zweiten Gruppe bei 16 Prozessoren

Setzt man die hier angegeben Testfälle mit `local`$_{1D}$ in Beziehung, dann erhält man die Faktoren, die in der Tabelle 12 angegeben sind.

Testfall	4 Prozessoren			8 Prozessoren			16 Prozessoren		
	512	1024	2048	512	1024	2048	512	1024	2048
copy1D2D$_1$	80	118	14	148	192	20	295	348	32
copy2D1D$_1$	82	99	9	220	256	24	552	509	42
copy1D2D$_{diag}$	1988	3910	780	2974	5994	1008	4449	9068	1865
copy2D1D$_{diag}$	1948	3831	763	2956	5958	1033	7184	14570	2499

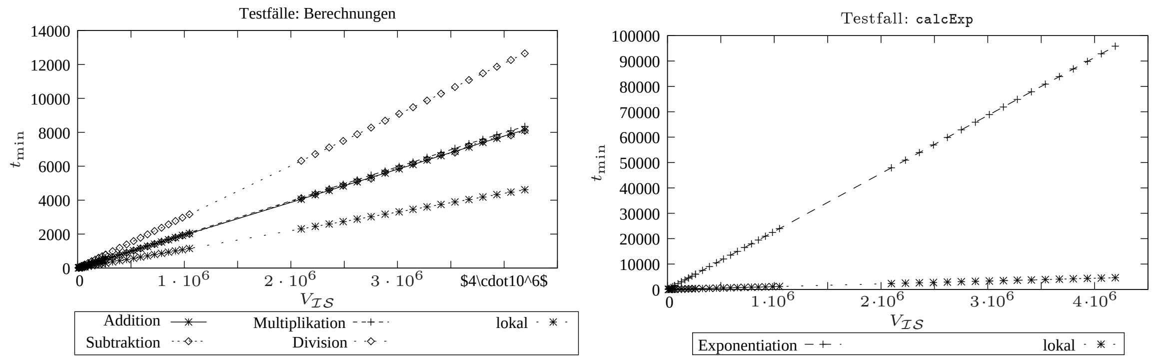
Tabelle 12: Faktoren für die Testfälle der Gruppe 2

6.3.4 Bewertung der Gruppe 3

Die Testfälle dieser Gruppe wurden ausschließlich auf einem Prozessor gemessen. Für die Auswertung wurde die Laufzeit gegenüber $V_{\mathcal{IS}}$ aufgetragen, wie Abbildung 21 zeigt. Setzt man die Laufzeit der Testfälle in Beziehung zu `local`$_{2D}$, dann erhält man die in der Tabelle 13 angegebenen Faktoren.

Testfall	Faktor
Addition	2
Subtraktion	2
Multiplikation	2
Division	3
Exponentiation	21

Tabelle 13: Faktoren der einzelnen Berechnungen für den Datentyp `REAL`

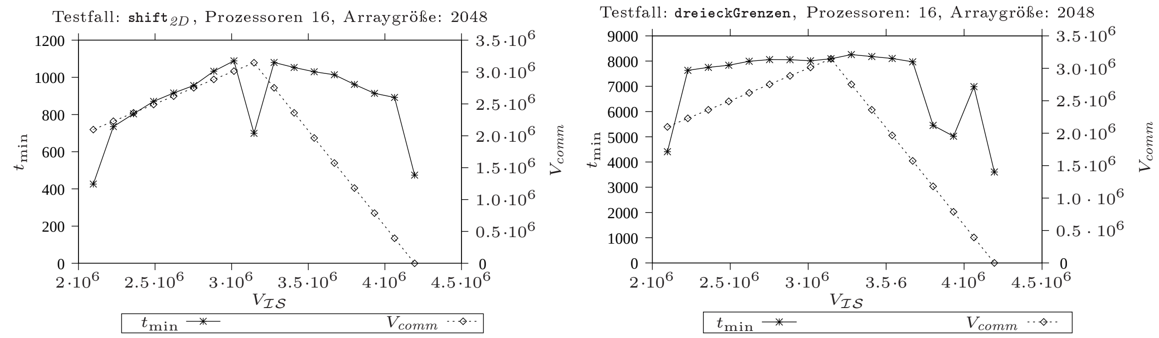


(a) (b)

Abbildung 21: Laufzeitverhalten der Testfälle der dritten Gruppe

6.3.5 Bewertung der Gruppe 4

In Abbildung 22(b) ist die Laufzeit des Testfalls `dreieckGrenzen` für die Arraygröße 2048 × 2048 gezeigt, wobei der Testfall auf 16 Prozessoren gemessen wurde. Zum Vergleich dazu ist die Laufzeit des Testfalls $\texttt{shift}_{2D}$ in Abbildung 22(a) gezeigt, da beide Testfälle das gleiche Kommunikationsmuster besitzen. Dabei ist zu erkennen, dass sich die Graphen vom Verlauf her zwar entsprechen, jedoch aufgrund der Skalierung der Zeitachse der Testfall `dreieckGrenzen` um eine Größenordnung langsamer als $\texttt{shift}_{2D}$ ist. Dies liegt an der schlechten Darstellung von nicht rechteckigen Iterationsräumen im HPF-Compiler ADAPTOR. (Vergleiche dazu auch Abschnitt 5.)



(a) Testfall $\texttt{shift}_{2D}$. (b) Testfall `dreieckGrenzen`

Abbildung 22: Vergleich der Testfälle $\texttt{shift}_{2D}$ und `dreieckGrenzen` für die Arraygröße 2048× 2048

7 Kostenmodell

In diesem Abschnitt soll ein Kostenmodell für den HPF-Compiler ADAPTOR erarbeitet werden, mit dem der in Abschnitt 3.4.1 erzeugte Kommunikationsgraph für eine mögliche Wahl von Platzierungen bewertet wird. Dazu werden zuerst als Vergleich zwei unterschiedliche Kostenmodelle dargestellt, bevor auf das Kostenmodell für ADAPTOR eingegangen wird.

7.1 Einfaches Kostenmodell für Nachrichtenaustausch

Foster stellt ein allgemeines, einfaches und intuitives Kostenmodell für Programme mit explizitem Nachrichtenaustausch vor [Fos94]. Es soll zur Bewertung von parallelen Algorithmen und deren Implementierung mit explizitem Nachrichtenaustausch, wie z.B. MPI [Mes95], dienen. Dazu wird für jeden Prozessor i die Zeit T^i_{comp}, die Prozessor i für Berechnungen benötigt, die Zeit T^i_{comm}, die Prozessor i für die Kommunikation aufwendet, und die Wartezeit T^i_{idle} bestimmt. Daraus ergibt sich für ein Programm, das auf n_{cpu} Prozessoren gestartet wird, folgende Formel für die Gesamtlaufzeit T:

$$T = \frac{1}{n_{cpu}} \left(\sum_{i=1}^{n_{cpu}} T^i_{comp} + \sum_{i=1}^{n_{cpu}} T^i_{comm} + \sum_{i=1}^{n_{cpu}} T^i_{idle} \right)$$

Da die Wartezeit T^i_{idle} eines Prozessors im Allgemeinen sehr schwierig zu bestimmen ist, setzt man $T_{idle} = 0$ und nimmt an, dass die Wartezeiten, wie z.B. das Warten auf die Vollendung einer Kommunikation, bereits in der Kommunikationszeit enthalten ist. Die Zeit T^i_{comp} entspricht der Anzahl der auf Prozessor i ausgeführten Berechnungsstatements.
Die Kosten für die Kommunikation werden für jedes Paar von Sende- und Empfangs-Statements explizit aus der Startup-Zeit t_s, die sich aus der Zeit für den Verbindungsaufbau und für die Bereitstellung der Puffer für die Kommunikation ergibt, aus der Länge der Kommunikation l in Bytes und aus der Bandbreite t_w durch folgende Formel bestimmt:

$$T_{comm} = t_s + t_w \cdot l \tag{21}$$

Die Werte für t_s und t_w ergeben sich aus der Architektur des Parallelrechners und können durch Benchmarks bestimmt werden.
Dieses Kostenmodell eignet sich gut zur Bewertung von Algorithmen und Programmen, die zur Implementierung Bibliotheken zum Nachrichtenaustausch, wie z.B. LAM [SL03], verwenden, da im Programm explizit die Anzahl und die Länge der Nachrichten bekannt sind.
Diese hier vorgeschlagene Modellierung ist für HPF zu detailliert, da für einen parallelen Schleifensatz weder die Anzahl noch die Länge der Nachrichten bekannt ist, weil dies von der vom HPF-Compiler erzeugten Kommunikationsstruktur abhängt. Wie in Abschnitt 5 gezeigt, hängt die vom Compiler erzeugte Kommunikation sowohl von der Möglichkeit ab, die Menge der zu kommunizierenden Elemente darzustellen, als auch von den verwendeten Optimierungen wie Message Vectorisation oder Message Coalescing. Zudem ist die Anzahl der Berechnungsstatements, die in T_{comp} eingehen würden, nicht bekannt, da diese von den Routinen der Laufzeitbibliothek des HPF-Compilers abhängig sind.
Außerdem zeigt die Auswertung der Messergebnisse des in Abschnitt 6 vorgestellten Benchmarks, dass die Laufzeit hauptsächlich vom Kommunikationsmuster abhängt und das Kommunikationsvolumen nur eine untergeordnete Rolle spielt.
Es wird daher ein gröberes Kostenmodell zur Bewertung der Kommunikation für einen HPF-Compiler benötigt.

7.2 Kostenmodell für Bulk-Synchronous Parallelism

Ein Kostenmodell, das von den einzelnen Nachrichten abstrahiert und nur das Kommunikationsvolumen berücksichtigt, wurde zur Programmiermethodik BSP entwickelt [Val90]. Ein BSP Programm gliedert sich in Supersteps, die sich in zwei Abschnitte zerlegen lassen: dem Berechnungsschritt folgt der Kommunikationsschritt mit abschließender Barriersynchronisation. Die Laufzeit T des gesamten Programms ergibt sich aus der Summe der Laufzeiten der Supersteps.
Die Laufzeit T_{step} eines Supersteps ergibt sich aus der Anzahl w der Berechnungen im Berechnungsschritt, aus der Anzahl h_s der gesendeten und aus der Anzahl h_r der empfangenen Daten und aus der Anzahl l der Schritte, die zur Barriersynchronisation benötigt werden, nach folgender Formel:

$$T_{step} = w + g \cdot \max(h_s, h_r) + l$$

Dabei bestimmt der Faktor g die Permeabilität des Netzwerkes und wird durch Benchmarks für einen Parallelrechner bestimmt. Er beschreibt folgendes Verhältnis:

$$g = \frac{\text{max. Schrittzahl von allen Prozessoren in der Sekunde}}{\text{max. Wortzahl kommunizierbar in einer Sekunde}}$$

Somit gibt der Faktor g an, um wieviel teurer eine Kommunikation im Vergleich zu einer Berechnung ist.
Dieses Kostenmodell benötigt zwar keine Informationen über die Anzahl und Länge der einzelnen Kommunikationen, aber es eignet sich dennoch nicht für HPF-Programme, da nicht zwischen verschiedenen Kommunikationsmustern unterschieden werden kann, sondern nur das maximale Kommunikationsvolumen in die Bewertung eingeht. Wie es sich bei dem Abschnitt 6 vorgestellten Benchmark gezeigt hat, ist die Laufzeit hauptsächlich vom Kommunikationsmuster abhängig und nicht vom Kommunikationsvolumen.

7.3 Kostenmodell für ADAPTOR

Basierend auf den Messergebnissen des Benchmarks in Abschnitt 6 soll in diesem Abschnitt ein Kostenmodell für den HPF-Compiler ADAPTOR vorgestellt werden, das den in Abschnitt 3.4.1 erzeugten Kommunikationsgraphen G_{comm} bewertet.
Um unterschiedliche Kommunikationsmuster bewerten zu können, die sich aus den Zugriffsfunktionen der verwendeten Arrays ergeben, wird eine Annahme über die Verteilung der Arrays/Templates benötigt. Ohne eine solche Annahme kann, wie bereits in Abschnitt 3.1.1 erwähnt, nur ausgesagt werden, ob Kommunikation auftritt oder nicht. Nimmt man an, dass die Arrays/Templates zyklisch auf die Prozessoren verteilt sind, lässt sich nur schwierig eine genauere Aussage treffen, da zwei nebeneinander liegende Arrayelemente abhängig von der Prozessoranzahl auf zwei völlig unterschiedlichen, nicht zwingend nebeneinander liegenden Prozessoren gespeichert sein können.
Aus diesem Grund beschränkt sich das Kostenmodell für ADAPTOR auf die Bewertung von blockverteilten Arrays/Templates, weil für diese Verteilung eine genauere Aussage über die Kommunikationsstruktur getroffen werden kann. Bei der Blockverteilung gilt nämlich die Eigenschaft, dass zwei nebeneinander liegende Arrayelemente auch auf zwei nebeneinander liegenenden Prozessoren gespeichert sind.
Daher wurde der Benchmark in Abschnitt 6 nur für blockverteilte Arrays durchgeführt. Bei der Auswertung der Messergebnisse in Abschnitt 6.3 zeigte es sich, dass keine direkte Abhängigkeit zwischen dem gesamten Kommunikationsvolumen und der Laufzeit der unterschiedlichen

Testfälle existiert. Unterschiedliche Zugriffsfunktionen, die das gleiche Kommunikationsvolumen erzeugen, haben jedoch starken Einfluss auf die Laufzeit. Man vergleiche dazu z.B. die Laufzeitmesswerte der beiden Testfälle `shift`$_{2D}$ und `general`$_{2D}$ in Abschnitt 6.3.2.
Aus diesem Grund wird im Folgenden eine Kommunikation nur nach dem Kommunikationsmuster beurteilt und das Kommunikationsvolumen ignoriert. Dabei ist zu beachten, dass das Kommunikationsmuster leicht aus der zur Kommunikationsrelation h_{comm} gehörenden Matrix $M_{h_{comm}} = \text{RelMat}(h_{comm})$ abgelesen werden kann. Dagegen ist die Berechnung des Kommunikationsvolumens sehr aufwenig, da zuerst das Polyeder P_{comm}, das die zu kommunizierenden Elemente enthält, zu erzeugen ist und dann aus P_{comm} das Ehrhart-Polynom berechnet werden muss, das das Kommunikationsvolumen darstellt [CL96]. Außerdem ist das erzeugte Ehrhart-Polynom von den Strukturparametern des Programmfragments abhängig, so dass im Allgemeinen ohne Kenntnis der Größenverhältnisse der Strukturparameter zwei durch Ehrhart-Polynome beschriebene Volumen nicht verglichen werden können. Eine mögliche Berechnungsmethode zur Bestimmung des Kommunikationsvolumens ist in Abschnitt 9.3 gezeigt.
Um das Kommunikationsmuster zu bestimmen, werden die Kommunikationsrelationen in folgende Klassen eingeteilt:

1. Eine Kommunikationsrelation h_{comm} setzt die Elemente zweier verschiedener Templates in Beziehung. Da beide Templates unterschiedlich verteilt sein können, wie in Abschnitt 3.1.1 erläutert, kann keine weitere Aussage über die Kommunikation getroffen werden. Um solche Kommunikationsmuster zu bewerten, werden die Ergebnisse der Testfälle `copy2D1D`$_1$ und `copy1D2D`$_1$ aus Gruppe 3 (siehe Abschnitt 6.1.2 und 6.3.3) verwendet. Eine Kommunikationsrelation h_{comm}, die diese Eigenschaft besitzt, wird durch den Wert `diffTemplates` klassifiziert.

2. Eine Kommunikationsrelation h_{comm} setzt die Elemente eines Templates in Beziehung zueinander. Da für zwei nebeneinanderliegene Template-Elemente gilt, dass sie auf nebeneinanderliegenden Prozessoren gespeichert sind, wird eine weitere Unterscheidung der Kommunikation nach Form der Kommunikationsrelation wie folgt getroffen:
   - Die Kommunikationsrelation beschreibt einen lokale Zugriff. (Klassifizierung: `lokal`)
   - Die Kommunikationrelation beschreibt eine Verschiebung der Werte entlang einer oder mehrerer Dimension um einen konstanten Wert (z.B. wie Testfall `shift`$_{2D}$ in Abschnitt 6.1.1). (Klassifizierung: `shift`)
   - Die Kommunikationsrelation beschreibt eine Permutation (z.B. wie Testfall `perm`$_{2D}$ in Abschnitt 6.1.1). (Klassifizierung: `permutation`)
   - Die Kommunikationsrelation beschreibt eine allgemeine, nicht von ADAPTOR analysierbare Kommunikation (z.B. wie `general`$_{2D}$ in Abschnitt 6.1.1). (Klassifizierung: `generell`)

Auf der Grundlage dieser Klassifizierung wird die Kommunikation bewertet.
Algorithmus 13 klassifiziert eine Kommunikationsrelation h_{comm}. Dabei wird zur Klassifizierung einer Kommunikationsrelation, die die Elemente eines $d_{\mathtt{T}}$-dimensionalen Templates `T` in Beziehung zueinander setzt, wie folgt vorgegangen:
Zuerst wird die zur Kommunikationsrelation h_{comm} gehörige Matrix $M = \text{RelMat}(h_{comm})$ erzeugt und in reduzierte Zeilen-Stufen-Form transformiert. Dem Template `T` sind die Spalten

`classify`(h_{comm})

Eingabe Eine Kommunikationsrelation h_{comm}.

Ausgabe Die Klassifizierung der Kommunikationsrelation h_{comm}, die einen Wert aus der Menge der möglichen Klassifizierungen {`lokal`, `shift`, `permutation`, `generell`, `diffTemplates` } annehmen kann.

Algorithmus

1. Bestimme die Templates $\mathtt{T}_1$ und $\mathtt{T}_2$, die durch h_{comm} in Beziehung stehen.
2. Sind $\mathtt{T}_1$ und $\mathtt{T}_2$ zwei unterschiedliche Templates, gib die Klassifizierung `diffTemplates` zurück.
3. Beschreiben $\mathtt{T}_1$ und $\mathtt{T}_2$ das gleiche Template, dann analysiere h_{comm} weiter: Dabei seien $p_l, \ldots, p_k$ die Dimensionen aus $\mathcal{P}$, die dem Template $\mathtt{T}_1$ zugeordnet sind, und sei $d_{\mathtt{T}_1} = k - l$ die Dimensionalität des Templates $\mathtt{T}_1$ und $d = n_{pDim} + n_b$.
   (a) Bestimme $M = \mathrm{RelMat}(h_{comm})$ und ermittle M' aus M, durch Übergang in reduzierte Zeilen-Stufen-Form. Die Elemente der $q \times (2 \cdot d)$ Matrix M' sind dargestellt durch:
   $$M' = (m'_{ij})_{\substack{1 \le i \le q \\ 1 \le j \le 2 \cdot d}}$$
   (b) Setze die Matrix $\hat{M} = M'$ und entferne alle Zeilen i aus $\hat{M}$, die folgende Form besitzen:
   $$\exists j \in \{l, \ldots k\} : \; m'_{ij} \neq 0 \quad \vee \quad \exists j \in \{d + l, \ldots d + k\} : \; m'_{ij} \neq 0$$
   Dann ist $\hat{M}$ eine $q' \times (2 \cdot d)$-Matrix.
   (c) Erstelle die Matrizen M_T, $M_{T'}$ und M_K aus der Matrix $\hat{M}$:
   $$M_T = (\hat{m}_{ij})_{\substack{1 \le i \le q' \\ l \le j \le k}} \qquad M_{T'} = (\hat{m}_{ij})_{\substack{1 \le i \le q' \\ d+l \le j \le d+k}} \qquad M_K = (\hat{m}_{ij})_{\substack{1 \le i \le q' \\ d+n_{pDim} \le j \le 2 \cdot d}}$$
   (d) Bewerte die Matrizen $M_T, M_{T'}$ und M_K wie folgt:
   - Erstelle die $d_{\mathtt{T}_1} \times d_{\mathtt{T}_1}$ Einheitsmatrix E.
   - Ist $M_T = E$ und $M_{T'} = -E$ und $M_K = 0$, dann gib die Klassifizierung `lokal` zurück.
   - Ist $M_T = E$ und $M_{T'} = -E$ und $M_K \neq 0$, gib die Klassifizierung `shift` zurück.
   - Ist $M_T = E$, ist $M_{T'} = -E$ eine Permutation der Matrix E und ist $M_K = 0$, dann gib die Klassifizierung `permutation` zurück.
   - Sonst gib die Klassifizierung `generell` zurück.

Algorithmus 13: Klassifizierung der Kommunikationsrelation

$p_l, \ldots, p_k$ bzw. die Spalten $p'_l, \ldots, p'_k$ der Matrix M zugeordnet. Da zur Bewertung der Kommunikation nur die Beziehungen zwischen den Dimensionen interessieren, die dem Template T zugeordnet sind, werden alle Zeilen aus M entfernt, die nicht die Dimensionen $p_l, \ldots, p_k$ bzw. $p'_l, \ldots, p'_k$ verwenden. Die Matrix M besitzt danach folgende Form:

$$
\begin{array}{cc}
 & \begin{array}{cccccccccccccccccccc} p_1 & \ldots & p_l & \ldots & p_k & \ldots & p_n & m_1 & \ldots & m_{n_b} & p'_1 & \ldots & p'_l & \ldots & p'_k & \ldots & p'_n & m'_1 & \ldots & m'_{n_b} \end{array} \\
M = & \begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 0 & 0 & * & * & * \\
0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 0 & 0 & * & * & * \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 0 & 0 & * & * & *
\end{pmatrix}
\end{array}
$$

Dabei bezeichnet $*$ beliebige rationale Einträge in der Matrix.
Um die Beziehung der Template-Elemente zueinander zu beschreiben, interessieren nur die Spalten $p_l, \ldots, p_k$, die Spalten $p'_l, \ldots, p'_k$ und die Spalten $m'_1, \ldots, m'_{n_b}$. Aus diesem Grund werden in Algorithmus 13 die Matrizen $M_T, M_{T'}$ und die Matrix M_K aus der Matrix $\hat{M}$ extrahiert.
Die Kommunikationsrelation wird wie folgt klassifiziert:

- Stellt die Kommunikationsrelation h_{comm} einen lokalen Zugriff dar, dann gilt für ein $\beta \in \mathcal{P}$: $h_{comm}(\beta) = \beta$. Daher beschreibt die Kommunikationsrelation h_{comm} folgende Beziehung:
$$\forall j \in \{l, \ldots, k\} : \; p_j = p'_j$$
Ist diese Beziehung zwischen den Template-Elementen gegeben, dann entsprechen M_T und $-M_{T'}$ der $n_{\mathtt{T}} \times n_{\mathtt{T}}$ Einheitsmatrix und die Matrix M_K besitzt nur Einträge mit dem Wert 0.

- Stellt die Kommunikationsreltation h_{comm} eine Verschiebung der Werte entlang einer oder mehrerer Dimension um einen konstanten Wert dar, dann beschreibt h_{comm} folgende Beziehung:
$$\forall j \in \{l, \ldots, k\} : \; p_j = p'_j + k_j$$
Dabei ist k_j ein linearer Ausdruck in den Strukturparametern. Ist diese Beziehung gegeben, dann entsprechen die Matrizen M_T und $-M_{T'}$ der $n_{\mathtt{T}} \times n_{\mathtt{T}}$ Einheitsmatrix und die Matrix M_K besitzt Einträge, die ungleich Null sind.

- Stellt die Kommunikationsrelation h_{comm} eine Permutation dar, dann beschreibt h_{comm} folgende Beziehung:
$$\forall j \in \{l, \ldots, k\} : \; p_j = p'_{\varrho(j)}$$
Dabei ist ϱ eine Permutatition der Menge $\{l, \ldots, k\}$. Ist diese Beziehung gegeben, dann entspricht die Matrix M_T der $n_{\mathtt{T}} \times n_{\mathtt{T}}$ Einheitsmatrix und $-M_{T'}$ einer Permutation der $n_{\mathtt{T}} \times n_{\mathtt{T}}$ Einheitsmatrix. Die Matrix M_K besitzt nur Einträge mit dem Wert 0.

- Alle anderen Formen von $M_T, M_{T'}$ und M_K werden als generelle Kommunikation eingestuft.

Ist eine Kommunikationsrelation h_{comm} mit `classify` klassifiziert worden, kann sie nach folgendem Schema bewertet werden: Zur Bestimmung der Kosten für Kommunikationsrelationen, die entweder als `lokal`, `shift`, `permutation` oder `generell` klassifiziert wurden, wird wie oben angegeben, die Auswertung der Gruppe 1 aus Abschnitt 6.3.2 verwendet. Die Kosten für die einzelnen Klassen ergeben sich aus dem Mittelwert der Faktoren der Testfälle $\texttt{shift}_{2D}$, $\texttt{perm}_{2D}$ und $\texttt{general}_{2D}$, die in Tabelle 10 auf Seite 77 aufgelistet sind. Die Kosten sind in Tabelle 14 zusammengefasst. Da die Faktoren in Abschnitt 6.3.2 bezüglich des Testfalls $\texttt{local}_{2D}$ berechnet wurden, betragen die Kosten für eine lokale Kommunikation 1.

Testfall	Kosten	Klassifizierung von h_{comm}
	1	$\texttt{classify}(h_{comm}) = \texttt{lokal}$
$\texttt{shift}_{2D}$	2	$\texttt{classify}(h_{comm}) = \texttt{shift}$
$\texttt{perm}_{2D}$	3	$\texttt{classify}(h_{comm}) = \texttt{permutation}$
$\texttt{general}_{2D}$	700	$\texttt{classify}(h_{comm}) = \texttt{generell}$
$\texttt{copy1D2D}_1$	146	$\texttt{classify}(h_{comm}) = \texttt{diffTemplates}$
$\texttt{copy2D1D}_1$		

Tabelle 14: Die Kosten für eine Kommunikationsrelation h_{comm}

Analog dazu ergeben sich die Kosten für eine Kommunikationsrelation, die mit `diffTemplates` klassifiziert wurde, aus dem Mittelwert der in Abschnitt 6.3.3 berechneten Faktoren für die Testfälle $\texttt{copy1D2D}_1$ und $\texttt{copy2D1D}_1$, die in Tabelle 12 aufgelistet sind.
Basierend auf diesen Faktoren werden die Kosten für einen Kommunikationsgraphen G_{comm} dadurch berechnet, dass die Kosten für alle Kommunikationsrelationen bestimmt und aufaddiert werden, wie in Algorithmus 14 gezeigt.

CostFunc$_{\texttt{ADAPTOR}}(G_{comm})$

Eingabe Der Kommunikationsgraph $G_{comm} = (V, E)$.

Ausgabe Die Kosten der durch den Kommunikationsgraph G_{comm} beschriebenen Kommunikation.

Algorithmus

1. Setze $c = 0$.
2. Für jede Kante $e \in E$ mit $e = (v, v', h_{comm})$:
   (a) Klassifiziere die Kommunikationsrelation: $class = \texttt{classify}(h_{comm})$
   (b) Bestimme die Kosten:

$$c = c + \begin{cases} 1 & \text{für } class = \texttt{lokal} \\ 2 & \text{für } class = \texttt{shift} \\ 3 & \text{für } class = \texttt{permutation} \\ 700 & \text{für } class = \texttt{generell} \\ 146 & \text{für } class = \texttt{diffTemplates} \end{cases}$$

3. Gib die berechneten Kosten c zurück.

Algorithmus 14: Die Kostenfunktion für den HPF-Compiler ADAPTOR basierend auf den Messergebnissen in Abschnitt 6.3

8 Experimenteller Laufzeitvergleich

In diesem Abschnitt soll das Ergebnis der Transformation eines Schleifensatzes aus einer astrophysikalischen Simulation durch LCCP gezeigt und der Laufzeitunterschied zwischen der transformierten und der originalen Version des Schleifensatzes bestimmt werden. Dazu wird ein Schleifensatz aus der sequenziellen Fortran77-Version des Programms zur Berechnung von Gravitationswellen ausgewählt, das an der Universität von Pittsburgh entwickelt wurde [GWI92, HM95]. Dieser Schleifensatz ist in Programm 8.1 angegeben.

Programm 8.1

```
DO k = 0, nz
   l2(0,k) = - 4.0 * (gm_south - g_south) / (dz * dz)
   DO j = 1, ny - 1
      l2(j,k) = - (((1.0 - y(j))*(1.0 - y(j))) * gyy(j,k) - &
                2.0 * y(j) * gy(j,k) + gzz(j,k) / (1.0 - y(j) * y(j)))
   END DO
   l2(ny,k) = - 4.0 * (gm_north - g_north) / (dz * dz)
END DO
```

Im Programmfragment 8.1 treten folgende Ausdrücke textuell mehrfach auf:

- der Lesezugriff auf das Array `y(j)`
- die Berechnung `(1.0 - y(j))`
- die Berechnung `y(j) * y(j)`

Deswegen werden für diese Ausdrücke von LCCP Hilfsarrays angelegt. Daneben werden folgende Ausdrücke mehrfach berechnet, da sie unabhängig von der äußeren Schleife mit Indexvariable `k` sind:

- die Berechnung `(1.0 - y(j))*(1.0 - y(j))`
- die Berechnung `(1.0 - y(j) * y(j))`
- die Berechnung `2.0 * y(j)`

Für diese Berechnungen werden ebenfalls von LCCP Hilfsarrays angelegt. Zudem werden folgende Berechnungen mehrfach durchgeführt, da sie nur Skalare verwenden, die von keiner Schleifenvariable abhängig sind:

- die Berechnung `(dz * dz)`
- die Berechnung `(- 4.0 * (gm_south - g_south) / (dz * dz))`
- die Berechnung `(- 4.0 * (gm_north - g_north) / (dz * dz))`

Diese Berechnungen können nicht zusammengefasst werden, weil die Implementierung der Methode LCCP zur Zusammenfassung der Berechnungen mehr als 2 GB Arbeitsspeicher benötigt, weshalb das Programm auf dem Rechner abbricht. Werden die Skalare ignoriert, dann benötigt die Implementierung von LCCP zur Zusammenfassung der Berechnungen genau 2 GB Arbeitsspeicher, so dass ein Ergebnis geliefert wird, das im Folgenden vorgestellt wird.
Programm 8.2 stellt das Ergebnis der Transformation durch LCCP dar.

Programm 8.2

```
DO p1 = 1, ny - 1
   array8(1,p1) = y(p1)
END DO
DO p1 = 1, ny - 1
   array7(2,p1) = 1.0 - array8(1,p1)            ! = 1.0 - y(p1)
END DO
DO p1 = 1, ny - 1
   array4(2,p1) = array8(1,p1) * array8(1,p1)   ! = y(p1) * y(p1)
END DO
DO p1 = 1, ny - 1
   array5(2,p1) = 2.0 * array8(1,p1)            ! = 2.0 * y(p1)
END DO
DO p1 = 1, ny - 1
   array6(3,p1) = array7(2,p1) * array7(2,p1)   ! = (1.0 - y(p1)) * (1.0 - y(p1))
END DO
DO p1 = 1, ny - 1
   array3(3,p1) = 1.0 - array4(2,p1)            ! = 1.0 - y(p1) * y(p1)
END DO
DO p1 = 0, nz
   l2(0,p1) = - 4.0 * (gm_south - g_south) / (dz * dz)
END DO
DO p1 = 0, nz
   DO p2 = 1, ny - 1
      l2(p2,p1) = -array6(3,p1) * gyy(p2,p1) - &
                  array5(2,p2) * gy(p2,p1) - gzz(p2,p1) / (array3(3,p1))
   END DO
END DO
DO p1 = 0, nz
   l2(ny,p1) = - 4.0 * (gm_north - g_north) / (dz * dz)
END DO
```

Für jede oben angegebene Berechnung wird ein Hilfsarray angelegt, das die Werte der Berechnungen aufnimmt, wobei für jedes Hilfsarray in Programm 8.2 durch einen Kommentar angegeben ist, welche Berechnung es aus dem Originalprogramm 8.1 darstellt. Jede Schleife kann parallel ausgeführt werden, da keine Abhängigkeiten zwischen den Iterationen der Schleifensätze existieren.
Die Verteilung der Arrays wird im Originalprogramm 8.1 so gewählt, dass der gesamte Schleifensatz ohne Kommunikation durchgeführt werden kann. Dazu wird das Template

```
!HPF$ TEMPLATE temp1(0:ny,0:nz)
```

angelegt, an dem alle Arrays ausgerichtet sind. Das 2-dimesionale Array `l2` wird daher mit der `ALIGN`-Direktive

```
!HPF$ ALIGN l2(i,j) WITH temp1(i,j)
```

auf das Template `temp1` ausgerichtet. Damit keine Kommunikation erzeugt werden muss, sind ebenfalls alle anderen 2-dimensionalen Arrays wie das Array `l2` auf das Template ausgerichtet. Da das 1-dimensionale Array `y` sowohl zur Berechnung der ersten als auch der zweiten Dimension des Arrays `l2` verwendet wird, ist das Array `y` auf die zweite Dimension des Templates repliziert:

```
!HPF$ ALIGN y(i) WITH temp1(i,*)
```

Bei der Generierung der möglichen Platzierungen, entsprechend Algorithmus 7 in Abschnitt 3.2.5, erhalten alle interessanten Knoten nur eine Platzierung, da die Propagierung in Richtung Ziel zur Quelle als auch in Richtung Quelle zum Ziel der Abhängigkeiten keine unterschiedlichen Platzierungen erzeugt, wobei jeder Knoten die gleiche Platzierung wie für das Array `y` erhält. Die gewählten Platzierungen sind für dieses Programm optimal, da sie zum einen keine Kommunikation erzeugen und zum anderen keine unnötige Replikation verwenden.
Um die Effektivität von LCCP zu untersuchen, wurde sowohl für die durch LCCP transformierte als auch für die originale Version des Programms Laufzeitmessungen durchgeführt. Damit der HPF-Compiler ADAPTOR effizienten Code für das Programm 8.1 erzeugen kann, wurde das Programm 8.1 zum Programm 8.3 umgeschrieben. Dabei wurden die äußeren Schleifen in drei unabhängige Schleifen aufgeteilt, damit ADAPTOR effizienten Code für die innere Schleife erzeugen kann. Zusätzlich wurden alle Schleifen mit der `INDEPENDENT`-Direktive als parallel ausführbar markiert. Dies ist daher möglich, da keine Abhängigkeiten zwischen den Iterationen des originalen Schleifensatzes existieren.

Programm 8.3

```
!HPF$ INDEPENDENT
DO k = 0, nz
   l2(0,k) = - 4.0 * (gm_south - g_south) / (dz * dz)
END DO
!HPF$ INDEPENDENT
DO k = 0, nz
   !HPF$ INDEPENDENT
   DO j = 1, ny - 1
      l2(j,k) = - (((1.0 - y(j))*(1.0 - y(j))) * gyy(j,k) - &
                2.0 * y(j) * gy(j,k) + gzz(j,k) / (1.0 - y(j) * y(j)))
   END DO
END DO
!HPF$ INDEPENDENT
DO k = 0, nz
   l2(ny,k) = - 4.0 * (gm_north - g_north) / (dz * dz)
END DO
```

Die Messung (vgl. Abschnitt 6.3) wurde auf dem hpcLine-Parallelrechner des Lehrstuhls für Programmierung an der Universität Passau durchgeführt, wobei die Laufzeit des Programms 10 mal gemessen wurde und als Messwert die minimale Laufzeit des Programms ausgegeben wurde. Um die Auswirkung der Einsparung der Berechnung untersuchen zu können, erfolgte die Messung für zwei unterschiedliche Problemgrößen, wobei den Parametern `ny` und `nz` folgende Werte zugewiesen wurden:

$$\begin{aligned} 6000 \times 6000 : &\quad \texttt{ny} = \texttt{nz} = 6000 \\ 9000 \times 9000 : &\quad \texttt{ny} = \texttt{nz} = 9000 \end{aligned}$$

Dabei konnte die Messung auf einem Prozessor nicht durchgeführt werden, da das Programm mit einem MPI-Fehler abbrach. Die Laufzeitmessungen für zwei Prozessoren lieferten für die Problemgröße 9000×9000 keine verwendbaren Werte, da die verwendeten Rechenknoten für die Arrays dieser Größenordnung nicht mit genügendem Arbeitsspeicher ausgerüstet waren. In Tabelle 15 sind die Laufzeiten in Millisekunden aufgelistet.

	Problemgröße 6000×6000		Problemgröße 9000×9000	
Prozessoren	LCCP	Original	LCCP	Original
2	2414	2752	-	-
4	1331	1535	3250	3747
8	901	969	1702	2147
16	591	761	1156	1472

Tabelle 15: Laufzeiten in Millisekunden der originalen und der durch LCCP transformierten Version des Programms 8.1

Zur Auswertung der Messung wurde der relative Speedup S_p bestimmt, der sich aus der Laufzeit T_1 des Programms auf einem Prozessor und der Laufzeit T_p des Programms auf p Prozessoren folgendermaßen berechnet [Fos94]:

$$S_p = \frac{T_1}{T_p}$$

Im Folgenden bezeichnen T_p^L die Laufzeit der durch LCCP transformierten Version und T_p^O die Laufzeit der originalen Version des Programms auf p Prozessoren. Da keine Laufzeitmessungen der Programme auf einem Prozessor vorliegen, wird die sequenzielle Laufzeit T_1 aus der Laufzeit der originalen Version des Programms unter Verwendung der wenigsten Prozessoren bestimmt.
Zur Berechnung des Speedups für die Problemgröße 6000×6000 wird die sequenzielle Laufzeit wie folgt bestimmt:

$$T_1 = T_2^O \cdot 2$$

Dabei wird angenommen, dass die Laufzeit T_2^O dadurch entstanden ist, dass das Programm auf zwei Prozessoren doppelt so schnell ist wie auf einem Prozessor. Dann berechnen sich der relative Speedup S_p^L für die durch LCCP transformierte Version und der relative Speedup S_p^O für die originale Version des Programms wie folgt:

$$\begin{aligned} S_p^O &= \frac{T_1}{T_p^O} \\ S_p^L &= \frac{T_1}{T_p^L} \end{aligned}$$

Der Speedup für die Problemgröße 9000 × 9000 wird analog dazu brechnet, wobei sich die sequenzielle Laufzeit T_1 aus der Laufzeit T_4^O ergibt, da für zwei Prozessoren keine Messwerte vorliegen:

$$T_1 = T_4^O \cdot 4$$

Tabelle 16 listet sowohl den so berechneten relativen Speedup als auch den Laufzeitunterschied zwischen der originalen und der durch LCCP transformierten Version unter Verwendung der in dieser Arbeit vorgestellten Platzierungsmethode für die eingeführten Hilfsarrays in Prozent auf. Dabei zeigt sich, dass die Einsparung der Berechnung eine Laufzeitverbesserung von durchschnittlich 18% gebracht hat.

	Problemgröße 6000 × 6000			Problemgröße 9000 × 9000		
	Verbesserung	rel. Speedup		Verbesserung	rel. Speedup	
Prozessoren	in %	LCCP	Original	in %	LCCP	Original
2	14	2.2	2.0	-	-	-
4	15	4.1	3.5	15	4.6	4
8	7	6.1	5.6	26	8.8	6.9
16	28	9.3	7.2	27	12.9	10.1

Tabelle 16: Verbesserung der LCCP-Version gegenüber dem Original und der relative Speedup der beiden Versionen

9 Thematisch verwandte Arbeiten

Die Generierung von Platzierungen für die im Polyedermodell dargestellten Berechnungen ist schon mehrfach betrachtet worden [Lam74, DR94]. Im Folgenden soll auf zwei Platzierungsmethoden eingegangen werden, die eine Ähnlichkeit mit dem in dieser Arbeit vorgestelltem Verfahren aufweisen: die Platzierungsmethode nach Feautrier [Fea93] (Abschnitt 9.1), die Platzierungen für Operationen im Polyedermodell bestimmt, und die Platzierungsmethode des dHPF-Compilers [AJMCY98] (Abschnitt 9.2), die zur Erzeugung von Platzierungen für die Berechnungen von parallelen Schleifen in HPF-Programmen eingesetzt wird. Abschließend wird in Abschnitt 9.3 eine Methode zur Berechnung des Kommunikationsvolumens in parallelen Schleifen gezeigt, in der ebenfalls zur Darstellung der Platzierungen ein Paar von linearen Abbildungen verwendet wird.

9.1 Platzierungsmethode nach Feautrier

Feautrier schlägt in seiner Veröffentlichung *„Toward Automatic Partitioniung Arrays in Distributed Memory Computers“* [Fea93] eine Methode vor, die automatisch für jedes Statement eines gegebenen Programmfragments, das den Eigenschaften in Abschnitt 2.3.2 entspricht, eine lineare Platzierungsfunktion π_S entsprechend Definition 2.22 berechnet.
Dabei wird als Modell zur Bewertung von Kommunikation angenommen, dass die Kosten zur Übermittlung einer Nachricht unabhängig von der Position des Quell- und des Zielprozessors sind, weshalb für lokale Arrayzugriffe keine und für nicht lokale Arrayzugriffe sehr hohe, aber konstante Kosten entstehen.
Aufgrund dieser Annahme werden die Platzierungen der Iterationen der Statements im Programm so gewählt, dass zwei voneinander abhängige Operationen auf dem gleichen Prozessor ausgeführt werden.
Dabei wird für jede Abhängigkeit $S \to S'$, beschrieben durch die h-Transformation $h = (D_h, f_h)$, folgende Bedingung aufgestellt:

$$\pi_S(f_h(\mathbf{i})) = \pi_{S'}(\mathbf{i}) \tag{22}$$

Erfüllen die beiden Platzierungen π_S und $\pi_{S'}$ Bedingung 22, dann ist garantiert, dass für Abhängigkeit h eine Iteration $\mathbf{i}$ des Statements S' auf dem gleichen Prozessor ausgeführt wird wie die Iteration $f_h(\mathbf{i})$ des Statements S.
Die Platzierungsfunktionen für die Statements werden anschließend so gewählt, dass sie alle Bedingungen erfüllen, die sich aus den Abhängigkeiten ergeben.
Aufgrund der gegebenen Abhängigkeiten kann dabei folgende Problematik auftreten: Werden alle Abhängigkeiten zur Bestimmung der Platzierung π_S berücksichtigt, dann können die Iterationen des Statements S nur auf einem einzigen Prozessor ausgeführt werden, weshalb keine Parallelität mehr ausgenutzt werden kann.
Zur Lösung dieser Problematik werden nicht alle sondern nur so viele Abhängigkeiten zur Bestimmung der Platzierungsfunktionen verwendet, dass durch die Platzierung π_S nicht alle Iterationen des Statements S auf einen Prozessor abgebildet werden. Für die nicht berücksichtigten Abhängigkeiten muss Kommunikation erzeugt werden. Um diese möglichst gering zu halten, werden die Abhängigkeiten nach folgender Heuristik ausgewählt: Zuerst sind diejenigen Abhängigkeiten zu berücksichtigen, die als sehr teuer eingestuft werden, da sie ein großes Kommunikationsvolumen erzeugen. Diese Abhängigkeiten werden von den innersten Schleifen getragen, da für jede Iteration der äußeren Schleifen Kommunikation zu erzeugen ist.

Die hier vorgestellte Platzierungsmethode nach Feautrier erzeugt für jedes Statement S eine Platzierung π_S, die die Iterationen des Statements S auf ein virtuelles mehrdimensionales Prozessorfeld abbildet, wobei zu beachten ist, dass für jedes Statement das gleiche Prozessorfeld verwendet wird.

Genau wie die oben dargestellte Methode nach Feautrier geht auch diese Arbeit davon aus, dass Kommunikation zu erzeugen ist, sobald Berechnungen nicht auf exakt den gleichen Koordinaten innerhalb eines Polyeders platziert werden. Im Gegensatz zu Feautrier's Methode werden in dieser Arbeit auch Replikation betrachtet und mit Hilfe eines flexiblen Kostenmodells Möglichkeiten zur Verfügung gestellt, um unterschiedliche Kommunikationsmuster zu ermitteln und gegeneinander zu evaluieren. Die Flexibilität ist dabei jedoch durch exponentielle Laufzeit der Platzierungsmethode zu bezahlen.

9.2 Platzierungsmethode des dHPF-Compilers

Der dHPF-Compiler [AJMCY98, AMC98, AMC01] für High Performance Fortran ist an der Rice Universität mit dem Ziel entwickelt worden, Optimierungstechniken zu untersuchen, um für ein breites Spektrum an verschiedenen Anwendungen effizienten parallelen Code zu generieren, wobei nur eine minimale Restrukturierung des originalen Fortran-Programms von Seiten des Programmierers erforderlich sein soll. Um dies zu erreichen, ist zur internen Darstellung der parallelen Schleifen und zur Codegenerierung teilweise das Polyedermodell eingesetzt und eine Methode zur Verteilung der Berechnungen entworfen worden, die eine Verallgemeinerung der Owner-Computes-Rule ist (vgl. Abschnitt 5.1).

In diesem Abschnitt soll nur die Methode der Verteilung der Berechnungen berücksichtigt werden, da sie der in dieser Arbeit vorgestellten Platzierungsmethode ähnlich ist. Die Verteilung der Iterationen der Statements auf die Prozessoren wird im Folgenden „Partitionierung der Berechnungen“ (CB) genannt. Dabei beschreibt eine CB(S) eines Statements S, die Verteilung der Iterationen des Statements S auf die Prozssoren.

Die im dHPF-Compiler verwendete Partitionierung der Berechnungen ist eine Verallgemeinerung der Owner-Computes-Rule, da der dHPF-Compiler ein Statement entsprechend der Verteilung eines der in dem Statement verwendeten Arrays auf die Prozessoren verteilen kann. Als Beispiel sei folgender Schleifensatz gegeben:

```
   DO i=1,n
S:    A(i) = B(i) + C(i)
   END DO
```

Der dHPF-Compiler kann das Statement S entsprechend der Verteilung der Arrays `A`, `B` oder `C` auf die Prozessoren verteilen. Die Owner-Computes-Rule ist also ein einfacher Spezialfall dieser Methode, da dabei nur die Verteilung des Arrays `A` berücksichtigt wird. Zusätzlich lässt das Modell der Partitionierung der Berechnungen zu, dass die CB eines Statements als eine Vereinigung mehrerer `ON HOME`-Direktiven dargestellt werden kann.

Der dHPF-Compiler unterscheidet bei der Berechnung von CBs für ein Statement Zugriffe auf privatisierbare und nicht privatisierbare Arrays. Ein privatisierbares Array wird dazu verwendet, temporäre Werte in einem Schleifensatz zu halten. Ein Array ist daher genau dann in einer Schleife privatisierbar, wenn jedem Element des Arrays, das in einer Iteration der Schleife verwendet wird, auch vorher in dieser Iteration ein Wert zugewiesen worden ist. Außerdem wird auf die in der Schleife berechneten Werte außerhalb der Schleife nicht zugegriffen. Für State-

ments, die Zuweisungen an privatisierbare Arrays repräsentieren, werden die CBs so berechnet, dass beim Lesezugriff auf die privatisierten Arrays keine Kommunikation erzeugt wird.
Zusätzlich berechnet der dHPF-Compiler CBs für Verzweigungstatements (wie `DO` und `IF`), um effizienten parallelen Code zu erzeugen. Aus diesen Gründen werden die Statements in zwei Klassen aufgeteilt:

1. Die Klasse der primären Statements enthält
   - Zuweisungen zu nicht privatisierbaren Arrays,
   - `CALL`-Statements und
   - Ein/Ausgabe-Statements.
2. Die Klasse der Hilfsstatements enthält alle anderen Statements, die nicht zu den primären Statements zählen, wie Zuweisungen zu privatisierbaren Arrays oder wie Verzweigungstatements. Die Platzierung dieser Statements wird durch Propagierung der Platzierungen der primären Statements berechnet.

Zuerst soll der Algorithmus zur Berechnung von Platzierungen der primären Statements vorgestellt werden. Für jede Schleife (in bottom-up Reihenfolge) in einer Prozedur arbeitet der Algorithmus wie folgt: Zuerst wird eine Menge von verschiedenen Platzierungsmöglichkeiten für jedes primäre Statement in der Schleife berechnet, wobei sich die verschiedenen Möglichkeiten aus den Platzierungen aller im Statement verwendeten Arrays ergeben, die nicht privatisierbar sind. Dann wird für alle Statements in der Schleife für alle möglichen Kombinationen von Platzierungsmöglichkeiten die entstehende Kommunikation betrachtet und diejenige Kombination von Platzierungen für die Statements ausgewählt, die eine Kommunikation mit minimalen Kosten besitzt. Dazu wird als Kostenfunktion die Kommunikationsfrequenz bestimmt.
Nachdem die Partitionierung der Berechnungen für die primären Statements ausgewählt wurde, berechnet der nachfolgende Algorithmus die Platzierungen der Hilfsstatements. Dieser Algorithmus hat zwei grundlegende Ziele:

- Das erste Ziel ist, die mögliche Parallelität bei gegebenem Kontrollfluss vollständig auszunutzen, wobei die Semantik des Programms erhalten bleiben muss. Um dies zu erreichen, wird die minimale Menge von Prozessoren ausgewählt, auf denen ein Kontrollflusstatement auszuführen ist.
- Das zweite Ziel ist, dass die Werte der privatisierbaren Variablen nur auf den Prozessoren berechnet werden, die die Werte der privatisierbaren Variablen verwenden, um teure Replikation von Berechnungen oder um Kommunikation für diese Variablen zu vermeiden.

Um die für die primären Statements bereits berechneten Platzierungen zu den Hilfsstatements zu propagieren, wird der CB-Abhängigkeitsgraph $G_{CB} = (V, E)$ erzeugt, dessen Knotenmenge V alle Statements im Programm enthält. In den CB-Abhängigkeitsgraph wird genau dann eine Kante von $v \rightarrow v'$ mit $v, v' \in G_{CB}$ einfügt, wenn die Platzierung von v' aus der Platzierung von v zu berechnen ist.
Der G_{CB} wird aus dem Kontrollflussgraphen und der Single-Assignment-Form des Programms konstruiert. Dabei interessiert die Konstruktion von G_{CB} aus dem Kontrollflussgraphen nicht, da sie nur zur korrekten Erzeugung von Code verwendet wird. Von Bedeutung ist die Konstruktion aus der Single-Assignment-Form des Programms, da sie zur Behandlung der privatisierbaren Arrays Anwendung findet. Für jede skalare Variable, die in einer Schleife privatisierbar

ist, wird die Kante $v \rightarrow v'$ in E eingefügt, wenn v das Statement ist, in dem der skalaren Variable ein Wert zugewiesen wird, und v' ein Statement ist, in dem die skalare Variable gelesen wird. Zudem werden für privatisierbare Arrays solche Kanten eingefügt, wobei eine genauere Behandlung von den Zugriffsfunktionen benötigt wird, um nicht notwendige Replikation zu vermeiden.

Ist der CB-Abhängigkeitsgraph G_{CB} erzeugt, werden die Platzierungen eines Knotens durch den Graphen propagiert. Dabei wird ein Zyklus im Graphen speziell behandelt: Allen Knoten im Zyklus wird die gleiche Platzierung zugewiesen, die ausschließlich von den Indexvariablen der Schleifen abhängen darf, die alle Statements des Zyklus umschließen. Daher werden zuerst die Platzierung von Knoten außerhalb des Zyklus zu Knoten innerhalb des Zyklus propagiert. Anschließend werden für jeden Knoten alle Schleifenindexvariablen, die keine Schleifenvariablen von umgebenden Schleifen sind, eliminiert, indem sie durch den Bereich ersetzt werden, der durch die Unter- und Obergrenzen der entsprechenden Schleifen gegeben ist. Abschließend wird jedem Knoten im Zyklus die Vereinigung aller Platzierungen der Knoten im Zyklus zugewiesen.

Die Platzierungsmethode des dHPF-Compilers erzeugt ähnlich wie die in dieser Arbeit vorgestellte Methode dadurch Platzierungen für Hilfsarrays[3], dass Platzierungen in einem CB-Abhängigkeitsgraphen propagiert werden. Der Unterschied zu der in dieser Arbeit vorgestellten Methode liegt darin, dass der dHPF-Compiler zur Berechnung der Platzierungen der Hilfsarrays nur Platzierungen derjenigen Statements berücksichtigt, die Werte von den Hilfsarrays lesen. Dagegen berücksichtigt das in dieser Arbeit vorgestellte Vorgehen auch zusätzlich Platzierungen der Arrays, die zur Berechnung der Werte für die Hilfsarrays verwendet werden.

9.3 Evaluierung der Kommunikation für den HPF-Builder

Der HPF-Builder, der an der Universität von Lille entwickelt worden ist, stellt für ein HPF-Programm die Verteilung der Arrays auf die Prozessoren graphisch dar [DL97]. Dadurch dass die durch `ALIGN`-Direktiven definierten Beziehungen der unterschiedlichen Arrayelemente zueinander graphisch dargestellt werden, erhält der Programmierer von HPF-Programmen Hilfestellung bei der Verteilung der Arrays.

Um eine gegebene Verteilung im HPF-Builder zu bewerten, soll, wie Boulet und Redon [BR98] vorschlagen, das Kommunikationsvolumen eines parallelen Schleifensatzes bestimmt werden. Die Autoren nehmen an, dass dabei ein kleineres Kommunikationsvolummen eine bessere Laufzeit verursacht. So lässt sich der Programmierer das Kommunikationsvolumen für unterschiedliche Verteilungen berechnen und kann die Verteilung wählen, die das geringste Kommunikationsvolumen besitzt.

Das Kommunikationsvolumen wird auf der Ebene der Sprache HPF berechnet, indem die `ALIGN`-, `DISTRIBUTE`- und die `PROCESSORS`-Direktive berücksichtigt werden. Als Modellierung des Iterationsraums der Schleifen kommt das Polyedermodell (siehe Abschnitt 2.3) zur Anwendung. Es werden als einzige kommunikationserzeugende Statements Zuweisungen betrachtet. Die zu analysierenden Statements im Programm werden in Storage-Statements aufgeteilt, die nur zwei unterschiedlich verteilte Arrays referenzieren. Ein Storage-Statement S hat folgende Form:

$$S: \qquad \mathtt{A}(\varphi_{\mathtt{A}}(\mathbf{i})) = f_S(\mathtt{B}(\varphi_{\mathtt{B}}(\mathbf{i})))$$

Dabei sind $\varphi_{\mathtt{A}}$ und $\varphi_{\mathtt{B}}$ die Zugriffsfunktionen der beiden Arrays, $\mathbf{i}$ der Iterationsvektor mit

[3] Die Hilfsarrays werden im dHPF-Compiler privatisierbare Arrays genannt.

$\mathbf{i} \in \mathcal{IS}(S)$ und f_S die durch das Statement S gegebene Berechnungsfunktion. Die Beziehungen der Arrayelemente zwischen beiden Arrays eines Storage-Statements sind bekannt, weil beide Arrays an das gleiche Template `T` ausgerichtet sind.
Da in HPF Replikation darstellbar ist (siehe Abschnitt 2.1.4) und dies bei der Berechnung der Kommunikation berücksichtigt werden soll, wird zur Darstellung der Verteilung die `ALIGN`-Direktive verwendet. Wie in dieser Arbeit in Abschnitt 3.1.1 vorgestellt, werden auch hier zur Modellierung von `ALIGN`-Direktiven Paare von linearen Abbildungen verwendet. So wird die Verteilung des Arrays `A` durch das Paar $r_{\mathtt{A}} = (r_{\mathtt{A}_L}, r_{\mathtt{A}_R})$ und die des Arrays `B` durch das Paar $r_{\mathtt{B}} = (r_{\mathtt{B}_L}, r_{\mathtt{B}_R})$ dargestellt.
Um die Kommunikationskosten zu bestimmen, wird die Anzahl der nötigen Kommunikationen zwischen den Template-Elementen gezählt. Dazu wird zuerst für ein Template-Element $\mathbf{j} \in \mathcal{D}_{\mathtt{T}}$ die Menge W der Schleifeniterationen bestimmt, die einen Wert von `A` berechnen, der diesem Template-Element $\mathbf{j}$ zugeordnet ist. Die Menge W enthält genau dann mehrere Elemente, wenn eine Dimension des Arrays `A` repliziert gespeichert ist. Die Menge W ergibt sich aus der Platzierung $r_{\mathtt{A}}$ und der in Statement S verwendeten Zugriffsfunktion $\varphi_{\mathtt{A}}$ wie folgt:

$$W = (\varphi_{\mathtt{A}}^{-1} \circ r_{\mathtt{A}_L}^{-1} \circ r_{\mathtt{A}_R})(\mathbf{j})$$

Für diese Menge W sind die Arrayelemente von `B` zu bestimmen, die für diese Schleifeniterationen gelesen und möglicherweise kommuniziert werden müssen. Dazu wird für jede Iteration $\mathbf{i} \in W$ bestimmt, auf welchem Template-Element das durch $\mathbf{i}$ referenzierte Arrayelement von `B` liegt. Die Menge $R_{\mathbf{i}}$ der Template-Elemente, auf denen das Arrayelement von `B` für die Iteration $\mathbf{i}$ liegt, wird wie folgt bestimmt:

$$R_{\mathbf{i}} = r_{\mathtt{B}_R}^{-1} \circ r_{\mathtt{B}_L} \circ \varphi_{\mathtt{B}}(\mathbf{i})$$

Gilt $\mathbf{j} \in R_{\mathbf{i}}$ für ein gerade berechnetes $R_{\mathbf{i}}$, dann ist die Kommunikation für die Iteration $\mathbf{i}$ und für das Template-Element $\mathbf{j}$ lokal, da die entsprechenden Arrayelemente beide dem gleichen Template-Element zugeordnet sind. Somit berechnet sich für ein Template-Element $\mathbf{j}$ die Menge $C_{\mathbf{j}}$ der zu kommunizierenden Elemente folgendermaßen:

$$C_{\mathbf{j}} = \left\{ (\mathbf{j}, \mathbf{i}) \mid \mathbf{i} \in (\varphi_{\mathtt{A}}^{-1} \circ r_{\mathtt{A}_L}^{-1} \circ r_{\mathtt{A}_R})(\mathbf{j}), \;\; \mathbf{j} \notin r_{\mathtt{B}_R}^{-1} \circ r_{\mathtt{B}_L} \circ \varphi_{\mathtt{B}}(\mathbf{i}) \right\}$$

Um das Gesamtvolumen zu berechnen, ist das Kommunikationsvolumen für jedes Template-Elements zu berücksichtigen. Somit berechnet sich das Gesamtvolumen der Kommunikation für das Storage-Statement S aus der Vereinung aller $C_{\mathbf{j}}$ wie folgt:

$$C = \bigcup_{\mathbf{j} \in \mathcal{D}_{\mathtt{T}}} C_{\mathbf{j}}$$

Da die Menge C durch Polyeder dargestellt wird, kann ihr Volumen durch Ehrhart-Polynome [CL96] berechnet werden.
Ähnlich wie beim HPF-Builder verwendet auch die in dieser Arbeit vorgestellte Methode zur Darstellung einer Relation ein Paar von linearen Abbildungen, um Replikation repräsentieren zu können. Doch betrachtet sie kein Kommunikationsvolumen, da bei der Auswertung des Benchmarks für ADAPTOR keine direkte Abhängigkeit zwischen der Laufzeit und dem Kommunikationsvolumen festgestellt werden konnte.

10 Zusammenfassung

In dieser Arbeit wurde eine Platzierungsmethode für Berechnungsinstanzen vorgestellt, die zur Platzierung derjenigen Stelleninstanzmengen eingesetzt wird, für die die Methode LCCP Hilfsarrays anlegt. Dabei wird eine Kostenfunktion zur Berechnung der Platzierungen verwendet, die der Platzierungsmethode Informationen über die günstigste Verteilung liefert. Als Fallbeispiel für eine Kostenfunktion wurde in dieser Arbeit ein Kostenmodell erstellt, das den HPF-Compiler ADAPTOR und einen hpcLine-Parallelrechner mit SCI-Netz berücksichtigt.
Die hier vorgestellte Platzierungsmethode verwendet als Darstellung des Programmfragments den reduzierten Stelleninstanzgraphen, wobei bereits ausgewählte Knoten eine initiale Platzierung besitzen. Aus dieser initialen Platzierung wird die Platzierung der von LCCP als interessant markierten Stelleninstanzmengen in drei Schritten berechnet:

1. Für jede interessante Stellenistanzmenge wird eine Menge von möglichen Platzierungen dadurch bestimmt, dass die initialen Platzierungen entlang der Kanten im reduzierten Stelleninstanzgraphen zu den interessanten Knoten ohne initiale Platzierung propagiert werden.

2. Im zweiten Schritt erfolgt die Nachbearbeitung der Menge der möglichen Platzierungen. Da bei der Propagierung Platzierungen entstanden sein können, die nicht durch `ALIGN`-Direktiven darzustellen sind, wird die Menge möglicher Platzierungen eines jeden Knoten auf nicht HPF-konforme Platzierungen untersucht und diese zu HPF-konformen transformiert.

3. Im dritten Schritt wird für jede Kombination von möglichen Platzierungen die auftretende Kommunikation berechnet und durch die Kostenfunktion bewertet. Dabei wird diejenige Kombination von Platzierungen gewählt, die durch die Kostenfunktion mit den geringsten Kosten bewertet wurde.

Im ersten Schritt trat bei der Propagierung folgende Problematik auf: Es können Platzierungen erzeugt werden, die Template-Elemente referenzieren, die nicht im entsprechenden Template zur Verfügung stehen, was der HPF-Standard nicht zulässt. Es zeigte sich dabei, dass diese Problematik durch Partitionierung des reduzierten Stelleninstanzgraphen zwar behoben werden kann, dass der Graph aber möglicherweise unendlich oft zu partitionieren ist, damit die Platzierungen auf beliebigen Wegen propagiert werden können. Um die Platzierungen sowohl in Richtung der Abhängigkeit von der Quelle zum Ziel als auch vom Ziel zur Quelle im reduzierten Stelleninstanzgraphen zu propagieren, wird die Propagierung in zwei Schritten durchgeführt: Zuerst erfolgen die Partitionierung und Propagierung der initialen Platzierungen in Richtung Quelle zum Ziel der Abhängigkeiten worauf sich die Partitionierung und Propagierung der initialen Platzierungen in Richtung Ziel zur Quelle anschließen. Damit wird garantiert, dass Platzierungen erzeugt werden, die nur Template-Elemente referenzieren, die im entsprechenden Template zur Verfügung stehen.
An dieser Stelle wäre es interessant, weiter zu untersuchen, ob es sinnvoll ist, die Propagierung dahingehend zu erweitern, dass nicht nur die initialen Platzierungen sondern alle bereits erzeugten Platzierungen nach einer Partitionierung des Graphen propagiert werden. Ist dies sinnvoll, dann ist zu untersuchen, ob alle Platzierungen berücksichtigt werden können, indem die Propagierung öfter in abwechselnder Richtung durchgeführt wird. Allerdings ist dabei zu beachten, dass dabei eine große Anzahl von Partitionierungen enstehen kann, für die sowohl Platzierungen berechnet werden müssen als auch Code erzeugt werden muss.

Nachdem im ersten Schritt alle möglichen Platzierungen erzeugt worden sind, werden sie im zweiten Schritt nachbearbeitet. An dieser Stelle wäre noch zu untersuchen, ob durch die in Abschnitt 3.3.1 vorgestellte Kombination von Platzierungen effizientere Platzierungen konstruiert werden können.
Im letzten Schritt wird, basierend auf einer Kostenfunktion, die Kombination von Platzierungen ausgewählt, die in Beziehung auf diese Kostenfunktion die beste Kommunikation erzeugt. Zur Darstellung der Kommunikation zwischen zwei Knoten im reduzierten Stelleninstanzgraphen wird eine Kommunikationsrelation berechnet, die die Beziehung zwischen den Template-Elementen darstellt, auf die für diese Kommunikation zuzugreifen ist. Bei der Berechnung der Kommunikationsrelation ist bereits berücksichtigt, dass ein lesender Zugriff auf replizierte Daten ein lokaler Zugriff ist. Die gesamte Kommunikation des Programmfragments wird durch den Kommunikationsgraphen dargestellt, der von der Kostenfunktion bewertet wird.
Der HPF-Compiler ADAPTOR wurde in dieser Arbeit dazu verwendet, die von LCCP transformierten Programme für den Parallelrechner des Lehrstuhls zu übersetzen. Es wurde ein Benchmark für einen HPF-Compiler entwickelt, der aus verschiedenen Testfällen zur Bewertung von Kommunikationsmustern besteht. Dieser Benchmark kann für eine beliebige Kombination an HPF-Compiler und Parallelrechner verwendet werden, um die Kommunikation zu bewerten. In dieser Arbeit wurde der Benchmark für den Compiler ADAPTOR gemessen. Dabei zeigte es sich, dass die Kosten für die Kommunikation hauptsächlich vom Kommunikationsmuster und nicht vom Kommunikationsvolumen abhängen. Basierend auf der Auswertung der Messergebnisse dieses Benchmarks wurde ein Kostenmodell für ADAPTOR vorgestellt, das nur die Zugriffsmuster der Kommunikationsrelationen bewertet.
Zeigt der Benchnchmark die gleichen Ergebnisse für eine andere Kombination von HPF-Compiler und Parallelrechner, nämlich dass die Kosten für die Kommunikation hauptsächlich vom Kommunikationsmuster abhängen, dann kann das vorgestellte Kostenmodell auch hier verwendet werden, wobei nur die Koeffizienten im Kostenmodell entsprechend den neuen Messwerten abzuändern sind.
Auch wäre es interessant zu untersuchen, welche Verfeinerungen des Kostenmodells sinnvoll wären, z.B. wie die Replikation bewertet und im Bezug auf die Kommunikation sinnvoll gewichtet werden soll.
Darüber hinaus wurde anhand eines Schleifensatzes aus einer astrophysikalischen Simulation gezeigt, dass eine Laufzeitverbesserung zu erzielen ist, wenn der Schleifensatz mit LCCP transformiert worden ist und die Hilfsarrays Platzierungen erhalten haben, die nach dem Verfahren berechnet worden sind, das in dieser Arbeit vorgestellt wurde.
Die Laufzeittests, die während der Implementierung der in dieser Arbeit vorgestellten Platzierungsmethode für verschiedene kleine synthetische Testprogramme durchgeführt wurden, haben gezeigt, dass gute Laufzeitverbesserungen ähnlich wie in Abschnitt 8 erzielt werden können, wenn Berechnungen eingespart werden, die entweder „teure" Berechnungen wie z.B. die Berechnung von Sinus oder Arrayzugriffe wie `C(i,i)` darstellten, für die ADAPTOR sehr ineffizienten parallelen Code erzeugt (vgl. Testfall $\texttt{general}_{2D}$ aus Abschnitt 6.3.2).
Es zeigte sich jedoch, dass „billige" Berechnungen wie z.B. die Addition oder Multiplikation um mindestens eine Größenordnung eingespart werden müssen, ehe eine Verbesserung der Laufzeit erreicht werden kann, da ansonsten der Zugriff auf die von LCCP eingeführten Hilfsarrays mehr Zeit verbrauchen würde, als durch die Einsparung der Berechnung gewonnen wird. Dieser Effekt wird zusätzlich verstärkt, wenn Partitionierung bei der Berechnung von Platzierungen benötigt wurde.
Um Beispiele aus wissenschaftlichen Anwendungen zu finden, wurden im Vorfeld dieser Arbeit

verschiedene numerische Anwendungen untersucht, wobei auffiel, dass zur Effizienzsteigerung dort per Hand jede Berechnung, die mehrfach benötigt wird, in Hilfsarrays bzw. Hilfsvariablen zwischengespeichert wird. Die Methode LCCP leistet diese Optimierung vollständig automatisch.

Notation

Graphen

V	Die Menge der Knoten eines Graphen
E	Die Menge der Kanten eines Graphen
v	Ein Knoten im Graphen mit $v \in V$
e	Eine Kante im Graphen mit $e \in E$
G	Ein Graph $G = (V, E)$
G_{OI}	Ein Stelleninstanzgraph
G_{ROI}	Ein reduzierter Stelleninstanzgraph
O_v	Die Stelleninstanzmenge eines Knoten des G_{ROI}
G_{COI}	Ein kondensierter Stelleninstanzgraph
G_{RCOI}	Ein reduzierter, kondensierter Stelleninstanzgraph
G_{comm}	Ein Kommunikationsgraph
G_{comp}	Ein kompaktifizierter Graph

Polyedermodell

`A`	Ein beliebiges Array
`T`	Ein beliebiges Template
$\mathcal{D}_{\mathtt{A}}$	Der Datenraum des Arrays `A`
$\varphi_{\mathtt{A}}(\mathbf{i})$	Die Zugriffsfunktion eines Arrays `A` mit dem Indexvektor $\mathbf{i}$
S	Ein Statement
$\mathcal{IS}(S)$	Die Indexmenge eines Statements
$\mathbf{i}$	Indexvektor, $\mathbf{i} \in \mathcal{IS}$
Ω	Die Menge der Operationen eines Programmfragmentes
$o_1 \delta o_2$	Die Abhängigkeit der beiden Operationen $o_1, o_2 \in \Omega$: o_2 ist abhängig von o_1
$h = (D_h, f_h)$	Eine h-Transformation h, wobei D_h die Definitionsmenge der linearen Funktion f_h ist, die die Abhängigkeit beschreibt
D_h	Der Definitionsraum einer h-Transformation h
θ_S	Der Schedule des Statements S
π_S	Die Platzierung des Statements S als lineare Funktion

π_S	Die Platzierung des Arrays `A` als lineare Funktion
n_v	Die Anzahl der Indizes im Programmfragment
n_b	Die Anzahl der Parameter im Programmfragment
n_{cpu}	Die Anzahl der Prozessoren
m_c	Der Parameter m_c repräsentiert die Konstante 1
m_∞	Der Parameter m_∞ repräsentiert die Konstante ∞

LCCP

$\mathcal{F}$	Die Menge der Funktionssymbole
τ	Ein Term, konstruiert aus der Menge $\mathcal{F} \cup \{;\}$
$\mathrm{Occ}(\tau)$	Die Funktion, die die Stelle eines Terms τ zurückgibt
OS	Die Menge aller Stelleninstanzen
α	Eine Stelleninstanz mit $\alpha \in OS$
$\alpha_1 \Delta \alpha_2$	Die Abhängigkeit der beiden Stelleninstanzen $\alpha_1, \alpha_2 \in OS$: α_2 ist abhängig von α_1
$\pi_{Occ}(\alpha)$	Die Projektion auf die Dimension, die die Stelle der Stelleninstanz repräsentiert
$\mathrm{Idx}(O_v)$	Die Projektion auf die Indexmenge einer Stelleninstanz Menge O_v. Dabei werden alle Gleichungen entfernt, die den Parameter m_∞ referenzieren
$r = (r_L, r_R)$	Eine lineare Relation

Platzierungen

$\mathcal{P}$	Der Raum der virtuellen Prozessoren $\mathcal{P}$, der alle Dimensionen enthält, die für die im Programmfragment verwendeten Templates benötigt werden
n_{pDim}	Die Anzahl der Prozessordimensionen von $\mathcal{P}$
$\tilde{\Phi}_{\mathtt{A}} = (\tilde{\Phi}_{\mathtt{A}_{OI}}, \tilde{\Phi}_{\mathtt{A}_P})$	Eine Platzierung eines Arrays `A`, dargestellt als lineare Relation
$\Phi v = (\Phi_{v_{OI}}, \Phi_{v_P})$	Platzierung einer Stelleninstanzmenge eines Knoten v im G_{ROI}
$\mathcal{A}_v$	Die Menge aller möglichen Platzierungen eines Knoten eines reduzierten Stelleninstanz Graphen mit $\mathcal{A}_v = \{\Phi_v^1, \ldots, \Phi_v^n\}$
$\mathrm{sel}_i(\mathcal{A}_v)$	Diese Funktion wählt die i-te Platzierung Φ_v^i von $\mathcal{A}_v$ aus

Literatur

[AJMCY98] Adve, Vikram, Guohua Jin, John Mellor-Crummey und Qing Yi: *High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes.* In: *Proceedings of Supercomputing 98: High Performance Computing and Networking*, Orlando, FL, November 1998.

[AMC98] Adve, Vikram und John Mellor-Crummey: *Using Integer Sets for Data-Parallel Program Analysis and Optimization.* In: *SIGPLAN Conference on Programming Language Design and Implementation*, Seiten 186–198, 1998.

[AMC01] Adve, Vikram und John Mellor-Crummey: *Advanced Code Generation for High Performance Fortran.* In: *Compiler Optimizations for Scalable Parallel Systems Languages*, Seiten 553–596, New York, USA, 2001. Springer-Verlag New York.

[BHdW+95] Bailey, David, T. Harris, Rob Van der Wigngaart, William Saphir, Alex Woo und Maurice Yarrow: *The NAS Parallel Benchmarks 2.0.* Technical Report NAS-95-01, NASA Ames Research Center, 1995.

[BR98] Boulet, Pierre und Xavier Redon: *Communication Pre-Evaluation in HPF.* In: *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Seiten 263–272, London, UK, 1998. Springer-Verlag.

[Bra00] Brandes, Thomas: *ADAPTOR Distributed Array Library (DALIB) Version 8.0.* GMD - German National Research Center for Information Technology, Dezember 2000. Manual.

[BZ94] Brandes, Thomas und Falk Zimmermann: *ADAPTOR-A Transformation Tool for HPF Programms.* In: K.M.Decker und R.M.Rehmann (Herausgeber): *Programming Environments for Massively Parallel Distributed Systems*, Seiten 91–96. Birkhäuser Verlag, 1994.

[CFH+92] Choudhary, Alok, Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles H. Koelbel, Sanjay Ranka und Chau-Wen Tseng: *Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines.* In: *Fourth Symposium on the Frontiers of Massively Parallel Computation*, Seiten 4–11, McLean, Va., 1992. Washington, D.C.: IEEE Computer Society Press.

[CGP96] Coelho, Fabien, Cecile Germain und Jean-Louis Pazat: *State of the Art in Compiling HPF.* In: *The Data Parallel Programming Model*, Lecture Notes in Computer Science, Seiten 104–133. Springer Verlag, 1996.

[CL96] Clauss, Philippe und Vincent Loechner: *Parametric Analysis of Polyhedral Iteration Spaces.* In: *IEEE International Conference on Application Specific Array Processors, ASAP'96.* IEEE Computer Society, August 1996.

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

[DL97] DEKEYSER, JEAN-LUC und CHRISTIAN LEFEBVRE: *HPF-Builder: A Visual Environment to Transform Fortran 90 Codes to HPF.* The International Journal of Supercomputer Applications and High Performance Computing, 11(2):95–102, Summer 1997.

[DR94] DARTE, ALAIN und YVES ROBERT: *Mapping Uniform Loop Nests onto Distributed Memory Architectures.* Parallel Compututing, 20(5):679–710, 1994.

[Fea91] FEAUTRIER, PAUL: *Dataflow Analysis of Array and Scalar References.* International Journal of Parallel Programming, 20(1):23–53, 1991.

[Fea92a] FEAUTRIER, PAUL: *Some Efficient Solutions to the Affine Scheduling Problem: Part I. Onedimensional Time.* International Journal of Parallel Programming, 21(5):313–348, 1992.

[Fea92b] FEAUTRIER, PAUL: *Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time.* International Journal of Parallel Programming, 21(6):389–420, 1992.

[Fea93] FEAUTRIER, PAUL: *Toward Automatic Partitioning of Arrays on Distributed Memory Computers.* In: *ICS '93: Proceedings of the 7th international conference on Supercomputing*, Seiten 175–184, New York, NY, USA, 1993. ACM Press.

[FGL01a] FABER, PETER, MARTIN GRIEBL und CHRISTIAN LENGAUER: *A Closer Look at Loop-Carried Code Replacement.* In: *Proc. GI/ITG PARS'01*, PARS-Mitteilungen Nr.18, Seiten 109–118. Gesellschaft für Informatik e.V., November 2001.

[FGL01b] FABER, PETER, MARTIN GRIEBL und CHRISTIAN LENGAUER: *Loop-Carried Code Placement.* In: SAKELLARIOU, RIZOS, JOHN KEANE, JOHN GURD und LEN FREEMAN (Herausgeber): *Euro-Par 2001: Parallel Processing*, Lecture Notes in Computer Science 2150, Seiten 230–234. Springer-Verlag, 2001.

[FGL04] FABER, PETER, MARTIN GRIEBL und CHRISTIAN LENGAUER: *Polyhedral Loop Parallelization: The Fine Grain.* In: GERNDT, MICHAEL und EDMOND KEREKU (Herausgeber): *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, Seiten 25–36. LRR-TUM, Technische Universität München, Juli 2004.

[Fos94] FOSTER, IAN: *Designing and Bulding Parallel Programs.* Addition Wesley, 1994.

[FvDF+94] FOLEY, JAMES, ANDRIES VAN DAM, STEVEN K. FEINER, JOHN F. HUGHES und RICHARD L. PHILLIPS: *Introduction to Computer Graphics.* Addison-Wesley, 1994.

[GL94] GRIEBL, MARTIN und CHRISTIAN LENGAUER: *On the Space-Time Mapping of WHILE-Loops.* Parallel Processing Letters, 4:221–232, 1994.

[GL95] GRIEBL, MARTIN und CHRISTIAN LENGAUER: *A Communication Scheme for the Distributed Execution of Loop Nests with* while *Loops.* International Journal of Parallel Programming, 23(5):471–495, 1995.

[GL96] GRIEBL, MARTIN und CHRISTIAN LENGAUER: *The Loop Parallelizer LooPo.* In: GERNDT, MICHAEL (Herausgeber): *Proc. Sixth Workshop on Compilers for Parallel Computers*, Band 21 der Reihe *Konferenzen des Forschungszentrums Jülich*, Seiten 311–320. Forschungszentrum Jülich, 1996.

[GLW98] GRIEBL, MARTIN, CHRISTIAN LENGAUER und SABINE WETZEL: *Code Generation in the Polytope Model.* In: *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Seite 106, Washington, DC, USA, 1998. IEEE Computer Society.

[Gri96] GRIEBL, MARTIN: *The Mechanical Parallelization of Loop Nests Containing* while *Loops.* Doktorarbeit, University of Passau, 1996. also available as technical report MIP-9701.

[Gri04] GRIEBL, MARTIN: *Automatic Parallelization of Loop Programs for Distributed Memory Architectures.* University of Passau, 2004.

[GWI92] GÓMEZ, ROBERTO, JEFFREY WINICOUR und R. ISAACSON: *Evolution of scalar fields from characteristic data.* Journal of Computational Physics, 98(1):11–25, 1992.

[Hig93] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification, Version 1.0.* Technischer Bericht CRPC-TR92225, Houston, Tex., 1993.

[Hig97] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification, Version 2.0*, 1997.

[HM95] HAUPT, TOM und MARK MILLER: *Gravitational Wave Extraction - Code.* `http://www.npac.syr.edu/projects/bbh/PITT_CODES/pitt_f77.html`, August 1995.

[Int91] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO/IEC 1539:1991*, 1991.

[Int97] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO/IEC 1539-1:1997*, 1997.

[Int04] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO/IEC 1539-1:2004*, 2004.

[KLS+94] KOEBEL, CHARLES, DAVID LOVEMAN, ROBERT SCHREIBER, GUY STEELE und MARY ZOSEL: *The High Performance Fortran Handbook.* MIT Press, Cambridge, Massachusetts, USA, 1994.

[KM98] KOWALSKY, HANS-JOACHIM und GERHARD O. MICHLER: *Lineare Algebra.* de-Gruyter, 1998.

[Lam74] LAMPORT, LESLIE: *The Parallel Execution of DO Loops.* Communications of the ACM, 17(2):83–93, 1974.

[Len93] LENGAUER, CHRISTIAN: *Loop Parallelization in the Polytope Model.* In: BEST, E. (Herausgeber): *CONCUR'93*, Lecture Notes in Computer Science 715, Seiten 398–416. Springer-Verlag, 1993.

[LG95] LENGAUER, CHRISTIAN und MARTIN GRIEBL: *On The Parallelization of Loop Nests Containing* WHILE *Loops.* In: *PAS '95: Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, Seite 10, Washington, DC, USA, 1995. IEEE Computer Society.

[Max46] MAXWELL, EDWIN: *Methods of Plane Projective Geometry Based on the Use of General Homogeneous Coordinates.* Cambridge University Press, 1946.

[Mes95] MESSAGE PASSING INTERFACE FORUM: *MPI: A Message Passing Interface Standard.* Technical Report, University of Tennessee, Knoxville, 1995.

[MR98] METCALF, MICHAEL und JOHN REID: *Fortran 90/95 explained.* Oxford science Publications, 3rd Auflage, 1998.

[MvR91] MEHROTRA, PIYUSH und JOHN VAN ROSENDALE: *Programming Distributed Memory Architectures Using Kali*, Band Advances in Languages and Compilers for Parallel Processing, Kapitel 19. Pitman Publishing, 1991.

[QRW00] QUILLERÉ, FABIEN, SANJAY RAJOPADHYE und DORAN WILDE: *Generation of Efficient Nested Loops from Polyhedra.* International Journal of Parallel Programming, 28(5):469–498, 2000.

[Sed03] SEDGEWICK, ROBERT: *Algorithms in Java*, Band 5. Addison Wesley, 3 Auflage, Juli 2003.

[SL03] SQUYRES, JEFFREY M. und ANDREW LUMSDAINE: *A Component Architecture for LAM/MPI.* In: *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Nummer 2840 in *Lecture Notes in Computer Science*, Seiten 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[SPB93] SU, ERNESTO, DANIEL J. PALERMO und PRITHVIRAJ BANERJEE: *Automating Parallelization of Regular Computations for Distributed-Memory Multicomputers in the PARADIGM Compiler.* In: *Proceedings of the 1993 International Conference on Parallel Processing*, Band II - Software, Seiten II–30–II–38, Boca Raton, FL, 1993. CRC Press.

[Sti05] STILLER, ANDREAS: *Von Xeon, Xen und Cell.* c't Magazin, 4/05:20–30, 2005.

[Uni05] UNIVERSITY, INDIANA: *MPI Implementation List.* `http://www.lam-mpi.org/mpi/implementations/fulllist.php`, August 2005.

[Usm87] USMANI, RIAZ A.: *Applied linear Algebra.* Marcel Dekker Inc., 1987.

[Val90] VALIANT, LESLIE G.: *A Bridging Model for Parallel Computation.* Communications of the ACM, 33(8):103–111, 1990.

[Wol95] WOLFE, MICHAEL: *Building High Performance Compilers For Parallel Computing.* Addition Wesley, 1995.

Index

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als soche gekennzeichnet sind, sowie dass die Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 09. November 2005

____