UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

Transformation von Shared-Memory-Programmen zu Distributed-Memory-Programmen

Diplomarbeit

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

Betreuer: Dr. Martin Griebl

Bearbeiter: Peter Faber

25. November 1997

Vorbemerkungen

Danksagung Zunächst möchte ich Herrn Prof. Christian Lengauer, Ph.D., meinen Dank aussprechen, insbesondere für seine zügige Korrektur und die Tatsache, daß es auch Stellen in dieser Arbeit gab, die danach nicht rot waren. Zu großem Dank verpflichtet bin ich besonders meinem Betreuer Dr. Martin Griebl, der mir in vielen Diskussionen mit seinem Rat zur Seite stand und mich oftmals auf den Boden der Tatsachen zurückholte – oder dies zumindest versuchte ... Meinen besten Dank auch an alle anderen Mitglieder des *LooPo*-Teams: Sven Anders, Andreas Dischinger, Nils Ellmenreich, Max Geigl, Robert Günz, Harald Keimer, Klaus Köbler, Radko Kubias, Wolfgang Meisl, Frank Schuler, Martina Schumergruber, Sabine Wetzel, Christian Wieninger und Alex Wüst. In diesem Team mitzuwirken, war mir ein Vergnügen. So ist diese Arbeit ein Ergebnis, jedoch weniger der Grund für meine Tätigkeit im *LooPo*-Team.

Bemerkungen zur Typographie Diese Arbeit wurde mit LaTeX 2ε unter Verwendung des AMS-LaTeX-Pakets erstellt.
Zur Hervorhebung bestimmter Wörter bzw. Textabschnitte werden dabei unterschiedliche Fonts verwandt:

- Zentrale Begriffe werden grundsätzlich *kursiv* gedruckt.
- Bei der Definition eines neuen Begriffs wird dieser **fett** gedruckt.
- Einmalige Hervorhebungen sind ebenfalls **fett** gedruckt.
- Programmtext ist in `Typewriter`-Schrift gesetzt:

  `for` $i = l$ `to` u

- In Formeln werden
  - gängige Funktionsnamen im Basisfont gedruckt: $\mathrm{ggT}(6, 9)$,
  - lediglich hier verwendete Funktions- oder Variablennamen *kursiv* gedruckt: $\mathit{extremize}(x, \min)$.

Zusammenfassung

Massiv parallele Rechnerarchitekturen mit verteiltem Speicher sind kostengünstiger und einfacher zu realisieren und zu erweitern als solche mit gemeinsamem Speicher. Allerdings ist die Programmierung paralleler Systeme mit verteiltem Speicher eine fehleranfällige Tätigkeit, da der Benutzer durch das Einfügen expliziter Kommunikationsroutinen in den Code garantieren muß, daß alle für eine Berechnung nötigen Daten auf dem berechnenden Prozessor vorhanden sind. Ziel dieser Arbeit ist es, aus Programmen, die Parallelität explizit über Parallelschleifen ausdrücken, jedoch von gemeinsamem Speicher ausgehen, insbesondere der Ausgabe von *LooPo*, SPMD-Programme mit expliziter Kommunikation zu generieren, die auf solchen Distributed-Memory-Maschinen ausgeführt werden können.

Inhaltsverzeichnis

1 Einführung **1**

2 Grundlagen **2**
2.1 Notation und Definitionen 2
2.2 Mathematische Grundlagen 7
2.2.1 Rechenregeln zur Gaußklammerfunktion 7
2.2.2 Matrixinversion 13

3 Spezifikation **16**
3.1 Maschinenmodell 16
3.2 Eingabespezifikation 16
3.3 Globale und lokale Räume 19
3.4 Prinzipielles Ausführungsmodell 20

4 Anweisungsverteilung **22**
4.1 Reduktion rationaler Schrittweiten 22
4.2 Prozessoren in $\mathbb{N}^m$ 24
4.3 Dreistufenmodell 25
4.4 Basisblöcke 29
4.5 Anweisungsverteilung in *LooPo* 31

5 Datenverteilung **34**
5.1 Dynamische und statische Datenallokation 34
5.2 Berechnung einer statischen Datenallokation 35
5.3 Darstellung einer *Modularabbildung* als Allokation 39
5.3.1 Darstellung einer blockzyklischen Allokation 40
5.3.2 Verallgemeinerung der blockzyklischen Allokation 41
5.3.3 Allokation durch mehrdimensionale *Modularabbildungen* . . . 47
5.4 Reallokation und Host-Kommunikation 58
5.5 Datenverteilung in *LooPo* 59

6 Kommunikation **62**
6.1 Synchronisationspunkte 62
6.2 `LOAD` und `STORE` 65
6.3 `LOAD` und `STORE` bei allgemeinen `par`-Schleifen 67
6.4 Alternative Plazierungen für `get` und `put` 69
6.5 Vorausberechnung des Kommunikationspartners 72
6.5.1 Vorausberechnung bei *Modularabbildungen* 73
6.5.2 Programmtransformationen 85
6.6 Kommunikation in *LooPo* 88

7 Zusammenfassung **93**

1 Einführung

Massiv parallele Rechnerarchitekturen bieten heute große Rechenleistung für rechenintensive Aufgaben. Insbesondere Architekturen mit verteiltem Speicher haben dabei gegenüber Architekturen mit gemeinsamem Speicher Vorteile, sowohl was die Kosten angeht als auch was die Skalierbarkeit betrifft. Allerdings ist die Programmierung paralleler Systeme eine komplexe und fehleranfällige Tätigkeit. Insbesondere gilt dies wiederum für Maschinen mit verteiltem Speicher, wo der Benutzer durch das Einfügen expliziter Kommunikationsroutinen in den Code garantieren muß, daß alle für eine Berechnung nötigen Daten auf dem berechnenden Prozessor vorhanden sind.

Um allgemeine sequentiell entwickelte Algorithmen auch auf Parallelrechnern ausführbar zu machen, wurden Methoden entwickelt, die gleichzeitige Ausführbarkeit verschiedener Iterationen einer Schleife zur parallelen Ausführung zu nutzen.

Die Sprache `Kali` [MR91] verwendet vom Programmierer explizit als parallel ausführbar markierte Schleifen, in denen alle Iterationen voneinander unabhängig sein müssen, um SPMD-Code[1] für Parallelrechner zu erzeugen. Weitere Projekte, wie das `SUPERB`-Projekt an der Universität Bonn [ZBG88], der `Fortran D`-Compiler der Rice-University Houston [CFH$^+$92] und das `PARADIGM`-Projekt der University of Illinois in Urbana Champaign [SPB93], die auf der automatischen Erkennung parallel ausführbarer Programmschleifen basieren, führten zur Entwicklung des `HPF`-Standards[2] [Hig97].

Prinzipiell ermitteln all diese Ansätze lediglich, ob es möglich ist, die Iterationen einer Schleife gleichzeitig durchzuführen. Es gibt in vielen Anwendungen jedoch auch Schleifensätze, in denen dies für keine Schleife von sich aus gilt. Dennoch können auch solche Schleifensätze „versteckte“ Parallelität enthalten. Durch Restrukturierung dieser Schleifensätze kann diese Parallelität explizit gemacht werden.

Das von Lamport [Lam74] begründete Polyedermodell bietet eine Grundlage für die Restrukturierung des Programmcodes, so daß auch implizite Parallelität ausgenutzt werden kann. Es führte zu einer Reihe von Algorithmen, die im theoretischen Modell eines n-dimensionalen Raumes das von den Schleifen gebildete Polyeder transformieren, und so einen hohen (u.U. maximalen) Grad an impliziter Parallelität ausnutzen können. Solche Verfahren werden beispielsweise in [Len93], [Ban93], [Fea92a], [Fea92b] und [DV96] beschrieben (weitere Beschreibungen dieser Verfahren – insbesondere im Hinblick auf die Implementierung – finden sich in [Wie95] und [Mei96]). Diese Verfahren werden im *LooPo*-Projekt des Lehrstuhls für Programmierung der Universität Passau implementiert. Das Ergebnis dieser Algorithmen besteht (nach dem in [Wet95] beschriebenen Scanning der resultierenden Polytope) aus einem Programmstück, das parallel ausführbare Programmteile in Form von parallelen Schleifen ausdrückt, wobei Datenzugriffe immer noch auf gemeinsamem Speicher basieren. Ziel der vorliegenden Arbeit ist es, aus solchen Programmen, insbesondere der Ausgabe von *LooPo*, SPMD-Programme mit expliziter Kommunikation zu generieren, die auf Parallelrechnern mit verteiltem Speicher ausführbar sind.

[1] SPMD steht für Single Processor Multiple Data.
[2] `HPF` steht für `High Performance Fortran`.

2 Grundlagen

2.1 Notation und Definitionen

Im folgenden sollen die grundlegenden in dieser Arbeit verwendeten Begriffe definiert werden.

Wie im Polyedermodell üblich, definieren wir den Indexraum einer Anweisung:

Definition 2.1 *S sei eine Anweisung, die von m Schleifen umgeben ist, von denen jeweils ein Index $i_1, \ldots, i_m$ gebunden wird. Die Untergrenze der j-ten Schleife sei l_j, die Obergrenze u_j (dabei müssen l_j und u_j nicht zur Compilezeit bekannt sein – insbesondere ist u_j bei einer* `while`*-Schleife auch zur Laufzeit zunächst nicht bekannt). Der* **Indexraum** *I_S der Anweisung S ist dann*

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

Wird S von keiner Schleife umgeben, so ist $I_S = \{0\}$. Wir sagen auch, der Indexraum I_S werde von den S umgebenden Schleifen **aufgespannt***.*

Wir beschäftigen uns mit Programmstücken, sogenannten *Projekten*, von denen jedes für sich einen eigenen Indexraum definiert:

Definition 2.2 *Ein* **Projekt** *besteht aus einem nicht notwendigerweise perfekt verschachtelten Schleifensatz, dessen Rumpf nur Schleifen,* `if`*-Anweisungen, Zuweisungen oder Prozeduraufrufe (ohne Rückgabewert) enthält.*
Die in einem Projekt *definierten Variablen werden wie folgt unterteilt:*

1. **Indizes** *sind die Variablen, die in den Schleifen durchgezählt werden und so einen Indexraum aufspannen.*

2. **Blobs** *sind Strukturparameter: dies sind Skalare, die innerhalb des* Projekts *nur gelesen, aber nicht modifiziert werden.*

3. **Arrays** *können in dem* Projekt *gelesen und geschrieben werden.*

Schleifen haben grundsätzlich eine konstante Zahl als Schrittweite (also keine Variable irgendeiner Art) und unterteilen sich in folgende Typen:

1. `par`**-Schleifen** *sind Schleifen, deren Iterationen von jeweils unterschiedlichen Prozessoren parallel ausgeführt werden; die Abbruchbedingung wird vor dem Eintritt in die Schleife einmal berechnet.*

2. `for`**-Schleifen** *sind Schleifen, deren Iterationen in aufsteigender Reihenfolge zeitlich versetzt ausgeführt werden; die Abbruchbedingung wird vor dem Eintritt in die Schleife einmal berechnet.*

3. `while`**-Schleifen** *sind Schleifen mit einer Ausführungsbedingung statt fest vorgegebener Grenzen; die Bedingung einer* `while`*-Schleife ist unabhängig von Indexwerten evtl. umliegender* `par`*-Schleifen; die Abbruchbedingung wird bei jeder Iteration der Schleife neu berechnet.*

Prozeduren und Funktionen, die in einem Projekt *aufgerufen werden, dürfen nicht auf (globale) Arrays zugreifen (weder lesend noch schreibend). Ausnahmen sind lediglich ihnen übergebene Array-Elemente; dabei dürfen Funktionen auch auf diese Elemente nicht schreibend zugreifen.*

Die Bedingung, die wir an `while`-Schleifen stellen, scheint zunächst unmotiviert. Wir werden in Abschnitt 6.5.2 jedoch die hieraus folgende Tatsache verwenden, daß jede `while`-Schleife auf jedem Prozessor die gleiche Anzahl an Iterationen durchläuft. Die durch *LooPo* generierten *Projekte* erfüllen alle diese Bedingung (siehe [Gei97]).

Definition 2.3 *Der* **Datenraum** *eines Arrays A mit Untergrenzen* $(l_1, \ldots, l_n)$ *und Obergrenzen* $(u_1, \ldots, u_n)$ *ist* $Idx(A) = (\prod i : i \in \{1, \ldots, n\} : \{l_i, \ldots, u_i\})$. *Blobs, Indizes und Skalare haben den Datenraum* $\{0\}$.

Definition 2.4 *Seien S und T zwei Anweisungen und I der Indexraum, der von den Schleifen aufgespannt wird, die sowohl S als auch T umgeben. In S und T seien die – mindestens einen schreibenden Zugriff umfassenden – Arrayzugriffe*

$$A[\varphi_S] \text{ bzw.}$$
$$A[\varphi_T]$$

enthalten, wobei φ_S *und* φ_T *beliebige Funktionen von I in den Datenraum von A seien. Dann heißt T* **abhängig** *von S, i.Z.* $S\delta T$, *falls*

$$(\exists i, j : i, j \in I : \varphi_S(i) = \varphi_T(j) \wedge (i \prec j \vee (i = j \wedge S \sqsubseteq T)))$$

wobei $\prec$ *die lexikographische Striktordnung darstellt und* $\sqsubseteq$ *die durch die Reihenfolge im Programmtext gegebene Ordnung auf den Anweisungen.*

Bemerkung: Gibt es in obiger Definition keine Schleife, die sowohl S als auch T umgibt, so ist T abhängig von S, falls

$$\varphi_S = \varphi_T \wedge S \sqsubseteq T$$

In der Literatur gibt es verschiedene Interpretationen des *Tragens* einer Abhängigkeit. Zur Unterscheidung verwenden wir den Begriff *enthalten*. Für *tragen* verwenden wir die Definition aus [Ban93]:

Definition 2.5 *Eine Abhängigkeit ist in einer Schleife* **enthalten**, *falls der Distanzvektor in der dieser Schleife entsprechenden Komponente nicht* 0 *ist. Eine Abhängigkeit wird von einer Schleife* **getragen**, *falls sie die äußerste Schleife ist, die die Abhängigkeit* enthält. *Die äußerste sequentielle Schleife, die die Abhängigkeit* enthält, **trägt die sequentielle Komponente** *dieser Abhängigkeit.*

Parallele und sequentielle Schleifen Wir gehen in dieser Arbeit davon aus, daß Parallelität durch explizite Parallelschleifen erreicht wird. Die grundsätzliche Semantik einer Parallelschleife ist dabei, daß ihr Rumpf für jeden Wert des durch die Schleife gebundenen Indexes von einem anderen Prozessor auszuführen ist. Die Reihenfolge der Anweisungen im *Projekt* bestimmt immer auch die Reihenfolge der Abarbeitung. Das heißt auch, daß alle Anweisungen, die sich vor einer Parallelschleife befinden, abgearbeitet sein müssen, bevor irgendein Prozessor die Schleife ausführt. Genauso muß jeder Prozessor den Rumpf einer Parallelschleife beendet haben, bevor die folgenden Anweisungen abgearbeitet werden. Parallelschleifen haben also eine `fork-join`-Semantik, d.h. es wird implizit vor und nach jeder Parallelschleife synchronisiert. Dabei bezeichnen wir Parallelschleifen, die keine Abhängigkeit *tragen*, als `doall`-Schleifen. Parallelschleifen, die eine Abhängigkeit *tragen*, werden als `doacross`-Schleifen bezeichnet. Ebenso notieren wir solche Schleifen auch im Programmtext; eine Parallelschleife, über deren Eigenschaft als `doall` oder `doacross`-Schleife keine Aussage gemacht werden soll, wird einfach mit `par` notiert. Eine genauere Definition der Semantik einer Parallelschleife folgt in Abschnitt 3.2. So wie Parallelität durch parallele Schleifen angegeben wird, wird Sequentialität durch sequentielle Schleifen angegeben: abgesehen von der Reihenfolge der Anweisungen im *Projekt* wird die Abarbeitungsreihenfolge von Anweisungen nur noch davon bestimmt, daß alle Instanzen von Anweisungen, zwischen denen Abhängigkeiten existieren, die in einer sequentiellen Schleife *enthalten* sind, in der Reihenfolge abzuarbeiten sind, die durch die lexikographische Ordnung auf der Projektion der Indexvektoren auf die durch sequentielle Schleifen aufgespannten Dimensionen gegeben wird. Für eine einzelne Anweisung in einer Schleife wird die Ausführungsreihenfolge also einzig und allein von den umgebenden sequentiellen Schleifen bestimmt, keinesfalls von irgendwelchen umgebenden Parallelschleifen. Eine sequentielle Schleife wird mit `for` notiert. Eine Schleife, über deren Eigenschaft als Parallelschleife oder sequentielle Schleife wir keine Aussage machen wollen, wird mit `do` notiert.

Mathematische Notationen Eine Funktion f mit Quellraum Q und Zielraum Z wird notiert mit:

$$f : Q \to Z$$

Bildet f dabei ein $x \in Q$ nach $f(x)$ ab, so notieren wir diese Zuordnung durch

$$f : x \to f(x)$$

(wenn die Zuordnung allein ohne Funktionsnamen verwendet wird, entfällt dabei „f :“) oder auch gemeinsam mit der Typisierung als

$$f : Q \to Z : x \to f(x)$$

Für beliebige m, n und $Q \subseteq \mathbb{Q}^n, Z \subseteq \mathbb{Q}^m$ definieren wir die Nullfunktion

$$0 : Q \to Z : x \to 0$$

Die Definition der unteren Gaußklammer ($\lfloor \cdot \rfloor$) dehnen wir durch komponentenweise Anwendung auf Vektoren aus $\mathbb{R}^n$ aus (wir werden hier jedoch nur Vektoren aus $\mathbb{Q}^n$ verwenden).

Definition 2.6 *Wir definieren eine Division auf ganzzahligen Vektoren*

$$\mathrm{DIV} : \mathbb{Z}^n \times (\mathbb{Z} \setminus \{0\})^n \to \mathbb{Z}^n : ((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \to (c_1, \ldots, c_n)$$
$$\textit{mit } c_i = \mathrm{sgn}(b_i) \cdot \left\lfloor \frac{a_i}{|b_i|} \right\rfloor$$

und benutzen die Infixnotation a/b für $\mathrm{DIV}(a, b)$.
Außerdem wird der Modulo-Operator auf ganzzahligen Vektoren definiert durch:

$$\mathrm{MOD} : \mathbb{Z}^n \times (\mathbb{Z} \setminus \{0\})^n \to \mathbb{N}^n : ((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \to (c_1, \ldots, c_n)$$
$$\textit{mit } c_i = a_i - (a_i/b_i) \cdot b_i$$

Auch hier erlauben wir Infixnotation durch $a\%b$ statt $\mathrm{MOD}(a, b)$.

Man beachte, daß $(\forall (a, b) : (a, b) \in \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) : 0 \leq a\%b < |b|)$. Außerdem gilt für $b < 0$, $b \not| a$:

$$a/b \neq \left\lfloor \frac{a}{b} \right\rfloor.$$

Wäre $a/b = \lfloor \frac{a}{b} \rfloor$ (auch für negative b), würde lediglich

$$(\forall (a,b) : (a,b) \in \mathbb{Z}\times(\mathbb{Z} \setminus \{0\}) : -|b| < a\%b < |b|)$$

gelten.

In Erweiterung der Definition der modularen Abbildung von Lee und Fortes in [LF94] definieren wir eine *Modularabbildung* wie folgt:

Definition 2.7 *Eine* **Modularabbildung** *ist eine Funktion*

$$M : Q\subseteq\mathbb{Z}^n \to Z\subseteq\mathbb{Z}^m : x \to \lfloor A(x) \rfloor \% b,$$

wobei $A : Q \to \mathbb{Q}^m$ eine affine Abbildung und der **Modulo-Vektor** *b aus $\mathbb{N}^m$ ist.*

Eine Schranke für den extremalen Wert eines Ausdrucks definieren wir durch eine *extremize*-Funktion, die im wesentlichen alle Indizes auf syntaktischer Ebene durch ihre Extremalwerte ersetzt:

Definition 2.8 *Für einen arithmetischen Ausdruck A, repräsentiert in Präfixnotation, in den gebundenen Schleifenindizes $(i_1, \ldots, i_n)$ mit den unteren Grenzen $(l_1, \ldots, l_n)$, oberen Grenzen $(u_1, \ldots, u_n)$ und Schrittweiten $(s_1, \ldots, s_n)$ sei die extremize-Funktion mit $\omega \in \Omega = \{\min, \max\}$ und $\overline{\omega} \in \Omega \setminus \{\omega\}$ definiert als:*

$$\begin{aligned}
\mathit{extremize}(A,\omega) \quad &\equiv \quad XT_0(A,\omega), \ \textit{wobei} \\
XT_0(A,\omega) \quad &\equiv \quad \begin{cases}
A & \textit{falls } A \textit{ frei von } \{i_1, \ldots, i_n\},\ A = \emptyset, \\
 & A \textit{ Funktionsaufruf (außer } +,-,*,\div,\min, \\
 & \max, \lfloor\cdot\rfloor, \lceil\cdot\rceil) \textit{ oder Arrayzugriff} \\
u_j & \textit{falls } A = i_j,\ \begin{cases} (s_j > 0 \wedge \omega = \max) & \vee \\ (s_j < 0 \wedge \omega = \min) \end{cases} \\
l_j & \textit{falls } A = i_j,\ \begin{cases} (s_j < 0 \wedge \omega = \max) & \vee \\ (s_j > 0 \wedge \omega = \min) \end{cases} \\
XT_1(A,\omega) & \textit{falls } A = \psi(A_1, \ldots, A_k)
\end{cases} \\
\textit{und} \\
XT_1(A,\omega) \quad &\equiv \quad \begin{cases}
\psi(XT_0(A_1,\omega), \ldots, XT_0(A_k,\omega)) & \textit{falls } \psi \in \{\lfloor\cdot\rfloor, \lceil\cdot\rceil \\
 & \quad \min,\max,+,*\} \\
\psi(XT_0(A_1,\omega), XT_0(A_2,\overline{\omega})) & \textit{falls } \psi \in \{-,\div\} \\
 & \quad \wedge k = 2 \\
\psi(\overline{\omega}) & \textit{falls } \psi \equiv - \wedge k = 1
\end{cases}
\end{aligned}$$

Bemerkung: Für jeden Indexvektor i und jeden Ausdruck $A(i)$, der aus Maxima, Minima und Affinkombinationen von gebundenen Indizes besteht, gilt

$$\mathit{extremize}(A, \min) \leq A(i) \leq \mathit{extremize}(A, \max)$$

Dennoch gilt nicht unbedingt für einen Schleifenindex k und einen davon abhängigen Ausdruck $A(k)$, daß $\mathit{extremize}(A(k), \min) = \inf(A(k))$ oder $\mathit{extremize}(A(k), \max) = \sup(A(k))$, denn z.B. für den Schleifensatz

`for` $t = 0$ `to` 10
$\quad \ldots t + \max(A[42], 0) - t \ldots$

gilt $\sup_{t\in\{0,\ldots,10\}}(t+\max(A[42],0)-t)=\max(A[42],0)$, jedoch

$extremize(t+\max(A[42],0)-t,\max)\equiv 10+\max(A[42],0)-0=10+\max(A[42],0)$

Diese Ungleichheit liegt an der einfachen syntaktischen Transformation. Für eine algebraisch exakte Maximierung (oder Minimierung) wäre (je nach Komplexität erlaubter Ausdrücke und Schleifengrenzen) ein komplizierteres Termersetzungssystem nötig.

Wir verwenden außerdem folgende Definitionen bzw. Notationen:

1. E bezeichnet die Einheitsmatrix (oder ggf. einen Ausschnitt daraus).
2. Die Projektion eines Vektors $v=(v_1,\ldots,v_n)$ auf seine i-te Komponente wird mit $\pi_i(v)=v_i$ bezeichnet. Wir erweitern diese Definition um Matrizen, indem $\pi_i(M)$ die i-te Zeile einer Matrix M ergibt.
3. Für $a,b\in\mathbb{Z}$ ist
$$\{a,\ldots,b\}=\{i\in\mathbb{Z}\mid a\leq i\leq b\}$$
Damit gilt für $b>a$:
$$\{b,\ldots,a\}=\emptyset$$
4. $\mathrm{Ke}\,(F)$ bezeichnet den Kern einer (affinen) Abbildung F.
5. $\mathrm{Bi}\,(F)$ bezeichnet das Bild einer (affinen) Abbildung F.
6. $\mathrm{Span}(U)$ bezeichnet eine Basis für den Unterraum U.
7. $\mathrm{GL}(n;K)$ bezeichnet die Gruppe der invertierbaren $n\times n$-Matrizen über dem Ring K.
8. Die Kardinalität einer endlichen Menge M wird mit $\#\,(M)$ notiert. Für unendliche Mengen gilt $\#\,(M)=\infty$.
9. Für jede Funktion $f:Q_1\times\ldots\times Q_n\to Z$ definieren wir für $L_i\subseteq Q_i$:
$$f(\ldots,L_i,\ldots)=\{f(\ldots,l_i,\ldots)\mid l_i\in L_i\}$$
Dies gilt insbesondere für binäre Verknüpfungen (wie Addition und Multiplikation):
Ist $M\subseteq K\ni x$ und $\odot$ eine binäre Verknüpfung, definiert als $\odot:K\times K\to K$, dann ist das Resultat der binären Verknüpfung $M\odot x$ bzw. $x\odot M$, durch elementweise Verknüpfung definiert:
$$M\odot x=\{m\odot x\mid m\in M\}$$
$$x\odot M=\{x\odot m\mid m\in M\}$$
10. Das Urbild eines Punktes $z\in Z$ unter einer nichtinvertierbaren Funktion $\varphi:Q\to Z$ ist die Menge
$$\varphi^{-1}(z)=\varphi^{-1}(\{z\})=\{q\in Q\mid\varphi(q)=z\}$$
11. Der größte gemeinsame Teiler ggT und das kleinste gemeinsame Vielfache kgV werden durch Anwendung des Absolutbetrages auf die Argumente auf die ganzen Zahlen erweitert.
12. Für einen Term T bezeichnet
$$T[x/y]$$
den Term, der aus T durch Ersetzung von x durch y hervorgeht. Für $x=(x_1,\ldots,x_m)$ und $y=(y_1,\ldots,y_m)$ definieren wir abkürzend
$$T[x/y]\equiv T[x_1/y_1]\ldots[x_m/y_m]$$

2.2 Mathematische Grundlagen

Im Laufe dieser Arbeit werden wir mehrfach auf *Modularabbildungen* oder Spezialfälle solcher stoßen. Insbesondere wird dabei das Urbild eines Punktes unter dieser Abbildung von Interesse sein. Obwohl sich *Modularabbildungen* nicht im allgemeinen invertieren lassen, kann man dennoch das exakte Urbild eines Punktes unter einer *Modularabbildung* angeben. Zunächst befassen wir uns jedoch mit einfacheren Spezialfällen.

2.2.1 Rechenregeln zur Gaußklammerfunktion

Ein Grund, weshalb eine *Modularabbildung* nicht invertierbar sein kann, ist die Existenz der Gaußklammerfunktion $\lfloor\cdot\rfloor$ in ihrer Definition. Da wir jedoch nur mit ganzen und rationalen Zahlen arbeiten, können wir die folgenden Rechenregeln für die Gaußklammerfunktion anwenden:

Lemma 2.9 *Seien $a, b \in \mathbb{Z}$, dann gilt*

$$\left\lceil \tfrac{a}{b} \right\rceil = \left\lfloor \tfrac{a-\operatorname{sgn}(b)}{b} \right\rfloor + 1 \tag{1}$$

und

$$\left\lfloor \tfrac{a}{b} \right\rfloor = \left\lceil \tfrac{a+\operatorname{sgn}(b)}{b} \right\rceil - 1 \tag{2}$$

Beweis:

Wir zeigen nur Gleichung (1): Gleichung (2) folgt entsprechend (bzw. direkt durch Anwendung von (1)). Im Beweis sei $m \in \mathbb{Z}$. Sei zunächst $b > 0$. Dann gilt:

$$\begin{array}{rcl}
\left\lceil \frac{a}{b} \right\rceil = m & & \\
 & \Leftrightarrow & \\
m-1 & < \frac{a}{b} \leq & m \\
 & \Leftrightarrow & \\
(m-1)\cdot b & < a \leq & m \cdot b \\
 & \Leftrightarrow & \\
(m-1)\cdot b + 1 & \leq a < & m \cdot b + 1 \\
 & \Leftrightarrow & \\
(m-1)\cdot b & \leq a-1 < & m \cdot b \\
 & \Leftrightarrow & \\
m-1 & \leq \frac{a-1}{b} < & m \\
 & \Leftrightarrow & \\
\left\lfloor \frac{a-1}{b} \right\rfloor + 1 = m & &
\end{array}$$

also $\left\lceil \frac{a}{b} \right\rceil = m \Leftrightarrow \left\lfloor \frac{a-1}{b} \right\rfloor + 1 = m$ und so $\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a-1}{b} \right\rfloor + 1$.

Entsprechend gilt für $b < 0$:

$$\begin{array}{rcl}
\lceil \frac{a}{b} \rceil = m & & \\
& \Leftrightarrow & \\
m-1 & < \frac{a}{b} \leq & m \\
& \Leftrightarrow & \\
(m-1) \cdot b & > a \geq & m \cdot b \\
& \Leftrightarrow & \\
(m-1) \cdot b - 1 & \geq a > & m \cdot b - 1 \\
& \Leftrightarrow & \\
(m-1) \cdot b & \geq a+1 > & m \cdot b \\
& \Leftrightarrow & \\
m-1 & \leq \frac{a+1}{b} < & m \\
& \Leftrightarrow & \\
\lfloor \frac{a+1}{b} \rfloor + 1 = m & &
\end{array}$$

Damit gilt insgesamt $\lceil \frac{a}{b} \rceil = \left\lfloor \frac{a-\mathrm{sgn}(b)}{b} \right\rfloor + 1$. ✓

Lemma 2.10 *Seien $a, c \in \mathbb{Z}^m, b \in \mathbb{Z}$, dann gilt:*

$$\lfloor \tfrac{a}{b} \rfloor \% c = \left\lfloor \frac{a\%(b \cdot c)}{b} \right\rfloor \% c \tag{3}$$

Beweis:

Da MOD-Operator, Gaußklammer und Skalarmultiplikation komponentenweise definiert sind, ist die i-te Komponente der linken Seite $l = (l_1, \ldots, l_m)$:

$$l_i = \left\lfloor \frac{\pi_i(a)}{\pi_i(b)} \right\rfloor \% \pi_i(c)$$

Damit gilt für alle $z_i \in \mathbb{Z}$:

$$\begin{aligned}
l_i &= \left\lfloor \frac{\pi_i(a)}{\pi_i(b)} \right\rfloor \% \pi_i(c) \\
&= (\left\lfloor \frac{\pi_i(a)}{\pi_i(b)} \right\rfloor + z_i \cdot \pi_i(c)) \% \pi_i(c) \\
&= \left\lfloor \frac{\pi_i(a)}{\pi_i(b)} + z_i \cdot \pi_i(c) \right\rfloor \% \pi_i(c) \\
&= \left\lfloor \frac{\pi_i(a) + z_i \cdot (\pi_i(b) \cdot \pi_i(c))}{\pi_i(b)} \right\rfloor \% \pi_i(c)
\end{aligned}$$

$$\overset{l_i \text{ unabhängig von } z_i}{\Longrightarrow}$$

$$\left\lfloor \frac{\pi_i(a)}{\pi_i(b)} \right\rfloor \% \pi_i(c) = \left\lfloor \frac{\pi_i(a) \% (\pi_i(b) \cdot \pi_i(c))}{\pi_i(b)} \right\rfloor \% \pi_i(c)$$

✓

Lemma 2.11 *Seien $a, b, c \in \mathbb{Z}$. Dann gilt:*

$$(a+b)\%c = (a + b\%c)\%c \tag{4}$$
$$(a \cdot b)\%c = (a \cdot (b\%c))\%c \tag{5}$$
$$a\%c = (a\%(b \cdot c))\%c \tag{6}$$

Beweis:

1. (4) gilt, weil nach Definition des MOD-Operators ein $z \in \mathbb{Z}$ existiert, so daß:

$$\begin{aligned}(a + b\%c)\%c &= (a + b + z \cdot c)\%c \\ &= (a + b)\%c\end{aligned}$$

2. Zu (5): Es ist

$$\begin{aligned}(a \cdot b)\%c &= (\operatorname{sgn}(a)(\sum n : n \in \{1, \dots, |a|\} : b))\%c \\ &= (\operatorname{sgn}(a)(\sum n : n \in \{1, \dots, |a|\} : b\%c))\%c \\ &= (a \cdot (b\%c))\%c\end{aligned}$$

3. Zu (6): Es ist

$$\begin{aligned}(a\%(b \cdot c))\%c &= (a + (-n \cdot c))\%c \text{ für ein } n \in \mathbb{Z} \\ &\overset{(4)}{=} (a + ((-n \cdot c)\%c))\%c \\ &= a\%c\end{aligned}$$

✓

Lemma 2.12 *Seien $a = (a_1, \dots, a_n), c = (c_1, \dots, c_n) \in \mathbb{Z}^n$. Dann ist mit $b = \operatorname{kgV}(c_1, \dots, c_n)$:*

$$a\%c = (a\%(b, \dots, b))\%c$$

Beweis:

Für jede Komponente i gilt:

$$\begin{aligned}\pi_i((a\%(b, \dots, b))\%c) &= (a_i\%(m_i \cdot c_i))\%c_i \text{ mit } \frac{b}{c_i} = m_i \in \mathbb{Z} \\ &\overset{\text{Lemma 2.11 (6)}}{=} a_i\%c_i\end{aligned}$$

✓

Damit können wir nun den folgenden Satz zeigen:

Satz 2.13 *Sei $P = (p_{i,j}) \in \mathbb{Z}^{m \times m}, c = (c_1, \dots, c_m), x = (x_1, \dots, x_m) \in \mathbb{Z}^m, d \in \mathbb{Z}$. Dann gilt mit $g = \operatorname{kgV}(c_1, \dots, c_m)$:*

$$\left\lfloor \tfrac{P \cdot x}{d} \right\rfloor \%c = \left\lfloor \tfrac{P \cdot (x\%(g \cdot d, \dots, g \cdot d))}{d} \right\rfloor \%c$$

Beweis:

Nach Lemma 2.12 ist

$$\left\lfloor \tfrac{P \cdot x}{d} \right\rfloor \%c = (\left\lfloor \tfrac{P \cdot x}{d} \right\rfloor \%(g, \dots, g))\%c$$

Nach Lemma 2.10 gilt:

$$\left\lfloor \tfrac{P \cdot x}{d} \right\rfloor \%(g, \dots, g) = \left\lfloor \tfrac{(P \cdot x)\%(g \cdot d, \dots, g \cdot d)}{d} \right\rfloor \%(g, \dots, g)$$

Die i-te Komponente von $(P \cdot x)\%(g \cdot d, \dots, g \cdot d)$ ist aber:

$$\begin{aligned}&\pi_i((P \cdot x)\%(g \cdot d, \dots, g \cdot d)) = ((\sum j : j \in \{1, \dots, n\} : p_{i,j} \cdot x_j))\%(g \cdot d) \\ &\overset{\text{Lemma 2.11 (4)}}{=} ((\sum j : j \in \{1, \dots, n\} : (p_{i,j} \cdot x_j)\%(g \cdot d)))\%(g \cdot d) \\ &\overset{\text{Lemma 2.11 (5)}}{=} ((\sum j : j \in \{1, \dots, n\} : (p_{i,j} \cdot (x_j\%(g \cdot d)))\%(g \cdot d)))\%(g \cdot d) \\ &\overset{\text{Lemma 2.11 (4)}}{=} ((\sum j : j \in \{1, \dots, n\} : (p_{i,j} \cdot (x_j\%(g \cdot d)))))\%(g \cdot d) \\ &= \pi_i((P \cdot (x\%(g \cdot d, \dots, g \cdot d)))\%(g \cdot d, \dots, g \cdot d))\end{aligned}$$

Damit gilt:

$$\left\lfloor \tfrac{P\cdot x}{d} \right\rfloor \%(g,\ldots,g) = \left\lfloor \tfrac{(P\cdot(x\%(g\cdot d,\ldots,g\cdot d)))\%(g\cdot d,\ldots,g\cdot d)}{d} \right\rfloor \%(g,\ldots,g)$$
$$\overset{\text{Lemma 2.10}}{=} \left\lfloor \tfrac{P\cdot(x\%(g\cdot d,\ldots,g\cdot d))}{d} \right\rfloor \%(g,\ldots,g)$$

Wegen Lemma 2.12 können wir auf den letzten MOD-Operator wieder verzichten:

$$\left\lfloor \tfrac{P\cdot x}{d} \right\rfloor \%c = (\left\lfloor \tfrac{P\cdot(x\%(g\cdot d,\ldots,g\cdot d))}{d} \right\rfloor \%(g,\ldots,g))\%c$$
$$= \left\lfloor \tfrac{P\cdot(x\%(g\cdot d,\ldots,g\cdot d))}{d} \right\rfloor \%c$$

✓

Satz 2.14 *Gegeben sei eine Funktion*

$$F : \mathbb{Z} \to \mathbb{Z} : j \to \left\lfloor \tfrac{c\cdot j+k}{d} \right\rfloor$$

mit $\mathbb{Z} \ni c, d \neq 0$*. Dann ist das Urbild von* $m \in \mathbb{Z}$ *unter* F*:*

$$F^{-1}(m) = \begin{cases} \{\left\lfloor \frac{d\cdot m-k-\operatorname{sgn}(c)}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{(m+1)\cdot d-k-\operatorname{sgn}(c)}{c} \right\rfloor\} & \textit{falls } \operatorname{sgn}(c) = \operatorname{sgn}(d) \\ \{\left\lfloor \frac{(m+1)\cdot d-k}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{m\cdot d-k}{c} \right\rfloor\} & \textit{falls } \operatorname{sgn}(c) \neq \operatorname{sgn}(d) \end{cases}$$

Beweis:

Es gilt

$$\begin{array}{ccc} \left\lfloor \frac{c\cdot j+k}{d} \right\rfloor & = & m \\ & \Leftrightarrow & \\ m & \leq \frac{c\dot{j}+k}{d} < & m+1 \end{array} \tag{7}$$

Wir unterscheiden die Fälle

1. $c, d > 0$:

   Gleichung (7) gilt gdw.

$$\begin{array}{ccc} \dfrac{m\cdot d-k}{c} & \leq j < & \dfrac{(m+1)\cdot d-k}{c} \\ & \Leftrightarrow & \\ \left\lceil \frac{m\cdot d-k}{c} \right\rceil & \leq j < & \left\lceil \frac{(m+1)\cdot d-k}{c} \right\rceil = \left\lfloor \frac{(m+1)\cdot d-k-1}{c} \right\rfloor + 1 \\ & \Leftrightarrow & \\ \left\lfloor \frac{m\cdot d-k-1}{c} \right\rfloor + 1 & \leq j \leq & \left\lfloor \frac{(m+1)\cdot d-k-1}{c} \right\rfloor \end{array}$$

   Also $j \in F^{-1}(m) \Leftrightarrow j \in \{\left\lfloor \frac{m\cdot d-k-1}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{(m+1)\cdot d-k-1}{c} \right\rfloor\}$.

2. $c, d < 0$:

   Gleichung (7) gilt gdw.

$$\begin{array}{ccc} m\cdot d & \geq c\cdot j+k > & (m+1)\cdot d \\ & \Leftrightarrow & \\ \dfrac{m\cdot d-k}{c} & \leq j < & \dfrac{(m+1)\cdot d-k}{c} \\ & \Leftrightarrow & \\ \left\lceil \frac{m\cdot d-k}{c} \right\rceil & \leq j < & \left\lceil \frac{(m+1)\cdot d-k}{c} \right\rceil = \left\lfloor \frac{(m+1)\cdot d-k+1}{c} \right\rfloor + 1 \\ & \Leftrightarrow & \\ \left\lfloor \frac{m\cdot d-k+1}{c} \right\rfloor + 1 & \leq j \leq & \left\lfloor \frac{(m+1)\cdot d-k+1}{c} \right\rfloor \end{array}$$

   Also $j \in F^{-1}(m) \Leftrightarrow j \in \{\left\lfloor \frac{m\cdot d-k+1}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{(m+1)\cdot d-k+1}{c} \right\rfloor\}$.

3. $c < 0, d > 0$:

Gleichung (7) gilt gdw.

$$\begin{array}{ccc} m \cdot d & \leq c \cdot j + k < & d \cdot m + 1 \\ & \Leftrightarrow & \\ \dfrac{m \cdot d - k}{c} & \geq j > & \dfrac{(m+1) \cdot d - k}{c} \\ & \Leftrightarrow & \\ \left\lfloor \frac{m \cdot d - k}{c} \right\rfloor & \geq j > & \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor \\ & \Leftrightarrow & \\ \left\lfloor \frac{m \cdot d - k}{c} \right\rfloor & \geq j \geq & \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor + 1 \end{array}$$

Also $j \in F^{-1}(m) \Leftrightarrow j \in \{\left\lfloor \frac{m \cdot d - k}{c} \right\rfloor, \ldots, \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor + 1\}$.

4. $c > 0, d < 0$:

Gleichung (7) gilt gdw.

$$\begin{array}{ccc} m \cdot d & \geq c \cdot j + k > & (m+1) \cdot d \\ & \Leftrightarrow & \\ \dfrac{m \cdot d - k}{c} & \geq j > & \dfrac{(m+1) \cdot d - k}{c} \\ & \Leftrightarrow & \\ \left\lfloor \frac{m \cdot d - k}{c} \right\rfloor & \geq j > & \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor \\ & \Leftrightarrow & \\ \left\lfloor \frac{m \cdot d - k}{c} \right\rfloor & \geq j \geq & \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor + 1 \end{array}$$

Also $j \in F^{-1}(m) \Leftrightarrow j \in \{\left\lfloor \frac{m \cdot d - k}{c} \right\rfloor, \ldots, \left\lfloor \frac{(m+1) \cdot d - k}{c} \right\rfloor + 1\}$.

Damit gilt die Behauptung. ✓

Lemma 2.15 *Gegeben sei eine Funktion*

$$F : \mathbb{Z} \rightarrow \mathbb{Z} : j \rightarrow \left\lfloor \frac{c \cdot j + k}{d} \right\rfloor = \left\lfloor \frac{c^* \cdot j}{d^*} + \frac{k}{d} \right\rfloor$$

mit $0 \neq c^* \in \mathbb{Z}, d^* \in \mathbb{N} \setminus \{0\}$ *gekürzt. Dann gilt für alle* $z \in \mathbb{Z}$

$$F(j + z \cdot d^*) = F(j) + z \cdot c^* \tag{8}$$

$$F^{-1}(F(j + z \cdot d^*)) = F^{-1}(F(j)) + z \cdot d^* \tag{9}$$

und damit

$$\#\left(F^{-1}(F(j + z \cdot d^*))\right) = \#\left(F^{-1}(F(j))\right) \tag{10}$$

Beweis:

1. (8) gilt wegen:

$$\begin{array}{rcl} F(j + z \cdot d^*) & = & \left\lfloor \frac{c^* \cdot (j + z \cdot d^*)}{d^*} + \frac{k}{d} \right\rfloor \\ & = & \left\lfloor \frac{c^* \cdot j}{d^*} + \frac{c^* \cdot z \cdot d^*}{d^*} + \frac{k}{d} \right\rfloor \\ & \overset{\frac{c^* \cdot z \cdot d^*}{d^*} \in \mathbb{Z}}{=} & F(j) + c^* \cdot z \end{array}$$

2. Sei $F(j) = m$. Dann gilt:

$$F^{-1}(F(j + z \cdot d^*)) \stackrel{(8)}{=} F^{-1}(F(j) + z \cdot c^*)$$

$$\begin{array}{cl}
\stackrel{\substack{\text{Satz 2.14,} \\ d\cdot(m+z\cdot c^*)= \\ d\cdot m+d\cdot z\cdot c^*}}{=} & \begin{cases} \{\left\lfloor \frac{d\cdot m+d\cdot z\cdot c^*-k-\operatorname{sgn}(c)}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{(m+1)\cdot d+d\cdot z\cdot c^*-k-\operatorname{sgn}(c)}{c} \right\rfloor\} & \text{falls } \frac{c}{d} > 0 \\ \{\left\lfloor \frac{(m+1)\cdot d+d\cdot z\cdot c^*-k}{c} \right\rfloor + 1, \ldots, \left\lfloor \frac{m\cdot d+d\cdot z\cdot c^*-k}{c} \right\rfloor\} & \text{sonst} \end{cases} \\
= & \begin{cases} \{\left\lfloor \frac{d\cdot m-k-\operatorname{sgn}(c)}{c} \right\rfloor + 1 + z\cdot d^*, \ldots, \left\lfloor \frac{(m+1)\cdot d+-k-\operatorname{sgn}(c)}{c} \right\rfloor + z\cdot d^*\} & \frac{c}{d} > 0 \\ \{\left\lfloor \frac{(m+1)\cdot d+-k}{c} \right\rfloor + 1 + z\cdot d^*, \ldots, \left\lfloor \frac{m\cdot d+-k}{c} \right\rfloor + z\cdot d^*\} & \text{sonst} \end{cases} \\
= & F^{-1}(F(j)) + z \cdot d^*
\end{array}$$

Damit gilt (9).

3. (10) folgt direkt aus (9).

✓

Satz 2.16 *Sei F wie in Lemma 2.15. Seien $0 < p \in \mathbb{N}$ und $l \in \mathbb{Z}$ gegeben. Dann ist*

$$s = \frac{\operatorname{kgV}(p, c^*)}{|c^*|} \cdot d^*$$

minimal in $\mathbb{N}$ mit

1. *$(\exists n : 0 < n \in \mathbb{N} : F(l + s) = F(l) + \operatorname{sgn}(c^*) \cdot n \cdot p)$ und*
2. *$d^* | s$ (d^* teilt s)*

Beweis:

1. (2) gilt nach Definition von s.
2. (1) gilt nach Lemma 2.15, Gleichung (9), mit

$$z = \frac{\operatorname{kgV}(p, c^*)}{c^*}$$

und weil $p | \operatorname{kgV}(p, c^*)$.

3. Zu zeigen ist noch die Minimalität. Sei $\mathbb{N} \ni x < s$, so daß (1) und 2 auch für x gelten. Dann gilt wegen $x | d^*$ auch

$$x \leq \frac{\operatorname{kgV}(p, c^*)}{|c^*|} \cdot d^* - d^*$$

Dann ist mit $K = \{0, \ldots, \operatorname{kgV}(p, c^*) \cdot \frac{d^*}{|c^*|} - d^*\}$ zu zeigen, daß

$$(\forall x : x \in K : (\nexists n : 0 < n \in \mathbb{N} : F(l + x) = F(l) + \operatorname{sgn}(c^*) \cdot n \cdot p))$$

Sei $x = v \cdot d^*$, $v \in \frac{K}{d^*}$. Es gilt:

$$\begin{array}{rcl}
F(l + x) & = & F(l + v \cdot d^*) \\
 & \stackrel{\text{Lemma 2.15 (8)}}{=} & F(l) + v \cdot c^* \\
 & \text{und} & \\
F(l) + \operatorname{sgn}(c^*) \cdot n \cdot p & = & F(l) + v \cdot c^* \\
 & \Leftrightarrow & \\
\operatorname{sgn}(c^*) \cdot n \cdot p & = & v \cdot c^* \\
 & \Leftrightarrow & \\
 & p | (v \cdot |c^*|) & \text{mit } v \in \{0, \ldots, \frac{\operatorname{kgV}(p, c^*)}{|c^*|} - 1\}
\end{array}$$

(a) Sei $v = 1$. Dann ist $\frac{\mathrm{kgV}(p,c^*)}{|c^*|} - 1 = \frac{|c^*|}{|c^*|} - 1 = 0$ und deshalb $F(l+x) = F(l+0) \neq F(l) + \mathrm{sgn}(c^*) \cdot n \cdot p$

(b) Sei $v \neq 1$. Dann ist

$$v = \frac{\mathrm{kgV}(c^*, p)}{c^*} \cdot z \text{ mit } z \in \mathbb{N}$$

was für $v \neq 0$ ein Widerspruch ist zu $v \in \{0, \ldots, \frac{\mathrm{kgV}(p,c^*)}{|c^*|} - 1\}$; für $v = 0$ ist $F(l+x) = F(l) \neq F(l) + \mathrm{sgn}(c^*) \cdot n \cdot p$.

✓

Satz 2.17 *Gegeben sei eine Funtion F wie in Lemma 2.15. Dann gilt*

$$\#\left(\{F(i) \mid i \in \{0, \ldots, d^* - 1\}\}\right) = \min(|c^*|, d^*)$$

Beweis:

1. Sei $|c^*| \geq d^*$. o.B.d.A. sei $c^* > 0$ (für $c^* < 0$ ändert sich unten nur die Richtung der Ungleichungen, nicht die Aussage über die Monotonie). Dann gilt für alle $i \in \mathbb{Z}$:

$$\begin{aligned} F(i) &= \left\lfloor \tfrac{c^*}{d^*} \cdot i + \tfrac{k}{d} \right\rfloor \\ &= \left\lfloor \tfrac{c^*}{d^*} \cdot (i-1) + \tfrac{k}{d} + \tfrac{c^*}{d^*} \right\rfloor \\ &\overset{\text{Def. }\%}{=} \left\lfloor \tfrac{c^*}{d^*} \cdot (i-1) + \tfrac{k}{d} + \left\lfloor \tfrac{c^*}{d^*} \right\rfloor + \tfrac{c^* \% d^*}{d^*} \right\rfloor \\ &\geq \left\lfloor \tfrac{c^*}{d^*} \cdot (i-1) + \tfrac{k}{d} \right\rfloor + \left\lfloor \tfrac{c^*}{d^*} \right\rfloor \\ &> F(i-1) \end{aligned}$$

   F ist also streng monoton in $\mathbb{Z}$. D.h. $F(i) \in \{F(i) \mid i \in \{0, \ldots, d^*-1\}\} \Leftrightarrow i \in \{0, \ldots, d^*-1\}$ und damit $\#\left(\{F(i) \mid i \in \{0, \ldots, d^*-1\}\}\right) = d^*$.

2. Sei nun $|c^*| < d^*$. Dann gilt die Implikation:

$$F(i) \neq F(i + \mathrm{sgn}(c^*)) \Rightarrow F(i) \quad = \quad F(i + \mathrm{sgn}(c)) - 1$$

   D.h. an jeder Stelle i, an der sich durch Addieren (oder Subtrahieren) von 1 (man beachte, daß F auf $\mathbb{Z}$ definiert ist) der Funktionswert von F ändert, ändert sich der Funktionswert betragsmäßig um 1. Deshalb ist die Anzahl unterschiedlicher Funktionswerte:

$$\begin{aligned} \#\left(\{F(i) \mid i \in \{0, \ldots, d^*-1\}\}\right) &= |F(d^*) - F(0)| \\ &= \left| \left\lfloor \tfrac{c^* \cdot d^*}{d^*} + \tfrac{k}{d} \right\rfloor - \left\lfloor \tfrac{k}{d} \right\rfloor \right| \\ &= \left| \left\lfloor c^* + \tfrac{k}{d} \right\rfloor - \left\lfloor \tfrac{k}{d} \right\rfloor \right| \\ &= \left| c^* + \left\lfloor \tfrac{k}{d} \right\rfloor - \left\lfloor \tfrac{k}{d} \right\rfloor \right| \\ &= |c^*| \end{aligned}$$

✓

2.2.2 Matrixinversion

Im Laufe der Arbeit werden wir allgemeine affine Funktionen betrachten. Dabei stellt sich auch die Frage nach ihrer Invertierbarkeit bzw. nach einem einzelnen Urbildpunkt zu einem gegebenen Bildpunkt. Aus der linearen Algebra ist bekannt, daß eine Matrix A (bzw. die von ihr gegebene lineare Abbildung) invertierbar ist gdw. A quadratisch ist und vollen Rang hat. Sie besitzt ein (nicht eindeutiges) Rechtsinverses A^R, wenn sie höchstens so viele Zeilen wie Spalten und maximalen Rang hat. Allgemein kann man die Inverse einer Matrix $A \in \mathbb{Q}^{m \times n}$ finden (bzw. ihre Existenz widerlegen), indem man, wie in [BS91], S.154, beschrieben, die linearen Gleichungssysteme

$$A \cdot X = E_{m,m}$$

nach X auflöst (sofern möglich), wobei für die i-te Spalte x_i von X der inhomogene Anteil des linearen Gleichungssystems $A \cdot x_i = e_i$ aus der i-ten Spalte von E besteht. Für eine möglicherweise nichtinvertierbare lineare Funktion zeigt dieser Ansatz die Bedingungen dafür auf, daß ein Punkt aus dem Bild der Funktion stammt:
Sei $A \in \mathbb{Q}^{m \times n}$ und $y \in \mathbb{Q}^m$. Dann gilt

$$\begin{array}{rcl} A \cdot x_i & = & y \\ & \Leftrightarrow & \\ Z \cdot A \cdot x_i & = & Z \cdot y \\ (\text{wobei } Z \cdot A & \text{in} & \text{Zeilenstufenform}) \end{array}$$

Nun läßt sich durch Lösen des Gleichungssystems, das aus den Zeilen von $Z \cdot A$ besteht, die nicht nur aus Nullen bestehen, jeweils ein $x_i \in \mathbb{Q}^n$ bestimmen, so daß für diese Zeilen j gilt:

$$\pi_j(Z \cdot A \cdot x_i) = \pi_j(Z \cdot y)$$

Die Gleichung

$$Z \cdot A \cdot x_i = Z \cdot y$$

gilt für ein solches x_i genau dann, wenn für alle ausschließlich aus Nullen bestehenden Zeilen k von $Z \cdot A$ gilt, daß

$$\pi_k(Z \cdot y) = 0$$

D.h. $y \in \mathrm{Bi}(A) \Leftrightarrow$

$$(\forall j : j \in \{1, \ldots, m\} : (\forall k : k \in \{1, \ldots, n\} : (Z \cdot A)_{j,k} = 0) \Rightarrow \pi_j(Z \cdot y) = 0)$$

Anders ausgedrückt heißt dies, daß für jeden Bildpunkt y von A eine Zeile j in $Z \cdot y$ gleich 0 ist, falls die Zeile j in $Z \cdot A$ gleich $(0, \ldots, 0)$ ist; d.h. für alle Bildpunkte von A existiert eine Lösung des Gleichungssystems. Wir setzen nun für y der Reihe nach die Einheitsvektoren e_i ein und lösen das Gleichunssystem $A \cdot x_i = e_i$, also $Z \cdot A \cdot x_i = Z \cdot e_i$ für die Zeilen (oder Gleichungen) j mit $\pi_j(Z \cdot A) \neq (0, \ldots, 0)$ (dieses eingeschränkte Gleichungssystem ist immer lösbar!). Die Matrix X, die sich aus Aneinanderreihung der Lösungsvektoren x_i ergibt, repräsentiert eine Abbildung, die für jeden Punkt y, an dem das Gleichungssystem $A \cdot x = y$ gelöst werden kann, eine Lösung $x = X \cdot y$ bestimmt. Denn nach Konstruktion von X kann sich für jedes $y \in \mathbb{Q}^n$ das Produkt $Z \cdot A \cdot X \cdot y$ von $Z \cdot y$ nur in den letzten $m - \mathrm{rang}(A)$ Zeilen unterscheiden; diese sind in $Z \cdot A \cdot X \cdot y$ immer 0, für $y \in \mathrm{Bi}(A)$ sind die entsprechenden Zeilen von $Z \cdot y$ aber ebenfalls 0. D.h. für $y \in \mathrm{Bi}(A)$ ist $Z \cdot A \cdot X \cdot y = Z \cdot y$, also $A \cdot X \cdot y = y$. Das bedeutet:

$$y \in \mathrm{Bi}(A) \Rightarrow A \cdot X \cdot y = y$$

Für eine auf diese Art bestimmte Matrix X gilt also auf jeden Fall:

$$A \cdot X \cdot A = A \tag{11}$$

da nach Konstruktion von X für jedes $y = A \cdot x$ auch $A \cdot X \cdot A \cdot x = A \cdot X \cdot y = y = A \cdot x$ gilt. Eine solche Matrix X, die (11) erfüllt, nennen wir wie in [Usm87] eine **verallgemeinerte Inverse** zu A.

Beispiel 2.18 *Sei* $A = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$*;* $Z \cdot A$ *ist in Zeilenstufenform für* $Z = \begin{pmatrix} 1 & 0 \\ -1 & 2 \end{pmatrix}$.
Lösen der Gleichungssysteme $Z \cdot A \cdot x_i = Z \cdot e_i$*, also*

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix} \cdot x_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{12}$$

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix} \cdot x_2 = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \tag{13}$$

ergibt als mögliche Lösung $x_1 = \frac{1}{2}$ *bzw.* $x_2 = 0$ *(*$X = \begin{pmatrix} \frac{1}{2} & 0 \end{pmatrix}$*) und als Bedingung für den Bildpunkt:*

$$y \in \mathrm{Bi}(A) \Leftrightarrow -\pi_1(y) + 2 \cdot \pi_2(y) = 0$$

also „erste Komponente eines Bildpunktes ist immer das Doppelte der zweiten", wie auch unmittelbar einsichtig.

Andererseits ist es natürlich möglich, daß es mehrere Urbildpunkte für einen Bildpunkt gibt – und so die Matrix X nicht eindeutig ist, die wir mit diesem Verfahren berechnen. Solche Mehrdeutigkeiten kommen als Wahlmöglichkeiten beim Auflösen des Gleichungssystems vor. Wir wählen eine eindeutige Matrix aus, indem wir für jeden beliebig wählbaren Eintrag die 0 wählen.
Im allgemeinen ist eine verallgemeinerte Inverse also nicht eindeutig (es gab u.U. Wahlfreiheiten für einige Matrixeinträge). Das obige Verfahren ergibt jedoch aufgrund des Einsetzens von 0 für jeden frei wählbaren Eintrag eine eindeutige Matrix, die wir mit A^r bezeichnen. Hat die Matrix A maximalen Rang, so ist A^r ein Rechtsinverses. Ist sie zusätzlich quadratisch, so ist $A^r = A^{-1}$. Wie jede Matrix definiert A^r eine lineare Abbildung. Diese Abbildung invertiert die durch A gegebene Abbildung φ an allen Stellen, wo diese invertierbar ist; d.h. sie liefert an jedem Bildpunkt von φ einen Urbildpunkt, der – falls es mehrere gibt – in möglichst vielen möglichst hochdimensionalen[3] Komponenten den Wert 0 hat.

[3]Beim Auflösen des Ungleichungssystems werden zunächst Werte für die „am weitesten rechts stehenden" Variablen eingesetzt, was bei der Vektorschreibweise den höher numerierten Dimensionen entspricht.

3 Spezifikation

3.1 Maschinenmodell

Als Parallelrechnerarchitektur gehen wir von einer Menge von Prozessoren mit privatem Speicher und ohne gemeinsamen Speicherbereich aus. Jeder Prozessor hat eine ihm eineindeutig zugeordnete Identifikationsnummer aus $\{0, \ldots, p_{max-1}\}$, wobei p_{max} die Anzahl der Prozessoren des Parallelrechners ist. Diese Identifikation wird von allen gängigen Parallelrechner-Systemen unterstützt, so auch von dem Betriebssystem `Parix` ([Par93]) und den Kommunikationsroutinen der `MPI`-Library ([Pac95], [Mes95]). Da es keinen gemeinsamen Speicher gibt, kann auf dieser Maschine Zugriff auf gemeinsam genutzte Daten nur durch das Einfügen von Kommunikationsanweisungen in den Programmcode realisiert werden. Zur Kommunikation zwischen Prozessoren setzen wir die folgenden Funktionen voraus:

`send`	synchrones (blockierendes) Senden eines Wertes zu einem beliebigen anderen Prozessor
`receive`	synchrones (blockierendes) Empfangen eines Wertes von einem beliebigen anderen Prozessor
`broadcast`	kollektive Kommunikation eines sendenden Prozessors mit allen anderen Prozessoren (muß von allen Prozessoren ausgeführt werden)
`barrier`	Synchronisation aller Prozessoren innerhalb einer vorher definierten Untergruppe (muß von allen Prozessoren dieser Gruppe ausgeführt werden)

Die tatsächliche Rechnerarchitektur darf sich von dieser eindimensionalen Identifikation unterscheiden – der physikalischen Maschine könnte beispielsweise ein n-dimensionales Gitter zugrundeliegen. Ähnlich wie mehrdimensionale Arrays auf einen eindimensionalen Speicher abgebildet werden, werden wir später die im allgemeinen mehrdimensionale Gitterstruktur, die aus verschachtelten Parallelschleifen entsteht, auf diese eindimensionale Prozessor-ID abbilden. Dabei kann eine gitterförmige Maschinenarchitektur unterstützt werden.
Außerdem nehmen wir an, daß der Parallelrechner über einen ausgezeichneten Prozessor mit einem sequentiellen Hostrechner verbunden ist, der die Versorgung mit Daten gewährleistet und die Ausführung von *Projekten* auf dem Parallelrechner steuert. Diese Aufgabe kann prizipiell auch von einem ausgezeichneten Prozessor der Parallelrechnerarchitektur übernommen werden.

3.2 Eingabespezifikation

Unser Ausgangspunkt ist eine Sequenz von *Projekten*. Dabei soll in jedem *Projekt* für jede von i Parallelschleifen umgebene Parallelschleife `par` $p = l_p$ `to` u_p gelten:

- $\mathit{extremize}(p, \min) \geq 0$
- $\mathit{extremize}(p, \max)$ ist ein konstanter arrayfreier Ausdruck
- $\mathit{extremize}(p, \max)$ ist kleiner als die Anzahl der Prozessoren, die für die $i+1$-te Prozessordimension auf dem Parallelrechner zur Verfügung stehen

Das heißt, daß die Obergrenze einer Parallelschleife im Programmtext durch einen konstanten Ausdruck gegeben werden muß oder durch einen Ausdruck aus Maxima, Minima und (möglicherweise Blobs enthaltenden) Affinkombinationen von Indizes umliegender Schleifen, deren Extremalwert wiederum dieser Bedingung genügt. Der Extremalwert ist hier für ein Auftreten des Indexes mit nichtnegativen

Koeffizienten der höchste Wert[4] des Schleifenindexes und für ein Auftreten mit negativem Vorzeichen der geringste Wert.
Ist für die Schachtelungtiefe i der Wert S_i das Maximum aller Werte, die der Index einer Parallelschleife mit i umgebenden Parallelschleifen annehmen kann, so muß außerdem $(\prod i : i \in \{0, \ldots, m-1\} : \lfloor S_i \rfloor + 1)$ kleiner oder gleich der Anzahl der Prozessoren des Parallelrechners sein.

Beispiel 3.1 *Das Programm*

$t_1 := 0$;
`while` $t_1 < 10$
  `for` $t_2 = t_1$ `to` $2 \cdot t_1$
    `par` $p_1 = 0$ `to` t_2
      `skip;`

stellt eine illegale Eingabe dar, weil p_1 nicht durch eine ganze Zahl beschränkt ist: die Bedingung der `while`-Schleife kann nicht berücksichtigt werden.

Beispiel 3.2 *Betrachten wir den folgenden Programmtext:*

`for` $t = l$ `to` u
  `par` $p_1 = l$ `to` $2 \cdot t$
    `par` $p_2 = 2$ `to` $p_1 - t$
      `skip;`
  `par` $p_3 = 3 \cdot t$ `to` $3 \cdot t$
    `skip;`

Hier muß l nichtnegativ sein, da es als Untergrenze für die p_2-Schleife verwendet wird (wir nehmen an, daß u ebenfalls nichtnegativ ist); das geschätzte Maximum der Obergrenzen für nicht in andere Parallelschleifen geschachtelte Parallelschleifen ist

$$extremize(\max(2 \cdot u, 3 \cdot u)) = \max(2 \cdot u, 3 \cdot u) = 3 \cdot u,$$

das geschätzte Maximum für einfach geschachtelte Parallelschleifen ist

$$\begin{aligned} extremize(p_1 - t, \max) &= extremize(p_1, max) - extremize(t, \min) \\ &= 2 \cdot u - l, \end{aligned}$$

also $2 \cdot u - l$. Damit müssen auf dem Parallelrechner mindestens

$$(3 \cdot u + 1) \cdot (2 \cdot u - l + 1) = 6u^2 + 5 \cdot u - 3 \cdot u \cdot l + 1$$

Prozessoren vorhanden sein. Für die erste Dimension müssen $3 \cdot u + 1$ Prozessoren zur Verfügung stehen, und für die zweite $2 \cdot u - l + 1$.

Größe des Prozessorfeldes Mit S_i als per *extremize*-Funktion geschätztes Maximum aller Indizes, die von einer Parallelschleife mit $i-1$ weiteren umgebenden Parallelschleifen gebunden wird, wird für jedes *Projekt* ein virtuelles Prozessorfeld der folgenden Gitterform definiert:

$$\{(p_1, \ldots, p_m) \mid (\forall i : i \in \{1, \ldots, m\} : 0 \leq p_i \leq \lfloor S_i \rfloor)\},$$

wobei m die maximale Anzahl ineinander verschachtelter Parallelschleifen in dem Projekt darstellt. Jedem Prozessor der Zielmaschine wird eineindeutig ein Vektor aus $\mathbb{N}^m$ zugeordnet (näheres zu dieser Zuordnung in Abschnitt 4.2). Damit werden jedem Prozessor eines *Projekts* auch m Koordinaten (die des Vektors) zugewiesen. Man beachte, daß dabei auch Parallelschleifen der Form

[4] Man beachte, daß der höchste Wert einer Schleife mit negativer Schrittweite von ihrer Untergrenze gegeben wird.

```
par p_1 = 10 to 15
  par p_2 = p_1 to 15
    skip;
```

ein Prozessorfeld der Form

$$P = \{(p_1, p_2) \mid 0 \leq p_1 \leq 15, 0 \leq p_2 \leq 15\}$$

aufspannen. Andernfalls könnten *Modularabbildungen* nicht als Abbildungen in das Prozessorfeld verwendet werden: Im allgemeinen bilden *Modularabbildungen* auch auf 0 ab. Darum **muß** das Prozessorfeld auch die 0-Koordinaten umfassen, wenn *Modularabbildungen* als Abbildungen in das Prozessorfeld verwendet werden sollen. (Wir verwenden *Modularabbildungen* als Datenallokation, siehe unten.)

Rationale Schrittweiten Die in den *Projekten* auftretenden Schleifen dürfen zunächst rationale Grenzen und Schrittweiten enthalten. Dabei zählt eine Schleife der Form

do $i = \frac{l}{q_l}$ to $\frac{u}{q_u}$ step $\frac{s}{q_s}$

die Indexpunkte

$$\{\frac{l}{q_l} + n \cdot \frac{s}{q_s} \mid n \in \{0, \ldots, \left\lfloor \tfrac{q_s}{s} \cdot (\tfrac{u}{q_u} - \tfrac{l}{q_l}) \right\rfloor\}\}$$

auf, wobei es unerheblich ist, ob es sich hier um eine sequentielle oder um eine parallele Schleife handelt. Alle Indexpunkte $\frac{a}{b}$ sind dabei prinzipiell zu demselben Raumzeitpunkt $\lfloor \frac{a}{b} \rfloor$ auszuführen. Dies entspricht der Interpretation rationaler Raum- und Zeitpunkte durch automatische Scheduling- und Allokationsmethoden wie den in [Wie95] und [Mei96] beschriebenen, die im *LooPo*-Projekt Anwendung finden. Handelt es sich bei der Schleife also um eine Parallelschleife mit $j-1$ umgebenden Parallelschleifen, so ist jeder rationale Indexpunkt $\frac{a}{b}$ auf den Prozessoren $\{p \in P \mid \pi_j(p) = \lfloor \frac{a}{b} \rfloor\}$ auszuführen.

Anweisungsallokation In einem *Projekt* können Anweisungen auftreten, die nicht von einer maximalen Anzahl von Parallelschleifen umgeben sind. Obwohl wir insgesamt von einem m-dimensionalen Prozessorfeld ausgehen, kann es also sein, daß an diesem Punkt im Code lediglich ein j-dimensionales Prozessorfeld $(0 \leq j < m)$ durch die umliegenden Parallelschleifen aufgespannt wird (Anweisungen, die von keiner Parallelschleife umgeben sind, werden nur von einem Prozessor ausgeführt). Dieses kleinere Prozessorfeld muß also in dasjenige eingebettet werden, auf dem das *Projekt* auszuführen ist. Wir bestimmen, daß dieses Prozessorfeld durch Anlegen an den Ursprung eingebettet werden soll. Damit sind Zuweisungen und Prozeduraufrufe, die von $j < m$ Parallelschleifen umgeben sind, nur von den Prozessoren p mit

$$(\forall k : k \in \{j+1, \ldots, m\} : \pi_k(p) = 0)$$

auszuführen. Diese Festlegung ist nicht rein willkürlich; denn nach der Definition des Prozessofeldes gibt es für jedes $0 \leq k < m$ und alle von umgebenden Parallelschleifen aufgezählten Koordinaten einen Prozessor, der diese Bedingung erfüllt (falls der Indexraum nicht leer ist). Da aber auch Parallelschleifen der Form par $p = 0$ to 0 in einem *Projekt* auftreten können, ist dies für andere ganze Zahlen nicht gewährleistet.

Datenallokation Durch Parallelschleifen wird also eine Anweisungsallokation gegeben. Zusätzlich definieren wir nun auch eine Datenallokation wie in [ZC93]:

Definition 3.3 *Sei A eine Variable mit Datenraum I und P ein Prozessorfeld. Dann ist eine Datenallokation für Variable A gegeben durch eine totale Abbildung*

$$\Phi_A : I \to \mathcal{P}(P)$$

wobei $\mathcal{P}(P)$ die Potenzmenge von P bezeichnet.

Ein Prozessor x mit $\Phi_A(i) = x$ heißt **Owner** des Datums $A[i]$. Dabei sagen wir, A werde **repliziert** gespeichert, falls $(\forall i : i \in I : \#(\Phi_A(I)) > 1)$ [ZC93]. Gilt $(\forall i : i \in I : \#(\Phi_A(I)) = 1)$, so nennen wir auch die Abbildung $I \to P : i \to p$ mit $p \in \Phi_A(i)$ eine Datenallokation für A und bezeichnen sie ebenfalls mit Φ_A. In dieser Arbeit gehen wir grundsätzlich davon aus, daß Indizes und Blobs repliziert gespeichert werden. Für Arrays muß eine Datenallokation bestimmt werden. Diese kann für ein Array A mit Datenraum I in Form einer affinen Abbildung $\Phi_{A,\mathit{aff}} : I \to \mathbb{Z}^m$ gegeben sein oder wird wie in Abschnitt 5.2 beschrieben berechnet. Da der Zielraum einer affinen Abbildung außerhalb des Prozessorfeldes liegen kann, wird für jede affine Abbildung eine Datenallokation durch

$$\Phi_A : I \to P : i \to \lfloor \Phi_{A,\mathit{aff}}(i) \rfloor \% \lfloor S \rfloor$$

berechnet, wobei in $S = (S_1, \ldots, S_m)$ der Wert S_i wieder das durch *extremize* abgeschätzte Maximum aller Indizes darstellt, die von einer Parallelschleife gebunden sind, die ihrerseits von $i - 1$ weiteren Parallelschleifen umgeben ist.

3.3 Globale und lokale Räume

Unser Ziel ist ein SPMD-Programm, das Parallelität dadurch erreicht, daß nicht auf jedem Prozessor des Parallelrechners alle Vektoren des Indexraums aufgezählt werden. Außerdem soll nicht auf jedem Prozessor jedes Array vollständig gespeichert werden (gewissermaßen soll also nicht der gesamte Datenraum auf jedem Prozessor „aufgezählt“ werden). Dies ergibt eine Einteilung in globale und (für jeden Prozessor) lokale Räume.

Durch die in Abschnitt 3.2 beschriebenen Anweisungsallokationen (durch Parallelschleifen) wird für Anweisungen eine Partitionierung des Indexraums in Unter-Indexräume angegeben, die auf den verschiedenen Prozessoren durchlaufen werden sollen, wie in Abbildung 1 dargestellt. Von nichtganzzahligen Schrittweiten abgesehen besteht jeder dieser Unterräume prinzipiell aus der Menge der Indexvektoren, die sich nur in den von sequentiellen Schleifen aufgespannten Dimensionen unterscheiden. Dies sind also die Indexvektoren, deren Projektion auf die von Parallelschleifen aufgespannten Dimensionen den gleichen – von einer Anweisungsallokation Ψ gegebenen – Wert haben. Die Funktion β aus Abbildung 1 ist in unserem Fall die Identität, da die sequentiellen Schleifen genau die entsprechenden Koordinaten des Indexraums aufzählen und jede Prozessorkoordinate ebenfalls genau den Indexwert der entsprechenden Parallelschleife darstellt, der auf diesem Prozessor aufgezählt werden soll. Entsprechend wird, wie Abbildung 1 zeigt, für jedes Array durch eine Datenallokation Φ eine Partitionierung des Datenraums in Unter-Datenräume angegeben, von denen jeder auf einem anderen Prozessor zu speichern ist. Dabei kann es zweckmäßig sein, den zu speichernden Unter-Datenraum durch eine Abbildung α auf einen anderen Raum abzubilden – beispielsweise durch Translation um die untere Grenze, um bei Programmiersprachen wie `C` immer eine untere Grenze von 0 garantieren zu können. Wir nennen den ursprünglichen Datenraum D_G des Arrays (wie es in der Eingabe logisch spezifiziert wurde) den **globalen Datenraum**

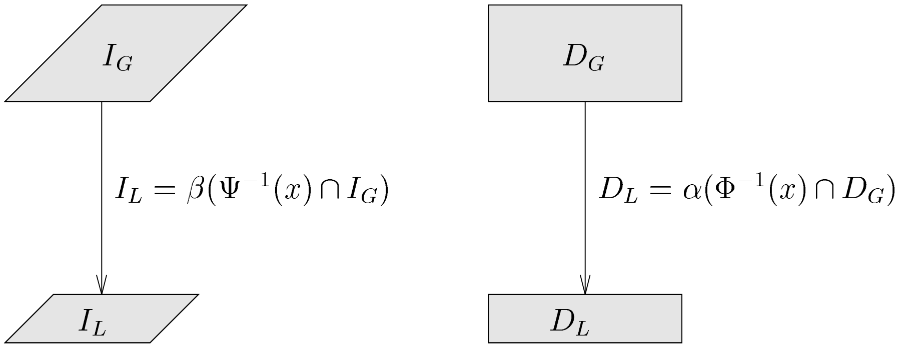


Abbildung 1: Globaler und lokaler Index- bzw. Datenraum (für Prozessor x)

und den aus dem zu speichernden Unterraum durch α für jeden Prozessor gegebenen physikalisch anzulegenden Datenraum D_L den **lokalen Datenraum**. Entsprechend heißt der gesamte Indexraum eines *Projekts* I_G **globaler Indexraum** und der auf einem Prozessor auszuführende Unter-Indexraum I_L **lokaler Indexraum**. Wenn wir von dem Datenraum eines Arrays sprechen, meinen wir damit – falls nicht anders angegeben – den globalen Datenraum D_G^{orig} des Arrays, mit dem es deklariert wurde. Unter dem Indexraum eines *Projekts* verstehen wir – falls nicht anders angegeben – den globalen Indexraum I_G, also ebenfalls die Eingabe nach Abschnitt 3.2 (über einen evtl. vorhandenen ursprünglichen Indexraum, aus dem der Indexraum des *Projekts* mit Schedules und Allokationen hervorgegangen sein könnte, haben wir prinzipiell keine Information).

Ein beliebiger Schleifensatz läßt sich im allgemeinen jedoch nicht sofort durch Angabe von `par`-Schleifen semantikerhaltend partitionieren. Hierzu ist u.U. eine Transformation des Indexraums notwendig, wie sie im Polyedermodell von Schedulern und Allokatoren (vgl. [Wie95], [Mei96]) berechnet werden. Bei einer begrenzten Anzahl an Prozessoren ist außerdem eine weitere Partitionierung des Indexraums nötig, die dafür sorgt, daß Teile von parallel ausführbaren Schleifen sequentiell ausgeführt und so weniger Prozessoren benötigt werden (vgl. [Sch97]). In Abschnitt 5.3 werden wir sehen, daß auch zur verteilten Speicherung eines Arrays ein Basiswechsel des Datenraums mit einer Abbildung R^{-1} nötig sein kann, wie in Abbildung 2 skizziert. Prinzipiell kann so auch der transformierte globale Datenraum D_G nicht rechteckig sein; wir gehen in diesem Fall jedoch davon aus, daß er durch Vergrößerung auf ein umliegendes Rechteck auf diese Form gebracht wurde: das hat den Vorteil, daß unabhängig von der Prozessorkoordinate die gleichen Grenzen für das Array angenommen werden können.[5] Auch der Datenraum kann für die Anzahl der Prozessoren zu groß sein. Gelöst wird dieses Problem durch „Tiling" des Datenraums, indem statt affinen Abbildungen als Datenallokation wie in 3.2 beschrieben ausschließlich *Modularabbildungen* verwendet werden.

3.4 Prinzipielles Ausführungsmodell

Das Ziel dieser Arbeit soll ein SPMD-Programm sein, das auf einem Parallelrechner mit verteiltem Speicher ausgeführt werden kann. Für jedes *Projekt* soll dieses Programm dabei äquivalente Funktionalität zur Verfügung stellen. Hierzu ist es nötig, gemeinsamen Speicher durch Einfügen von Kommunikationsroutinen zu simulieren.

[5] Dies ist ein **Muß**, falls alle Prozessoren das gleiche Programm ausführen und das Array statisch angelegt werden soll.

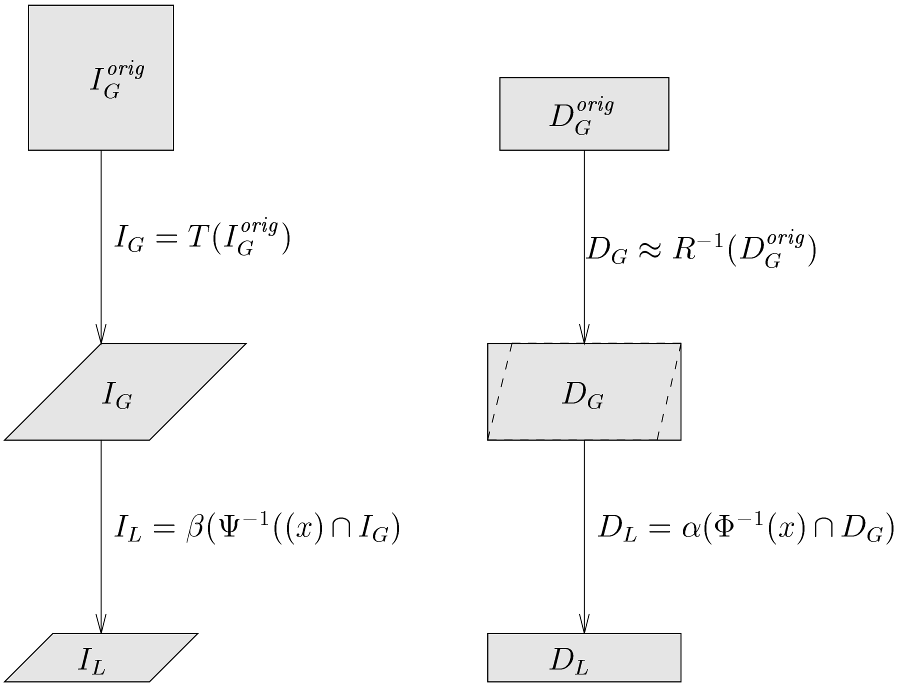


Abbildung 2: Globaler und lokaler Index- bzw. Datenraum (für Prozessor x) mit Transformationen

Die Korrektheit muß dabei durch Synchronisation mittels dieser Kommunikationsroutinen oder ggf. explizit mittels `barrier`-Befehl gewährleistet werden. Da es sich um ein SPMD-Programm handelt, wird Parallelität dadurch erreicht, daß dieses eine Programm (auch als *Node-Programm* bezeichnet) auf jedem der zur Verfügung stehenden Prozessoren des Parallelrechners gestartet wird. Der Bereich, der durch einen von einer Parallelschleife gebundenen Index durchgezählt wird, wird dabei auf jedem Prozessor auf einen anderen Teilbereich eingeschränkt (wir haben jedoch gesehen, daß dieser Bereich auch mehrelementig sein kann, da wir auch rationale Schrittweiten zulassen).

4 Anweisungsverteilung

Nachdem wir einer Eingabe nun eine Semantik zuordnen können, soll in diesem Kapitel darauf eingegangen werden, wie diese Semantik durch ein SPMD-Programm ausgedrückt werden kann, das auf dem Parallelrechner ausgeführt wird. In Abschnitt 4.1 gehen wir auf die Umwandlung rationaler Schrittweiten in ganzzahlige Schrittweiten ein. Abschnitt 4.2 beschreibt kurz die Abbildung eines m-dimensionalen Prozessorfeldes auf das eindimensionale Maschinenmodell. Abschnitt 4.3 stellt dann eine Strategie vor, nach der jeder Prozessor bestimmen kann, welchen Programmteil er auszuführen hat. Abschnitt 4.5 beschreibt die Realisierung im *LooPo*-Projekt.

4.1 Reduktion rationaler Schrittweiten

Beschreibung einer Schleife Wie in Abschnitt 3.2 beschrieben, erlauben wir Schleifen mit rationalen Grenzen und Schrittweiten als Eingabe. Solche Konstruktionen werden von den meisten imperativen Programmiersprachen jedoch nicht akzeptiert. In diesem Abschnitt beschreiben wir darum, wie die rationalen Grenzen und Schrittweiten auf ganzzahlige Werte reduziert werden können. Eine beliebige Schleife

`do` $j = \frac{l_1}{q_l}$ `to` $\frac{u_1}{q_u}$ `step` $\frac{s_1}{q_s}$
  $\ldots A[\varphi(j)] \ldots$

definiert eine Menge von Schleifendurchläufen $\mathbb{N} \supseteq I = \{0, \ldots, \left\lfloor (\frac{u_1}{q_u} - \frac{l_1}{q_l}) \cdot \frac{q_s}{s_1} \right\rfloor\}$ und zwei Abbildungen

$$\begin{array}{llll} \Theta : & I \rightarrow \mathbb{Q} & : i \rightarrow \frac{l_1}{q_l} + i \cdot \frac{s_1}{q_s} \\ \Psi : & I \rightarrow \mathbb{N} & : i \rightarrow \lfloor \Theta(i) \rfloor \end{array}$$

Dabei bildet die Funktion Θ den Schleifendurchlauf i auf den rationalen Indexwert ab, mit dem die Schleife durchlaufen werden soll. Je nachdem, ob es sich bei der Schleife um eine sequentielle oder parallele Schleife handelt, bildet Ψ den i-ten Schleifendurchlauf auf den Zeitpunkt oder die Prozessorkoordinate für die Ausführung des Rumpfes ab. Um einfacher rechnen zu können, bringen wir die Schleife zunächst auf ihren Hauptnenner $q = kgV(q_l, q_u, q_s) > 0$ (für $a, b \in \mathbb{Z}$ gilt $kgV(a, b) = kgV(|a|, |b|)$). Die Abbildung Θ ist dann:

$$\begin{array}{rcl} \Theta : I \rightarrow \mathbb{Q} : i & \rightarrow & \frac{l_1 \cdot \frac{q}{q_l}}{q} + i \cdot \frac{s_1 \cdot \frac{q}{q_s}}{q} \\ & = & \frac{l + i \cdot s}{q} \end{array}$$

mit $l = l_1 \cdot \frac{q}{q_l}$, $s = s_1 \cdot \frac{q}{q_s}$. Außerdem sei $u = u_1 \cdot \frac{q}{q_u}$. Wir können nun diese eine Schleife in zwei Schleifen überführen, von denen die äußere das Bild von Ψ durchzählt, während die innere Schleife die Zähler z der zugehörigen Indexwerte aufzählt. Dieses Verfahren entspricht einer Art Strip-Mining (vgl. z.B. [Wol95]). Wie in Abschnitt 3.2 erwähnt, sind die Anweisungen im Rumpf der inneren Schleife dann eigentlich für jeden Durchlauf zu demselben Raumzeitpunkt auszuführen; die innere Schleife müßte korrekterweise also als quasiparallele Schleife interpretiert werden. Mit dieser Art der Parallelität wollen wir uns in dieser Arbeit jedoch nicht befassen. Darum genügt es, sie als sequentielle Schleife auszuführen. Die äußere Schleife hingegen repräsentiert die gleiche Alloktion in der Raumzeit wie die ursprüngliche Schleife. Sie ist darum genau dann parallel auszuführen, wenn auch die ursprüngliche Schleife eine Parallelschleife war. Im Rumpf dieser Schleifen ist dann jedes Vorkommen von i durch $\frac{z}{q}$ zu ersetzen.

Schleifengrenzen Die Grenzen der äußeren Schleife sind klar: es müssen alle Werte aufgezählt werden, die im Bild von Ψ liegen. Die untere Grenze ist $\left\lfloor \frac{l_1}{q_l} \right\rfloor$, die obere Grenze ist $\left\lfloor \frac{u_1}{q_u} \right\rfloor$, und die Schrittweite ist $\mathrm{sgn}(\frac{s_1}{q_s})$. Die innere Schleife muß für jeden Wert m, den die äußere Schleife aufzählt, für $\frac{s_1}{q_s} > 0$ die Menge

$$\{l, \ldots, u\} \cap (q \cdot \Theta(\Psi^{-1}(m)))$$

durchzählen, bzw. für $\frac{s_1}{q_s} < 0$ die Menge

$$\{u, \ldots, l\} \cap (q \cdot \Theta(\Psi^{-1}(m)))$$

Für das Urbild von Ψ gilt nach Satz 2.14:

$$\Psi^{-1}(m) = \begin{cases} \{\left\lfloor \frac{q \cdot m - l - 1}{s} \right\rfloor + 1, \ldots, \left\lfloor \frac{(m+1) \cdot q - l - 1}{s} \right\rfloor\} & \text{falls } \mathrm{sgn}(s) = 1 \\ \{\left\lfloor \frac{(m+1) \cdot q - l}{s} \right\rfloor + 1, \ldots, \left\lfloor \frac{m \cdot q - l}{s} \right\rfloor\} & \text{falls } \mathrm{sgn}(s) = -1 \end{cases}$$

($\mathrm{sgn}(s)$ bzw. $\mathrm{sgn}(q)$ kann aus der Formel gestrichen werden, da $\mathrm{sgn}(q) = 1$.) Damit ist

$$q \cdot \Theta(\Psi^{-1}(m)) = \begin{cases} \{l + (\left\lfloor \frac{m \cdot q - l - 1}{s} \right\rfloor + 1) \cdot s, \ldots, l + \left\lfloor \frac{(m+1) \cdot q - l - 1}{s} \right\rfloor \cdot s\} \\ \qquad \text{falls } \mathrm{sgn}(s) = 1 \\ \\ \{l + (\left\lfloor \frac{(m+1) \cdot q - l}{s} \right\rfloor + 1) \cdot s, \ldots, l + \left\lfloor \frac{m \cdot q - l}{s} \right\rfloor \cdot s\} \\ \qquad \text{falls } \mathrm{sgn}(s) = -1 \end{cases}$$

Dabei ist die Schrittweite s (Schrittrichtung wird beibehalten, jedoch nur noch der Zähler durchgezählt). Der gesamte Schleifensatz lautet also für $\mathrm{sgn}(s) = 1$:

$\texttt{do } m = \left\lfloor \frac{l}{q} \right\rfloor \texttt{ to } \left\lfloor \frac{u}{q} \right\rfloor$
$\quad \texttt{for } z = \max(l, l + (\left\lfloor \frac{m \cdot q - l - 1}{s} \right\rfloor + 1) \cdot s) \texttt{ to } \min(u, l + \left\lfloor \frac{(m+1) \cdot q - l - 1}{s} \right\rfloor \cdot s)$
$\qquad\qquad \texttt{step } s$
$\qquad \ldots A[\varphi(\frac{z}{q})] \ldots$

und für $\mathrm{sgn}(s) = -1$:

$\texttt{do } m = \left\lfloor \frac{l}{q} \right\rfloor \texttt{ to } \left\lfloor \frac{u}{q} \right\rfloor \texttt{ step } -1$
$\quad \texttt{for } z = \max(u, l + \left\lfloor \frac{m \cdot q - l}{s} \right\rfloor \cdot s) \texttt{ to } \min(l, l + (\left\lfloor \frac{(m+1) \cdot q - l}{s} \right\rfloor + 1) \cdot s) \texttt{ step } s$
$\qquad \ldots A[\varphi(\frac{z}{q})] \ldots$

Schleifenvertauschung Unter Umständen wird durch dieses Verfahren eine `doacross`-Schleife in zwei Schleifen umgewandelt. Von dieser Schleife *getragene* Abhängigkeiten könnten aussehen, wie in Abbildung 3 skizziert. Solche Abhängigkeiten werden beispielsweise von dem *Projekt*

$\texttt{par } p = 0 \texttt{ to } 1 \texttt{ step } \frac{1}{2}$
$\quad \texttt{for } t = 0 \texttt{ to } 2$
$\qquad A[t, 2 \cdot p] := A[t-1, 2 \cdot p - 1] + A[t-1, 2 \cdot p + 1];$

erzeugt. Dieses *Projekt* würde durch obiges Verfahren umgeschrieben zu

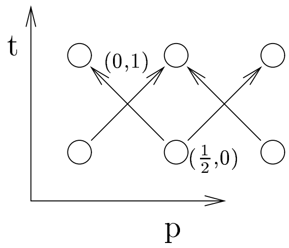


Abbildung 3: Abhängigkeiten, die bei der (teilweisen) Ausführung der p-Schleife als sequentielle Schleife nicht korrekt wiedergegeben werden.

```
par p = 0 to 1
  for t_n = max(0, 2·p) to min(2, 2·p + 1)
    for t = 0 to 2
      A[t, t_n] := A[t − 1, t_n − 1] + A[t − 1, t_n + 1];
```

Da die t_n-Schleife aber sequentiell und nicht quasiparallel ausgeführt wird, wird der Indexpunkt $(0,1)$ vor dem Punkt $(\frac{1}{2},0)$ ausgeführt, was nicht semantikerhaltend ist. Darum muß in einem solchen Fall die innere Schleife in die am weitesten außen liegende Zeitschleife hineingezogen werden, für die gilt, daß alle von der ursprünglichen Parallelschleife *getragenen* Abhängigkeiten in dieser oder einer umliegenden sequentiellen Schleife liegen. Dabei ist zu beachten:

- Taucht der Index der hineinzuziehenden Schleife in einer Schleifengrenze auf, in deren Schleife hineingezogen werden soll, so ist dieser Index durch den entsprechenden Extremalwert zu ersetzen. Die durch die unveränderten Schleifengrenzen gegebenen Bedingungen werden in eine Menge C von Bedingungen aufgenommen.
- Jede von dem Index der hineinzuziehenden Schleife abhängige Bedingung einer `if`- oder `while`-Anweisung, in die die Schleife hineingezogen werden soll, gehört zu der Bedingungsmenge C. Aus der `if`- oder `while`-Anweisung ist diese Bedingung zu löschen.
- Der Rumpf der hineingezogenen Schleife ist durch eine `if`-Anweisung zu umschließen, die alle Bedingungen aus C ($\wedge$-verknüpft) abprüft.

In obigem Beispiel-*Projekt* werden keine besonderen Bedingungen erzeugt. Das Resultat lautet hierfür also:

```
par p = 0 to 1
  for t = 0 to 2
    for t_n = max(0, 2·p) to min(2, 2·p + 1)
      A[t, t_n] := A[t − 1, t_n − 1] + A[t − 1, t_n + 1];
```

Von nun an gehen wir davon aus, daß diese Transformation auf ein zu betrachtendes *Projekt* **schon angewandt** wurde. Damit können wir für jede betrachtete Schleife **ganzzahlige Schleifengrenzen und Schrittweiten** voraussetzen.[6]

4.2 Prozessoren in $\mathbb{N}^m$

Wie in Abschnitt 3.2 beschrieben, gehen wir von einer parallelen Maschine aus, deren Prozessoren eindimensional durchnumeriert sind und deren Identifikationsnummer durch eine Abbildung $P_V : \mathbb{N} \rightarrow \mathbb{N}^m$ einer Position in dem durch ein *Projekt*

[6] Die Reduktion rationaler Schrittweiten in *LooPo* wurde von Robert Günz [Gün96] implementiert.

definierten m-dimensionalen Gitter zugeordnet wird. Ist für $i \in \{1, \ldots, m\}$ die maximale Gitterkoordinate in Dimension i durch S_i gegeben[7], kann P_V durch die Abbildung

$$x \rightarrow ((x/p(1))\%S_1, \ldots, (x/p(n))\%S_m)$$

und die Umkehrung durch

$$y \rightarrow (\sum i : i \in \{1, \ldots, m\} : \pi_i(y) \cdot p(i))$$

gegeben werden, wobei $p(i) = (\prod j : j \in \{i+1, \ldots, m\} : S_j)$. Kann die Zielarchitektur durch ein Gitter $\{(P_1, \ldots, P_u) \mid (\forall i : i \in \{1, \ldots, u\} : 0 \leq P_i < R_i)\}$ mit $m \leq u \in \mathbb{N}$ modelliert werden, so ist es auch möglich, diese Architektur besser zu unterstützen (z.B. durch Bestimmung einer anderen Funktion für p). Ist p_{max} die Anzahl der Prozessoren des Parallelrechners,[8] so wird durch $P_V(p_{max})$ eine (u.U. echte) Obermenge des Prozessorfeldes generiert. Wir bezeichnen $P_V(p_{max})$ im folgenden als das **erweiterte Prozessorfeld** eines *Projekts*.

4.3 Dreistufenmodell

Unser Ziel ist ein SPMD-Programm, das auf dem gesamten erweiterten Prozessorfeld ausgeführt wird. Welcher Prozessor des Parallelrechners dabei welche Anweisung eines *Projekts* auszuführen hat, läßt sich zunächst an folgendem Beispiel verdeutlichen:

Beispiel 4.1 *Gegeben sei eine Parallelrechnerarchitektur mit 32 Prozessoren und das* Projekt

S_1 :
`for` $t = 0$ `to` 9
S_2 :
  `par` $p_1 = t$ `to` 9
S_3 :
    $B[t, p_1] := 42$;
S_4 :
    `if` $p_1 = 7$ `then`
S_5 :
      `par` $p_2 = 0$ `to` 2
S_6 :
        $A[t, p_1, p_2] := 2 \cdot A[t-1, p_1, p_2]$;

Dann besteht das Prozessorfeld P aus den virtuellen Prozessoren $P = \{(p_1, p_2) \mid 0 \leq p_1 \leq 9, 0 \leq p_2 \leq 2\}$, die auf 30 der 32 Prozessoren abgebildet werden. Zwei der Prozessoren des Parallelrechners werden für die Laufzeit des Projektes *also untätig bleiben.*

Das *Projekt* aus Beispiel 4.1 besteht aus den Zuweisungen S_3 und S_6 und den Kontrollanweisung S_1, S_2, S_4 und S_5. Die Kontrollanweisungen führen auf jedem Prozessor (u.U. in Abhängigkeit von dessen Position im Prozessorfeld) jeweils eine Einschränkung ein, die bestimmt, ob im Rumpf befindliche Zuweisungen oder Prozeduraufrufe auszuführen sind, und eine weitere, die bestimmt, ob im Rumpf befindliche weitere Kontrollanweisungen ausgeführt werden sollen (welche zusätzliche Einschränkungen definieren können).

[7]Die minimale Gitterkoordinate ist immer 0.

[8]Diese muß mindestens so groß sein wie $(\prod i : i \in \{1, \ldots, m\} : S_i)$, da sonst nicht genügend Prozessoren zur Ausführung des Programms vorhanden sind.

Wir gehen den Programmtext in der folgenden Tabelle schrittweise durch:

Befehl	Maske für Kontroll-anweisungen	Maske für Zuweisungen	Begründung/Bemerkung
S_1	$0 \leq p_1 \leq 9$, $0 \leq p_2 \leq 2$	$p_1 = p_2 = 0$	Das gesamte Prozessorfeld kann für folgende Anweisungen benötigt werden, es ist aber noch keine Dimension des Prozessorfeldes aufgespannt.
S_2	$t \leq p_1 \leq 9$, $0 \leq p_2 \leq 2$	$t \leq p_1 \leq 9$, $p_2 = 0$	Die erste Dimension des Prozessorfeldes ist aufgespannt, die zweite aber nicht; deshalb führen nur Prozessoren mit $p_2 = 0$ Zuweisungen aus; es können aber noch alle Prozessoren mit $p_1 \geq t$ durch eine evtl. folgende Parallelschleife aufgezählt werden.
S_3	$t \leq p_1 \leq 9$, $0 \leq p_2 \leq 2$	$t \leq p_1 \leq 9$, $p_2 = 0$	Zuweisung S_3 wird gemäß Bedingung (nur eine Raumdimension aufgezählt) nur von Prozessoren mit $p_2 = 0$ ausgeführt.
S_4	$t \leq p_1 = 7$, $0 \leq p_2 \leq 2$	$t \leq p_1 = 7$, $p_2 = 0$	Neue Einschränkung $p_1 = 7$ ändert den Kontrollfluß: auch die zweite Prozessordimension kann im Rumpf nur noch von Prozessoren mit $p_1 = 7$ aufgespannt werden.
S_5	$t \leq p_1 = 7$, $0 \leq p_2 \leq 2$	$t \leq p_1 = 7$, $0 \leq p_2 \leq 2$	Die zweite Prozessordimension wird aufgespannt; damit werden Zuweisungen nicht mehr nur von Prozessoren mit $p_2 = 0$ ausgeführt, sondern von allen, die diese Stelle überhaupt erreichen.
S_6	$t \leq p_1 = 7$, $0 \leq p_2 \leq 2$	$t \leq p_1 = 7$, $0 \leq p_2 \leq 2$	Zuweisung S_6 wird gemäß Bedingung von allen Prozessoren ausgeführt, die aufgrund der Kontrollanweisungsbedingungen bis an diese Stelle gekommen sind.

Welche Anweisung von welchen Prozessoren des virtuellen Prozessorfeldes auszuführen ist, kann damit der folgenden Tabelle entnommen werden:

Anweisung	ausführende Prozessoren
S_1	$P = \{(p_1, p_2) \mid 0 \leq p_1 \leq 9, 0 \leq p_2 \leq 2\}$
S_2	P
S_3	$\{(p_1, 0) \mid t \leq p_1 \leq 9\}$
S_4	$P \cap \{(p_1, p_2) \mid t \leq p_1 \leq 9\}$
S_5	$\{(7, p_2) \mid 0 \leq p_2 \leq 2 t \leq 7\}$
S_6	$\{(7, p_2) \mid 0 \leq p_2 \leq 2, t \leq 7\}$

Hierbei wird deutlich, daß Kontrollanweisungen (hier die Bedingungsauswertung des `ifs` in S_4 und des `pars` in S_5) durchaus von Prozessoren ausgeführt werden müssen, die nach unserer Definition der Semantik einer Parallelschleife **keine** Berechnungen durchführen dürfen (z.B. muß S_4 auch von $(t, 1)$ ausgeführt werden, obwohl dieser S_3 **nicht** ausführen darf). Jedoch dürfen auch diese Anweisungen nicht einfach vom gesamten Prozessorfeld ausgeführt werden (beispielsweise wird S_5 für $t = 8$ von keinem einzigen Prozessor des gesamten Prozessorfeldes ausgeführt – für $t \leq 7$ gibt es Prozessoren, die diese Anweisung ausführen). Außerdem gibt es, wie schon erwähnt, zwei Prozessoren des Parallelrechners, die nicht zum Prozessorfeld gehören und deshalb auch keine der Anweisungen ausführen dürfen.

Drei Ausführungsebenen Für die Ausführung von Anweisungen kann sich ein Prozessor also beim Bearbeiten einer `par`-Schleife (oder beim Eintritt in ein *Projekt*) für eine oder mehrere von drei Stufen qualifizieren:

> Stufe I: Der Prozessor gehört zum Prozessorfeld und kann Arraysektionen speichern. Nur Stufe I-Prozessoren bearbeiten den Code eines *Projektes*.
>
> Stufe II: Der Prozessor $(p_1, \ldots, p_m)$ erfüllt die Voraussetzungen aller umliegenden `if`-Anweisungen und aller umliegenden Parallelschleifen, d.h. für jede Schleife `par` $k_i = l_i$ `to` u_i `step` s_i, die von $i - 1$ Parallelschleifen umgeben wird, gilt $l_i \leq p_i \leq u_i$ und $p_i \% s_i = 0$. Stufe II-Prozessoren führen Kontrollanweisungen (und ggf. deren Rumpf) aus.
>
> Stufe III: zusätzlich zu den Bedingungen der Stufe II gilt $(\forall j : j \in \{i + 1, \ldots, m\} : p_j = 0)$. Stufe III-Prozessoren führen Zuweisungen und Prozeduraufrufe aus.

Veranschaulichung Die Menge der Stufe I-Prozessoren enthält also die Menge der Stufe II-Prozessoren, die wiederum auch alle Stufe III-Prozessoren enthält. Die Abbildungen 4–6 skizzieren dieses Dreistufenmodell.

Den Abbildungen 5 und 6 liegt dabei das Beispiel 4.1 zugrunde. Die beiden für das Prozessorfeld überzähligen Prozessoren wurden hier willkürlich in das Gitter eingepaßt. Da ihre Koordinaten in diesem Gitter größer sind als die, die durch Parallelschleifen aufgezählt werden können, können diese Prozessoren dann von dem *Projekt* durch eine Abfrage ausgeschlossen werden, die alle Prozessorkoordinaten auf die im Prozessorfeld zulässigen Maximalwerte hin überprüft. In jedem Zeitschritt wird zunächst die $(p1, 0)$-Reihe $p1 \geq t$ aktiv, um Anweisung S_3 auszuführen. Die Prozessoren $(p1, p2)$ mit $p1 \geq t$ müssen evtl. auch die Anweisungen S_5 und S_6 ausführen und sind darum beim Eintritt in die Schleife S_2 der Stufe II zugeordnet worden. Für $t \leq 7$ werden dann die Prozessoren $\{(7, p_2) \mid 0 \leq p_2 \leq 2\}$ aktiviert, um S_6 auszuführen. Alle anderen Prozessoren erreichen die qualifizierende Schleife S_5 nicht.

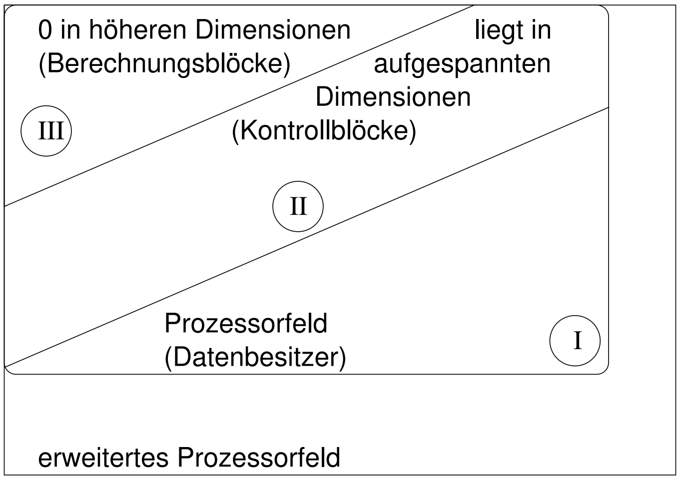


Abbildung 4: Das Dreistufenmodell: Stufe I-Prozessoren besitzen Daten und müssen darum evtl. Kommunikationen ausführen; Stufe II-Prozessoren müssen Kontrollblöcke ausführen; Stufe III-Prozessoren müssen Kontroll- und Berechnungsblöcke ausführen.

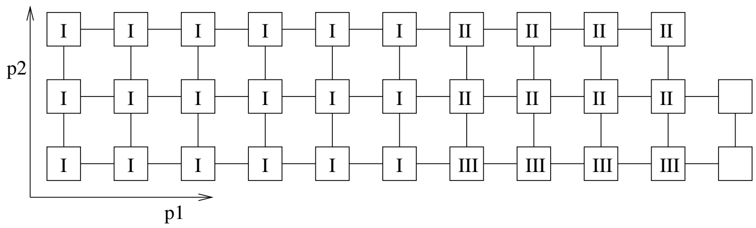


Abbildung 5: Erweitertes Prozessorfeld für Anweisung S_3 bei $t = 6$: Nur die Prozessoren (p_1, p_2) mit $t \leq p1 \leq 9$ können evtl. noch Zuweisungen ausführen, nur die, bei denen zusätzlich $p_2 = 0$ ist, führen S_3 tatsächlich aus.

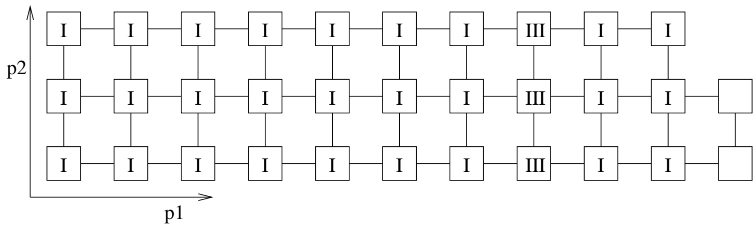


Abbildung 6: Erweitertes Prozessorfeld für Anweisung S_6 bei $t = 6$: Aus dem Prozessorfeld für S_3 werden alle Prozessoren mit $p_1 \neq 7$ ausgeblendet; dafür gibt es keine Bedingung für p_2, um die Zuweisung auszuführen.

4.4 Basisblöcke

Im letzten Abschnitt haben wir gesehen, daß Kontrollanweisungen auf einem Prozessor unter anderen Bedingungen auszuführen sind als Zuweisungen oder Prozeduraufrufe. Die Anweisungen eines *Projektes* lassen sich also in zwei Klassen einteilen. Diese Einteilung wollen wir in diesem Abschnitt vornehmen. Wir definieren:

Definition 4.2 *Eine Berechnung zur Auswertung der Bedingung einer Kontrollanweisung nennen wir* **Kontrollberechnung**.

Bemerkung: Es ist möglich, daß mehrere Berechnungen nötig sind, um Daten für eine einzige Anweisung zu berechnen, beispielsweise die Obergrenzenberechnung in einer Schleife der Form

`do` $i = 0$ `to` $\max(\min(4, 7 + j - m), \lfloor j \cdot \pi \rfloor, A[j])$

Definition 4.3 *Unter einem* **Basisblock** *verstehen wir eine maximale lineare Codesequenz von Kontrollberechnungen, sowie eine maximale, lineare kontrollberechnungsfreie Codesequenz.*

Im folgenden soll die Bezeichnung *Berechnungsblock* für einen kontrollberechnungsfreien *Basisblock* gelten. Einen Kontrollberechnungen enthaltenden *Basisblock* nennen wir *Kontrollblock*.

Wie wir gesehen haben, müssen bei der Ausführung von Parallelschleifen in einem SPMD-Programm Kontrollberechnungen durch andere `if`-Anweisungen als Zuweisungen oder Prozeduraufrufe maskiert werden (prinzipiell müßte jede Anweisung durch ein eigenes `if` maskiert werden): die Bedingung für Kontrollanweisungen ist die Qualifikation als Stufe II-Prozessor, die Bedingung für Zuweisungen und Prozeduraufrufe ist die Qualifikation als Stufe III-Prozessor. Jedoch ist diese Maske für die Stufe III-Bedingung für alle Anweisungen eines *Berechnungsblocks* gleich. Es genügt also, jeden *Berechnungsblock* in eine entsprechende `if`-Anweisung einzuschließen. Da außerdem die Bedingung für die Ausführung von Kontrollanweisungen durch die Bedingung der Berechnungsblöcke impliziert wird, genügt es für das Abprüfen der Stufe II-Bedingung, alle Anweisungen im Rumpf einer Parallelschleife durch ein einziges `if` zu maskieren, das diese Stufe II-Bedingung überprüft.

Der gesamte Code eines *Projektes* ist zusätzlich einmal durch eine `if`-Anweisung zu umschließen, die garantiert, daß nur die Prozessoren das Programmstück ausführen, die irgendwann von einer Parallelschleife aufgespannt werden. Insgesamt können dann alle `par`-Schleifen durch diese `if`-Anweisungen (zusammen mit den Berechnungen für die entsprechenden Qualifikationen) ersetzt werden.

Dieses Vorgehen entspricht dem Maskieren und Zusammenfassen, wie es bei `HPF`-Compilern angewandt wird, um ein gegebenes Programm zu parallelisieren (vgl. [CK88], [CFH+92], [ZC93], [SPB93], [Su93]). Die Art der Maskierung unterscheidet sich hier jedoch von der in `HPF`, da es dort keine expliziten Parallelschleifen gibt und der Compiler die entsprechenden Masken ausschließlich aufgrund der Datenverteilung ermittelt (vgl. [Hig97]). In folgendem Beispiel soll das prinzipielle Vorgehen eines `HPF`-Compilers kurz verdeutlicht werden:

Beispiel 4.4 *Ein* Projekt *mit* `ON`*-Klauseln als* `HPF`*-Äquivalent zu Parallelschleifen und eine mögliche vom* `HPF`*-Compiler generierten SPMD-Version für gemeinsamen Speicher (die durch* `ifs` *garantiert, daß jeder Prozessor nur die ihm zugeteilten Anweisungen ausführt):*

```
S1 :
for t = 0 to 9
S2 :
  for p1 = t to 9
    ON PROCESSORS(p1, :) BEGIN
S3 :
      ON PROCESSORS(p1, 0)
      A[p1, t] := 42 · t + p1 ;
S4 :
      ON PROCESSORS(p1, 0)
      B[p1, t] := A[p1, t − 3] + 22;
S5 :
      if t + p1 < 9 then
S6 :
        for p2 = t to 9
          ON PROCESSORS(p1, p2)
          BEGIN
S7 :
            T[p2, t] := B[p2, t] + A[p2, t];
S8 :
            A[p2, t] := T[p2, t]/2;
          END ON
S9 :
      ON PROCESSORS(p1, 0)
      A[p1, t] := 2 · A[p1, t];
    END ON
```

In der SPMD-Version:

```
S1 :
for t = 0 to 9
S2 :
  for p1 = t to 9
    if MyID1 = p1 then
S3 :
      if MyID1 = p1
        ∧ MyID2 = 0 then
        A[p1, t] := 42 · t + p1 ;
S4 :
      if MyID1 = p1
        ∧ MyID2 = 0 then
        B[p1, t] := A[p1, t − 3] + 22;
S5 :
      if t + p1 < 9 then
S6 :
        for p2 = t to 9
S7 :
          if MyID1 = p1
            ∧ MyID2 = p2 then
            T[p2, t] := B[p2, t] + A[p2, t];
S8 :
            A[p2, t] := T[p2, t]/2;
S9 :
      if MyID1 = p1
        ∧ MyID2 = 0 then
        A[p1, t] := 2 · A[p1, t];
```

$MyID_i$ bezeichnet dabei die Position des ausführenden Prozessors in Dimension i. Durch Zusammenfassen der Masken und Vereinfachung der Schleifengrenzen gegen die Vereinigung der Masken im Rumpf (vgl. [Su93]) entsteht dann etwa ein SPMD-Programm wie im folgenden links abgedruckt; weitere Vereinfachungen können auch auf ein Programm führen wie auf der rechten Seite zu sehen:

S_1 :
`for` $t = 0$ `to` 9
S_2 :
  `for` $p_1 = \max(MyID_1, t)$ `to` $\min(MyID_1, 9)$
S_3 :
    `if` $MyID_2 = 0$ `then`
      $A[p_1, t] := 42 \cdot t + p_1$;
S_4 :
    `if` $MyID_2 = 0$ `then`
      $B[p_1, t] := A[p_1, t-3] + 22$;
S_5 :
    `if` $t + p_1 < 9$ `then`
S_6 :
      `for` $p_2 = \max(MyID_2, t)$ `to` $\min(MyID_2, 9)$
S_7 :
        $T[p_2, t] := B[p_2, t] + A[p_2, t]$;
S_8 :
        $A[p_2, t] := T[p_2, t]/2$;
S_9 :
    `if` $MyID_2 = 0$ `then`
      $A[p_1, t] := 2 \cdot A[p_1, t]$;

S_1 :
`for` $t = 0$ `to` 9
S_3/S_4 :
  `if` $t \leq MyID_1 \leq 9 \wedge MyID_2 = 0$ `then`
    $A[MyID_1, t] := 42 \cdot t + MyID_1$;
    $B[MyID_1, t] := A[MyID_1, t-3] + 22$;
S_5 :
  `if` $t \leq MyID_1 \leq 9$ `then`
    `if` $t + MyID_1 < 9$ `then`
S_7/S_8 :
      `if` $t \leq MyID_2 \leq 9$ `then`
        $T[MyID_2, t] := B[MyID_2, t] + A[MyID_2, t]$;
        $A[MyID_2, t] := T[MyID_2, t]/2$;
S_9 :
  `if` $t \leq MyID_1 \leq 9 \wedge MyID_2 = 0$ `then`
    $A[MyID_1, t] := 2 \cdot A[MyID_1, t]$;

Die eingesetzten `if`*-Anweisungen, die durch die* `ON`*-Klausel forciert wurden, entsprechen hier genau der von uns getroffenen Abstufung in die drei Ausführungsebenen. Der* `HPF`*-Compiler erreicht eine Partitionierung des Indexraums also, indem alle Anweisungen durch* `if`*-Anweisungen maskiert werden, die für jeden Prozessor alle nicht auszuführenden Indexvektoren „ausblenden“. Bedingungen, die für alle Anweisungen im Rumpf einer Schleife gleich sind, können dann evtl. die Anzahl der auszuführenden Iterationen einer Schleife verringern und müssen dann nicht mehr abgeprüft werden. Wir haben eine feste Partitionierung vorgegeben und können eine* `par`*-Schleife direkt in eine* `if`*-Anweisung umformen. Allerdings werden wir später sehen, daß auch Prozessoren, die nicht explizit von einer* `par`*-Schleife aufgezählt werden, als* Owner *durchaus Anweisungen ausführen müssen; hierfür werden wir im Gegensatz zu* `HPF`*-Compilern, die den Indexraum möglichst weit einschränken, den vorgegebenen Indexraum (in möglichst geringem Maße) erweitern.*

4.5 Anweisungsverteilung in *LooPo*

Entsprechend dem in Abschnitt 4.4 gezeigten Vorgehen werden `if`-Anweisungen lediglich beim Eintreten in den Rumpf einer Parallelschleife (zum Abprüfen auf Stufe II) und um jeden *Berechnungsblock* (zum Abprüfen auf Stufe III) generiert. In Beispiel 4.4 ergibt das Code der Form:

S_1 :
`for` $t = 0$ `to` 9
S_2 :
  $P_{II}[0] := t \leq MyID_1 \leq 9$;

$P_{III}[0] := \mathit{MyID}_2 = 0$;
`if` $P_{II}[0]$ `then`
S_3/S_4 :
`if` $P_{III}[0]$ `then`
$A[\mathit{MyID}_1, t] := 42 \cdot t + \mathit{MyID}_1$;
$B[\mathit{MyID}_1, t] := A[\mathit{MyID}_1, t-3] + 22$;
S_5 :
`if` $t + \mathit{MyID}_1 < 9$ `then`
S_6 :
$P_{II}[1] := t \leq \mathit{MyID}_2 \leq 9$;
$P_{III}[1] := \text{true}$;
`if` $P_{II}[1]$ `then`
S_7/S_8 :
`if` $P_{III}[1]$ `then`
$T[\mathit{MyID}_2, t] := B[\mathit{MyID}_2, t] + A[\mathit{MyID}_2, t]$;
$A[\mathit{MyID}_2, t] := T[\mathit{MyID}_2, t]/2$;
S_9 :
`if` $P_{III}[0]$ `then`
$A[\mathit{MyID}_1, t] := 2 \cdot A[\mathit{MyID}_1, t]$;

Bei der Ausgabe eines *Projekts* durch *LooPo* müssen alle drei Qualifikationsstufen aus Abschnitt 4.3 durch `if`-Anweisungen abgeprüft werden:

Für jede Parallelschleife wird der Aufruf an eine Funktion des Laufzeitsystems ausgegeben. Diese Funktion berechnet die Bedingungen für die Stufen II und III für diese Parallelschleife. Die Bedingung für Stufe II wird dann einem Skalar zugewiesen, die Bedingung für Stufe III einem Element in einem Array der Länge m. Außerdem gibt die Funktion die Position des ausführenden Prozessors in der der Schachtelungstiefe entsprechenden Dimension des Prozessorfeldes als Wert für den durch die Schleife gebundenen Index zurück. Der Wahrheitswert für Stufe II wird direkt nach Aufruf dieser Funktion von einer `if`-Anweisung abgeprüft, die den gesamten Rumpf der Parallelschleife umfaßt. Um die Stufe III-Qualifikation abzuprüfen, wird jeder *Berechnungsblock* von einer entsprechende `if`-Anweisung umgeben. Außerdem ist vorher die Qualifikation eines Prozessors auf Stufe I zu überprüfen. Stufe I bestimmt darüber, ob der Prozessor überhaupt irgendwelchen Code des *Projekts* auszuführen hat. Dies wird zur Laufzeit überprüft, indem jedem Prozessor gemäß Abschnitt 4.2 durch eine injektive Funktion eine Position in einem m-dimensionalen Gitter zugeordnet wird, dessen Ausdehnung in jeder Komponente zur Compilezeit durch die Abschätzung mittels der *extremize*-Funktion bestimmt wurde. Müßte die Position eines Prozessors in diesem Gitter aufgrund dieser Abbildung in einer Komponente einen größeren Wert haben als die Ausdehnung des Gitters in dieser Dimension, gehört der Prozessor nicht zum Prozessorfeld. In diesem Fall wird durch das Laufzeitsystem eine Variable auf false gesetzt. Diese Variable wird von einem `if` abgeprüft, das das restliche *Projekt* einschließt.

Das Resultat:

Beispiel 4.5 *Die Ausgabe für das Programmsegment aus Beispiel 4.4 (nach Entfernen eingefügter Kommunikations- und Synchronisationsanweisungen):*

```
LooPo_beginSection(2,10,10);
if(LOOPO_SECTION_OK)
 {
  for(t=0;t<=9;t++)
   {
```

```
    p1=LooPo_parBegin(t,9);
    if(LOOPO_PARFOR_OK)
     {
      if(LOOPO_COMPUTE_OK)
       {
        loopo_targout_local_0=42*t+p1;
        loopo_targout_local_1=22+loopo_targout_local_2;
       }
      if(p1+t < 9)
       {
        p2=LooPo_parBegin(t,9);
        if(LOOPO_PARFOR_OK)
         {
          if(LOOPO_COMPUTE_OK)
           {
            loopo_targout_local_4=loopo_targout_local_5+loopo_targout_local_6;
            loopo_targout_local_5=((double)(loopo_targout_local_4))/2;
           }
         }
        LooPo_parEnd();
       }
      if(LOOPO_COMPUTE_OK)
       {
        loopo_targout_local_0=2*loopo_targout_local_0;
       }
     }
    LooPo_parEnd();
   }
 }
LooPo_endSection();
}
```

`LooPo_beginSection(2,10,10)` *legt ein 2-dimensionales Gitter mit* 10×10 *Prozessoren an. Durch* `LooPo_parBegin` *werden die Stufe II- und Stufe III-Bedingungen berechnet und die Koordinate des Prozessors in der durch die* `par`*-Schleife aufgezählten Dimension zurückgegeben.* `if(LOOPO_PARFOR_OK)` *prüft die Stufe II-Bedingung einer zuvor ausgeführten* `LooPo_parBegin`*-Anweisung aus, und* `if(LOOPO_COMPUTE_OK)` *testet die zur aktuellen Schachtelungstiefe gehörende Stufe III-Bedingung ab.*

5 Datenverteilung

Wie in Abschnitt 3.2 schon beschrieben, gehen wir davon aus, daß für jedes Array eines *Projekts* eine statische Datenallokation (in Form einer *Modularabbildung*) vorliegt. Dennoch wollen wir in Abschnitt 5.1 auch kurz auf dynamische Datenallokation eingehen. In Abschnitt 5.2 wird beschrieben, wie für jedes Array eine Datenallokation ermittelt werden kann. Abschnitt 5.3 befaßt sich mit den Problemen, die sich bei einer *Modularabbildung* als Datenallokation ergeben können, und Abschnitt 5.4 behandelt die Kommunikation mit dem Hostrechner (die nötig ist, um die Daten, die bearbeitet werden sollen, auf den Parallelrechnerprozessoren sichtbar zu machen) als Sonderfall der Reallokation. In 5.5 wird die Implementation in *LooPo* beschrieben.

5.1 Dynamische und statische Datenallokation

Um gemeinsamen Speicher auf verteiltem Speicher zu simulieren, muß bei dem Zugriff auf ein nicht lokal vorhandenes Datum Kommunikation eingesetzt werden. Diese kann hardware- oder softwaregestützt (z.B. durch Laufzeitumgebungen wie `SAM` [SL94]) implizit stattfinden oder durch direkt in den Code eingesetzte explizite Kommunikationsanweisungen realisiert werden. In jedem Fall muß zur Laufzeit bekannt sein, wo sich das Datum, das zu einer Berechnung gebraucht wird, befindet (bzw. eine aktuelle Kopie des Datums). Zu jedem Zeitpunkt kann die aktuelle Verteilung eines Arrays durch eine Datenallokation wie in Definition 3.3 angegeben werden. Derart allgemeine Allokationen sind jedoch in der Handhabung problematisch, wie Beispiel 5.1 zeigt.

Beispiel 5.1 *Betrachten wir folgendes Programmfragment:*

for $t = 0$ to 10
  par $p = 0$ to 10
    if $t < 5$ then
S_1 :
      $A[p,t] := 42$;
S_2 :
    $A[p-1, t-1] := A[p-1, t-1] + 2$;

Hier könnte Array A grundsätzlich aufgrund der Owner Computes Rule (OCR) verteilt werden. Dabei treten aber folgende Probleme auf:

- *Da nicht alle Elemente berechnet werden, kann durch bloße Anwendung der OCR mangels Totalität keine Datenallokation ermittelt werden – dennoch werden auch Elemente gelesen, für die nicht aufgrund der OCR eine Allokation ermittelt werden kann.*
- *Eine exakte Allokationsfunktion kann für $t \geq 5$ lediglich als abschnittsweise affine Funktion angegeben werden, da $A[10,t]$ zuletzt auf Prozessor 10 geschrieben wird, andere Array-Elemente $A[i,t]$ mit $i < 10$ jedoch auf Prozessor $i+1$. Zusätzlich ändert sich die Datenallokation der Array-Elemente $A[i,j]$ mit $0 \leq i \leq 10, 0 \leq j \leq 4$ von einer Iteration der t-Schleife zur nächsten, auch wenn diese Allokation als abschnittsweise affine Abbildung gegeben ist.*

Wie wir an obigem Beispiel sehen, ist eine so allgemeine dynamische Datenallokation

1. schwer zu berechnen und

2. schlecht geeignet, den *Owner*-Prozessor eines Datums zur Laufzeit zu ermitteln (hier insbesondere für die rechte Seite der Anweisung S_2).

Außerdem resultiert das erste Problem des Beispiels somit auch in der Frage wann und wie die Daten vom Hostrechner geholt werden sollen. Denn die benötigten Array-Elemente müßten – sofern sie noch nicht auf dem Parallelrechner vorhanden sind – erst vom Hostrechner geholt weden, was u.U. eine andere Art der Kommunikation erfordert. Dabei ist es jedoch möglich, daß – wie im Beispiel 5.2 für Anweisung S_2 der Fall – Teile des benötigten Arrays nur auf dem Parallelrechner in einer aktuellen Kopie vorhanden sind, andere Teile jedoch nur auf dem Hostrechner. Zusätzlich müßte Kommunikationsgenerierung bei dieser strikten OCR-Allokation im allgemeinen Fall spekulativ vorgehen, wie folgendes Besipiel zeigt:

Beispiel 5.2 *In dem Programmfragment*

```
par p = 0 to 10
  t := 0;
  while FeelLikeIt(t, p)
S1
    A[t, p] := 42;
    t := t + 1;
S2
  A[p − 1, 6] := A[p − 1, 6] · 9;
```

ist erst zur Laufzeit bekannt, welche Elemente vor Ausführung von S_2 von $p-1$ zu empfangen sind und welche nicht. Und auch das kann nur ermittelt werden, wenn p auch FeelLikeIt$(t, p-1)$ kennt oder ein Laufzeitsystem ein komplexeres Protokoll unterstützt, wonach zuerst $p-1$ nach dem Element „gefragt“ wird und im Falle einer negativen Antwort auf die nächstmögliche Quelle zugegriffen wird.

Aufgrund dieser Probleme bei dynamischer Datenallokation gehen wir davon aus, daß für jedes Array eine – totale – statische Datenallokation vorgegeben ist. Die Daten des Hostrechners können somit alle zu Beginn des Eintritts in ein *Projekt* auf dem Parallelrechner verteilt werden.

5.2 Berechnung einer statischen Datenallokation

In Abschnitt 3.2 wurde festgelegt, daß für jedes Datum eine Datenallokation bekannt sein soll (wobei Indizes und Blobs repliziert gespeichert werden): denn für jedes Datum, das dem Parallelrechner bekannt sein soll, muß auch sein Speicherplatz (in der gesamten mehrere lokale Speicher umfassenden Architektur) bekannt sein. Zwar liefert die Allokationsmethode von Dion und Robert (vgl. [DR96], beschrieben auch in [Mei96]) eine affine Allokation für Arrays, jedoch gilt dies nicht für alle Arrays. Außerdem ergeben Allokationsmethoden wie die von Lamport [Lam74] oder Feautrier [Fea92a], [Fea92b] (beschrieben auch in [Wie95]) gar keine statische Datenallokation. Da wir dennoch **irgendeine** Allokation für solche Arrays brauchen, sollen diese gegebenenfalls wenigstens nach einer naiven Methode verteilt werden. Wir wollen dabei versuchen, möglichst eine Allokation zu finden, die die Owner Computes Rule erfüllt.

Wie Beispiel 5.1 zeigt, ist dies mit einer statischen Datenallokation nicht immer möglich. Betrachten wir jedoch das Programmfragment

```
for t = l_t to u_t
  par p = l_p to u_p
    A[4 · t + 2 · p, 2 · p − 1] := A[4 · t, 2 · p] · 105;
```

so ist es hier tatsächlich möglich, mit einer statischen affinen Datenallokation

$$(i,j) \rightarrow \left\lfloor \tfrac{1}{2} \cdot j \right\rfloor + 1$$

die OCR zu erfüllen (und entsprechend auch mit der nach Abschnitt 3.2 kanonisch daraus erzeugten *Modularabbildung*).

Allgemein kann eine solche der OCR entsprechende Allokationsfunktion dann einfach ermittelt werden, wenn

1. es mindestens einen Schreibzugriff auf das Array gibt,
2. alle Schreibzugriffe mit derselben affinen Indexfunktion φ stattfinden und
3. die Indexfunktion φ invertierbar ist in dem Sinne, daß die Projektion von φ^{-1} auf von Parallelschleifen aufgespannte Dimensionen eine Funktion ist; d.h. alle Schreibzugriffe finden auf demselben Prozessor statt.

Dabei gehen wir davon aus, daß die Funktion φ in homogenen Koordinaten angegeben ist. D.h. Quell- und Zielraum werden um eine Dimension vergrößert, die den Konstantenanteil beschreibt; eine affine Funktion wird so als lineare Funktion eines um eine Dimension größeren Raumes betrachtet (vgl. [Wet95]).

Beispiel 5.3 *Die Indexfunktionen für die Arrayzugriffe in*

```
for t = 0 to 42
  par p1 = 5 to 10
    par p2 = 5 to 10
      A[p1, p2, t] := ... ;
      B[2 · p1 − 1, 4 · p2 − 2] := ... ;
```

werden durch die folgenden Matrizen gegeben:

$$\text{für } A\text{:} \quad \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{für } B\text{:} \quad \begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 4 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei stellt jeweils die i-te Zeile (bis auf die letzte) die Indexfunktion in der i-ten Arraydimension dar (die letzte Zeile korrespondiert zu der letzten Spalte der Matrix, die für die homogene Darstellung hinzugefügt wurde). In jeder Zeile stellt die erste Spalte den Koeffizienten von t in der affinen Indexfunktion dar, die zweite den Koeffizienten von p_1, die dritte den von p_2 und die letzte Spalte die additive Konstante.

Eine der OCR entsprechende Allokation Φ ordnet jedem Wert der Array-Indizes $i = (i_1, \ldots, i_n)$ Werte $p = (p_1, \ldots, p_m)$ für Parallelschleifenindizes in folgender Weise zu: Die Indexfunktion φ – im Quellraum eingeschränkt auf parallele Indizes (sequentielle werden als Parameter angesehen) – muß den für ein bestimmtes i ermittelten p-Wert wieder auf i abbilden. D.h. prinzipiell sollte

$$\varphi \circ \Phi = \mathrm{id}$$

gelten – dies läßt sich jedoch nicht immer garantieren, da φ nicht invertierbar sein muß. Allerdings müssen auch nicht tatsächlich alle $i \in \mathbb{Z}^n$ so abgebildet werden, sondern nur die, die im Bild von φ liegen, also bei stattfindenden Schreibzugriffen tatsächlich verwendet werden (so muß bei dem konstanten Schreibzugriff $C[0] := \ldots;$ lediglich das Element 0 derart abgebildet werden). Es muß also für alle $i \in \mathrm{Bi}(\varphi)$ gelten:

$$\varphi \circ \Phi(i) = i$$

Oder anders ausgedrückt: wenn wir φ auf parallele Indizes einschränken (und sequentielle Indizes als Parameter ansehen) gilt

$$\varphi \circ \Phi \circ \varphi(p_1, \ldots, p_m) = \varphi(p_1, \ldots, p_m)$$

Wird φ durch die Matrix A gegeben, Φ durch die Matrix X, so ist die gesuchte Eigenschaft der Matrix X also, daß

$$A \cdot X \cdot A = A$$

Dies spezifiziert für X genau die Bedingung, die von der in Abschnitt 2.2.2 berechneten verallgemeinerten Inversen A^r erfüllt wird. Eine Allokation mit den gewünschten Eigenschaften wird also durch $\pi(A^r)$ gegeben, wobei π auf von Parallelschleifen aufgespannte Dimensionen projiziert (denn die Allokation muß das Prozessorfeld als Zielraum haben). In obigem Beispiel läßt sich für das Array A die affine Indexfunktion $(t, p_1, p_2) \rightarrow (p_1, p_2, t)$ trivialerweise invertieren zu $i = (i_1, i_2, i_3) \rightarrow (i_3, i_1, i_2)$.

Allerdings kann es auch sein, daß die Indexfunktion φ nicht allgemein invertierbar ist. Das in Abschnitt 2.2.2 verwendete Verfahren zur Bestimmung einer Inversen liefert dabei die notwendigen (und hinreichenden) Bedingungen dafür, daß ein Punkt im Bild der Indexfunktion liegt (also geschrieben wird). Da es auch Array-Elemente geben kann, die zwar gebraucht, aber nicht geschrieben werden, setzen wir die Allokationsfunktion einfach fort, indem wir A^r als affinen Teil der Allokationsfunktion für das betrachtete Array verwenden, die ermittelten Bedingungen jedoch ignorieren. Da für jeden Bildpunkt der Indexfunktion durch A^r ein gültiger Urbildpunkt bestimmt wird, beeinträchtigt dies nicht die OCR: gibt es keinen gültigen Urbildpunkt, so liegt das betreffende Array-Element nicht im Bild der Indexfunktion und wird somit auch nicht geschrieben, hat also nach der OCR keinen eindeutig bestimmbaren Besitzer. Es ist zudem möglich, daß während des Auflösens eines linearen Gleichungssystems eine Prozessorkoordinate frei gewählt werden kann, auf die die Funktion abbilden soll. Bei A^r sind solche Koordinaten in der Abbildung grundsätzlich auf 0 gesetzt. In 3.2 hatten wir festgelegt, daß Berechnungen, die nicht von der vollen Anzahl an Parallelschleifen umgeben sind, von den Prozessoren ausgeführt werden sollen, die in den entsprechenden höheren Dimensionen die Koordinate 0 haben. Kommen in der Indexfunktion also alle Indizes von umgebenden Parallelschleifen vor, werden alle Array-Elemente, auf die (evtl.) geschrieben wird, auf den tatsächlich schreibenden Prozessor gelegt.

Beispiel 5.4 *Betrachten wir weiter das Programmstück aus Beispiel 5.3:*

```
for t = 0 to 42
  par p1 = 5 to 10
    par p2 = 5 to 10
      A[p1, p2, t] := ... ;
      B[2 · p1 − 1, 4 · p2 − 2] := ... ;
```

Wir wollen eine Allokation für das Array B bestimmen. Die Indexfunktion φ ist wie in Beispiel 5.3 gegeben durch die Matrix

$$\begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 4 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Mit $Z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ -2 & 1 & 0 \end{pmatrix}$ *kann diese Matrix auf die Zeilenstufenform*

$$\begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

gebracht werden. Diese Matrix können wir nun für jede Spalte von Z nach einem Vektor x_i auflösen, dessen Projektion auf parallele Indizes die Allokationsfunktion für die i-te Arraydimension beschreibt. Die linearen Gleichungssysteme sind

$$\begin{aligned}
\begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot x_1 &= \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} \\
\begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot x_2 &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \\
\begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot x_3 &= \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}
\end{aligned}$$

Das erste Gleichungssystem macht dabei eine Aussage über die Verteilung in der ersten Arraydimension, das zweite über die zweite Arraydimension und das dritte über den jeweils zu addierenden Offset (da es sich auf den durch homogene Koordinaten hinzugefügten Konstantenanteil bezieht). Damit ist durch die jeweils letzte Zeile der Gleichungssysteme eine Bedingung für $B[i, j, k]$ gegeben, die besagt, daß nur Elemente mit $-2 \cdot i + 1 \cdot j = 0$ geschrieben werden. Die zweite Prozessorkoordinate ist dafür immer beliebig. Wir setzen sie auf 0. Die somit eindeutig bestimmten Lösungen für x_i sind:

$$x_1 = \begin{pmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 0 \end{pmatrix} \quad x_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad x_3 = \begin{pmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 1 \end{pmatrix}$$

Die Projektion auf von Parallelschleifen aufgespannte Dimensionen entspricht dem Herausstreichen der Zeilen aus diesen Lösungen, die für nichtparallele Indizes, Blobs oder den Konstantenanteil stehen. Das sind die erste und die letzte Zeile (diese stehen für den t-Wert bzw. den Konstantenanteil). Die Projektion ergibt also:

$$x_1 = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix} \quad x_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad x_3 = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}$$

Der affine Allokationsanteil Φ_{aff} ist damit gegeben durch die Matrix

$$\begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 \end{pmatrix} \text{ also } \Phi_{aff} : (i, j) \rightarrow (\frac{1}{2} \cdot i + \frac{1}{2}, 0)$$

und die gemäß Abschnitt 3.2 erzeugte Modularabbildung *ist*

$$\Phi : (i, j) \rightarrow (\lfloor \tfrac{1}{2} \cdot i + \tfrac{1}{2} \rfloor, 0)\%(10, 10)$$

Damit wird die zweite Arraydimension komplett auf die Prozessoren abgebildet, deren zweite Koordinate im Prozessorfeld 0 ist und die deshalb die Zuweisung ausführen werden. Somit ist jeder Schreibzugriff lokal.

Da unsere Eingabe jedoch allgemeiner sein kann als obige Restriktionen zur einfachen Erstellung einer Allokation, wir aber eine Datenallokation für jedes Array brauchen, müssen noch die folgenden Probleme beachtet werden:

- φ ist nichtaffin: In diesem Fall setzen wir willkürlich etwaige nichtaffine Anteile in der Indexfunktion auf 0, um wieder eine affine (wenn auch im allgemeinen modifizierte) Indexfunktion zu erhalten.
- Es gibt mehrere Schreibzugriffe auf ein Array: In diesem Fall verwenden wir willkürlich den ersten Schreibzugriff, der gleichzeitig ein Lesezugriff ist (dies ist nur als Argument eines Prozeduraufrufs möglich) oder – falls es keinen solchen Zugriff gibt – den ersten Schreibzugriff im Programmtext.
- Es gibt keinen Schreibzugriff: In diesem Fall verwenden wir den ersten Lesezugriff im Programmtext.
- Es gibt weder Lese- noch Schreibzugriffe: In diesem Fall ist es zwar fraglich, warum das Array deklariert wurde, wir alloziieren es jedoch einfach auf Prozessor $(0,\ldots,0)$, um für jedes deklarierte Array eine Allokation gemäß Definition 3.3 zu haben.

Auf diese Weise kann für jedes noch nicht mit einer Allokationsfunktion versehene Array – nach Anwendung der unteren Gaußklammer und Hinzufügen der Grenzen des Prozessorfeldes als Modulo-Vektor – eine *Modularabbildung* als Allokation ermittelt werden. Wir gehen von nun an also davon aus, daß jedes Array durch eine *Modularabbildung* alloziiert wird.

5.3 Darstellung einer *Modularabbildung* als Allokation

Wir können nun also davon ausgehen, daß für jedes Array eine Datenallokation in Form einer *Modularabbildung* Φ gegeben ist. Um jedoch festzustellen, welche Elemente eines Arrays auf einem bestimmten Prozessor gespeichert werden sollen, und auf diesem Prozessor entsprechend Platz für den lokal zu speichernden Teil des Arrays zu schaffen, müssen wir die inverse Funktion von Φ anwenden. Da es sich bei Φ nicht um eine bijektive lineare Abbildung handelt, ist dies nicht einfach durch Invertieren möglich. Außerdem wollen wir nicht auf jedem Prozessor Platz für das gesamte Array anlegen. Darum brauchen wir eine Darstellung der Allokationsfunktion (bzw. ihrer – mengenwertigen – Inversen), die geeignet ist, für jeden Prozessor ein Array zu bestimmen, das den lokalen Anteil des gesamten Arrays enthält.

Die Allokationsfunktion $\Phi : \mathbb{Z}^n \to \mathbb{N}^m$ kann angesehen werden als ein m-Tupel von Allokationsfunktionen $\varphi_j : \mathbb{Z}^n \to \mathbb{N}$, von denen jede die Allokation des Arrays in der entsprechenden Prozessordimension bestimmt. Im allgemeinen ist der Quellraum dieser Abbildungen mehrdimensional. Ist das Bild einer solchen Abbildung konstant, bedeutet dies, daß das Array in der entsprechenden Dimension des Prozessorfeldes nicht verteilt ist: Nur Prozessoren mit der entsprechenden Koordinate in dieser Raumdimension speichern Elemente des Arrays. Ist die Funktion nicht konstant, ist das Array in dieser Dimension verteilt, d.h. Prozessoren mit unterschiedlichen Koordinaten in dieser Dimension speichern unterschiedliche Teile des Arrays. Sind die φ_j eindimensionale Funktionen $\mathbb{Z} \to \mathbb{N}$, deren Quellraum aus jeweils unterschiedlichen Arraydimensionen besteht, kann man die auf einem Prozessor zu speichernde Datenmenge ermitteln als das kartesische Produkt der Urbilder der einzelnen Prozessorkoordinaten unter der jeweiligen Allokationsfunktion, wobei nichtverteilte Arraydimensionen – also solche, die nicht Quellraum für ein φ_j sind – in ihrer gesamten Ausdehnung an dem Produkt teilnehmen.

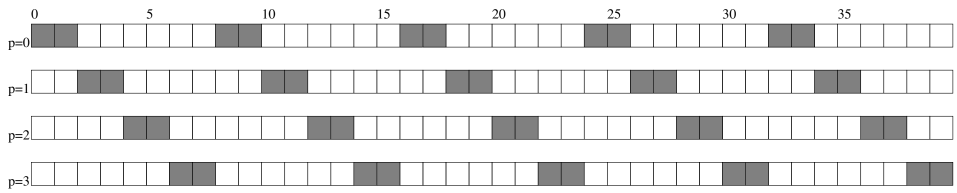


Abbildung 7: Blockzyklische Verteilung nach $i \rightarrow \lfloor \frac{1}{2} \cdot i \rfloor \% 4$

In den folgenden Abschnitten beschäftigen wir uns zunächst mit verteilten Arraydimensionen. In Abschnitt 5.3.3 gehen wir dann darauf ein, wie wir eine allgemeine *Modularabbildung* in eine Form bringen können, die es uns ermöglicht, sie mit unseren Methoden insgesamt korrekt zu interpretieren. Unser Ziel ist dabei immer eine einheitliche Beschreibung, welche Daten aufgrund der gegebenen Allokation auf welchem Prozessor zu speichern sind.

5.3.1 Darstellung einer blockzyklischen Allokation

Gehen wir zunächst von einer einfachen blockzyklischen Verteilung des Arrays aus. Eine solche Verteilung wird gegeben durch eine Funktion

$$\Phi : i = (i_1, \ldots, i_n) \rightarrow (\phi_1(i_{\sigma(1)}), \ldots, \phi_n(i_{\sigma(n)}))$$

wobei σ eine Permutation ist und in jeder Arraydimension j die Allokationsfunktion für diese Arraydimension gegeben wird durch

$$\phi_j : x \rightarrow \left\lfloor \frac{c_j \cdot x + k_j}{d_j} \right\rfloor \% p_j$$

Dabei ist $c_j, k_j \in \mathbb{Z}$ und $p_j, d_j \in \mathbb{N}$. Außerdem gilt für die blockzyklische Allokation $c_j = 1$. Im folgenden werden wir **immer** $d_j > 0$ **annehmen**. Ist eine Funktion obiger Form mit $d_j < 0$ gegeben, kann diese leicht auf die Form $d_j > 0$ gebracht werden. Außerdem betrachten wir nun immer nur eine Dimension des Arrays bzw. des Prozessorfeldes. Wenn also von einem Prozessor Nummer x_j die Rede ist, so gilt die Aussage für alle Prozessoren, deren Koordinate in der betrachteten Dimension x_j ist.

Bild 7 zeigt eine blockzyklische Verteilung entlang einer Arraydimension mit der Blockgröße 2 und 4 Prozessoren (also Modulo-Faktor 4). In [RB95] wird für eine solche Allokation mit der *PITFALLS*-Repräsentation eine einfache Darstellungsform vorgestellt. Diese basiert auf den zusammenhängenden Bereichen des Arrays, die auf demselben Prozessor gespeichert werden sollen, und stellt einen einfachen affinen Zusammenhang her zwischen der Prozessornummer und dem ersten auf dem Prozessor zu speichernden Array-Element:

Definition 5.5 *Ein* **Line Segment** *ist ein zusammenhängender Bereich* $\{l, \ldots, r\}$ *von Elementen einer Arraydimension. Notation:* (l, r).

Eine **FAmiLy of Line Segments** *(***FALLS***), gegeben durch ein ganzzahliges Tupel* (l, r, s, n)*, ist eine Menge von* Line Segments

$$\{(l + s \cdot i, r + s \cdot i) \mid i \in \{0, \ldots, n-1\}\}$$

Die Länge einer FALLS *ist* $r - l + 1$.

Zusammenhängende Array-Elemente, die auf demselben Prozessor gespeichert werdden sollen, lassen sich also durch ein *Line Segment* beschreiben; mehrere *Line*

Segments gleicher Länge mit konstantem Abstand zueinander ergeben dementsprechend eine *FALLS*. Abbildung 7 zeigt auch, daß tatsächlich u.U. ganze *FALLS* auf dem gleichen Prozessor zu speichern sind. Dies liegt an der zyklischen Verteilung, die durch die MOD-Operation in der Allokationsfunktion ausgedrückt wird. Jeder Prozessor x_j hat dabei die Elemente aufzunehmen, die aufgrund des affinen Anteils der Allokationsfunktion (mit Gaußklammer) auf einen der Prozessoren $x_j + \mathbb{Z} \cdot p$ (bei p Prozessoren) abgebildet werden. Sei nun $\phi_{j,\%} : i \to \left\lfloor \frac{c_j \cdot i + k_j}{d_j} \right\rfloor$ die Allokationsfunktion ohne Modulo-Operator. Dann ist bzgl. dieser Funktion die Menge der auf einem Prozessor x_j zu speichernden Array-Elemente nach Satz 2.14:

$$\begin{aligned}\phi^{-1}_{j,\%}(x_j) &= \begin{cases}\{\left\lfloor \frac{d_j \cdot x_j - k_j - \mathrm{sgn}(c_j)}{c_j} \right\rfloor + 1, \ldots, \left\lfloor \frac{(x_j+1) \cdot d_j - k_j - \mathrm{sgn}(c_j)}{c_j} \right\rfloor\} & \text{falls } \frac{c_j}{d_j} > 0 \\ \{\left\lfloor \frac{(x_j+1) \cdot d_j - k_j}{c_j} \right\rfloor + 1, \ldots, \left\lfloor \frac{x_j \cdot d_j - k_j}{c_j} \right\rfloor\} & \text{sonst}\end{cases} \\ &= \{d_j \cdot x_j - k_j, \ldots, d_j \cdot (x_j + 1) - k_j - 1\}\end{aligned}$$

Diese Menge beschreibt ein *Line Segment* (l_{x_j}, r_{x_j}), dessen Länge damit als d_j gegeben ist.

Ein weiteres *Line Segment* (L, R) wird durch ϕ_j auf denselben Wert (bzw. Prozessor) abgebildet, falls $L = l_{x_j} + p \cdot i$ mit $i \in \mathbb{Z}$. Die Schrittweite zwischen zwei *Line Segments* eines *FALLS* ist für eine blockzyklische Allokation abhängig davon, welches auf ein *Line Segment* (l_0, r_0) folgende Array-Element durch $x_j \to \left\lfloor \frac{c_j \cdot x_j + k_j}{d_j} \right\rfloor$ als erstes wieder auf ein $\phi_j(l_0) + i \cdot p$ abgebildet wird. Nach Satz 2.16 ist dies

$$s = \frac{\mathrm{kgV}(p, c^*)}{|c^*|} \cdot d^* = p \cdot d^*$$

Die Anzahl n der zu speichernden *Line Segments* (also die Anzahl der Prozessoren $x_j + i \cdot s$ mit $i \in \mathbb{Z}$, die in $\mathrm{Bi}(\phi_j)$ liegen) berechnet sich aus der Untergrenze l und der Obergrenze u des Arrays durch $\left\lceil \frac{u-l}{s} \right\rceil$. d hat bei einer blockzyklischen Verteilung immer den Wert der Blockgröße, also d_j.

In diesem und den folgenden Abschnitten gehen wir davon aus, daß das jeweilige Array in jeder Dimension „passend" deklariert ist, d.h. daß die Untergrenze immer auf den Beginn eines *Line Segments* fällt und die Obergrenze immer auf das Ende des letzten *Line Segments* desselben *FALLS*. Ist dies im tatsächlichen Programm nicht der Fall, kann diese Eigenschaft durch entsprechendes Vergrößern des Arrays garantiert werden.

Die Arraysektionen, die bei einer blockzyklischen Verteilung auf einem Prozessor gespeichert werden sollen, lassen sich also durch eine *FALLS* beschreiben. Die *FALLS*, die die Verteilung auf einem bestimmten Prozessor beschreibt, läßt sich durch einen affinen Zusammenhang aus der Prozessorkoordinate ermitteln:

Definition 5.6 *Eine* **Processor Index Tagged FAmiLy of Line Segments** *(***PITFALLS***)* **im engeren Sinne** *ist gegeben durch ein ganzzahliges Tupel* (l, r, s, n, d_p, p) *und definiert für jeden Prozessor* $x_j \in \{0, \ldots, p-1\}$ *die* x_j*-te* FALLS

$$F^{x_j} = (l + d_p \cdot x_j, r + d_p \cdot x_j, n, s)$$

Somit kann jede blockzyklische Allokation durch eine *PITFALLS* ausgedrückt werden, die für jeden Prozessor eine *FALLS* definiert, welche die Array-Elemente beschreibt, die auf dem Prozessor zu speichern sind.

5.3.2 Verallgemeinerung der blockzyklischen Allokation

***FALLS* mit konstanter Länge** Mit der *PITFALLS*-Repräsentation aus [RB95] können wir also blockzyklische Allokationen darstellen, jedoch noch keine allgemeinen *Modularabbildungen*. Wir wollen jetzt die Einschränkung aufheben, daß der

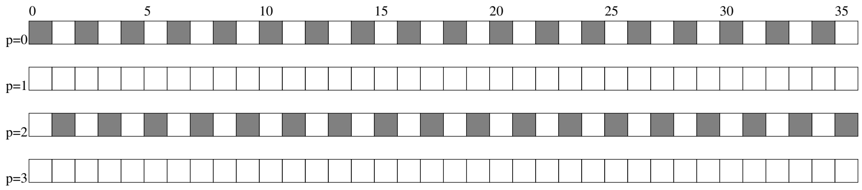


Abbildung 8: Verteilung nach $i \to \lfloor 2 \cdot i \rfloor \% 4$

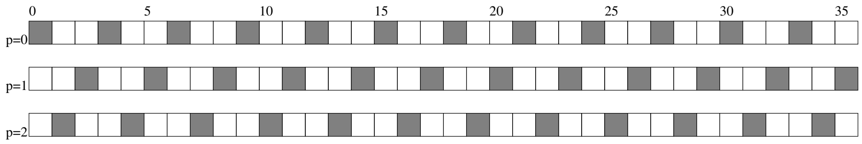


Abbildung 9: Verteilung nach $i \to \lfloor 2 \cdot i \rfloor \% 3$

lineare Koeffizient 1 sein muß. Wir lassen also ein Tupel von Funktionen ϕ_j zu, dessen Komponenten die Form

$$\phi_j : i \to \left\lfloor \frac{c_j \cdot i + k_j}{d_j} \right\rfloor \% p_j = \left\lfloor \frac{c_j^* \cdot i}{d_j^*} + \frac{k_j}{d_j} \right\rfloor \% p_j$$

haben, wobei $c_j, k_j \in \mathbb{Z}$ und $d_j, p_j \in \mathbb{N}$ ist und $c^* \in \mathbb{Z}$, $d^* \in \mathbb{N}$ gegeneinander gekürzt sind. Da nun $c_j \neq 1$ zugelassen ist, kann es Prozessoren geben, auf denen kein einziges Element gespeichert werden soll, wie in Abbildung 8 am Beispiel $i \to \lfloor 2 \cdot i \rfloor \% 4$ zu sehen ist. Dies kann durchaus abhängig von der Anzahl der Prozessoren sein, wie Abbildung 9 zeigt. Diese Verteilungen lassen sich allerdings immer noch durch jeweils eine *FALLS* für einen Prozessor ausdrücken; es kommen lediglich Prozessoren hinzu, für die die *FALLS* leer ist. Wir können die Verteilung also durch eine modifizierte *PITFALLS* beschreiben:

Definition 5.7 *Eine* **(verallgemeinerte) Processor Index Tagged FAmiLy of Line Segments** *(***PITFALLS***) ist gegeben durch ein ganzzahliges Tupel* $(l, r, s, n, d_p, p, p_s, p_f)$ *und definiert für jeden Prozessor* $x_j \in \{0, \ldots, p-1\}$ *eine* FALLS*:*

$$\begin{aligned} F^{\rho}_{x_j} &= \begin{cases} F^{\rho(x_j)} & \textit{falls } \rho(x_j) < p \\ \emptyset & \textit{sonst} \end{cases} \\ &= \begin{cases} (l + d_p \cdot \rho(x_j), r + d_p \cdot \rho(x_j), n, s) & \textit{falls } \rho(x_j) < p \\ (0, -1, 0, 0) & \textit{sonst} \end{cases} \end{aligned}$$

mit

$$\rho : \mathbb{N} \to \mathbb{Z} : i \to \min(\{h \mid (p_f + p_s \cdot h) \% p = i\} \cup \{p\})$$

In einer *PITFALLS* bestimmt so p_f den ersten Prozessor, auf dem überhaupt Daten des Arrays zu speichern sind. Die Prozessorschrittweite p_s bestimmt, welcher Prozessor dann den nächsten Block der durch die *PITFALLS* beschriebenen Arraysektion speichert; dieser Block beginnt d_p Array-Elemente später als der aktuelle Block. Prozessoren, die keine Array-Elemente speichern, werden so „übersprungen“. Für jeden Prozessor liefert ρ also die Nummer der *FALLS*, die auf diesem Prozessor gespeichert wird (oder p, falls es keine solche gibt – die größtmögliche Nummer für eine *FALLS* ist $p-1$). Dabei ist immer darauf zu achten, daß die Prozessoranzahl

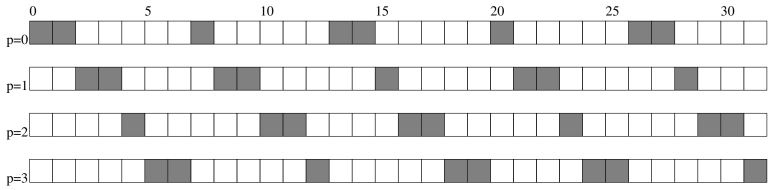


Abbildung 10: Verteilung nach $i \to \lfloor \frac{5 \cdot i}{8} \rfloor \% 4$

nicht überschritten wird (daher der Modulo-Operator in ρ). Eine verallgemeinerte *PITFALLS* ordnet so jedem Prozessor die zugehörige *FALLS* zu, wenn alle die Verteilung beschreibenden *Line Segments* gleich lang sind.

***FALLS* mit nichtkonstanter Länge** Eine allgemeine eindimensionale *Modularabbildung* $x \to \lfloor \frac{c \cdot x + k}{d} \rfloor \% p$ kann jedoch auch unregelmäßigere Verteilungen beschreiben. Abbildung 10 gibt hierfür ein Beispiel. Die Verteilung kann hier nicht durch eine einzige *PITFALLS* beschrieben werden, da die zusammenhängenden Bereiche, die auf einem Prozessor gespeichert werden nicht alle gleich lang sind. Dennoch handelt es sich auch hierbei um eine regelmäßige Verteilung. In dem Beispiel der Abbildung 10 kann die gesamte Verteilung anhand von 5 (verallgemeinerten) *PITFALLS* beschrieben werden:

1. $(0, 1, 32, 1, 2, 4, 5, 0)$
2. $(2, 3, 32, 1, 2, 4, 5, 1)$
3. $(4, 4, 32, 1, 2, 4, 5, 2)$
4. $(5, 6, 32, 1, 2, 4, 5, 3)$
5. $(7, 7, 32, 1, 2, 4, 5, 0)$

Jede *PITFALLS* kann als Familie von *Line Segments* nur eine Vereinigung von Arraysektionen darstellen, die konstanten Abstand s voneinander haben und gleich lang sind ($r - l + 1$ Elemente). Allerdings unterliegen diese Arraysektionen in ihrer Länge und Verteilung wiederum einem regelmäßigen Schema:

- Gehört ein Array-Element l zu einem Urbild $F_j^{-1}(x_j)$ (mit $F_j(i) = \left\lfloor \frac{c^* \cdot i}{d^*} + \frac{k}{d} \right\rfloor$), so gehören auch alle Elemente $l + d_j^* \cdot \mathbb{Z}$ zu Urbildern $F_j^{-1}(y)$ mit $\#\left(F_j^{-1}(x_j)\right) = \#\left(F_j^{-1}(y)\right)$; d.h. alle Urbilder von Array-Elementen $l + d_j^* \cdot \mathbb{Z}$ unter F_j gehören zu *Line Segments* gleicher Länge.

- Nach Satz 2.16 beträgt der kürzeste Abstand zwischen den ersten Elementen zweier *Line Segments*, die wegen obiger Beobachtung gleich lang sein müssen und die durch ϕ_j auf den gleichen Wert abgebildet werden, konstant
$$s = \frac{\mathrm{kgV}(p_j, c_j^*)}{|c_j^*|} \cdot d_j^*$$
Elemente.

- Da die *Line Segments*, von denen wir wissen, daß sie gleich lang sind, im Abstand von d_j^* Array-Elementen liegen (gemessen vom ersten Element eines *Line Segments* zum ersten Element des nächsten *Line Segments*), gibt es nach Satz 2.17 in diesem Zyklus von d_j^* Array-Elementen genau
$$\min(|c_j^*|, d_j^*)$$

möglicherweise unterschiedlich lange *Line Segments* (die durch F_j auf unterschiedliche Werte abgebildet werden).

Die Verteilung aufgrund jeder der Familien von potentiell unterschiedlich langen *Line Segments* läßt sich nach den folgenden Formeln mit einer (verallgemeinerten) *PITFALLS* beschreiben (l_A, u_A sind die Unter- bzw. Obergrenze des Arrays in der betrachteten Dimension, q ist die Nummer der *PITFALLS*, wobei 0 für die *PITFALLS* steht, die Array-Element 0 enthält):

$$l = \begin{cases} \left.\begin{cases} \left\lfloor \frac{d_j \cdot q - (k_j \% d_j) - 1}{c_j} \right\rfloor + 1 & \text{falls } c_j \geq 0 \\ \left\lfloor \frac{d_j \cdot (q+1) - (k_j \% d_j)}{c_j} \right\rfloor + 1 & \text{falls } c_j < 0 \end{cases}\right\} & \text{falls } |c_j| < d_j \\ q & \text{falls } |c_j| \geq d_j \end{cases} \tag{14}$$

$$r = \begin{cases} \left.\begin{cases} \left\lfloor \frac{d_j \cdot (q+1) - k_j \% d_j - 1}{c_j} \right\rfloor & \text{falls } c_j \geq 0 \\ \left\lfloor \frac{d_j \cdot q - k \% d_j}{c_j} \right\rfloor & \text{falls } c_j < 0 \end{cases}\right\} & \text{falls } |c_j| < d_j \\ q & \text{falls } |c_j| \geq d_j \end{cases} \tag{15}$$

$$s = \frac{\mathrm{kgV}(p, c^*)}{|c^*|} \cdot d^* \tag{16}$$

$$n = \left\lceil \frac{u_A - l_A + 1}{s} \right\rceil = \left\lfloor \frac{u_A - l_A}{s} \right\rfloor + 1 \tag{17}$$

$$d_p = d^* \tag{18}$$

$$p_s = c^* \tag{19}$$

$$p_f = \left\lfloor \frac{c \cdot l + k}{d} \right\rfloor \% p_n \tag{20}$$

Das Urbild einer eindimensionalen *Modularabbildung* läßt sich also durch eine Menge von *PITFALLS* beschreiben:

Definition 5.8 *Eine* **Set of Processor Index Tagged FamiLies of Line Segments** *(***SPITFALLS***) ist ein ganzzahliges Tupel* $(c, k, d, d^*, n, p_n, P_f, P_n, s, A)$, *welches die Verteilung eines Arrays in Dimension A beschreibt. Eine* SPITFALLS *definiert die* PITFALLS *Nummer* P_f *bis* $P_f + P_n - 1$ *nach den Gleichungen (14) bis (20), wobei* $c^* = \frac{c \cdot d^*}{d}$ *und* PITFALLS *Nummer* 0 *die* PITFALLS *ist, die das Array-Element Nummer* 0 *enthält.*

Ist $c = 0$, *so handelt es sich um eine konditionale* SPITFALLS, *die besagt, daß nur solche Prozessoren Daten zu speichern haben, deren der* SPITFALLS *entsprechende Koordinate* $\frac{k}{d} \% p_n$ *beträgt.*

Die Urbilder einer eindimensionalen *Modularabbildung* $\phi_j : i \rightarrow \left\lfloor \frac{c \cdot i + k}{d} \right\rfloor \% p_n = \left\lfloor \frac{c^*}{d^*} + \frac{k}{d} \right\rfloor$ (c^*, d^* gegeneinander gekürzt) lassen sich also komplett durch eine *SPITFALLS* beschreiben mit $P_f = \left\lfloor \frac{l_A \cdot \min(|c^*|, d^*)}{d^*} \right\rfloor$ und $P_n = \min(|c^*|, d^*)$. Denn mit diesen Werten ist P_f die *PITFALLS*, in der l liegt, und nach Satz 2.17 kann es P_n unterschiedliche Längen von *Line Segments* geben, so daß die Beschreibung durch P_n verschiedene *PITFALLS* vollständig ist.

Organisation des lokalen Speichers Kann die Allokation eines Arrays in jeder Dimension durch eine *SPITFALLS* beschrieben werden, so kann für jede Arraydimension Speicher so alloziiert werden, daß lediglich für die Elemente Speicher belegt wird, die durch die Datenallokation auch auf diesen Prozessor abgebildet werden.

Auf einem Prozessor werden allgemein zusammenhängende *Line Segments* des gesamten Arrays gespeichert. Es liegt also nahe, diese Bereiche weiterhin in zusammenhängenden Speicherbereichen zu halten. Eine *FALLS* kann dann aus mehreren

Line Segments zusammengesetzt werden, indem man jedem *Line Segment* eines *FALLS* eine Nummer zwischen 0 und $n-1$ zuordnet und so für diese Aufteilung eine weitere Arraydimension einführt. Schließlich kann man in einer dritten Dimension die Nummern der möglichen *PITFALLS* durchzählen (P_f bis $P_f + P_n - 1$). Auf diese Art läßt sich exakt der lokal zu speichernde Anteil einer Dimension des Arrays auf einem Prozessor in einem dreidimensionalen lokalen Array halten. Beginnen alle Arraydimensionen mit dem Index 0, so ergeben sich die lokalen Indizes für einen globalen Index i zu:

1. $i - \begin{cases} \left\lceil \dfrac{d \cdot \left\lfloor \frac{i \cdot \min(c^*, d^*)}{d^*} \right\rfloor - k\%d}{c} \right\rceil & \text{falls } c > 0 \\ \left\lceil \dfrac{d \cdot (\left\lfloor \frac{i \cdot \min(|c^*|, d^*)}{d^*} \right\rfloor + 1) - k\%d}{c} \right\rceil + 1 & \text{falls } c \leq 0 \end{cases}$
2. $\left\lfloor \frac{i - l_A}{s} \right\rfloor$
3. $\left\lfloor \frac{i \cdot \min(|c^*|, d^*)}{d^*} \right\rfloor$

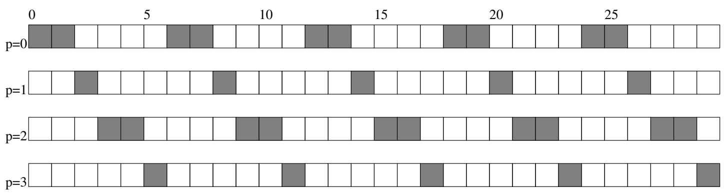
Obwohl diese exakte Berechnung der lokalen Indizes möglich ist, resultiert sie also in relativ aufwendigen Indexausdrücken.

Einfachere Indexausdrücke erhält man, wenn man jede Arraydimension folgendermaßen in 2 Dimensionen für das lokale Array überführt:

1. Die erste Dimension zählt die *Line Segments* von 0 bis $n-1$ durch. Der lokale Index für einen globalen Index i ist in dieser Dimension
$$\left\lfloor \frac{i - l_A}{s} \right\rfloor = (i - l_A)/s$$
2. Die zweite Dimension zählt die Position des globalen Indexes innerhalb der *Line Segments* durch, wobei jedoch immer P_n aufeinanderfolgende *Line Segments* aneinandergereiht sind, die Zählung also nicht am Ende eines *Line Segments* wieder mit 0 beginnt, sondern am Ende jedes P_n-ten *Line Segments*. Der lokale Index für einen globalen Index i ist in dieser Dimension
$$(i - l_A)\%d^*$$

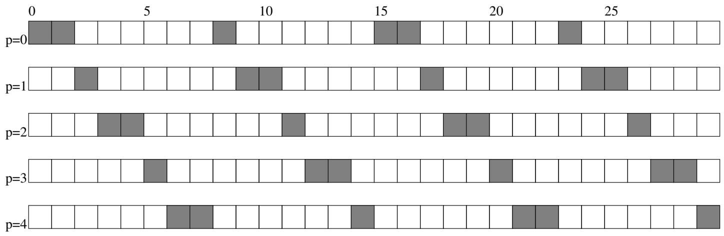
Während die zuerst vorgestellte Strategie jedoch exakt die Menge an Speicher benötigt, die der lokal zu speichernde Anteil des Arrays ausmacht, ist es bei der zweiten Strategie möglich, daß auf einem Prozessor zu viel Speicher belegt wird, wie folgendes Beispiel zeigt:

Beispiel 5.9 *Ein Array* $A[0 \dots 29]$ *sei alloziiert durch* $i \rightarrow \lfloor \frac{2}{3} \cdot i \rfloor \%4$. *Die Verteilung entspricht somit dem Bild:*



Auf den Prozessoren 0 *und* 2 *sind also je* 10 *Elemente zu speichern, auf Prozessor* 1 *und* 3 *hingegen nur je* 5. *Die zweidimensionale Strategie belegt jedoch auf jedem Prozessor Speicher für* $n \cdot d^* = 5 \cdot 3 = 15$ *Array-Elemente.*

Haben wir einen Prozessor mehr zur Verfügung, ist die Allokation also gegeben durch $\Phi : i \rightarrow \lfloor \frac{2}{3} \cdot i \rfloor \%5$, *ergibt sich folgendes Bild:*



In diesem Fall sind tatsächlich auf jedem Prozessor 6 *Elemente zu speichern, wie sie von der 2-dimensionalen Strategie auch angelegt werden.*

Angesichts der Ersparnis bei der Berechnung der Indexausdrücke erscheint dieser mögliche Mehraufwand im Speicherplatz jedoch vertretbar, zumal er nicht grundsätzlich einen Mehraufwand an Speicher nach sich zieht. Auch wenn nicht auf jedem Prozessor die gleiche Anzahl an Daten gespeichert werden müssen, kann bei diesem Ansatz zu großer Speicherverbrauch teilweise vermieden werden, indem für die zweite Dimension auf Prozessor x_j nicht grundsätzlich d^* Elemente angelegt werden, sondern D_{x_j} Elemente, wobei $D_{x_j} = \max(\{1 + (i - l_A)\%d^* \mid i \in \phi_j^{-1}(x_j)\})$.

Beispiel 5.10 *Verwenden wir mit* Φ *wie in Beispiel 5.9 (*$d^* = 3$*,* $l_A = 0$*,* $u_A = 29$*,* $n = 5$*) für die zweite lokale Arraydimension auf Prozessor x jeweils* $\max(\{(i - l_A)\%d^* \mid i \in \Phi^{-1}(x)\})$ *Elemente, dann ergibt sich für das Array aus diesem Beispiel der folgende Platzverbrauch:*

CPU	*Maximalwert* $(i - l_A)\%d^*$	*Speicherplatz*
0	$\max(\{(1-0)\%3\}) = 1$	$2 \cdot 5 = 10$
1	$\max(\{(2-0)\%3\}) = 2$	$3 \cdot 5 = 15$
2	$\max(\{(4-0)\%3\}) = 1$	$2 \cdot 5 = 10$
3	$\max(\{(5-0)\%3\}) = 2$	$3 \cdot 5 = 15$

Eine dritte Möglichkeit wird von `HPF`-Compilern für die Allokation nach *Modularabbildungen* durch Alignment des Arrays auf ein blockzyklisch verteiltes Template verwendet. In [Wol95] wird ein solcher Ansatz beschrieben, der ebenfalls keine optimale Speicherauslastung garantiert, jedoch in einfachen Indexausdrücken resultiert. Ein eindimensionales Array $A[l \ldots h]$, das gemäß der *Modularabbildung*

$$\Phi : i \rightarrow \left\lfloor \frac{c \cdot i + k}{d} \right\rfloor \% p$$

verteilt werden soll, wird in `HPF` durch folgende Deklaration beschrieben:

```
INTEGER::A[l:h]
!HPF$ PROCESSORS P[p]
!HPF$ TEMPLATE T[N]
!HPF$ DISTRIBUTE T[CYCLIC(d)] ONTO P
!HPF$ ALIGN A[i] WITH T[c*i+k]
```

Die Allokation geschieht dabei bzgl. eines **Ursprungs** t, für den prinzipiell beispielsweise der Indexwert des ersten Template-Elements verwendet werden kann, auf das ein Array-Element abgebildet wird; dies ist der Zähler der *Modularabbildung* angewandt auf die Untergrenze des Arrays. Jede Arraydimension $A[l \ldots h]$ wird auch hier in zwei Dimensionen $A[L_F \ldots L_L][O_F \ldots O_L]$ aufgeteilt:

$$L_F = \left\lfloor \frac{c \cdot l + k - t}{p \cdot d} \right\rfloor$$

$$L_L = \left\lfloor \frac{c \cdot h + k - t}{p \cdot d} \right\rfloor$$

$$F_F = 0$$

$$F_L = \left\lceil \frac{d}{c} \right\rceil - 1$$

(Für negative c sind L_F und L_L bzw. F_F und L_L zu vertauschen.) Die lokale Adresse (j_1, j_2) und der *Owner*-Prozessor p_0 für Element i ergibt sich dann aus

$$\begin{aligned} j_1 &= \left\lfloor \frac{c \cdot i + k - t}{p \cdot d} \right\rfloor \\ j_2 &= \left\lfloor \frac{(c \cdot i + k - t)\%d}{c} \right\rfloor \\ p_0 &= \left\lfloor \frac{(c \cdot i + k - t)\%(d \cdot p)}{d} \right\rfloor \end{aligned}$$

Die Untergrenzen der lokalen Arrays können ebenfalls auf 0 normalisiert werden, indem statt j_1 der Index $j_1 - L_1$ und statt j_2 der Index $j_2 - L_2$ verwandt wird mit

$$L_1 = \begin{cases} \left\lfloor \frac{c \cdot l + k - t}{d \cdot p} \right\rfloor & : \text{ falls } a > 0 \\ \left\lfloor \frac{c \cdot h + k - t}{d \cdot p} \right\rfloor & : \text{ falls } a < 0 \end{cases}$$

$$L_2 = \begin{cases} 0 & : \text{ falls } c > 0 \\ \left\lfloor \frac{d-1}{a} \right\rfloor & : \text{ falls } c < 0 \end{cases}$$

In [Wol95] wir der Ursprung t vom Compiler gewählt, was jedoch im allgemeinen dazu führen kann, daß nicht tatsächlich die obige Allokation Φ realisiert wird. Mit $t = 0$ ist dies jedoch gewährleistet.

Diese Methode entspricht der Vergrößerung des Speichers für jedes auf einem Prozessor zu speichernde *Line Segment* auf die maximale Größe, die unter den *Line Segments* erreicht wird. Jedoch wird Speicher nur für tatsächlich auf einem Prozessor alloziierte *Line Segments* belegt.

Beispiel 5.11 *Sei das Array* $A[0 \dots 29]$ *wieder alloziiert wie in Beispiel 5.9. Dann gilt:*

$$\begin{aligned} L_F &= \left\lfloor \frac{2 \cdot 0}{4 \cdot 3} \right\rfloor = 0 \\ L_L &= \left\lfloor \frac{2 \cdot 29}{4 \cdot 3} \right\rfloor = 4 \\ F_F &= 0 \\ F_L &= \left\lceil \frac{3}{2} \right\rceil - 1 = 1 \end{aligned}$$

Auf jedem Prozessor werden also $2 \cdot 5 = 10$ *Elemente angelegt; das sind* 5 *Elemente weniger als bei der zweiten Strategie. Haben wir hingegen wieder* 5 *Prozessoren zur Verfügung, also eine Allokation gemäß* $i \mapsto \left\lfloor \frac{2 \cdot i}{3} \right\rfloor \%5$*, so gilt:*

$$\begin{aligned} L_F &= \left\lfloor \frac{2 \cdot 0}{5 \cdot 3} \right\rfloor = 0 \\ L_L &= \left\lfloor \frac{2 \cdot 29}{5 \cdot 3} \right\rfloor = 3 \\ F_F &= 0 \\ F_L &= \left\lceil \frac{3}{2} \right\rceil - 1 = 1 \end{aligned}$$

Auf jedem Prozessor werden also $2 \cdot 4 = 8$ *Elemente angelegt; das sind* 2 *Elemente mehr als bei der zweiten Strategie.*

5.3.3 Allokation durch mehrdimensionale *Modularabbildungen*

Wir haben nun gesehen, wie für jeden Prozessor beschrieben werden kann, welche Array-Elemente auf diesem Prozessor zu speichern sind, falls die Allokation durch eine Sequenz von eindimensionalen *Modularabbildungen* beschrieben werden kann, und wie dann Speicherplatz für den lokal zu speichernden Anteil des Arrays eingerichtet werden kann. Allerdings wollen wir **allgemeine** *Modularabbildungen* als

Allokation verwenden. Dies sind im allgemeinen mehrdimensionale Abbildungen. In diesem Abschnitt wollen wir also darauf eingehen, wie man mehrdimensionale *Modularabbildungen* durch eine Sequenz von eindimensionalen *Modularabbildungen* ausdrücken kann, für die wir ja eine Lösung haben.

Spaltentransformation Aus der Linearen Algebra wissen wir, daß sich jede lineare Abbildung $\Phi : \mathbb{Q}^n \rightarrow \mathbb{Q}^m$ bzgl. einer bestimmten Basis für den Urbildraum durch eine Matrix in Spaltenstufenform ausdrücken läßt, also durch eine Matrix $A \in \mathbb{Q}^{m \times n}$ der Form:

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & \dots & 0 \\ a_{2,1} & \vdots & \vdots & 0 & \dots & 0 \\ \vdots & a_{i,2} & 0 & 0 & \dots & 0 \\ \vdots & a_{i+1,2} & \ddots & 0 & \dots & 0 \\ \vdots & \vdots & \dots & 0 & \dots & 0 \end{pmatrix}$$

Zu jeder Spalte j und jeder Zeile i mit Eintrag $a_{i,j} \neq 0$ kann eine Spalte $\sigma(i,j)$ bestimmt werden mit $\sigma(i,j)$ minimal in $\mathbb{N}$, so daß $a_{i,\sigma(i,j)} \neq 0$. Subtrahieren wir im Fall $\sigma(i,j) \neq j$ das $\frac{a_{i,j}}{a_{i,\sigma(i,j)}}$-fache der $\sigma(i,j)$-ten Spalte von der j-ten Spalte, erhalten wir eine Matrix $A' = (a'_{i,j})_{i \in \{1,\dots,m\}, j \in \{1,\dots,n\}}$, bei der jeder Beginn einer neuen „Stufe" in der „Spaltenstufenform" der einzige Nichtnull-Eintrag in der entsprechenden Zeile ist. Diese Matrix hat die Form:

$$A' = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & \dots & 0 \\ a_{2,1} & \vdots & \vdots & 0 & \dots & 0 \\ 0 & a_{i,2} & 0 & 0 & \dots & 0 \\ a_{i+1,1} & a_{i+1,2} & \ddots & 0 & \dots & 0 \\ \vdots & \vdots & \dots & 0 & \dots & 0 \end{pmatrix}$$

Es gilt also:

$$\begin{array}{l} (\forall j : 1 \leq j \leq n : i = \min(\{i' \mid a'_{i',j} \neq 0\}) \Rightarrow (\nexists j' : j' \in \{1,\dots,n\} \setminus \{j\} : a'_{i,j'} \neq 0)) \\ \wedge \\ i < j \Rightarrow a'_{i,j} = 0 \end{array} \tag{21}$$

Ist A' vollrangig, bedeutet dies, daß lediglich die Diagonalelemente der Matrix $a_{i,i} \neq 0$ sein können. Dies kann allerdings auch gelten, falls A' nicht vollen Rang hat. Unter Umständen gilt nach der Transformation auch:

$$\begin{array}{l} (\forall j : j \in \{1,\dots,n\} : \#(\{i \mid a'_{i,j} \neq 0\}) \leq 1) \\ \wedge \\ (\forall i : i \in \{1,\dots,m\} : \#(\{j \mid a'_{i,j} \neq 0\}) \leq 1) \end{array} \tag{22}$$

D.h. sowohl in jeder Spalte als auch in jeder Zeile der Matrix gibt es höchstens einen Nichtnull-Eintrag. Da die Matrix in „Spaltenstufenform" ist, sind die Nichtnull-Einträge verschiedener Zeilen dann in unterschiedlichen Spalten. In diesem Fall kann die durch die Matrix beschriebene Abbildung durch m lineare Funktionen ausgedrückt werden. Nehmen wir diese Spaltentransformation R auf der Matrix vor, die den linearen Anteil der Datenallokation für das Array A beschreibt, entspricht

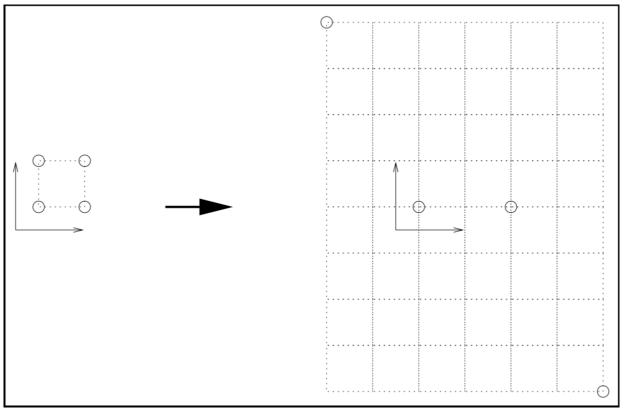


Abbildung 11: Anwendung der Spaltentransformation aus Beispiel 5.12

dies einem Basiswechsel, der einen Datenraum $R^{-1}(Idx(A))$ einführt bgzl. dessen die Allokationsfunktion angegeben wird. Gilt nach der Transformation die Bedingung (22), so läßt sich die Allokation bzgl. der neuen Basis durch *m SPITFALLS* beschreiben.

Damit sich auch die Indexausdrücke im *Projekt* auf die Basis des Datenraums beziehen, ist auf alle Indexausdrücke für dieses Array R^{-1} anzuwenden.

Stauchung der Basistransformation Bei der Transformation R kann es sich aber – wie Abbildung 11 zeigt – um eine allgemeine lineare Abbildung handeln. Das Gitter in Abbildung 11 repräsentiert die ganzzahligen Array-Indizes. Beispiel 5.12 zeigt dabei ein Problem auf: das gesamte Array kann durch Streckung in der linearen Abbildung stark vergrößert werden.

Beispiel 5.12 *Der lineare Anteil einer Datenallokation für ein Array $A[0\ldots2,0\ldots2]$ sei gegeben durch die Matrix*

$$A = \begin{pmatrix} 2 & -1 \\ -4 & 4 \end{pmatrix}$$

Diese Matrix kann in Diagonalform gebracht werden durch:

$$A \cdot R = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$$

mit

$$R = \begin{pmatrix} \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

also

$$R^{-1} = \begin{pmatrix} 4 & -2 \\ -4 & 4 \end{pmatrix}$$

Auftretende Indexausdrücke müssen also durch Multiplikation mit R^{-1} umgeschrieben werden, um dem Basiswechsel Rechnung zu tragen. Insbesondere gilt dies für

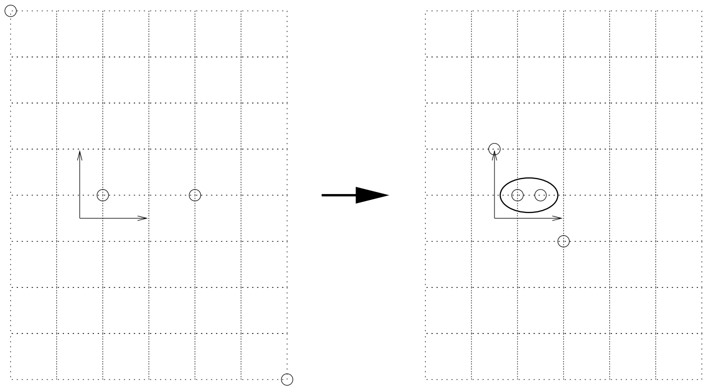


Abbildung 12: Zu starke Stauchung der Spaltentransformation aus Beispiel 5.12

die Arraydeklaration: Die neue Arraydeklaration ergibt sich dadurch, daß R^{-1} auf alle Eckpunkte des Arrays anzuwenden ist, zu $A[-4\dots 8, -8\dots 8]$. Damit wird aber aus dem 9-elementigen Array A ein 221-elementiges (wir gehen hier davon aus, daß weiterhin nur rechteckige Arrays deklariert werden können).

Ist der lineare Teil einer Datenallokation gegeben durch eine Matrix A und die Spaltentransformation durch eine invertierbare Matrix R, so läßt sich die Vergrößerung des Arrays aufgrund der Streckung durch R^{-1} verringern, indem R mit einem Faktor $d_{opt} \in \mathbb{N}$ multipliziert wird, während R^{-1} durch d_{opt} geteilt (die Abbildung also gestaucht) wird. Die erwünschten Eigenschaften von R und R^{-1} gehen dabei nicht verloren:

$$A \cdot R \cdot d_{opt} \quad \text{hat die gleiche Form wie} \quad A \cdot R \tag{23}$$

und

$$\frac{R^{-1}}{d_{opt}} \cdot R \cdot d_{opt} = E \tag{24}$$

Da jedoch ein Array $A[l\dots u]$ immer aus den Elementen $\{A[i] \mid i \in \{l, \dots, u\}\}$ besteht (es also keine Arrays mit rationalen Indizes gibt), bezieht sich ein Arrayzugriff mit rationalem Indexausdruck $A[\frac{i}{q}]$ immer auf das Array-Element $A[i/q]$. Darum muß $\frac{R^{-1}}{d_{opt}}$ von Anfang an noch zwei weitere Bedingungen erfüllen:

- Eine bijektive rationale lineare Abbildung $S_{\mathbb{Q}}$ mit $S_{\mathbb{Q}} \in GL(n, \mathbb{Q})$ kann durchaus zwei Werte $i \neq j \in \mathbb{Z}^n$ auf $S_{\mathbb{Q}}(i)$ und $S_{\mathbb{Q}}(j)$ so abbilden, daß

$$\left\lfloor S_{\mathbb{Q}}(i) \right\rfloor = \left\lfloor S_{\mathbb{Q}}(j) \right\rfloor$$

  In unserem Fall ist $S_{\mathbb{Q}} = \frac{R^{-1}}{d_{opt}}$, das heißt, daß Indexfunktionen, die Zugriffe auf zwei unterschiedliche Array-Elemente beschreiben, durch die Basistransformation so umgeschrieben werden können, daß sie sich wegen der Interpretation der Array-Indizes mit $\lfloor \cdot \rfloor$ nun auf das gleiche Array-Element beziehen. Dies entspricht in Bild 12 der Tatsache, daß zwei Array-Elemente – nämlich die aus $(0,0)$ bzw. $(1,1)$ entstandenen Elemente $(0,0)$ und $(\frac{1}{2}, 0)$ – in das gleiche Feld des ganzzahligen Gitters abgebildet werden. Die Stauchung der Spaltentransformation muß also mit einem so geringen Faktor geschehen, daß immer noch alle Array-Elemente in unterschiedliche Felder des Gitters abgebildet werden.

Die Basistransformation $S_{\mathbb{Q}} = \frac{R^{-1}}{d_{opt}}$ muß also auch die folgende Bedingung erfüllen:

$$(\forall a, b : a \neq b \in \mathbb{Z}^n : \left\lfloor S_{\mathbb{Q}}(a) \right\rfloor \neq \left\lfloor S_{\mathbb{Q}}(b)) \right\rfloor) \tag{25}$$

- Außerdem darf es keinen Arrayzugriff geben, für den durch das Umschreiben und die Interpretation von Indexausdrücken durch $\lfloor \cdot \rfloor$ ein anderer *Owner*-Prozessor bestimmt wird als zuvor. Wie in Beispiel 5.13 zu sehen, ist dies jedoch möglich, falls nichtganzzahlige Indexausdrücke entstehen. Darum ist die folgende Eigenschaft (26) zu erfüllen:

$$(\forall i : i \in \mathbb{Z}^n : A \cdot R \cdot d_{opt} \cdot \left\lfloor \frac{R^{-1}}{d_{opt}} \cdot i \right\rfloor = A \cdot i) \tag{26}$$

Beispiel 5.13 *Gegeben sei die Allokation*

$$A = \begin{pmatrix} 2 & 0 \\ 2 & 4 \end{pmatrix}$$

Diese Matrix kann mit

$$R = \begin{pmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{pmatrix} \quad \textit{und} \quad R^{-1} = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix}$$

umgeformt werden zu

$$A \cdot R = \begin{pmatrix} 2 & 0 \\ 0 & 4 \end{pmatrix}$$

Um die Allokation als $A \cdot R$ ausdrücken zu können, wird ein Arrayzugriff

`A[3,3]`

umgeschrieben zu

`A[`$\lfloor 1 \cdot 3 + 0 \rfloor$`,`$\lfloor \frac{1}{2} \cdot 3 + 3 \rfloor$`]`

also

`A[3,4]`

Gemäß $A \cdot R$ wird dieses Element aber auf

$$\begin{pmatrix} 2 & 0 \\ 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 6 \\ 16 \end{pmatrix}$$

abgebildet, während A für den ursprünglichen Arrayzugriff

$$\begin{pmatrix} 2 & 0 \\ 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 6 \\ 18 \end{pmatrix}$$

bestimmte.

Ist jedoch $\frac{R^{-1}}{d_{opt}} \in \mathbb{Z}^{n \times n}$, besteht also auch der Bruch $\frac{R^{-1}}{d_{opt}}$ nur aus ganzen Zahlen, so sind beide Bedingungen (25) und (26) trivialerweise erfüllt. Wählen wir also $d_{opt} = d_{\mathrm{ggT}}$ mit $\mathbb{Z}^{n \times n} \ni R^{-1} = (r'_{i,j})$ und

$$d_{\mathrm{ggT}} = \mathrm{ggT}(\{r'_{i,j} \mid i, j \in \{1, \ldots, n\}\} \setminus \{0\})$$

so besitzt $\frac{R^{-1}}{d_{opt}}$ alle gewünschten Eigenschaften – R^{-1} kann durch Multiplikation mit dem Hauptnenner und Division von R um denselben Faktor auf die benötigte

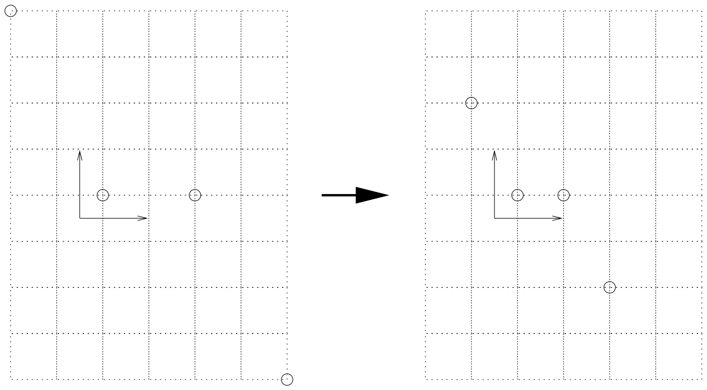


Abbildung 13: Optimierte Stauchung der Basistransformation aus Beispiel 5.12

ganzzahlige Form gebracht werden; wir wollen von hier ab annehmen, daß dies schon geschehen ist. Gerade für $S_{\mathbb{Q}} \in \mathbb{Z}^{n \times n}$ ist aber der Streckungsfaktor, mit dem das Array vergrößert wird, relativ groß.

Wie man leicht sieht, ist die Bedingung (26) für $j \in \mathbb{N}$ mit $e_j \in \mathrm{Ke}(A \cdot R)$ auch dann erfüllt, wenn $\pi_j(\frac{R^{-1}}{d_{opt}})$ nichtganzzahlige Koeffizienten enthält – denn das Ergebnis von $A \cdot R \cdot d_{opt} \cdot i$ wird durch $\pi_j(i)$ nicht beiinflußt, wenn e_j im Kern von $A \cdot R$ liegt. Um Bedingung (26) zu erfüllen genügt es also, wenn wir $d_{opt} = d'_{\mathrm{ggT}}$ wählen mit

$$d'_{\mathrm{ggT}} = \mathrm{ggT}(\{r'_{i,j} \mid i,j \in \{1,\ldots,n\}, e_i \notin \mathrm{Ke}(A \cdot R)\} \setminus \{0\})$$

Da $A \cdot R$ in Spaltenstufenform ist, genügt es hierfür, bei der Berechnung des größten gemeinsamen Teilers nur die ersten $\mathrm{rang}(A \cdot R)$ Zeilen von R^{-1} zu betrachten.

Haben wir auf diese Weise einen Faktor gefunden, besteht jedoch immer noch die Möglichkeit, daß $\frac{R^{-1}}{d_{opt}}$ die Bedingung (25) nicht erfüllt. Wir suchen also einen Stauchungsfaktor d_{inj}, der möglichst groß ist, so daß zwei unterschiedliche Elemente wieder auf unterschiedliche Felder des ganzzahligen Gitters abgebildet werden. Für

$$d_{opt} = \min(d'_{\mathrm{ggT}}, d_{inj})$$

sind dann alle Bedingungen erfüllt, wie in Abbildung 13 dargestellt.

Beispiel 5.14 *In Beispiel 5.12 kann die Matrix*

$$R^{-1} = \begin{pmatrix} 4 & -2 \\ -4 & 4 \end{pmatrix}$$

mit dem Faktor 2 *modifiziert werden zu*

$$(R')^{-1} = \begin{pmatrix} 2 & -1 \\ -2 & 2 \end{pmatrix}$$

entsprechend ist dann

$$R' = 2 \cdot R = \begin{pmatrix} 1 & \frac{1}{2} \\ 1 & 1 \end{pmatrix}$$

und die Matrix der Datenallokation ist:

$$A \cdot R' = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

R' ist hier ganzzahlig. Deshalb gilt immer noch:

$$(\forall i : i \in \mathbb{Z}^n : R' \cdot \lfloor (R')^{-1} \cdot i \rfloor = i)$$

und

$$(\forall a, b : a \neq b \in \mathbb{Z}^n : \lfloor R' \cdot a \rfloor \neq \lfloor R' \cdot b \rfloor)$$

Und damit auch die Bedingungen (25) und (26). Abbildung 13 zeigt diese Stauchung der Basistransformation um den Faktor 2. Die modifizierte Basistransformation führt auf die Arraygrenzen $A[-2 \ldots 4, -4 \ldots 4]$.

Das neue Array hat also 63 Elemente. Auf einem Prozessorfeld $\{(p_1, p_2) \mid 0 \leq p_1 \leq 4, 0 \leq p_2 \leq 8\}$ speichern so nach der durch $A \cdot R'$ gegebenen Allokation die Prozessoren $\{(p_1, p_2) \mid p_1 \in \{3, 4\}, p_2 \in \{0, \ldots, 8\}\}$ jeweils 2 Array-Elemente, während alle übrigen Prozessoren jeweils 1 Array-Element speichern.

Nun stellt sich die Frage, wie wir für eine gegebene Matrix $S \in \mathbb{Z}^{n \times n}$ zu dem maximalen Stauchungsfaktor d_{inj} kommen. Das genaue Kriterium liefert der folgende Satz:

Satz 5.15 *Sei $S = (s_{i,j}) \in \mathbb{Z}^{n \times n}$. Sei d_{inj} maximal, so daß für alle $a \neq b \in \mathbb{Z}^n$ gilt:*

$$\left\lfloor \frac{S}{d_{inj}} \cdot a \right\rfloor \neq \left\lfloor \frac{S}{d_{inj}} \cdot b \right\rfloor$$

Dann ist

$$d_{inj} = -1 + \min(\{d \mid (\exists x : x \in \mathbb{Z}^n : (\forall i :: -d < (\textstyle\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot \pi_j(x)) < d))\})$$

Beweis:

Wir beweisen die Behauptung, indem wir zunächst eine Abschätzung der gegebenen Eigenschaft ermitteln. Für die dann ihre Optimalität bewiesen wird (in dem Sinne, daß in den übrigen Fällen a, b nicht beliebig sein können). Schließlich wenden wir dann diese Abschätzung an, um den Satz zu zeigen.

Abschätzung der exakten Bedingung Wir suchen für $(s_{i,j})_{i,j \in \{1, \ldots, n\}} = S \in \mathbb{Z}^{n \times n}$ ein möglichst großes $d_{inj} \in \mathbb{N}$ mit

$$(\forall a, b : a \neq b \in \mathbb{Z}^n : \lfloor S(a) \rfloor \neq \lfloor S(b) \rfloor) \tag{27}$$

Diese Bedingung gilt genau dann, wenn für alle $(a_1, \ldots, a_n) = a \neq b = (b_1, \ldots, b_n) \in \mathbb{Z}^n$ gilt:

$$\left\lfloor \frac{(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j)}{d_{inj}} \right\rfloor \neq \left\lfloor \frac{(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j)}{d_{inj}} \right\rfloor$$
$$\overset{\text{Def. }\%}{\Leftrightarrow}$$
$$\frac{(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j)}{d_{inj}} - \frac{((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j))\% d_{inj}}{d_{inj}}$$
$$\neq$$
$$\frac{(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j)}{d_{inj}} - \frac{((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j))\% d_{inj}}{d_{inj}}$$
$$\Leftrightarrow$$
$$(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j) - (\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j)$$
$$\neq$$
$$((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j))\% d_{inj} - ((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j))\% d_{inj}$$
$$\Leftrightarrow$$
$$(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot (b_j - a_j))$$
$$\neq$$
$$((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j))\% d_{inj} - ((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j))\% d_{inj} \tag{28}$$

Bedingung (28) ist erfüllt, falls gilt:

$$|(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot (b_j - a_j))| \geq d_{inj} \tag{29}$$

Optimalität der Abschätzung Bedingung (29) stellt dabei eine optimale Abschätzung in dem Sinne dar, daß, wenn die obige Formel durch irgendwelche anderen Paare (S, d_{inj}) erfüllt sein soll, der Wertebereich[9] für a, b berücksichtigt werden muß, also die Bedingung (28) keinesfalls für **alle** $a \neq b \in \mathbb{Z}^n$ gelten kann:

Nehmen wir an, es gelte für alle $i \in \{1, \ldots, n\}$:

$$(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot (b_j - a_j)) \neq$$
$$((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j))\% d_{inj} - ((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot a_j))\% d_{inj}$$

wähle $a = (0, \ldots, 0)$. Es gilt also:

$$(\forall i :: (\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j) \neq ((\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j))\% d_{inj})$$

Wir betrachten nun lediglich die Fälle, in denen die obige Abschätzung nicht tatsächlich erfüllt war. D.h. es gilt:

$$(\not\forall i, b :: |(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j)| \geq d_{inj})$$

also

$$(\exists i, b :: |(\sum j : j \in \{1, \ldots, n\} : s_{i,j} \cdot b_j)| < d_{inj})$$

[9] Dies sind in unserem Fall die Arraygrenzen.

Wir betrachten die speziellen i und b aus dem Existenzquantor. D.h. es gilt o.B.d.A.:

$$0 \leq (\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot b_j) \leq d_{inj}$$

(gilt dies nicht für b, so gilt es für $-b$). Die gewünschte Ungleichung

$$(\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot b_j) \neq ((\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot b_j))\% d_{inj}$$

ist in diesem Fall äquivalent zu

$$(\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot b_j) \neq (\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot b_j)$$

was ein Widerspruch ist. Die Gültigkeit ist also von dem Wertebereich von a, b abhängig. Die Bedingung (29) stellt somit eine optimale Abschätzung dar.

Anwendung der Abschätzung Das größte d_{inj}, für das (29) gilt, ist aber $d_0 - 1$, wobei $d_0 \in \mathbb{N}$ minimal ist, so daß die Bedingung

$$(\forall i : i \in \{1,\ldots,n\} : |(\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot x_j)| < d_0) \tag{30}$$

noch eine ganzzahlige Lösung für $x = (x_1,\ldots,x_n)$ besitzt. Bedingung (30) ist äquivalent zu

$$(\forall i : i \in \{1,\ldots,n\} : -d_0 < (\sum j : j \in \{1,\ldots,n\} : s_{i,j} \cdot x_j) < d_0) \tag{31}$$

In Mengenschreibweise ergibt dies die Formulierung aus Satz 5.15. ✓

Mithilfe dieses Satzes können wir nun den Stauchungsfaktor ermitteln. Im Beweis wurde verwendet, daß $d_{inj} \in \mathbb{N}$ maximal ist, so daß Bedingung (29) erfüllt ist. Dabei muß $d_{inj} < \max(\{|s_{i,j}| \mid i,j \in \{1,\ldots,n\}\})$ sein, da man sonst leicht zwei Einheitsvektoren $e_{\sigma(1)} \neq e_{\sigma(2)}$ aussuchen kann, so daß für mindestens eine Kombination für a, b mit $a \in \{e_{\sigma(1)}, -e_{\sigma(1)}\}$ und $b \in \{e_{\sigma(2)}, -e_{\sigma(2)}\}$ die Bedingung

$$\left\lfloor \tfrac{S}{d_{inj}} \cdot a \right\rfloor \neq \left\lfloor \tfrac{S}{d_{inj}} \cdot b \right\rfloor$$

nicht mehr erfüllt sein wird. Zur genauen Bestimmung von d_{inj} können wir Satz 5.15 verwenden:

Die Ungleichung (31) läßt sich durch einen Lauf des Fourier-Motzkin-Algorithmus (vgl. [Ban93]) symbolisch nach x auflösen. Damit bleibt x in der Ausgabe des Algorithmus weiter von d abhängig. Es genügt also, für jeden in Frage kommenden Wert von d so lange alle Ungleichungen, die der Fourier-Motzkin-Algorithmus liefert, darauf abzutesten, ob sie ganzzahlige Lösungen besitzen, bis eine solche Lösung für jedes x_i gefunden werden kann. Ist dies für d_0 der Fall, war $d_{inj} = d_0 - 1$ der gesuchte maximale Nenner für S.

Zur Ermittlung des Wertes von d_{inj} können auch Methoden der parametrisierten linearen Optimierung herangezogen werden: beispielsweise kann für jeden möglichen Wert von d_{inj} der `PIP`-Algorithmus von Feautrier [Fea88] auf das gegebene Ungleichungssystem angewandt werden, um festzustellen, ob eine ganzzahlige Lösung existiert.

Zeilentransformation Obwohl es möglich ist, daß der lineare Anteil der Allokationsfunktion A der Bedingung (22) genügt, sich die Allokation somit also durch eine Sequenz von *SPITFALLS* (eine *SPITFALLS* für jede Dimension des Prozessorfeldes) ausdrücken läßt, ist dies nicht immer gewährleistet. Wir brauchen aber auch für diesen Fall eine Darstellungform für die Datenallokation.

Ist die Datenallokation durch eine komplexere Abbildung gegeben, so daß der lineare Anteil nicht Bedingung (22) genügt, ist dieser – nach der auf Seite 48 eingeführten Spaltentransformation (mit der im letzten Absatz beschriebenen Stauchung) – in einer Form gegeben wie

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & \dots & 0 \\ a_{2,1} & \vdots & \vdots & 0 & \dots & 0 \\ 0 & a_{i,2} & 0 & 0 & \dots & 0 \\ a_{i+1,1} & a_{i+1,2} & \ddots & 0 & \dots & 0 \\ \vdots & \vdots & \dots & 0 & \dots & 0 \end{pmatrix}$$

wobei $a_{i+1,1} \neq 0$. Es gibt also mehrere Nichtnull-Einträge in einer Zeile. Dies entspricht einer zusätzlichen Bedingung auf den Prozessor- bzw. Arraykoordinaten, die erfüllt sein muß, damit ein gegebener Prozessor x das Array-Element $A[i_1, \dots, i_n]$ speichert. Ist der lineare Anteil wie oben gegeben, lautet diese Bedingung:

$$\pi_{i+1}(x) = \lfloor a_{i+1,1} \cdot i_1 + a_{i+1,2} \cdot i_2 + \pi_i(k) \rfloor$$

wobei k die additive Konstante im affinen Teil der *Modularabbildung* ist. Um eine solche Allokation zu realisieren, kann man nun

- die Allokation derart abändern, daß das Array entlang der Prozessordimensionen, auf denen eine derartige Bedingung abzutesten ist, repliziert gespeichert wird,
- die bisherige Darstellung dahingehend ändern, daß die solchen Bedingungen entsprechenden Zeilen der Matrix separat gespeichert werden, oder
- die Allokationsbeschreibung mittels einer erneuten Transformation der darstellenden Matrix so umschreiben, daß die tarnsformierte Allokation dargestellt werden kann.

Der erste Ansatz scheidet allein schon wegen der Änderung der uns fest vorgegebenen Datenallokation aus. Der zweite Ansatz würde für jedes Array-Element eine Matrixmultiplikation bedeuten, über die bestimmt werden müßte, welcher Prozessor die (bis auf die Gaußklammer affine) Bedingung erfüllt. Wie wir später sehen werden, kann diese Multiplikation mit letzterem Ansatz – den wir im folgenden ausarbeiten wollen – etwas verschoben werden.

Aus der Linearen Algebra ist bekannt, daß sich jede Matrix einer linearen Abbildung bzgl. geeigneter Basen für Quell- und Zielraum so darstellen lassen, daß die Bedingung (22) gilt. Ist diese Bedingung nach der Spaltentransformation noch nicht erfüllt, kann sie mit einem geeigneten Basiswechsel im Zielraum noch erfüllt werden. Da der Zielraum einer Datenallokation das Prozessorfeld ist, entspricht ein Basiswechsel hier der Einführung von virtuellen Prozessoren, die von den realen Prozessoren des Prozessorfeldes simuliert werden müssen.

Da eine Datenallokation aber nicht durch eine einfache affine Funktion gegeben ist, sondern durch eine *Modularabbildung*, die zusammengesetzt ist aus einer rationalen affinen Funktion, einer Gaußklammerfunktion und einer MOD-Operation, genügt nicht einfach die Anwendung einer linearen Abbildung zur Simulation der virtuellen Prozessoren im Zielraum, wie es der Fall war bei den „virtuellen Indizes“ im Quellraum der Abbildung. Gehen wir nun also davon aus, daß eine Abbildung Φ gegeben ist mit

$$\Phi : \mathbb{Z}^n \to \mathbb{N}^m : i \to \left\lfloor \frac{A \cdot i + k}{d} \right\rfloor \% (p_1, \dots, p_m)$$

Gesucht sind nun Abbildungen $\Phi' : \mathbb{Z}^n \to \mathbb{N}^m$ und $\Omega : \mathbb{N}^m \to \mathbb{N}^m$ mit:

$$\Phi = \Omega \circ \Phi' \tag{32}$$

und

$$\Phi' : i \to \left\lfloor \frac{A'+k'}{d'} \right\rfloor \% (p'_1, \ldots, p'_m) \tag{33}$$

wobei A' Bedingung (22) erfüllt

Dabei wird Ω technisch dadurch realisiert, daß jeder physikalische Prozessor $(x_1, \ldots, x_m)$ u.U. mehrere virtuelle Prozessoren $\Omega^{-1}((x_1, \ldots, x_m))$ simuliert. Um Bedingung (33) zu erfüllen, reicht es aus – nachdem A schon der Bedingung (21) genügt –, A von links mit einer geeigneten Matrix L zu multiplizieren, welche lediglich aus elementaren Zeilentransformationen besteht. Wir gehen o.B.d.A. davon aus, daß $L \in \mathbb{Z}^{m\times m}$ (eine solche Matrix läßt sich immer finden). Dadurch erhalten wir zunächst den Zusammenhang

$$\Phi : i \to \left\lfloor \frac{L^{-1}A' \cdot i + k}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

Dabei wollen wir den Konstantenanteil in der eigentlichen Allokationsfunktion belassen und erhalten:

$$\Phi : i \to \left\lfloor \frac{L^{-1} \cdot (A' \cdot i + k')}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

Wir können nun

$$\Omega : x \to \left\lfloor \frac{L^{-1} \cdot x}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

definieren. Allerdings ist dann die Allokationsfunktion Φ' nach (32) zunächst nicht mehr als *Modularabbildung* gegeben, sondern durch die affine Abbildung $i \to A' \cdot i + k'$, was insbesondere $\Phi' \neq 0 \Rightarrow \#(\mathrm{Bi}(\Phi')) = \infty$ und damit im allgemeinen $\#(\Omega^{-1}(x)) = \infty$ zur Folge hat. Es müssen also u.U. unendlich viele virtuelle Prozessoren simuliert werden.

L^{-1} ist aus $\mathbb{Q}^{m\times m}$. Sei $L^{-1} = \frac{L_{\mathbb{Z}}}{d_L}$ mit $L_{\mathbb{Z}} \in \mathbb{Z}^{m\times m}$, dann kann man Φ schreiben als:

$$\Phi : i \to \left\lfloor \frac{L_{\mathbb{Z}} \cdot (A' \cdot i + k')}{d \cdot d_L} \right\rfloor \% (p_1, \ldots, p_m)$$

Nach Satz 2.13 können wir dann aber die Abbildung Φ für $\overrightarrow{q} = (q, \ldots, q)$ mit $q = d \cdot d_L \cdot \mathrm{kgV}(\{p_i \mid i \in \{1, \ldots, m\}\})$ umschreiben zu:

$$\Phi : i \to \left\lfloor \frac{L_{\mathbb{Z}} \cdot ((A' \cdot i + k') \% \overrightarrow{q})}{d \cdot d_L} \right\rfloor \% (p_1, \ldots, p_m)$$

also

$$\Phi : i \to \left\lfloor \frac{L^{-1} \cdot ((A' \cdot i + k') \% \overrightarrow{q})}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

Damit läßt sich Φ jetzt ausdrücken als

$$\Phi = \Omega \circ \Phi'$$

mit der *Modularabbildung*

$$\Phi' : i \to \left\lfloor \frac{A' \cdot i + k'}{1} \right\rfloor \% \overrightarrow{q}$$

und

$$\Omega : x \to \left\lfloor \frac{L^{-1} \cdot x}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

Durch die *Modularabbildung* Φ' kann die Allokation wie bisher ausgedrückt werden, ohne daß obige Probleme aufgrund der Unbeschränktheit von Φ' entstehen. Ω ist also eine *Modularabbildung*, kann aber auch als Hintereinanderausführung

$$\Omega = \Omega' \circ L^{-1}$$

mit der *Modularabbildung*

$$\Omega' : x \to \left\lfloor \frac{x}{d} \right\rfloor \% (p_1, \ldots, p_m)$$

betrachtet werden. Die Umkehrung dieser Hinteinanderausführung ergibt sich dann aus

$$(\Omega' \circ L^{-1})^{-1} = L \circ (\Omega')^{-1}$$

Der lineare Koeffizient in Ω' ist für jede Komponente 1. Damit läßt sich das Urbild eines Punktes unter Ω' durch eine einzige *FALLS* in einem virtuellen Prozessor-Array ausdrücken.

Insgesamt läßt sich nun also jede durch eine *Modularabbildung* gegebene Datenallokation durch eine Sequenz von *SPITFALLS* beschreiben, die für jede Prozessordimension eine *SPITFALLS* enthält. Hierbei simuliert u.U. jeder Prozessor eine Menge von virtuellen Prozessoren, die ihrerseits als Umkehrung einer *Modularabbildung* (definiert durch die Matrix L) gegeben wird. Um also festzustellen, auf welchem Prozessor welche Elemente eines Arrays zu speichern sind, ist es nicht nötig, für jedes Array-Element die Allokationsfunktion zu berechnen, was für jedes Array-Element eine Matrixmultiplikation bedeutet. Eine Matrixmultiplikation ist lediglich zur Simulation der virtuellen Prozessoren für jeden Urbildwert von $(\Omega')^{-1}(x)$ für einen physikalischen Prozessor x nötig. Die Matrixmultiplikation findet also anstatt für jedes Array-Element nur noch für jeden von einem realen Prozessor zu simulierenden virtuellen Prozessor statt (auf welcher Prozessoranordnung die Array-Elemente i.a. zyklisch verteilt sind).

5.4 Reallokation und Host-Kommunikation

Jedes *Projekt* definiert für jedes Array eine *Modularabbildung* als Datenallokation. Im allgemeinen ist es dabei nötig, zwischen der Ausführung der Projekte alle Arrays vom Parallelrechner auf den Hostrechner zu transferieren, und alle vom folgenden Projekt benötigten Arrays danach wieder auf den Parallelrechner aufzubringen. Dabei sind im allgemeinen für die Kommunikation mit dem Hostrechner andere Kommunikationsroutinen nötig als innerhalb des Parallelrechners.

Der Parallelrechner soll dabei (in *Projekten* formulierte) kurze Programmsequenzen eines größeren Programmes ausführen. Das Programm auf dem Host übernimmt dabei die Steuerung, wann welches *Projekt* auf dem Paradllelrechner zu starten ist.

Nachdem ein bestimmtes *Projekt* auf dem Parallelrechner aufgerufen wurde, müssen die Daten vom Hostrechner auf den Parallelrechner gemäß der für das auszuführende *Projekt* aktuellen Datenallokation übertragen werden. Da nur ein Prozessor des Parallelrechners mit dem Hostrechner verbunden ist, muß dieser Prozessor für alle Prozessoren des Parallelrechners die auf diesem Prozessor zu speichernden Daten vom Hostrechner abfragen und an den *Owner*-Prozessor weiterleiten.

Es ist jedoch auch möglich, daß mehrere *Projekte*, die das gleiche Prozessorfeld definieren, unmittelbar hintereinander ausgeführt werden sollen, so daß ein oder mehrere Arrays zwischen den Aufrufen der *Projekte* keinesfalls auf dem Hostrechner geändert werden konnten. In diesem Fall ist es prinzipiell nicht nötig, die Arrays nach Beendigung eines *Projekts* auf den Hostrechner aufzubringen, um sie unmittelbar danach wieder auf den Parallelrechner aufzuladen. Stattdessen können die Arrays prinzipiell direkt auf dem Parallelrechner von der Verteilung gemäß der alten Datenallokation Φ_{alt} auf die Verteilung gemäß der neuen Allokation Φ_{neu} umverteilt werden.[10] In diesem Fall kommuniziert potentiell jeder Prozessor mit jedem anderen. Hierfür muß jeder Prozessor x für jeden anderen Prozessor y seine Sendmenge

$$Send(x,y) = \Phi_{alt}^{-1}(x) \cap \Phi_{neu}^{-1}(y)$$

und seine Receivemenge

$$Recv(x,y) = \Phi_{alt}^{-1}(y) \cap \Phi_{neu}^{-1}(x)$$

kennen. Diese Mengen sind für den allgemeinen Fall von Φ_{neu} bzw. Φ_{alt} jedoch nicht einfach zu berechnen. Unterscheiden sich Φ_{neu} und Φ_{alt} aber nur in den beschreibenden *SPITFALLS* (und nicht in den notwendigen Spalten- bzw. Zeilentransformationen), so lassen sich aus den dadurch berechneten *PITFALLS* die Send- und Receivemengen nach dem *PITFALLS*-Intersection-Algorithmus aus [RB95] einfach angeben. Allerdings kann eine Entscheidung, welche Arrays zwischen welchen *Projekten* direkt auf dem Parallelrechner umzuverteilen sind, in diesem Rahmen nicht automatisch gefällt werden, da hier das Hostprogramm (mit Ausnahme der Teile, die durch die *Projekte* beschrieben werden) gänzlich unbekannt ist.

Die Kommunikation mit dem Hostrechner zum Aufladen der Arrays kann somit als Spezialfall einer solchen Reallokation mit $\Phi_{alt} : a \rightarrow k$ angesehen werden, wobei k die Koordinaten des ausgezeichneten Prozessors sind, der mit dem Hostrechner verbunden ist. Statt die Daten aus dem eigenen lokalen Speicher abzufragen (bzw. darin abzulegen), muß dann dieser Prozessor eine entsprechende Anfrage an den Hostrechner stellen; ansonsten kann exakt das gleiche Distributionsverfahren angewandt werden. Dabei ist zu beachten, daß vor bzw. nach jeder Redistributionsphase die Prozessoren des Parallelrechners synchronisiert werden müssen, um garantieren zu können, daß auf jedem Prozessor die aktuellen Daten vorhanden sind.

5.5 Datenverteilung in *LooPo*

LooPo verwendet grundsätzlich eine statische Datenallokation in Form einer *Modularabbildung* zur Allokation der Arrays. Replizierte Speicherung wird lediglich für Blobs (und Indizes) angewandt.

Der in *LooPo* integrierte Allokator nach der Methode von Dion und Robert (dessen Implementation in [Mei96] beschrieben ist) ermittelt für geeignete Arrays eine affine Abbildung als Allokation, die durch Anwendung der maximalen Grenzen des von dem *Projekt* aufgespannten Prozessorfeldes als Modulo-Vektor zu *Modularabbildung* umgeformt wird. Ist für ein Array keine Allokation vorgegeben, so wird eine solche nach dem in Abschnitt 5.2 beschriebenen Verfahren bestimmt.

[10] Bei der Implementation der *LooPo*-Library muß hier jedoch wegen der *SPITFALLS*-basierten Darstellung der Allokation das erweiterte Prozessorfeld und die Zeilen- und Spaltentransformationen für die Allokation der umzuverteilenden Variable in den *Projekten* identisch sein.

Im Ausgabeprogramm wird die Allokation für ein Array durch *SPITFALLS* ausgedrückt. Wie in Abschnitt 5.3 beschrieben, muß der lineare Anteil der Allokation hierfür (bzw. überhaupt zur Ermittlung lokal zu speichernder Abschnitte eines Arrays) durch Zeilen- und Spaltentransformationen in eine geeignete Form überführt werden. Der für die Spaltentransformation zu ermittelnde maximale Stauchungsfaktor aus Abschnitt 5.3.3 wird durch Iterieren über alle möglichen Werte bestimmt. In Abschnitt 5.4 wurde darauf hingewiesen, daß Arrays, die zwischen der Ausführung zweier *Projekte* garantiert unverändert bleiben, prinzipiell auch direkt auf dem Parallelrechner umverteilt werden können, ohne daß Daten mit dem Hostrechner ausgetauscht werden müßten. Wegen der *SPITFALLS*-basierten Darstellung der Allokation in der *LooPo*-Library müßte hierfür jedoch das erweiterte Prozessorfeld der *Projekte* identisch sein. Gleiches gilt für die Zeilen- und Spaltentransformationen der Allokation der umzuverteilenden Variablen. Außerdem steht *LooPo* selbst keine Information zur Verfügung, welche Arrays auf dem Hostrechner verändert werden, weshalb entsprechende Änderungen im SPMD-Code (vgl. Beispiel 6.13) manuell vorzunehmen wären.

Die Kommunikation des Parallelrechners mit dem Hostrechner findet über Internet-`sockets` statt. Für den Hostrechner wird zu diesem Zweck Code generiert, der in das ursprüngliche Programm (aus dem der Code für die *Projekte* entnommen wurde) einzusetzen ist:

Beispiel 5.16 *Unser Ausgangsprogramm ist eine Matrixmultiplikation der Form:*

```
#include<stdio.h>

main()
{
  int A[3][3];
  int B[3][3];
  int C[3][3];

  int n=3;

  int i,j,k;

  /* Matrixinitialisierung geloescht */

  for(i=0; i<n; i++)
    {
      for(j=0; j<n; j++)
        {
          for(k=0; k<n; k++)
            {
              C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }


  /* Ausgabe Endergebnis */
  for(i=0; i<n; i++)
    {
      for(j=0; j<n; j++)
        {
```

```
          printf("C[%d][%d]=%d\n",i,j,C[i][j]);
        }
    }
}
```

Der für den Hostrechner erzeugte Code ist:

```
/* put this at the beginning of your host program */
#include "LPOall.h"
#include "comm_all.h"
#include "pitfalls.h"
#include "symtab.h"

LooPo_initSymtabs(3-1,1-1);
LooPo_declareConstant(&(n),0,LOOPO_INT);
LooPo_declareArray(&(A),0,LOOPO_INT,2,0,2,0,2);
LooPo_declareArray(&(B),1,LOOPO_INT,2,0,2,0,2);
LooPo_declareArray(&(C),2,LOOPO_INT,2,0,2,0,2);

LooPo_connectPar(0);

/* replace call to loop number 0 by: */
LooPo_performPar(0);


/* put this at the end of your host program */
LooPo_disconnectPar();
LooPo_closeSymtabs();
```

Dabei baut `LooPo_initSymtabs` *eine Tabelle für die zu verwaltenden Daten auf, in der beispielsweise Typinformation und Größe der zu kommunizierenden Variablen gespeichert werden.* `LooPo_connectPar` *wartet auf eine Verbindungsanfrage seitens des Parallelrechners, und* `LooPo_performPar(`i`)` *schickt einen Befehl an den Parallelrechner, das* Projekt *Nummer i auszuführen. Die Datenallokation selbst ist zunächst lediglich auf dem Parallelrechner bekannt (wie in Beispiel 6.13 auf Seite 89 gezeigt), der entsprechende Anfragen zum Aufladen bzw. Zurückspeichern von Daten an den Hostrechner stellt; diese Anfragen werden ebenfalls innerhalb der Funktion* `LooPo_performPar` *verarbeitet.*

Zur lokalen Speicherung eines Arrays auf dem Parallelrechner wurde das zweite in Abschnitt 5.3 beschriebene Verfahren zur Organisation des lokalen Speichers implementiert; das Ausgabemodul `targout` ist auch für die dritte (in [Wol95] beschriebene) Strategie vorbereitet.

6 Kommunikation

Auf einem Distributed-Memory-Rechner werden Daten verteilt gespeichert. Wie diese Speicherung selbst realisiert werden kann, haben wir in Abschnitt 5.3 gesehen. Durch die Parallelschleifen eines *Projektes* werden aber auch Verteilungen für die auszuführenden Berechnungen angegeben. Dabei ist es möglich, daß ein Datum, das zur Abarbeitung einer Anweisung benötigt wird (z.B. weil es auf der rechten Seite einer Zuweisung steht und deshalb gelesen wird oder weil es auf der linken Seite einer Zuweisung steht und deshalb geschrieben wird), durch die Datenallokation nicht auf dem Prozessor plaziert worden ist, auf dem die Anweisung auszuführen ist. Es ist also klar, daß Kommunikation stattfinden muß, wenn ein Prozessor (hier *Computer* genannt) zur Durchführung einer Anweisung den Wert eines Datums braucht, das auf ihm nicht lokal vorhanden ist. Auch bei Maschinen mit gemeinsamem Speicher muß zumindest garantiert sein, daß kein Prozessor einen veralteten Wert eines Datums liest, der von einem anderen Prozessor evtl. noch nicht mit dem aktuellen Wert überschrieben wurde. In Abschnitt 6.1 befassen wir uns mit dieser prinzipiellen Problematik, bevor wir in Abschnitt 6.2 auf das Einfügen expliziter Kommunikationsanweisungen für Rechner mit verteiltem Speicher eingehen, die wir in den dann folgenden Abschnitten genauer betrachten werden. Abschnitt 6.6 beschreibt die Ausgabe eines *Projekts* mit Kommunikationsanweisungen durch das `targout`-Modul von *LooPo* anhand eines Beispiels.

6.1 Synchronisationspunkte

In Abschnitt 4.3 haben wir festgestellt, welche Prozessoren im SPMD-Modell welche Anweisungen auszuführen haben. Um eine korrekte Abarbeitung des Programms gewährleisten zu können, muß (zur Organisation der Prozessoren untereinander) jedoch auch irgendeine Form der Kommunikation zwischen Prozessoren stattfinden. Bei Maschinen mit gemeinsamem Speicher genügt dabei die Synchronisation der einzelnen Prozesoren miteinander. Für diesen Abschnitt gehen wir also davon aus, der Parallelrechner arbeite mit einem gemeinsamen Speicher.

Es muß nun gewährleistet werden, daß alle Anweisungen in der in Abschnitt 2.1 definierten Reihenfolge ausgeführt werden. Da der Parallelrechner aus einer asynchronen Maschine besteht, muß diese Ausführungsreihenfolge unter Umständen durch Einfügen von `barrier`-Anweisungen garantiert werden.

Anders als beispielsweise in `HPF`-Programmen, definiert dies **nicht** die gleiche Abarbeitungsreihenfolge wie die sequentielle Ausführung aller Schleifen.

Beispiel 6.1 *Das* Projekt *aus Abschnitt 4.1 (hier mit Schrittweite* 1 *und entsprechend erweiterter Parallelschleife)*

`par` $p = 0$ `to` 2
  `for` $t = 0$ `to` 2
    $A[t,p] := A[t-1,p-1] + A[t-1,p+1];$

kann nicht semantikerhaltend in das sequentielle Programm

`for` $p = 0$ `to` 2
  `for` $t = 0$ `to` 2
    $A[t,p] := A[t-1,p-1] + A[t-1,p+1];$

überführt werden, da sonst – im Gegensatz zum ursprünglichen Projekt *– der Punkt* $(1,0)$ *vor dem Punkt* $(0,1)$ *ausgeführt wird, was nicht semantikerhaltend ist, weil diese Iterationen nicht unabhängig voneinander sind.*

Wie aus der Beschreibung von sequentiellen und parallelen Schleifen in Abschnitt 2.1 hervorgeht, ist eine korrekte Ausführung des Programms dann garantiert, wenn

- für `par`-Schleifen eine `fork-join`-Semantik gewahrt bleibt und
- jede neue Iteration einer `for`-Schleife erst dann begonnen wird, wenn die aktuelle Iteration auf **jedem** Prozessor beendet ist.

Sehen wir den Indexraum eines gesamten *Projekts* mit n Schleifen als Polyeder in $\mathbb{Z}^n$ an (so daß zwei Schleifen, die von denselben weiteren Schleifen umgeben werden, unterschiedlichen Dimensionen des $\mathbb{Z}^n$ entsprechen), so wird die Reihenfolge, in der die Indexvektoren dieses Gesamtindexraums aufgezählt werden müssen, durch die Einschränkung der lexikographischen Striktordnung $\prec$ auf die von sequentiellen Schleifen aufgespannten Dimensionen gegeben. Das ist mit der Abbildung

$$\pi_i^S : \mathbb{Z}^n \rightarrow \mathbb{Z} : j \rightarrow \begin{cases} \pi_i(j) & \text{falls Dimension } i \text{ von \texttt{for}-Schleife aufgespannt wird} \\ 0 & \text{sonst} \end{cases}$$

also die Striktordnung $\prec_S$, die gegeben wird durch:

$$a \prec_S b : \Leftrightarrow (\pi_1^S(a), \ldots, \pi_n^S(a)) \prec (\pi_1^S(b), \ldots, \pi_n^S(b))$$

Unvergleichbare Indexvektoren können dabei in beliebiger Reihenfolge (bzw. parallel) ausgeführt werden (eine mögliche Ausführungsreihenfolge wird also durch die lexikographische Striktordnung $\prec$ definiert, wenn die Einträge der Indexvektoren so umsortiert werden, daß die durch sequentielle Schleifen aufgespannten Dimensionen vor den Einträgen für die von `par`-Schleifen aufgespannten Dimensionen stehen).

Für synchrone Rechnerarchitekturen mit verteiltem Speicher ist die Ausführungsreihenfolge schon bzgl. $\prec_S$ geordnet. Das heißt, auf einer asynchronen Rechnerarchitektur genügt es, jede Anweisung mit einer `barrier`-Anweisung abzuschließen, um die korrekte Ausführung des Programms zu gewährleisten. Unter der Annahme, daß ein Lese- bzw. Schreibzugriff auf ein Datum erst beendet ist, wenn er erfolgreich durchgeführt wurde (der Speicher sich also wieder in einem konsistenten Zustand befindet), können wir jedoch auf alle `barrier`-Anweisungen bis auf die letzte eines *Basisblocks* verzichten, da immer nur ein Prozessor eine Instanz einer Anweisung ausführt und sich die Menge der aktiven Prozessoren innerhalb eines *Basisblocks* nicht ändert. Außerdem ist es unerheblich, ob der Speicher nach den Zugriffen in einem gegebenen *Basisblock* schon wieder in einem konsistenten Zustand ist, falls die darin geschriebenen Daten nicht gebraucht werden. Zusätzlich genügt es aber für jede Abhängigkeit, die in einer umgebenden `for`-Schleife *enthalten* ist, wenn am Ende dieser Schleife synchronisiert wird. Also ist die Synchronisation am Ende eines *Basisblocks*, dessen Abhängigkeiten alle in umliegenden `for`-Schleifen *enthalten* sind, nicht notwendig. Wir kommen also zu dem Schluß, daß die Anweisungen eines *Projektes* auf asynchronen Maschinen in der korrekten Reihenfolge ausgeführt werden, falls an folgenden Stellen synchronisiert wird:

- am Ende jeder `for`-Schleife, die die sequentielle Komponente einer Abhängigkeit trägt, und
- am Ende jedes *Basisblocks*, von dem eine Abhängigkeit ausgeht, die nicht in einer umliegenden sequentiellen Schleife *enthalten* ist.

Die Synchronisation selbst führt jedoch auf ein weiteres Problem. Betrachten wir das *Projekt*

```
par p = 0 to 11
  for t = p + 10 to 20
    S1 :
    A[p, t] := 42 · A[p + 1, t − 10];
```

Dann ist bei der Ausführung nach dem Dreistufenmodell eine Synchronisation in der t-Schleife nicht über das gesamte Prozessorfeld $\{p \mid 0 \leq p \leq 10\}$ möglich, da nicht alle Prozessoren jede Iteration der t-Schleife ausführen. Dieses Problem kann auf zwei Arten gelöst werden:

- Der Indexraum wird zu einem massiven n-dimensionalen Quader (d.h. zu einem Quader, der aus **allen** ganzzahligen Punkten innerhalb der Grenzen besteht) ausgeweitet.
- `barrier`-Anweisungen beziehen sich grundsätzlich nur auf „aktive" Prozessoren.

Erstere Methode muß dadurch realisiert werden, daß die Unter- bzw. Obergrenze jeder `do`-Schleife auf ihr Minimum bzw. Maximum erweitert und die Schrittweite auf 1 gesetzt wird. Jeder *Berechnungsblock* ist dann durch eine `if`-Anweisung einzuschließen, die die Gültigkeit des Indexwertes abprüft, die vorher durch die `for`-Anweisung definiert war. Zusätzlich müssen bei jeder `if`-Anweisung alle Kontroll- und `barrier`-Anweisungen des `else`-Zweigs auch in den `then`-Zweig kopiert werden (und umgekehrt). In unserem Beispiel ergibt dieses Vorgehen dann:

```
par p = 0 to 11
  for t = 10 to 20
    if t ≥ p + 10 then
      S1 :
      A[p,t] := 42 · A[p + 1, t − 10];
      barrier;
    else
      barrier;
```

Diese Möglichkeit ist jedoch nicht direkt anwendbar auf Programmfragmente der Form

```
par p = 0 to 10
  while feelLikeIt(p)
    S1 :
    A[p,t] := 42 · A[p + 1, t − 10];
```

da für die `while`-Schleife a priori kein Maximum gebildet werden kann. Es müßte hier vor Beginn jeder Iteration der `while`-Schleife bestimmt werden, ob es noch Prozessoren gibt, die *Berechnungsblöcke* in der Schleife ausführen müssen.

Auf diesem Prinzip der Ermittlung noch aktiver Prozessoren fußt die zweite Methode: wir nehmen hier an, daß es kollektive Kontrollanweisungen `group`, `regroup` und `killgroup` gibt, die eine Gruppe definieren, auf die sich eine `barrier`-Anweisung bezieht (d.h. jeder Prozessor, der ein `barrier` ausführt, wird lediglich die Prozessoren aus dieser Gruppe informieren, daß er diese Anweisung erreicht hat, und auch nur auf die Information von Prozessoren aus dieser Gruppe warten). Die Anweisungen haben dabei folgende Bedeutung:

`group`(*cond*)	Aufbau einer neuen aktuellen Gruppe, die aus den Prozessoren besteht, für die *cond* gilt
`regroup`(*cond*)	Verkleinern der aktuellen Gruppe auf die Untermenge der Prozessoren, für die *cond* gilt
`killgroup`	Löschen der aktuellen Gruppe; die vor dem letzten `group`-Aufruf aktuelle Gruppe wird wieder als aktuell angesehen

Diese Anweisungen können so implementiert werden, daß jeder Prozessor auch die Bedingungen aller übrigen Prozessoren abprüft. Eine andere Möglichkeit (für verteilten Speicher) ist, jeden Prozessor lediglich die für ihn gültige Bedingung abprüfen zu lassen, wonach jeder Prozessor ein `broadcast` (bzw. – falls die Rechnerarchitektur dies unterstützt – eine entsprechende `all-to-all`-Kommunikation) mit dem Ergebnis an jeden anderen Prozessor der aktuellen Gruppe ausführt. Letzterer Ansatz würde synchronisierend wirken und so bei einer sequentiellen Schleife die `barrier`-Anweisung am Ende der Schleife unnötig machen (d.h. ersetzbar durch eine `barrier`-Anweisung **hinter** dem Schleifenende). In unserem Beispiel resultiert aus dieser Methode der folgende Code:

```
par p = 0 to 11
  group(p + 10 ≤ 20);
  for t = p + 10 to 20
    S_1 :
    A[p,t] := 42 · A[p + 1, t − 10];
    regroup (p + (t + 1) ≤ 20);
  killgroup
```

Diese Methode kann insbesondere bei Maschinen mit gemeinsamem Speicher vorteilhaft gegenüber erstgenannter Methode sein, da nicht immer das gesamte Prozessorfeld synchronisiert werden muß und sie keine Restriktionen auf die Eingabe erhebt. Wir werden später jedoch sehen, daß die erste Methode – die Erweiterung des Indexraums – zur Generierung von `send`- und `receive`-Anweisungen für Maschinen mit verteiltem Speicher ohnehin nötig ist.

6.2 LOAD und STORE

Wie schon erwähnt, unterscheidet sich bei uns (im Gegensatz zu HPF-Compilern) die Ausführungsreihenfolge im allgemeinen – für nicht strikt synchrone Programme – von der lexikographischen Ordnung auf den Indexvektoren. Wir betrachten dennoch zunächst den Fall, daß die Ausführungsreihenfolge mit dieser Ordnung zusammenfällt. In Abschnitt 6.3 werden wir dann den Fall von beliebig plazierten Parallelschleifen behandeln.

Auf Maschinen mit verteiltem Speicher muß der Zugriff auf ein Array i.a. durch eine oder mehrere Kommunikationsanweisungen implementiert werden, die das entsprechende Datum von einem anderen Prozessor in den lokalen Speicher kommunizieren. Callahan und Kennedy stellen in [CK88] ein Verfahren zum Umschreiben der Arrayzugriffe vor, das `LOAD`- und `STORE`-Anweisungen einfügt, um diese Kommunikation bei Beachtung der Owner Computes Rule durchzuführen. Dieses Verfahren, das hier kurz vorgestellt werden soll, bildet dort auch die Basis für die Parallelisierung eines Programms.

Die in [CK88] eingeführten Anweisungen `LOAD` und `STORE` haben (für nicht repliziert gespeicherte Arrays) die folgende Semantik:

```
procedure LOAD(A[I]: Array, tmp_A: lokales Array, pid: Int)
  if δ(I) = thisproc then
    if pid = thisproc then
      tmp_A := A[α(I)];
    else
      send A[α(I)] to pid
  else
    if pid = thisproc then
      receive tmp_A from δ(I);
```

```
procedure STORE(A[I]: Array, f(tmp_1,...,tmp_k): Ausdruck)
  if δ(I) = thisproc then
    A[α(I)] := f(tmp_1,...,tmp_k);
```

Dabei ist $\delta = P_V^{-1} \circ \Phi_A$ (mit P_V wie in Abschnitt 4.2) die Datenallokation des Arrays A, im Zielraum ausgedrückt durch die eindimensionale Prozessor-ID; *pid* ist eine Prozessor-ID, *thisproc* die Prozessornummer des abfragenden Prozessors und α wie in Abschnitt 3.3 die Funktion, die ein Element des globalen Datenraums eines Arrays auf das entsprechende Element aus dem lokalen Datenraum abbildet. Die Variablen tmp_A und $tmp_1, \ldots, tmp_k$ sind auf jedem Prozessor lokal gespeicherte (hier 0-dimensionale) Arrays, die den Wert des in der Berechnung verwendeten Array-Elements auf dem *Computer*-Prozessor enthalten.[11]

In [CK88] wird vor einer Zuweisung für jede Arrayreferenz auf der rechten Seite eine LOAD-Anweisung generiert und die Zuweisung selbst (nach Ersetzen der globalen Arrayzugriffe auf die eingefügten lokalen Arrays) durch eine entsprechende STORE-Anweisung ersetzt. Eine Zuweisung

$A[t,p] := A[t-1,p-1] + A[t-1,p+1];$

wird also – angenommen die OCR findet Anwendung – ersetzt durch das Programmfragment

LOAD($A[t-1,p-1]$, $tmp_{A[t-1,p-1]}$, $\Phi_A(t,p)$);
LOAD($A[t-1,p+1]$, $tmp_{A[t-1,p+1]}$, $\Phi_A(t,p)$);
STORE($A[t,p]$, $tmp_{A[t-1,p-1]} + tmp_{A[t-1,p+1]}$);

Wird dieses Programmfragment auf jedem Prozessor für jeden Indexvektor in der korrekten Reihenfolge ausgeführt – d.h. hier die durch die lexikographische Ordnung auf den Indexvektoren $\preceq$ definierte Reihenfolge –, gewährleistet dies eine korrekte Abarbeitung des Programms mit den entsprechenden Kommunikationen, die bei verteiltem Speicher nötig sind. Durch die Definition von STORE wird dabei die Owner Computes Rule forciert: die Zuweisung findet nur statt, wenn die STORE-Anweisung von dem *Owner* des zu bearbeitenden Datums aufgerufen wird. HPF-Compiler nutzen dies zur Parallelisierung, indem die Schleifengrenzen unter Berücksichtigung der if-Bedingungen in LOAD-/STORE-Anweisungen möglichst weit vereinfacht werden, so daß unnötige Schleifendurchläufe (bei denen die Auswertung der if-Anweisungen ergibt, daß keine Anweisungen auszuführen sind) verhindert werden; da das Resultat als SPMD-Programm ausgeführt wird, werden insgesamt alle Indexvektoren aufgezählt.

Allgemein müssen LOAD- und STORE-Anweisungen natürlich nicht nur für Zuweisungen, sondern entsprechend für jeden Lese- bzw. Schreibzugriff auf ein verteilt gespeichertes Array generiert werden.

Einen Sonderfall bilden dabei die Arrayzugriffe in der Bedingung einer while-Schleife: Wir wollen while-Schleifen als solche erhalten (und nicht etwa durch ein if und ein repeat-until ersetzen); dies bedeutet, daß die LOAD-Anweisungen für diese Arrayzugriffe **vor dem Ende** der while-Schleife[12] wiederholt werden müssen – und zwar **nach** einer ggf. stattfindenden Synchronisation. Sollte diese while Schleife

[11] Bei regulären Berechnungen kann für diese lokalen Arrays tmp_i bei der Deklaration des lokal zu speichernden Anteils des gesamten Arrays A ein über diesen durch die Allokation gegebenen Anteil hinausgehender Bereich, eine sogenannte Overlap-Area, eingerichtet werden (vgl. [ZBG88], [Su93]). Für die hier betrachteten allgemeinen Arrayzugriffe reicht dieser Ansatz jedoch nicht aus.

[12] Bei nichtaffinen for-Schleifen würde dies entsprechend gelten, wenn ihre Semantik wie in C definiert wäre (in C sind for-Schleifen zu while-Schleifen äquivalent, da die Abbruchbedingung in jeder Iteration neu berechnet wird); wir verwenden hier jedoch die Semantik von for-Schleifen wie in Pascal, wo die Anzahl der Schleifendurchläufe einmal zu Beginn der Schleife berechnet wird.

einen Zugriff enthalten, der in Speicherkonflikt mit einem in der Bedingung vorkommenden Zugriff steht (bzw. stehen kann), führt dies eine nicht *schleifengetragene* Abhängigkeit der beiden Zugriffe untereinander ein. Wir werden im folgenden einen konservativen Ansatz wählen und immer davon ausgehen, daß ein solcher Zugriff für alle Arrays aus der Bedingung existiert.

6.3 `LOAD` und `STORE` bei allgemeinen `par`-Schleifen

Wir haben im letzten Abschnitt gesehen, wie `LOAD`- und `STORE`-Anweisungen in den Programmcode eingefügt werden können, um eine korrekte parallele Abarbeitung des Programms nach der Owner Computes Rule bei verteiltem Speicher zu garantieren. In unserem Fall soll jedoch nicht die OCR forciert werden, sondern die durch die Parallelschleifen gegebene Anweisungsallokation.

Die Semantik einer `STORE`-Anweisung soll für uns von jetzt an also gegeben sein durch:

```
procedure STORE(A[I]: Array, f(tmp_1,...,tmp_k): Ausdruck, pid: Int)
  if δ(I) = thisproc then
    if pid = thisproc then
      A[α(I)] := f(tmp_1,...,tmp_k);
    else
      receive A[α(I)] from pid;
  else
    if pid = thisproc then
      send f(tmp_1,...,tmp_k) to δ(I)
```

D.h. eine `LOAD`-Anweisung sorgt dafür, daß ein Datum, das auf einem Prozessor benötigt wird, auf diesem Prozessor in einer „temporären" Variable zur Verfügung steht, egal, welcher Prozessor das Datum besitzt, und entsprechend sorgt eine `STORE`-Anweisung dafür, daß ein Wert, der einem Datum von irgendeinem Prozessor zugewiesen werden soll, aus einer „temporären" Variable dieses Prozessors gelesen und auf dem *Owner*-Prozessor dem Datum zugewiesen wird.

In unserer Eingabe werden die Berechnungen jedoch durch `par`-Schleifen auf die einzelnen Prozessoren verteilt. Und in Abschnitt 6.1 haben wir festgestellt, daß es im allgemeinen **nicht** möglich ist, eine Parallelschleife auf jedem Prozessor als sequentielle Schleife auszuführen (um für jeden Indexvektor die entsprechenden `LOAD`- und `STORE`-Anweisungen auf dem *Owner*-Prozessor ausführen zu können), da dies (falls in der Parallelschleife eine sequentielle Schleife geschachtelt ist) die Ausführungsreihenfolge auf illegale Weise abändert. Das *Projekt* aus Beispiel 6.1 etwa würde mit eingesetzten `LOAD`s und `STORE`s für die Datenallokation $\Phi_A : (i,j) \to 0$ als SPMD-Programm nach der Ersetzung der Parallelschleife folgendermaßen aussehen:

```
for p = 0 to 2
  for t = 0 to 2
    LOAD(A[t-1,p-1], tmp_A[t-1,p-1], p);
    LOAD(A[t-1,p+1], tmp_A[t-1,p+1], p);
    STORE(A[t,p], A[t-1,p-1] + A[t-1,p+1], p);
```

(Alle *pid*-Argumente haben also den Wert des Indexes, der im *Projekt* von der Parallelschleife durchgezählt wird). Da nun alle benötigten Daten grundsätzlich auf Prozessor 0 liegen, dieser jedoch die `LOAD`-Anweisungen in der Reihenfolge der sequentiellen Schleifen ausführt (also Punkt $(0,1)$ vor Punkt $(1,0)$) kann beispielsweise Prozessor 1 den Punkt $(1,0)$ erst durchführen, wenn Prozessor 0 Punkt $(0,1)$

ausgeführt hat; dieser Indexpunkt ist jedoch abhängig vom Punkt $(1, 0)$, so daß das Resultat nicht korrekt sein kann.

Zusätzlich ist dieser Ansatz für verschachtelte Arrayzugriffe ungeeignet, also für Arrayzugriffe der Form

$A[t, p] := A[A[p, t-1], t-1]$;

Denn bevor irgendein Prozessor die `LOAD`-Anweisung für $A[A[p, t-1], t-1]$ durchführen kann, muß er erst $A[p, t-1]$ laden. D.h. prinzipiell, daß jedem Prozessor jedes Element von A (bzw. ein dem Indexraum entsprechender Ausschnitt) bekannt sein muß (denn sowohl p als auch t muß auf jedem Prozessor variiert werden).

Im allgemeinen ist eine Vorhersage, welches Datum ein Prozessor braucht, also nicht möglich. Dennoch können `LOAD`- und `STORE`-Anweisungen mit `put`- und `get`-Funktionen realisiert werden, also Funktionen, die direkt auf den Speicher eines anderen Prozessors zugreifen. Solche Funktionen lassen sich realisieren durch einen auf jedem Prozessor zum übrigen Programm quasiparallel ablaufenden Thread der Form

```
procedure dserver(C: Kommandopuffer, M: Adresspuffer,
p: ID − Puffer, tmp: Datenpuffer)
  wait for request in (C, M, p, tmp)
  if C = "read" then
    tmp := α(M);
    send tmp to p
  else
    α(M) := tmp;
```

Dabei liefert $\alpha(M)$ hier die lokale Adresse des durch M definierten Datums. Werden auf jedem Prozessor lediglich die Indexpunkte wie in Abschnitt 4.3 beschrieben ausgeführt (also auf jedem Prozessor nur Indexvektoren aufgezählt, die bzgl. $\prec_S$ vergleichbar sind – weitere Indexpunkte wollten wir idealerweise nicht ausführen), kann die Funktion der `LOAD`s und `STORE`s dann also durch die Prozeduren `get` und `put` übernommen werden:

```
procedure get(M: globales Array − Element, tmp: lokales Array, p: Int)
  send ("read", M, thisproc, ·) to P_V^{-1} ∘ Φ(M)
  receive tmp from p;
```

```
procedure put(M: globales Array − Element, tmp: lokales Array, p: Int)
  send ("write", M, ·, tmp) to p
```

Dabei hat p – anders als bei der entsprechenden `LOAD`-Anweisung – grundsätzlich den Wert $\delta_{Name(M)}(Index(M)) = P_v^{-1} \circ \Phi_{Name(M)}(Index(M))$, als die Identifikationsnummer des *Owner*-Prozessors. Damit jederzeit aktuelle Werte der jeweiligen Variablen zur Verfügung stehen, muß hierbei sichergestellt sein, daß der Thread nach Ausführen einer `barrier`-Anweisung auf dem Prozessor alle evtl. noch ausstehenden Updates durchführt, bevor er neue Leseanfragen zuläßt. Die `put`-Anweisung wird dabei (da sowieso nur die Indexpunkte aufgezählt werden, für die die Zuweisung stattfinden soll) **hinter** einer Zuweisung plaziert. Die obige Zuweisung wird also umgeschrieben zu:

```
get A[t − 1, p − 1] from δ_A(t − 1, p − 1) into tmp_{A[t−1,p−1]};
get A[t − 1, p + 1] from δ_A(t − 1, p + 1) into tmp_{A[t−1,p+1]};
tmp_{A[t,p]} := tmp_{A[t−1,p−1]} + tmp_{A[t−1,p+1]};
put A[t, p] to δ_A(t, p) out of tmp_{A[t,p]};
```

6.4 Alternative Plazierungen für `get` und `put`

In Abschnitt 6.3 haben wir gesehen, wie gemeinsamer Speicher im allgemeinen Fall auf verteiltem Speicher simuliert werden kann. Dieser naive Ansatz hat jedoch eine Reihe von Schwachpunkten:

1. Wird ein Datum mehrfach gebraucht, werden hierfür auch mehrere `get`-Anweisungen generiert.
2. Es werden sehr viele sehr kleine Nachrichten verschickt; in der Praxis erweist sich dies als extrem langsam.
3. Für jede `get`-Operation ist auf dem diese Anweisung ausführenden Prozessor ein `send` und ein `receive` nötig.
4. Zur korrekten Verarbeitung der Anfragen ist auf jedem Prozessor ein eigener Thread nötig, sowie Synchronisation der Prozessoren untereinander.

Die Punkte 1–2 lassen sich durch Umsetzen der `get`- und `put`- Anweisungen verbessern, wie wir in diesem Abschnitt sehen werden. Auf die Punkte 3–4 werden wir in Abschnitt 6.5 eingehen.

Initiale Plazierung Wie schon beschrieben, muß bei der Umsetzung eines Programms auf verteilten Speicher vor jeden Lesezugriff (der dann durch einen Zugriff auf einen Puffer ersetzt wird) eine `LOAD`- bzw. `get`- Anweisung eingesetzt werden. Entsprechend muß nach jedem Schreibzugriff eine `STORE`- bzw. `put`-Anweisung eingesetzt werden. Dabei sind die `LOAD`- bzw. `get`-Anweisungen für einen Arrayzugriff M in absteigender Reihenfolge der maximalen Schachtelungstiefe von M in der entsprechenden Anweisung durchzuführen.

Beispiel 6.2 *Gegeben sei das* Projekt*:*

$\texttt{par}\ p = 1\ \texttt{to}\ 10$
$\quad\texttt{for}\ t = 1\ \texttt{to}\ 100$
$S_1:$
$\qquad A[p,C] := A[C,t-1] + B[p+t];$
$S_2:$
$\qquad A[p,t] := 2 \cdot A[p,t] + 2 \cdot B[p+t];$

Dann sind die `get`- *und* `put`-*Anweisungen* K_1–K_7 *folgendermaßen einzusetzen:*

$\texttt{par}\ p = 1\ \texttt{to}\ 10$
$\quad\texttt{for}\ t = 1\ \texttt{to}\ 100$
$K_1:$
$\qquad \texttt{get}\ C\ \texttt{from}\ \delta_C()\ \texttt{into}\ tmp_C;$
$K_2:$
$\qquad \texttt{get}\ A[tmp_C, t-1]\ \texttt{from}\ \delta_A(tmp_C, t-1)\ \texttt{into}\ tmp_{A[C,t-1]};$
$K_3:$
$\qquad \texttt{get}\ B[p+t]\ \texttt{from}\ \delta_B(p+t)\ \texttt{into}\ tmp_{B[p+t]};$
$S_1:$
$\qquad tmp_{A[p,C]} := tmp_{A[C,t-1]} + tmp_{B[p+t]};$
$K_4:$
$\qquad \texttt{put}\ A[p, tmp_C]\ \texttt{to}\ \delta_A(p, tmp_C)\ \texttt{out of}\ tmp_{A[p,C]};$
$K_5:$
$\qquad \texttt{get}\ A[p,t]\ \texttt{from}\ \delta_A(p,t)\ \texttt{into}\ tmp_{A[p,t]};$
$K_6:$
$\qquad \texttt{get}\ B[p+t]\ \texttt{from}\ \delta_B(p+t)\ \texttt{into}\ tmp_{B[p+t]};$

S_2 :

$\quad tmp_{A[p,t]} := 2 \cdot tmp_{A[p,t]} + 2 \cdot tmp_{B[p+t]}$;

K_7 :

$\quad$ `put` $A[p,t]$ `to` $\delta_A(p,t)$ `out of` $tmp_{A[p,t]}$;

In dem *Projekt* in Beispiel 6.2 muß die `get`-Anweisung K_1 vor der Anweisung K_2 ausgeführt werden, da der Arrayzugriff C in dem Arrayzugriff $A[p, C]$ geschachtelt vorkommt. Der Wert $B[p + t]$ wird zweimal verwendet – was zu zwei `get`-Anweisungen K_3 und K_6 führt.

Verschieben innerhalb eines *Basisblocks* Wie in [CK88] für die `LOAD`/`STORE`-Anweisungen festgestellt, können Optimierungsstrategien wie Common Subexpression Elimination und Code Motion auf `get`/`put`-Anweisungen genauso angewandt werden, wie auf jede andere Anweisung auch. Insbesondere ist es also möglich, diese Anwesiungen innerhalb eines *Basisblocks* zu verschieben. Hierfür können wir die folgende Regel aufstellen:

Regel 6.3 *Es gilt:*

1. *Eine* `get`*-Anweisung* `get` $M[i]$ `from` p `into` tmp_M *kann innerhalb eines* Basisblocks *so weit nach vorne verschoben werden, bis sie hinter einer Anweisung* K *steht mit:*
   (a) K *ist eine Anweisung* `put` $M[j]$ `to` y `out of` x, *wobei* i *und* j *den gleichen Wert annehmen können. Oder*
   (b) K *ist eine Anweisung* `get` N `from` y `into` x, *wobei* N *in* i *enthalten ist.*
2. *Eine* `put`*-Anweisung* `put` $M[i]$ `to` p `out of` tmp_M *kann innerhalb eines* Basisblocks *so weit nach hinten verschoben werden, bis sie vor einer Anweisung* K *steht mit:*
   (a) K *ist eine Anweisung* `get` $M[j]$ `from` y `into` x, *wobei* i *und* j *den gleichen Wert annehmen können. Oder*
   (b) K *ist eine Anweisung* `put` $M[j]$ `to` y `out of` x, *wobei* i *und* j *den gleichen Wert annehmen können.*

Damit gilt insbesondere folgende Regel:

Regel 6.4 *Existiert innerhalb eines* Basisblocks *keine nicht* schleifengetragene *Abhängigkeit, so können sämtliche*

1. `get`*-Anweisungen in der Reihenfolge der Schachtelungstiefe des betreffenden Arrayzugriffs am Anfang des* Basisblocks *ausgeführt werden,*
2. `put`*-Anweisungen in beliebiger Reihenfolge am Ende des* Basisblocks *ausgeführt werden.*

Die Bedingung der Regel 6.4 ist für alle von *LooPo* erzeugten *Projekte* gegeben, da die bei *LooPo* eingesetzten Scheduler und Allokatoren (vgl. [Wie95], [Mei96]) grundsätzlich zu *Projekten* führen, in denen alle Abhängigkeiten *schleifengetragen* sind. Ausnahmen bilden dabei Arrayzugriffe, die in der Bedingung einer `while`-Schleife bzw. den Grenzen einer (nichtaffinen) `for`-Schleife auftauchen: diese **müssen** am Ende der Schleife neu geladen werden – wird der Schleife ebenfalls ein Index zugeordnet, wie es auch bei `while`-Schleifen in [Gei97] der Fall ist, so müssen die `get`-Anweisungen **nach** dem Inkrementieren dieses Indexes stattfinden oder sich auf Indexausdrücke beziehen, in denen der Index i durch $i+1$ ersetzt ist: diese Anweisungen gehören vom Polyedermodell her schon zur nächsten Iteration der Schleife, müssen aber, da wir das `while`-Konstrukt so beibehalten wollen, noch vor Beendigung dieser Schleife ausgeführt werden. Ist es also möglich, die `get`- bzw. `put`-Anweisungen zu verschieben, so können dabei mehrere gleichartige `get`/`put`-Anweisungen durch eine einzige ersetzt werden:

Regel 6.5 *Zwei aufeinanderfolgende Anweisungen*

`get` M `from` p `into` tmp_M`;`
`get` M `from` p `into` tmp_M`;`

können ersetzt werden durch

`get` M `from` p `into` tmp_M`;`

Entsprechendes gilt für `put`.

Dies bedeutet – zusammen mit Regel 6.3 –, daß ein in einem Block von `get`- bzw. `put`-Anweisungen für jeden syntaktisch gleichen Arrayzugriff nur eine `get`- bzw. `put`-Anweisung vorhanden sein muß.

Somit können wir also unnötige Kommunikation vermeiden, indem `get`- und `put`-Anweisungen eliminiert werden, die sich auf den syntaktisch gleichen Arrayzugriff beziehen.

Verschieben zwischen Basisblöcken `get`- und `put`-Anweisungen zwischen gleichartig verschachtelten Basisblöcken A und B (d.h. Blöcken, deren Ausführung von den gleichen Kontrollanweisungen bestimmt werden), können `get`- und `put`-Anweisungen des Blocks A unter Beibehaltung ihrer Reihenfolge in Block B verschoben werden, falls sie sich nicht auf einen Arrayzugriff beziehen, für den eine nicht *schleifengetragene* Abhängigkeit zwischen A und einem Block C zwischen A und B (inklusive) besteht.

Herausziehen aus `for`-Schleifen Es ist unmittelbar einsichtig, daß sowohl `get` als auch `put`, falls ihre Argumente schleifeninvariant sind, sich aus einer sequentiellen Schleife herausziehen lassen, falls der bzw. die zugehörige(n) Arrayzugriffe nicht Ziel bzw. Quelle einer Abhängigkeit ist, deren sequentielle Komponente von dieser Schleife *getragen* wird. (Da der aktuelle Wert einer Variablen M auf dem berechnenden Prozessor immer in dem entsprechenden lokalen Array tmp_M zur Verfügung steht, kann diese Bedingung tatsächlich jedoch weiter abgeschwächt werden.)

Vektorisierung Auch wenn eine `get`- oder `put`-Anweisung nicht schleifeninvariant ist, kann sie über eine Schleife hinweggezogen werden, wenn dadurch nicht eine Abhängigkeit verletzt wird, deren sequentielle Komponente von dieser Schleife *getragen* wird. Dabei werden alle Array-Elemente kommuniziert, die während des Schleifendurchlaufs benötigt bzw. geschrieben werden. Hierfür wird statt eines Skalars ein ganzes Array als Puffer für die Kommunikationsanweisung verwendet. Diese Vektorisierung (vgl. [Wol95], [HKT91], [HKT92]) erhöht also die Größe der zu übertragenden Botschaften, verringert dafür aber in gleichem Maße die Häufigkeit der Übertragung, was aufgrund des Overheads für jede Kommunikationsanweisung einen Geschwindigkeitsgewinn zur Folge hat.

6.5 Vorausberechnung des Kommunikationspartners

Als ein Schwachpunkt der Kommunikation durch `get`- und `put`-Anweisungen wurde schon genannt, daß zur Implementation solcher Anweisungen auf dem Prozessor, der ein `get` ausführt, jeweils ein `send` und ein `receive` ausgeführt werden muß, und zusätzlich auf jedem Datenbesitzer (also prinzipiell auf jedem Prozessor) ein eigener Thread quasiparallel ablaufen muß, der die Anfragen verarbeitet. In diesem Abschnitt wollen wir – für reguläre Arrayzugriffe – solche Anweisungen durch explizite `send`/`receive`-Anweisungen ersetzen.

Wann ein Prozessor das `receive` einer `get`- bzw. `LOAD`-Anweisung oder das `send` einer `put`- bzw. `STORE`-Anweisung auszuführen hat und welcher Prozessor der Kommunikationspartner für diese Anweisung ist, ist klar: ausgeführt wird dieser Teil der Anweisung immer genau wie die `get`- oder `put`-Anweisung, und der Kommunikationspartner ist immer der *Owner* des Datums, der durch die Datenallokation fest vorgegeben ist. Die umgekehrte Richtung jedoch – die Anweisungen, die der *Owner* auszuführen hat – ist nicht ganz so klar: im allgemeinen kommt jeder Stufe I-Prozessor als *Owner* eines Datums in Betracht, wird jedoch nicht unbedingt von den umliegenden Parallelschleifen aufgezählt. Außerdem muß die Indexfunktion (und die Datenallokation) invertiert werden, um feststellen zu können, welcher Prozessor zu welchem Zeitpunkt welches Datum von dem gegebenen Prozessor benötigt. In Abschnitt 6.5.1 werden wir diese Berechnung des Kommunikationspartners eines Datenbesitzers beschreiben, um in dem Abschnitt 6.5.2 auf die hierfür notwendigen Programmtransformationen einzugehen.

Wir gehen dabei davon aus, daß **alle Schrittweiten** von `do`-Schleifen **positiv** sind. Ist dies für eine Schleife nicht der Fall, so ist die Untergrenze der Schleife als Obergrenze einzusetzen, die Schrittweite in ihren Absolutbetrag zu ändern und die Obergrenze wie folgt zu ersetzen:

Lemma 6.6 *Gegeben sei ein $m \in \mathbb{Z}$ und eine Funktion*

$$f : \mathbb{Z} \to \mathbb{Z} : i \to a \cdot i + b$$

Dann ist

$$n_0 = |a| \cdot \left\lfloor \frac{m-b-1}{|a|} \right\rfloor + |a| + b$$

minimal in $\mathrm{Bi}(f)$ *mit* $n_0 \geq m$ *und*

$$n_1 = |a| \cdot \left\lfloor \frac{m-b}{|a|} \right\rfloor + b$$

maximal in $\mathrm{Bi}(f)$ *mit* $n_1 \leq m$.

Dies ist unmittelbar einsichtig.

Die durch eine `do`-Schleife

`do` $i = l$ `to` u `step` s

wie in Abschnitt 4.1 definierte Abbildung

$$\Theta : i \to i \cdot s + l$$

ist in dem Fall einer negativen Schrittweite durch $l \in \mathbb{Z}$ und $\mathbb{Z} \ni s < 0$ definiert. Die die neue Schleife beschreibende Abbildung

$$\Theta' : i \to i \cdot |s| + l'$$

muß dabei lediglich die Bedingung

$$l' = \min(\{n \in \mathrm{Bi}(\Theta) \mid n \geq u\})$$

erfüllen. Damit berechnet sich nach Lemma 6.6 die neue Untergrenze aus

$$l' = |s| \cdot \left\lfloor \frac{u-l-1}{|s|} \right\rfloor + |s| + l$$

Die umgeformte Schleife mit positiver Schrittweite lautet also:

`do` $i = |s| \cdot \left\lfloor \frac{u-l-1}{|s|} \right\rfloor + |s| + l$ `to` l `step` $|s|$

Wir nehmen nun also an, daß alle Schrittweiten von `do`-Schleifen positiv sind.

6.5.1 Vorausberechnung bei *Modularabbildungen*

Für jeden Prozessor x und jeden Arrayzugriff $A[\varphi(i)]$ (mit $i \in \mathbb{Z}^{m+t}$) ist der *Owner* für das Datum, das bei dem Indexvektor i bearbeitet wird, gegeben durch $\Phi_A \circ \varphi(i)$. Ein Prozessor x, der Teile von A besitzt, muß nun ebenfalls alle Indexvektoren aus I aufzählen, für die Zugriffe auf den von ihm besessenen Teil des Arrays gemacht werden. Diese Menge ergibt sich aus

$$O(x) = (\Phi_A \circ \varphi)^{-1}(x) \cap I$$

Für jeden Vektor $x' \in O(x)$ ergibt sich dann der Kommunikationspartner (also der Prozessor, der den Zugriff auf das verteilte Array macht), aus der Projektion von x' auf die von `par`-Schleifen aufgespannten Dimensionen. Der Prozessor muß also lediglich für diese Indexvektoren den „*Owner*"-Teil der `LOAD`- bzw. `STORE`-Anweisung ausführen, während der Kommunikationspartner für diesen Arrayzugriff nun den anderen Teil dieser Anweisung (an Stelle eines `gets` bzw. `puts`) auszuführen hat. Der an diesen Prozessor zu sendende bzw. von ihm zu empfangene Wert ist $A[\varphi(c)]$.

Die Hintereinanderausführung von Indexfunktion und Datenallokation ist gegeben durch eine Funktion

$$\Phi_A \circ \varphi(i) : i \rightarrow \left\lfloor \frac{B \cdot i + k}{d} \right\rfloor \% p$$

Nun gilt klarerweise:

Lemma 6.7 *Gegeben sei eine Funktion*

$$F : \mathbb{Z}^w \rightarrow \mathbb{Z}^m : i \rightarrow G(i) \% p$$

mit $G : \mathbb{Z}^w \rightarrow \mathbb{Z}^m$. *Dann ist*

$$F^{-1}(x) = G^{-1}(x) + (\mathbb{Z} \cdot \pi_1(p), \ldots, \mathbb{Z} \cdot \pi_m(p))$$

Dies ist unmittelbar einsichtig.

Für diese Funktion G gilt hier wiederum

Lemma 6.8 *Gegeben sei eine Funktion*

$$G : \mathbb{Z}^w \rightarrow \mathbb{Z}^m : i \rightarrow \left\lfloor \frac{f(i)}{d} \right\rfloor$$

mit $f : \mathbb{Z}^w \rightarrow \mathbb{Z}^m$. *Dann gilt*

$$G^{-1}(x) = \bigcup_{j \in \{0, \ldots, d-1\}^w} f^{-1}(j + d \cdot x)$$

Dies ist unmittelbar einsichtig.

Damit ergibt sich für $O(x)$:

$$O(x) = \bigcup_{p' \in (x+\mathbb{Z} \cdot p_1, \ldots, \mathbb{Z} \cdot p_m)} \; \bigcup_{\overset{*}{p'} \in \{0, \ldots, d-1\}^w} (f^{-1}(p' \cdot d + \overset{*}{p'})) \cap I \qquad (34)$$

mit $f : \mathbb{Z}^w \to \mathbb{Z}^m : i \to B \cdot i + k$.

Da f eine affine Funktion ist, läßt sich das Urbild eines Punktes x unter f darstellen als eine spezielle Lösung l zusammen mit den Kern $\mathrm{Ke}(B)$ des linearen Teils von f: $f^{-1}(x) = l + \mathrm{Ke}(B)$. Dabei ist eine spezielle Lösung nach Abschnitt 2.2.2 gegeben durch $l = B^r \cdot (x - k)$. Ist die affine Abbildung durch eine Matrix D in homogenen Koordinaten gegeben, so ist eine spezielle Lösung gegeben durch $l = D^r \cdot x$.

Um die Menge der Indexpunkte zu ermitteln, bei denen ein Zugriff auf ein auf Prozessor x gespeichertes Datum stattfindet, genügt es also, für alle Punkte $x' \in O(x)$ auch auf x eine `LOAD`- bzw. `STORE`-Anweisung für den Zugriff mit Index $\varphi(x')$ auszuführen. Bei der Aufzählung von $O(x)$ ist jedoch darauf zu achten, daß auch die Menge $O(x)$ – genau wie I – in einer mit $\prec_S$ verträglichen Reihenfolge durchgezählt werden muß, um den durch die sequentiellen Schleifen gegebenen Schedule nicht zu verletzen. Dabei darf für die Indexpunkte aus $I \cap O(x)$ jedoch kein zusätzlicher Code erzeugt werden: diese Punkte müssen zur gleichen Zeit aufgezählt werden wie die anderen Punkte aus I, die auf x zu diesem Zeitpunkt aufzuzählen sind (d.h. es dürfen nicht zwei getrennte Mengen $\bigcup_{x \in P} O(x)$ und I aufgezählt werden, sondern eine Menge $\bigcup_{x \in P} O(x) \cup I$). Würden wir $O(x)$ exakt aufzählen, müßten hierfür sequentielle Schleifen generiert werden, deren Rumpf für die Bereiche, wo sich der lokal aufzuzählende Teil von $O(x)$ mit dem lokal aufzuzählenden Teil von I schneidet, mit dem Rumpf entsprechender, schon existierender sequentieller Schleifen zusammengefaßt wird. Dies kann beispielsweise dadurch zu Problemen führen, daß mehrere Prozessorkoordinaten in f abhängig von dem gleichen sequentiellen Index sein können, so daß auch die Schleifen für die obige Vereinigung über $p' \in x + \mathbb{Z} \cdot p$ aufeinander abgestimmt werden müßten.

Erster Ansatz Eine Methode, die auf jeden Fall zum Ziel führt, ist, auf jedem Prozessor grundsätzlich den **gesamten** Indexraum aufzuzählen, wobei – weil die Reihenfolge konform zur Striktordnung $\prec_S$ sein muß – alle Parallelschleifen (wie in Abschnitt 4.1) in den Rumpf evtl. darin geschachtelter sequentieller Schleifen zu verschieben und in `for`-Schleifen umzuwandeln sind (wobei Zuweisungen und Prozeduraufrufe im Schleifenrumpf mit den Parallelschleifen entsprechenden `if`s zu umgeben sind – durch diesen Vorgang wird aus dem *Projekt* ein Node-Programm). Die Anzahl tatsächlich aufzuzählender Indexpunkte kann dann mit exakt den Methoden verringert werden, wie sie bei der Parallelisierung von Schleifen durch `HPF`-Compiler angewandt werden (vgl. [Wol95], [ZBG88], [ZC93], [CK88], [CFH+92], [HKT91], [Su93], [SPB93]), wobei die Anweisungsverteilung schon durch die entsprechenden `if`-Anweisungen gegeben ist. Dieses Gebiet ist in vielen Arbeiten sehr gut untersucht worden. Dank diverser Optimierungsverfahren kann die Anzahl auszuführender Schleifendurchläufe in vielen wichtigen Fällen stark reduziert werden. Dennoch besteht der Nachteil, daß zunächst zumindest potentiell **alle** Iterationen auf jedem Prozessor ausgeführt werden müssen, obwohl der Benutzer für jede Iteration einer Parallelschleife genau **einen** Prozessor bestimmt hat, der diese Iteration durchführen soll.

„Scanning“ von O(x) Eine andere Methode besteht darin, zunächst zu versuchen, die Grenzen der Schleifen, die zum Aufzählen von $O(x)$ verwendet werden, schon bei der Generierung so zu wählen, daß möglichst wenige Punkte (möglichst nur $O(x)$) aufgezählt werden. Da es wie oben erwähnt jedoch problematisch ist, die von `for`-Schleifen aufgespannten Dimensionen von $O(x)$ mit dem übrigen Programmcode zu verschmelzen, werden wir auch in diesem Ansatz auf jedem Prozessor alle Zeitschritte aufzählen und lediglich für jeden Zeitschritt die von `par`-Schleifen aufgespannten Dimensionen $O_P(x)$ von $O(x)$ aufzählen. Das Prozessorfeld habe die Dimension m. Wir betten den Unterraum I_P von I, der aus den von `par`-Schleifen aufgespannten Dimensionen (u.U. weniger als m) gemäß Abschnitt 3.2 durch die Abbildung

$$\delta : I_P \rightarrow \mathbb{Z}^m : (i_1, \ldots, i_n) \rightarrow (i_1, \ldots, i_n, 0, \ldots, 0)$$

in $\mathbb{Z}^m$ ein und bezeichnen im folgenden mit I_P immer $\delta(I_P)$.

Wir fassen nun also die von `for`-Schleifen gebundenen Indizes als Parameter auf, so daß die Indexfunktion φ_P nun auf dem Unterraum I_P von I definiert ist, also

$$\varphi_P : I_P \subseteq \mathbb{Z}^m \rightarrow \mathbb{Z}^n$$

Die Hintereinanderausführung der Allokation mit dieser Indexfunktion sei gegeben durch

$$\Phi_A \circ \varphi_P : I_P \rightarrow \mathbb{Z}^m : i \rightarrow \left\lfloor \frac{B_P \cdot i + k}{d} \right\rfloor \% p$$

Die Punkte aus $O_P(x)$ müssen auf jedem Prozessor in der gleichen Reihenfolge aufgezählt werden, da es sonst möglich wäre, daß zwei Prozessoren, die beide Daten voneinander brauchen, beide zuerst versuchen, das entsprechende Datum an den jeweils anderen Prozessor zu schicken und dann erst zu empfangen. Man beachte, daß diese Notwendigkeit **nur** dann besteht, wenn **keine** asynchrone `send`-Anweisung zur Verfügung steht. Außerdem muß auch der Punkt $x \in O_P(x)$ aufgezählt werden (falls $x \in I_P$). In Abschnitt 5.3 haben wir gesehen, daß das Urbild eines Punktes unter einer *Modularabbildung* aus einer Menge von *Line Segments* unterschiedlicher Länge besteht. Intuitiv formuliert entsteht dabei eine „Lücke" zwischen zwei *Line Segments* dadurch, daß der Wert, auf den das nächste Element durch den affinen Anteil der Abbildung abgebildet wird, auf eine andere Zahl gerundet wird. Es liegt also nahe, $O_P(x)$ durch zwei Schleifensätze aufzuzählen:

1. Der erste Schleifensatz zählt die Bildpunkte

$$p' \in \begin{pmatrix} x_1 + i_1 \cdot p_1 \\ \vdots \\ x_m + i_m \cdot p_m \end{pmatrix} \cap \left\lfloor \frac{B_P(I_P)+k}{d} \right\rfloor$$

von $G(I_P)$ auf (und kehrt damit die %-Operation um); jede Schleife dieses Schleifensatzes zählt damit für die entsprechende Prozessordimension j die verschiedenen *Line Segments* des Urbildes auf (identifiziert durch p'_j mit $p'_j = G_{P,j}(LS)$, wobei LS ein *Line Segment*, $G_P = (G_{P,1}, \ldots, G_{P,m})$). Begrenzt wird der erste Schleifensatz durch die Extremalwerte von $G_P(i) = \left\lfloor \frac{B_P(i)+k}{d} \right\rfloor$, die für $i \in I_P$ erreicht werden können. Dabei müssen nur alle p_j Werte aufgezählt werden.

2. Der zweite Schleifensatz zählt für jeden Vektor $p' = (p'_1, \ldots, p'_m)$ alle möglichen Urbildpunkte

$$x' \in \bigcup_{\overset{*}{p'} \in \{0, \ldots, d-1\}^m} \left\lfloor B_P^r \cdot (d \cdot p' + \overset{*}{p'} - k) \right\rfloor + \mathrm{Ke}(B_P)$$

unter G auf; damit werden also alle zusammenhängenden Elemente eines *Line Segments* aufgezählt (falls die entsprechende Dimension nicht eine Komponente eines Kernvektors ist). Begrenzt wird der zweite Schleifensatz durch die jeweiligen Komponenten von $\lceil B_P^r \cdot (d \cdot p' - k) \rceil$ und

$\lfloor B_P^r \cdot (d \cdot p' + (d-1, \ldots, d-1) - k) \rfloor$ in Dimensionen, zu denen $\mathrm{Ke}(B_P)$ orthogonal liegt, und $(0, \ldots, 0)$ bzw. der Anzahl der Prozessoren in den anderen Dimensionen.

Der zweite Schleifensatz zählt also für jeden Zeitschritt und jeden Punkt p' die Punkte $x' \in O_P(x)$ auf. Man beachte, daß hierfür bei m umliegenden `par`-Schleifen um die `LOAD/STORE`-Anweisung lediglich m Prozessorkoordinaten aufzuzählen sind – dies ist u.U. geringer als die Dimensionalität des Prozessorfeldes: höhere Koordinaten sind nach Konvention in Abschnitt 3.2 auf 0 zu setzen.

Betrachten wir wieder die durch eine `do`-Schleife

`do` $i = l$ `to` u `step` s

gegebene Abbildung

$$\Theta : i \rightarrow i \cdot s + l$$

so wird diese Abbildung für die j-te Schleife des ersten Schleifensatzes durch folgende Bedingungen definiert:

$$s = p_i \tag{35}$$

$$\Theta(0) \geq \pi_j(\mathit{extremize}_x(\left\lfloor \tfrac{B_P(x)+k}{d} \right\rfloor, \min)) \tag{36}$$

$$(\exists i_0 : i_0 \in \mathbb{Z} : \Theta(i_0) = x_j) \tag{37}$$

Bedingung (37) entspricht der Forderung

$$\Theta : i \rightarrow f(i) + f_{\mathit{off}}$$

mit $f : i \rightarrow s \cdot i + x_j$ und $f_{\mathit{off}} | s$. Um eine Schleife generieren zu können, müssen wir nur noch die untere Grenze kennen, die in diesem Fall $f_{\mathit{off}} + x_j$ beträgt. Diese ergibt sich als das kleinste Element l_0 aus $\mathrm{Bi}(f)$, mit $l_0 \geq \mathit{extremize}_x(\pi_j(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor), \min)$. Dabei extremalisiert $\mathit{extremize}_x$ lediglich nach x (also den von Parallelschleifen gebundenen Indizes). Nach Lemma 6.6 ist die untere Grenze also:

$$p_j \cdot \left\lfloor \frac{\mathit{extremize}_x(\left\lfloor \pi_j(\frac{B_P(x)+k}{d}) \right\rfloor, \min) - x_j - 1}{p_j} \right\rfloor + p_j + x_j$$

Die Obergrenze ergibt sich entsprechend zu:

$$p_j \cdot \left\lfloor \frac{\mathit{extremize}_x(\left\lfloor \pi_j(\frac{B_P(x)+k}{d}) \right\rfloor, \max) - x_j}{p_j} \right\rfloor + x_j$$

Allerdings genügt es hier, da die Differenz zwischen Obergrenze und Untergrenze einer Schleife nicht durch die Schrittweite der Schleife teilbar sein muß, die Obergrenze

$$\mathit{extremize}_x(\left\lfloor \pi_j(\tfrac{B_P(x)+k}{d}) \right\rfloor, \max)$$

zu verwenden.

Entsprechend ist für eine das `LOAD`- bzw. `STORE` umgebende Parallelschleife

`par` $p_j = L_j$ `to` U_j `step` s_j

im zweiten Schleifensatz eine Schleife zu generieren, die einer Funktion $\Theta : i \rightarrow i \cdot a + l$ entspricht mit:

$$a = s_j \tag{38}$$
$$\Theta(0) \geq \lceil \pi_j(B_P^r \cdot d \cdot p' - k) \rceil \tag{39}$$
$$(\exists i_0 : i_0 \in \mathbb{Z} : \Theta(i_0) = L_j[x/x']) \tag{40}$$

Die untere Schleifengrenze (für Dimensionen, die senkrecht zum Kern von B_P stehen) berechnet sich demnach aus

$$s_j \cdot \left\lfloor \frac{\lceil \pi_j(B_P^r \cdot d \cdot p' - k) \rceil - L_j[x/x']}{s_j} \right\rfloor + s_j + L_j[x/x']$$

Qualitativ bedeutet dies, daß (für $\mathrm{Ke}(B_P) = \{(0, \ldots, 0)\}$ und `par`-Schleifen mit Grenzen $L_j \preceq U_j$ und Schrittweiten s_j) ein Schleifensatz der folgenden Form generiert werden müsste:

$$\texttt{for } p'_1 = p_1 \cdot \left\lfloor \frac{\mathit{extremize}_x(\pi_1(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \min)) - x_1 - 1}{p_1} \right\rfloor + p_1 + x_1$$
$$\texttt{to } \mathit{extremize}_x(\pi_1(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \max)) \texttt{ step } p_1$$
$$\vdots$$
$$\texttt{for } p'_m = p_m \cdot \left\lfloor \frac{\mathit{extremize}_x(\pi_m(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \min)) - x_m - 1}{p_m} \right\rfloor + p_m + x_m$$
$$\texttt{to } \mathit{extremize}_x(\pi_m(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \max)) \texttt{ step } p_m$$
$$\texttt{for } x'_1 = s_1 \cdot \left\lfloor \frac{\pi_1(\lceil B_P^r \cdot d \cdot p' - k \rceil) - L_1[x/x']}{s_1} \right\rfloor + s_1 + L_1[x/x']$$
$$\texttt{to } \min(U_1[x/x'], \pi_1(\lfloor B_P^r \cdot (p'_1 \cdot d + d - 1, p'_2 \cdot d, \ldots, p'_m \cdot d) - k \rfloor)) \texttt{ step } s_1$$
$$\vdots$$
$$\texttt{for } x'_m = s_m \cdot \left\lfloor \frac{\pi_m(\lceil B_P^r \cdot d \cdot p' - k \rceil) - L_m[x/x']}{s_m} \right\rfloor + s_m + L_m[x/x']$$
$$\texttt{to } \min(U_m[x/x'], \pi_m(\lfloor B_P^r \cdot (p'_1 \cdot d, \ldots, p'_{m-1} \cdot d, p'_m \cdot d + d - 1) - k \rfloor)) \texttt{ step } s_m$$
$$\texttt{/* kommuniziere mit } x' \texttt{ */}$$
$$\vdots$$
$$\vdots$$

Man beachte, daß auch dieses Verfahren nicht garantieren kann, daß alle Iterationen tatsächlich benötigt werden: Im Schleifeninneren muß u.U. abgeprüft werden, ob der ermittelte Wert aus I ist, da die entsprechende Anweisung grundsätzlich auch durch eine oder mehrere `if`-Anweisungen umgeben sein kann. Außerdem werden auch bei dieser Lösung grundsätzlich schon zu viele Möglichkeiten im zweiten Schleifensatz aufgezählt, da hierfür die spezielle Lösung des linearen Anteils l auch rationale Werte annehmen kann, also nicht aus I stammt; andererseits kann es aber auch für ein solches l noch ein $z \in \mathbb{Q} \cap \mathrm{Ke}(B_P)$ geben mit $l + z \in I$. Darum müssen für die Prozessorkoordinaten, die nicht orthogonal zum Kern liegen, alle möglichen Werte aufgezählt werden und – falls der Kern nichttrivial ist – die Bedingung

$$p' = \left\lfloor \frac{B_P \cdot x' + k}{d} \right\rfloor \quad \text{bzw.} \quad x = \left\lfloor \frac{B_P \cdot x' + k}{d} \right\rfloor \% p$$

abgeprüft werden.

Damit können wir nun für jeden Prozessor x die Menge seiner Kommunikationspartner $O_P(x)$ aufzählen; falls B_P einen trivialen Kern besitzt ist diese Aufzählung auch exakt, d.h. es gibt keine überflüssigen Iterationen. Steht auf dem Parallelrechner jedoch keine asynchrone Kommunikation zur Verfügung, so muß $O_P(x)$, wie schon erwähnt, zusätzlich auf jedem Prozessor in der gleichen Reihenfolge aufgezählt werden.

Spaltentransformation Um auf jedem Prozessor die Vektoren $x' \in O_P(x)$ in der gleichen Reihenfolge aufzuzählen, soll die Reihenfolge der Aufzählung durch die lexikographische Ordnung auf $O_P(x)$ gegeben werden. Der erste Schleifensatz kann in jedem Fall auf jedem Prozessor in der gleichen Reihenfolge aufgezählt werden. Jedoch muß B_P^r als Funktion nicht unbedingt monoton sein. D.h. die lexikographisch aufsteigende Folge von p' garantiert noch nicht die gleiche Ordnung von Vektoren $x' \in O_P$ auf jedem Prozessor. Hat die Matrix B_P^r jedoch die Form aus Satz 6.9, ist diese Eigenschaft gewährleistet:

Satz 6.9 *Sei $\mathbb{Q}^{m\times m} \ni A = (a_{i,j})$ eine Matrix in Spaltenstufenform (wie in Abschnitt 5.3.3), so daß*

$$(\forall j : j \in \{1, \ldots, m\} : a_{\Lambda(j),j} > 0)$$

wobei $\Lambda : \mathbb{N} \to \mathbb{N} : j \to \min(\{i \mid a_{i,j} \neq 0\})$. Dann ist die durch A gegebene lineare Abbildung monoton wachsend bzgl. der lexikographischen Ordnung $\preceq$ und bzgl. der ersten $\mathrm{rang}(A)$ *Komponenten streng monoton wachsend bzgl. der lexikographischen Striktordnung $\prec$.*

Beweis:

1. Sei $m = 1$. Dann ist die Behauptung klar.
2. Sei $m > 1$. Dann hat die durch A gegebene lineare Abbildung aufgrund der Spaltenstufenform von A die Form

   $$A : \mathbb{Q}^m \to \mathbb{Q}^m : i = (i_1, \ldots, i_m) \to \begin{pmatrix} A_{m-1}(i_1, \ldots, i_{m-s}) \\ A_m \end{pmatrix}$$

   wobei $A_{m-1} : \mathbb{Q}^{m-s} \to \mathbb{Q}^{m-1}$ eine monoton wachsende (bzgl. der ersten $m-s$ Komponenten streng monoton wachsende) lineare Abbildung ist und

   $$A_m : \mathbb{Q}^m \to \mathbb{Q} : i \to (\sum j : j \in \{1, \ldots, m\} : a_{m,j} \cdot \pi_j(i))$$

   wobei $a_{m,m-s} \geq 0$ und $j > m - s \Rightarrow a_{m,j} = 0$.
   Seien nun $(i_1, \ldots, i_m) \preceq (l_1, \ldots, l_m) \in \mathbb{Q}^m$. Gilt $(i_1, \ldots, i_{m-s}) \prec (l_1, \ldots, l_{m-1})$, so gilt nach Voraussetzung auch

   $$A_{m-1}((i_1, \ldots, i_{m-s})) \prec A_{m-1}((l_1, \ldots, l_{m-s}))$$

   und darum

   $$A_m((i_1, \ldots, i_{m-1})) \prec A_m((l_1, \ldots, l_{m-1}))$$

   Gilt also $(i_1, \ldots, i_{m-s}) = (l_1, \ldots, l_{m-s})$, so gilt wegen $a_{m,m-s} \geq 0$ auch

   $$A_m(i) \preceq A_m(j)$$

   Für $a_{m,m-s} > 0$ (dies ist der Fall für $\mathrm{rang}(A) \geq m - s$) gilt sogar

   $$A_m((i_1, \ldots, i_m)) < A_m((l_1, \ldots, l_m))$$

Und wegen der Invarianz der ersten $m-1$ Komponenten von A bzgl. i_{m-s} bzw. j_{m-s} gilt damit

$$A((i_1,\ldots,i_m)) \preceq A((l_1,\ldots,l_m))$$

✓

Ist also B_P^r in einer solchen Spaltenstufenform gegeben, werden bei dieser Strategie mit p' auch die Vektoren $x' \in O_P(x)$ in lexikographischer Ordnung aufgezählt. Aus der Linearen Algebra wissen wir aber, daß sich jede Matrix A durch eine Spaltentransformation mittels einer unimodularen Matrix R in eine Matrix $A \cdot R$ in Spaltenstufenform bringen läßt. Durch weitere unimodulare Spaltentransformationen (Multiplikation aller Spalten j, für die $a_{\Lambda(j),j} < 0$ gilt, mit -1) kann auch die zusätzliche Bedingung aus Satz 6.9 durch $B_P^r \cdot R$ für eine Spaltentransformation R erfüllt werden. Dies bedeutet aber, daß im ersten Schleifensatz nicht mehr ein $L \leq p' \leq U$ aufgezählt werden muß, sondern ein p'', zwischen den Grenzen $R^{-1} \cdot L$ und $R^{-1} \cdot U$; hierbei sind ggf. untere und obere Grenze einer Dimension zu vertauschen. Um den Indexraum der Schleife bei der Vertauschung nicht zu ändern, muß hierbei aber die Differenz zwischen Ober- und Untergrenze der Schleife ein Vielfaches der Schrittweite sein. Deshalb sind die Werte für U hier – anders als auf Seite 76 – direkt aus Lemma 6.6 zu ermitteln. Wegen der Unimodularität von R und damit R^{-1} bleiben die Werte, die p' durchlaufen muß, ganzzahlig (können also weiterhin durch eine `for`-Schleife aufgezählt werden). Allerdings ist für diesen Schleifensatz die Schrittweite anzupassen, da nun eine Linearkombination von Prozessorkoordinaten aufgezählt werden muß: die neue Schrittweite für eine Schleife der Tiefe $j \in \{1,\ldots,m\}$ ergibt sich mit $R^{-1} = (r'_{i,j})$ zu

$$\text{ggT}(\{r'_{j,q} \cdot p_q \mid q \in \{1,\ldots,m\}\})$$

Durch diese neue Schrittweite können auch Werte für p'' aufgezählt werden, die nicht im Bild von $i \to R^{-1} \cdot \left\lfloor \frac{B_P \cdot i + k}{d} \right\rfloor$ liegen. Hierfür müssen für alle $j \in \{1,\ldots,m\}$ auch die Bedingungen

$$\pi_j(R \cdot p'') \in \mathbb{Z} \tag{41}$$

$$(\pi_j(R \cdot p'') - L_{p',j}) \% p_j = 0 \tag{42}$$

gelten, wobei $L_{p',j}$ die untere Grenze für p'_j ist. Bedingung (41) gilt aufgrund der Unimodularität von R. Bedingung (42) ist dagegen (frühestens) zwischen dem ersten und dem zweiten Schleifensatz abzuprüfen, falls diese Bedingung nicht ohnehin garantiert werden kann. Garantiert wird sie etwa dadurch, daß $\pi_j(R \cdot R^{-1}) = e_j$ – genauer ist die Bedingung (42) genau dann abzuprüfen, wenn

$$max(\{r'_{j,q} \cdot p_q \mid q \in \{1,\ldots,m\}\}) \neq \text{ggT}(\{r'_{j,q} \cdot p_q \mid q \in \{1,\ldots,m\}\} \setminus \{0\})$$

Da $B_P^r \cdot R$ in Spaltenstufenform und damit invariant bzgl. der letzten $m - \text{rang}(B_P^r)$ Komponenten eines Eingabevektors ist, genügt es aber, für alle $j > \text{rang}(B_P^r)$ lediglich $\pi_j(p'') = 0$ anzunehmen und p'' nicht in diesen Komponenten aufzuzählen.

Die endgültigen Koordinaten der *Computer*-Prozessoren ergeben sich dann in denen Dimensionen, die nicht orthogonal zum Kern von B_P liegen, durch den gesamten aktiven Bereich des Prozessorfeldes. Entsprechend wie bei den p''-Werten, ist es auch hier möglich, x'-Werte aufzuzählen, die nicht zum Urbild unter $\Phi_A \circ \varphi_P$ gehören – beispielsweise würde ein Basisvektor $(1,\ldots,1)$ dafür sorgen, daß das **gesamte** Prozessorfeld aufgezählt würde. Darum muß auch hier – für solche Komponenten – überprüft werden, ob diese Prozessorkoordinate tatsächlich zum Urbild von $\Phi_A \circ \varphi_P$ für den aktuellen p''-Wert gehört.

Für Koordinaten j, die orthogonal zum Kern liegen, kann der genaue Bereich für x_j berechnet werden, der durchlaufen werden muß; beginnt bei der aktuellen

Koordinate keine neue Stufe in $B_P \cdot R$ oder teilt d alle Matrixeinträge in dieser Zeile von $B_P \cdot R$, so gibt es nur einen möglichen Wert für x_j:

$$x'_j = s_j \cdot \left\lfloor \frac{\lceil B_P^r \cdot R \cdot d \cdot p'' - k \rceil - L_j[x/x'] - 1}{s_j} \right\rfloor + s_j + L_j[x/x']$$

Allerdings ist in diesem Fall zu überprüfen, ob dieser Wert tatsächlich einen aktiven Prozessor bezeichnet. Ansonsten müssen alle ganzzahligen Werte aus dem oben beschriebenen Bereich durchlaufen werden (wobei auch hier nicht über die Grenze der aktiven Prozessoren hinaus gezählt werden darf).

In jedem Fall müssen für alle möglichen Werte für Kommunikationspartner noch die Bedingungen aus umgebenden `if`-Anweisungen abgefragt werden.

Prinzipiell ist also für $\mathbb{Z}^m \ni L_{p'} = (L_{p',j})_{j \in \{1,\ldots,m\}}$ und $\mathbb{Z}^m \ni U_{p'} = (U_{p',j})_{j \in \{1,\ldots,m\}}$ mit

$$L_{p',j} = p_j \cdot \left\lfloor \frac{extremize_x(\pi_j(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \min)) - x_j - 1}{p_j} \right\rfloor + p_j + x_j$$

und

$$U_{p',j} = p_j \cdot \left\lfloor \frac{extremize_x(\left\lfloor \pi_j(\frac{B_P(x)+k}{d}) \right\rfloor, \max) - x_j}{p_j} \right\rfloor + x_j$$

ein Schleifensatz der folgenden Form zu generieren:

`for` $p''_1 = \min(\pi_1(R^{-1} \cdot L_{p'}), \pi_1(R^{-1} \cdot U_{p'}))$
 `to` $\max(\pi_1(R^{-1} \cdot L_{p'}), \pi_1(R^{-1} \cdot U_{p'}))$ `step` $\mathrm{ggT}(\{r'_{1,q} \cdot p_q \mid q \in \{1,\ldots,m\}\} \setminus \{0\})$
 $\vdots$
  `for` $p''_m = \min(\pi_m(R^{-1} \cdot L_{p'}), \pi_m(R^{-1} \cdot U_{p'}))$
 `to` $\max(\pi_m(R^{-1} \cdot L_{p'}), \pi_m(R^{-1} \cdot U_{p'}))$ `step` $\mathrm{ggT}(\{r'_{m,q} \cdot p_q \mid q \in \{1,\ldots,m\}\} \setminus \{0\})$
   `/*` m `Gleichungen! */`
   `if` $(R \cdot p'' - L_{p'})\%p = 0$ `then`
    `for` $x'_1 = s_1 \cdot \left\lfloor \frac{\pi_1(\lceil B_P^r \cdot R \cdot d \cdot p'' - k \rceil) - L_1[x/x'] - 1}{s_1} \right\rfloor + s_1 + L_1[x/x']$
 `to` $\min(U_1[x/x'], \pi_1(\lfloor B_P^r \cdot R \cdot (p''_1 \cdot d + d - 1, p''_2 \cdot d, \ldots, p''_m \cdot d) - k \rfloor))$ `step` s_1
    $\vdots$
     `for` $x'_m = s_m \cdot \left\lfloor \frac{\pi_m(\lceil B_P^r \cdot R \cdot d \cdot p'' - k \rceil) - L_m[x/x'] - 1}{s_m} \right\rfloor + s_m + L_m[x/x']$
 `to` $\min(U_m[x/x'], \pi_m(\lfloor B_P^r \cdot R \cdot (p''_1, \ldots, p''_{m-1} \cdot d, p''_m \cdot d + d - 1) - k \rfloor))$ `step` s_m
      `/*` m `Gleichungen! */`
      `if` $\left\lfloor \frac{B_P \cdot x' + k}{d} \right\rfloor \%p = x$ `then`
       `/* kommuniziere mit` x' `*/`
    $\vdots$
 $\vdots$

Betrachten wir einen reinen *Owner*-Prozessor, der selbst keine Berechnungen durchführt, so genügt es prinzipiell auch, lediglich die Codesequenzen aus den der Kommunikation entsprechenden `LOAD`- oder `STORE`-Anweisungen im Rumpf dieser verschachtelten `for`- und `if`-Anweisungen auszugeben, statt der vollständigen `LOAD`s bzw. `STORE`s.

Insgesamt kann eine solche Codesequenz, die eine `get`- bzw. `put`-Anweisung durch eine `LOAD`- bzw. `STORE`-Anweisung auf einem reinen *Owner*-Prozessor ersetzt, durch den folgenden Algorithmus 6.10 erstellt werden. Dabei wird schon durch die Tatsache, daß diese Anweisung im *Ownerpart* eines Programmsegmentes steht, die Bedingung

$$pid \neq thisproc$$

erfüllt; da der ausführende Prozessor garantiert *Owner* des Datums ist, ist zusätzlich

$$\delta(I) = thisproc$$

erfüllt – es wird also lediglich die `send`-Anweisung eines `LOAD`s bzw. die `receive`-Anweisung eines `STORE`s ausgeführt. Solche Einschränkungen, die bei der Codegenerierung beachtet werden können, notieren wir im folgenden für $\Gamma \in \{\texttt{LOAD}, \texttt{STORE}\}$ und $\approx \in \{=, \neq\}$ durch:

$$^{\approx}\Gamma, \text{ falls } \delta(I) \approx thisproc \text{ gilt}$$
$$_{\approx}\Gamma, \text{ falls } pid \approx thisproc \text{ gilt}$$

Algorithmus 6.10

Eingabe: Anweisung $\Gamma \in \{\texttt{get}, \texttt{put}\}$ für Arrayzugriff $A[\varphi]$ mit
$\Phi_A \circ \varphi : i \rightarrow \left\lfloor \frac{B_P(i)+k}{d} \right\rfloor \% p$ (Spaltenstufenform von B_P^r wie in Satz 6.9
$(B_P^r \cdot R) = (b_{i,j}^R)$, $R^{-1} = (r'_{i,j})$),
aus `par`-Schleifen stammende
Indizes $(x_1, \ldots, x_m)$ begrenzt durch $(L_1, \ldots, L_m) \leq x \leq (U_1, \ldots, U_m)$
mit Schirttweiten $(s_1, \ldots, s_m)$ (dabei ist $L_j, U_j = 0$ für Dimensionen j,
die nicht von umgebenden Parallelschleifen aufgespannt werden),
Bedingungen C_A aus umgebenden `if`- und `else`-Bedingungen

Ausgabe: Schleifensätze zum Aufzählen von Kommunikationspartnern
in lexikographischer Reihenfolge und
Senden bzw. Empfang durch `LOAD`- bzw. `STORE`-Anweisung auf reinem
Owner-Prozessor

$\mu := extremize_x(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, min)$;
$M := extremize_x(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, max)$;
`for` $i = 1$ `to` m
  $L_{p',i} :=$ ‘‘$p_i \cdot \left\lfloor \frac{\pi_i(\mu)-x_i-1}{p_i} \right\rfloor + p_i + x_i$‘‘;
  $U_{p',i} :=$ ‘‘$p_i \cdot \left\lfloor \frac{\pi_i(M)-x_i}{p_i} \right\rfloor + x_i$‘‘;
`/* verwende` $p''_i = 0$ `bei Multiplilaktionen statt */`
`/* symbolischen Wert` p''_i`! */`
`for` $i = \text{rang}(B_P^r) + 1$ `to` m
  $p''_i := 0$;
$Cond :=$ ’’true’’;
`for` $i = 1$ `to` $\text{rang}(B_P^r)$
  `/* symbolische Matrixmultiplikation */`
  `emit ‘‘for` $p'' = min(\pi_i(R^{-1} \cdot L_{p'}), \pi_i(R^{-1} \cdot U_{p'})$
 `to` $max(\pi_i(R^{-1} \cdot L_{p'}), \pi_i(R^{-1} \cdot U_{p'}))$
 `step` $\text{ggT}(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\} \setminus \{0\})$’’;
  `if` $\text{ggT}(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\} \setminus \{0\}) \neq \max(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\})$ `then`

```
    Cond := Cond + ‘‘ ∧((π_i(R · p'') − L_{p',i})%p_i = 0’’;
if Cond ≠ ‘‘true’’ then
  emit ‘‘if Cond then’’;
/* lastCol ist die Nummer der neu */
/* zu beginnenden Stufe in B_P · R */
lastCol := 1;
Cond := ‘‘true‘‘;
for i = 1 to m
  if (∃K : K ∈ Span(Ke(B_P)) : π_i(K) ≠ 0) then
    emit ‘‘for x'_i = L_i to U_i step s_i’’;
    Cond := Cond + ‘‘ ∧π_i(⌊(B_P·x'+k)/d⌋%p) = x_i)‘‘;
  else
    if (b^R_{i,lastCol} = 0) ∨ (d = ggT({d, b^R_{i,1}, ..., b^R_{i,lastCol}} \ {0})) then
      emit ‘‘x'_i := s_i · ⌊(π_i(⌈B^r_P · R · d · p'' − k⌉) − L_i[x/x'] − 1)/s_i⌋ + s_i + L_i[x/x'];’’;
      emit ‘‘if x'_i ≤ U_i[x/x'] then’’;
    else
      emit ‘‘for x'_i = s_i · ⌊(π_i(⌈B^r_P · R · d · p'' − k⌉) − L_i[x/x'] − 1)/s_i⌋ + s_i + L_i[x/x']
 to min(U_i[x/x'], π_i(⌊B^r_P · R · (p''_1 · d, ..., p''_lastCol · d + d − 1, ..., p''_m · d) − k⌋)) step s_i’’;
  if b^R_{i,lastCol} ≠ 0 then
    lastCol := lastCol + 1;
if Cond ≠ ‘‘true‘‘ then
  emit ‘‘if Cond then’’;
if C_A ≠ ∅ then
  emit ‘‘if C_A[x/x'] then’’;
if Γ = get then
  emit ‘‘=/≠LOAD(A[φ], tmp_A[φ], x')’’;
else
  emit ‘‘=/≠STORE(A[φ], tmp_A[φ], x')’’;
if C_A ≠ ∅ then
  emit ‘‘endif’’;
if Cond ≠ ‘‘true‘‘ then
  emit ‘‘endif’’;
emit Endmarkierungen für alle geöffneten fors und ifs;
```

Es gibt jedoch auch Prozessoren, die nicht nur *Owner* eines Datums sind, sondern **zusätzlich** auch Daten benötigen, um selbst Berechnungen durchführen zu können. Dennoch müssen auch diese Prozessoren u.U. Daten an andere Prozessoren verschicken. Der Schleifensatz für solche Prozessoren unterscheidet sich also von dem bisherigen lediglich dadurch, daß auch der Wert x in der x'-Schleife aufgezählt werden muß. Dies läßt sich dadurch realisieren, daß eine Boolesche Variable eingeführt wird, die anzeigt, ob der Wert x schon erreicht wurde; falls dies nicht der Fall ist, x' jedoch schon einen lexikographisch mindestens so hohen Wert hat, muß die `LOAD`- bzw. `STORE`-Anweisung für den Wert x ausgeführt werden, bevor sie für x' ausgeführt wird; die eingeführte Boolesche Variable ist dann entsprechend abzuändern. In dem Fall $x = x'$ ist die `LOAD`- oder `STORE`-Anweisung lediglich ein einziges Mal auszuführen. Wurde überhaupt kein $x' \succeq x$ aufgezählt, so muß diese Anweisung nach dem Ende des Schleifensatzes für x ausgeführt werden.

Dies führt zu Schleifensätzen der folgenden Form:

```
doLS := true;
for p''_1 = min(π_1(R^{-1} · L_{p'}), π_1(R^{-1} · U_{p'}))
```

```
to max(π_1(R^{-1}·L_{p'}), π_1(R^{-1}·U_{p'})) step ggT({r'_{1,q}·p_q | q ∈ {1,...,m}} \ {0})
  ⋮
    for p''_m = min(π_m(R^{-1}·L_{p'}), π_m(R^{-1}·U_{p'}))
to max(π_m(R^{-1}·L_{p'}), π_m(R^{-1}·U_{p'})) step ggT({r'_{m,q}·p_q | q ∈ {1,...,m}} \ {0})
      /* m Gleichungen! */
      if (R·p'' − L_{p'})%p = 0 then
        for x'_1 = s_1·⌊(π_1(⌈B^r_P·R·d·p'' − k⌉) − L_1[x/x'] − 1)/s_1⌋ + s_1 + L_1[x/x']
to min(U_1[x/x'], π_1(⌊B^r_P·R·(p''_1·d + d − 1, p''_2·d, ..., p''_m·d) − k⌋)) step s_1
          ⋮
            for x'_m = s_m·⌊(π_m(⌈B^r_P·R·d·p'' − k⌉) − L_m[x/x'] − 1)/s_m⌋ + s_m + L_m[x/x']
to min(U_m[x/x'], π_m(⌊B^r_P·R·(p''_1, ..., p''_{m-1}·d, p''_m·d + d − 1) − k⌋)) step s_m
              /* m Gleichungen! */
              if ⌊(B_P·x' + k)/d⌋%p = x then
                if doLS then
                  if x' ≻ x then
                    /* Kommunikation für Berechnung auf x */
                    /* mit Φ_A ∘ φ_P(x) */
                    doLS := false;
                  else
                    doLS := (x ≠ x');
                /* kommuniziere mit x' (x = x' möglich!) */
          ⋮
  ⋮
if doLS then
  /* Kommunikation für Berechnung auf x mit Φ_A ∘ φ_P(x) */
```

Algorithmus 6.11 liefert eine Codesequenz für einen solchen Prozessor, der sowohl *Owner* ist als auch aktiver (berechnender) Prozessor.

Algorithmus 6.11

Eingabe: Anweisung $\Gamma \in \{\texttt{get}, \texttt{put}\}$ für Arrayzugriff $A[\varphi]$ mit
$\Phi_A \circ \varphi : i \mapsto \left\lfloor \frac{B_P(i)+k}{d} \right\rfloor \% p$ (Spaltenstufenform von B_P^r wie in Satz 6.9
$(B_P^r \cdot R) = (b_{i,j}^R)$, $R^{-1} = (r'_{i,j})$),
aus `par`-Schleifen stammende
Indizes $(x_1, \ldots, x_m)$ begrenzt durch $(L_1, \ldots, L_m) \leq x \leq (U_1, \ldots, U_m)$
mit Schirttweiten $(s_1, \ldots, s_m)$ (dabei ist $L_j, U_j = 0$ für Dimensionen j,
die nicht von umgebenden Parallelschleifen aufgespannt werden),
Bedingungen C_A aus umgebenden `if`- und `else`-Bedingungen

Ausgabe: Schleifensätze zum Aufzählen von Kommunikationspartnern
in lexikographischer Reihenfolge und
Senden bzw. Empfang durch `LOAD`- bzw. `STORE`-Anweisung auf reinem
Owner-Prozessor

$\mu := \mathit{extremize}_x(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \mathit{min});$

$M := \mathit{extremize}_x(\left\lfloor \frac{B_P(x)+k}{d} \right\rfloor, \mathit{max});$

for $i = 1$ to m
  $L_{p',i} :=$ ''$p_i \cdot \left\lfloor \frac{pi_i(\mu)-x_i-1}{p_i} \right\rfloor + p_i + x_i$'';
  $U_{p',i} :=$ ''$p_i \cdot \left\lfloor \frac{\pi_i(M)-x_i}{p_i} \right\rfloor + x_i$'';
/* verwende $p''_i = 0$ bei Multiplilaktionen statt */
/* symbolischen Wert p''_i! */
for $i = \mathrm{rang}(B^r_P) + 1$ to m
  $p''_i := 0$;
/* $doLS$ gibt an, ob Γ für x noch durchzuführen ist */
emit ''$doLS := \mathrm{true}$;'';
for $i = 1$ to $\mathrm{rang}(B^r_P)$
  /* symbolische Matrixmultiplikation */
  emit ''for $p'' = min(\pi_i(R^{-1} \cdot L_{p'}), \pi_i(R^{-1} \cdot U_{p'})$
 to $max(\pi_i(R^{-1} \cdot L_{p'}), \pi_i(R^{-1} \cdot U_{p'}))$
 step $\mathrm{ggT}(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\} \setminus \{0\})$'';
$Cond :=$ ''true'';
for $i = 1$ to m
  if $\mathrm{ggT}(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\} \setminus \{0\}) \neq \max(\{r'_{i,q} \cdot p_q \mid q \in \{1, \ldots, m\}\})$ then
    $Cond := Cond +$ ''$\wedge ((\pi_i(R \cdot p'') - L_{p',i}) \% p_i = 0$'';
if $Cond \neq$ ''true'' then
  emit ''if $Cond$ then'';
/* $lastCol$ ist die Nummer der neu */
/* zu beginnenden Stufe in $B_P \cdot R$ */
$lastCol := 1$;
$Cond :=$ ''true'';
for $i = 1$ to m
  if $(\exists K : K \in \mathrm{Span}(\mathrm{Ke}(B_P)) : \pi_i(K) \neq 0)$ then
    emit ''for $x'_i = L_i$ to U_i step s_i'';
    $Cond := Cond +$ ''$\wedge \pi_i(\left\lfloor \frac{B_P \cdot x' + k}{d} \right\rfloor \% p) = x_i$'';
  else
    if $(b^R_{i,lastCol} = 0) \vee (d = \mathrm{ggT}(\{d, b^R_{i,1}, \ldots, b^R_{i,lastCol}\} \setminus \{0\}))$ then
      emit ''$x'_i := s_i \cdot \left\lfloor \frac{\pi_i(\lceil B^r_P \cdot R \cdot d \cdot p'' - k \rceil) - L_i[x/x'] - 1}{s_i} \right\rfloor + s_i + L_i[x/x']$;'';
      emit ''if $x'_i \leq U_i[x/x']$ then'';
    else
      emit ''for $x'_i = s_i \cdot \left\lfloor \frac{\pi_i(\lceil B^r_P \cdot R \cdot d \cdot p'' - k \rceil) - L_i[x/x'] - 1}{s_i} \right\rfloor + s_i + L_i[x/x']$
 to $\min(U_i[x/x'], \pi_i(\lfloor B^r_P \cdot R \cdot (p''_1 \cdot d, \ldots, p''_{lastCol} \cdot d + d - 1, \ldots, p''_m \cdot d) - k \rfloor))$ step s_i'';
  if $b^R_{i,lastCol} \neq 0$ then
    $lastCol := lastCol + 1$;
if $Cond \neq$ ''true'' then
  emit ''if $Cond$ then'';
if $C_A \neq \emptyset$ then
  emit ''if $C_A[x/x']$ then'';
emit ''if $doLS$ then'';
emit ''if $x \prec x'$ then'';
if $\Gamma =$ get then
  emit ''$\underline{\neq}$LOAD($A[\varphi_P], tmp_{A[\varphi_P]}, x$)'';
else
  emit ''$\underline{\neq}$STORE($A[\varphi_P], tmp_{A[\varphi_P]}, x$)'';
emit ''$doLS := \mathrm{false}$;'';
emit ''else'';
emit ''$doLS := (x \neq x')$;'';

```
emit ''endif'';
emit ''endif'';
if Γ = get then
  emit ''=LOAD(A[φ_P],tmp_A[φ_P],x')'';
else
  emit ''=STORE(A[φ_P],tmp_A[φ_P],x')'';
/* Ende für C_A */
if C_A ≠ ∅ then
  emit ''endif'';
/* Ende für Cond */
if Cond ≠ ''true'' then
  emit ''endif'';
emit Endmarkierungen für alle geöffneten fors und ifs;
/* evtl. wurde kein x' ⪰ x aufgezählt */
emit ''if doLS then'';
if Γ = get then
  emit ''≢LOAD(A[φ_P],tmp_A[φ_P],x)'';
else
  emit ''≢STORE(A[φ_P],tmp_A[φ_P],x)'';
emit ''endif'';
```

Auf diese Weise kann für jeden affinen Arrayzugriff die Menge der Kommunikationspartner für einen gegebenen Prozessor x in lexikographischer Reihenfolge aufgezählt werden. Wenn bei der Codegenerierung die Tests bezüglich der lexikographischen Ordnungsrelation noch weiter bei der Generierung der `LOAD`- und `STORE`-Anweisungen berücksichtigt werden, kann der Code für diese Anweisungen auch noch besser (auf jeweils ein `send`, `receive` oder eine Zuweisung) vereinfacht werden.

Bemerkung: Es sei an dieser Stelle nochmals darauf hingewiesen, daß die lexikographische Ordnung lediglich bei synchroner Kommunikation erforderlich ist und entfallen kann, falls die Architektur ein nichtblockierendes `send` in Verbindung mit einem blockierenden `receive` bereitstellt (was für heutige Architekturen beispielsweise durch `MPI` [Mes95] meist der Fall ist).

In diesem Fall können die Schleifensätze generiert werden, als gelte $R^{-1} = E$; d.h. insbesondere, daß die Schrittweite für die p''_i bindende Schleife p_i ist und die Grenzen direkt durch $L_{p',i}$ und $U_{p',i}$ gegeben sind.

6.5.2 Programmtransformationen

Jeder Prozessor x des Prozessorfeldes kann als *Owner* eines Datums auftreten, das prinzipiell zu jeder Zeit von einem anderen Prozessor benötigt werden kann, auch wenn x zu diesem Zeitpunkt und an dieser Stelle des Codes nicht von den umliegenden Parallelschleifen aufgespannt wird (also nicht zur Stufe II oder III gehört). Da nun also auch von Anweisungen die Rede sein wird, die von Prozessoren ausgeführt werden, die nicht von `par`-Schleifen aufgespannt werden, betrachten wir von nun an ein *Projekt* als **Node-Programm**, also in der in Abschnitt 4.3 beschriebenen SPMD-Form. Wir nehmen außerdem an, das *Projekt* sei um `barrier`-, `get`- und `put`-Anweisungen erweitert worden. Außerdem gehen wir davon aus, daß alle in `for`-Schleifen vorkommenden Schrittweiten positiv sind; ist dies nicht der Fall, sind die Schleifenparameter wie in Abschnitt 6.5 besprochen zu ersetzen.

Definition 6.12 *Sei B ein Programmfragment mit* `get`- *und* `put`-*Anweisungen. Dann ist der* **Ownerpart** *von B, $Ownerpart(B)$, das Programmfragment, das aus B durch Entfernen aller Zuweisungen und Prozeduraufrufe (ausgenommen* `get`-, `put`- *und* `barrier`-*Anweisungen) entsteht, wobei alle* `get`- *und alle* `put`-*Anweisungen mit einer Markierung* `Ownerpart` *zu versehen sind.*

Erweiterung von `for`-Schleifen Da ein *Owner*-Prozessor prinzipiell zu jedem Zeitpunkt ein Datum von einem anderen Prozessor empfangen oder an ihn senden muß, sind alle sequentiellen Schleifen auf jedem Prozessor gleich oft zu durchlaufen. Das heißt:

1. Jede `for`-Schleife

   ```
   for t = l(x) to u step s
     B
   ```

   deren Untergrenze von einem aus `par`-Schleifen stammenden Indexvektor x abhängt, wird ersetzt durch

   ```
   for t = extremize_x(l(x), min) to l(x) - 1
     Ownerpart(B)
   for t = l(x) to u
     if (t - l(x))%s = 0 then
       B
     else
       Ownerpart(B)
   ```

   wobei $extremize_x$ lediglich nach x extremalisiert.

2. Jede `for`-Schleife, deren Obergrenze von x abhängt

   ```
   for t = l to u(x) step s
     B
   ```

   wird ersetzt durch

   ```
   for t = l to u(x) step s
     B
   for t = u(x) + s - (u(x) - l(x))%s to extremize_x(u(x), max) step s
     Ownerpart(B)
   ```

Im allgemeinen können natürlich Kombinationen dieser beiden Fälle auftreten.

Erweiterung von `while`-Schleifen Aus dem gleichen Grund wie `for`-Schleifen müßten eigentlich auch `while`-Schleifen erweitert werden. Da für `while`-Schleifen jedoch zur Compilezeit keine maximale Obergrenze ermittelt werden kann, haben wir in Abschnitt 2.1 festgelegt, daß die Bedingung einer `while`-Schleife nicht von dem Wert eines von einer `par`-Schleife gebundenen Indexes abhängig sein darf. Damit ist von Anfang an gewährleistet, daß jeder Prozessor die maximale Anzahl von Iterationen durch die `while`-Schleife ausführt.

Erweiterung von `if`-Anweisungen Auch `if`-Anweisungen können von Prozessorkoordinaten abhängig sein und so die Anzahl der Ausführungen des Rumpfes prozessorabhängig machen (was beispielsweise auch wegen einer möglicherweise in einem `if` verschachtelten `for`-Schleife, die durch eine `barrier`-Anweisung alle Stufe I-Prozessoren synchronisieren muß,[13] unerwünscht wäre). Darum ersetzen wir jede `if`-Anweisung

`if` $cond(x)$ `then`
  B
`else`
  C

deren Bedingung von einem Indexvektor x abhängt, der aus ein `par`-Schleifen stammt, durch

`if` $cond(x)$ `then`
  B
  $Ownerpart(C)$
`else`
  $Ownerpart(B)$
  C

Bei der soeben durchgeführten Erweiterung der `for`-Anweisung hatten wir in Punkt 1 eine `if`-Anweisung eingefügt. Würden wir die obige `if`-Regel auch auf diese erzeugte Anweisung „`if` $(t-l(x))\%s = 0$ `then`" anwenden, wäre der `else`-Zweig dieses eingefügten `if`s überflüssig. Auf die angegebene Art (einen `else`-Zweig beim Erweitern der `for`-Schleifen einzufügen) genügt es jedoch, zuerst alle `if`-Anweisungen zu erweitern und dann alle `for`-Schleifen. Die Erweiterungen dürfen nicht beliebig überlappend ausgeführt werden, da das Verfahren sonst nicht terminieren würde.

Umwandlung von `get`/`put` zu `LOAD`/`STORE` In Abschnitt 6.5.1 haben wir eine Methode entwickelt, die Menge der Kommunikationspartner eines Prozessors in lexikographischer Reihenfolge aufzuzählen; dabei sind unterschiedliche Schleifensätze nötig für reine *Owner*-Prozessoren auf der einen Seite und auf der anderen Prozessoren, die auch Berechnungen durchzuführen haben. Jede `get`- oder `put`-Anweisung Γ mit passender (d.h. affiner) Indexfunktion können wir nun auf diese Weise zu einer `LOAD`- bzw. `STORE`-Anweisung umschreiben, indem sie durch die Ausgabe von Algorithmus 6.11 bzw. – falls Γ mit der Markierung `Ownerpart` versehen ist – durch die Ausgabe von 6.10 ersetzt wird.[14]

Bemerkung: Da die bei der Erweiterung des Indexraums verwandte *extremize*-Funktion nur eine Abschätzung der exakten Schleifengrenzen liefert, kann eine erweiterte Schleife zu viele Iterationen aufzählen. Darum ist die exakte Bedingung einer Schleife

`for` $i = l(x)$ `to` $u(x)$ `step` s

in der Form

$$\{i \geq l[x/x'], i \leq u[x/x'], (i - l[x/x'])\%s = 0\}$$

[13] In Abschnitt 6.1 hatten wir gesehen, daß die auf asynchronen Machinen notwendige Synchronisation entweder diese Indexraumerweiterung oder wiederholte Regruppierung erfordert; wir gehen hier davon aus, daß keine Regruppierung stattfindet.

[14] Falls bei der ursprünglichen Indexraumerweiterung auch mit `Ownerpart` markierte `get`- und `put`-Anweisungen erzeugt wurden, die (wegen zu komplexer Indexfunktion) nicht durch `LOAD`/`STORE` ersetzt wurden, sind diese noch aus dem Programmtext zu entfernen.

für die Algorithmen 6.10 und 6.11 zu den Bedingungen aus C_A hinzuzufügen. Affine $l(x)$ und $u(x)$ könnten prinzipiell auch stattdessen in die Grenzberechnung in diesen Algorithmen eingehen.

Bemerkung: Unter Umständen kann durch das Signalschema, das durch die expliziten `send`- und `receive`-Anweisungen der `LOAD`s bzw. `STORE`s forciert wird, die Synchronisation mittels `barrier` unnötig werden. In folgenden Fällen ist die `barrier`-Anweisung am Ende eines *Basisblocks* B überflüssig:

- B befindet sich am Ende einer sequentiellen Schleife S und jeder Zugriff[15] (in Quelle und Ziel) jeder Abhängigkeit, die von einer Anweisung innerhalb von S ausgeht und deren sequentielle Komponente von S getragen wird, ist durch `LOAD`- bzw. `STORE` realisiert.[16]
- B befindet sich nicht am Ende einer Schleife und jeder Zugriff[17] (in Quelle und Ziel) jeder Abhängigkeit, die nicht von einer umgebenden Schleife *getragen* wird und die von einer Anweisung innerhalb von B ausgeht, ist durch `LOAD`- bzw. `STORE` realisiert.

Bemerkung: Ist für die Indexfunktion φ bzw. ihren linearen Anteil φ_{lin} und die Allokation Φ_A bzw. deren linearen Anteil $\Phi_{A,lin}$ der Kern der Hintereinanderausführung $\Phi_{A,lin} \circ \varphi_{lin}$ nichttrivial, so werden nach dieser Methode u.U. sehr viele Prozessorkoordinaten unnötigerweise durch die x'-Schleifen aufgezählt. Im Extremfall kann dies, wie erwähnt, das Aufzählen des gesamten Prozessorfeldes – und zwar für jeden Wert von p'' – nach sich ziehen. Für diesen Fall ist diese Methode also nicht empfehlenswert.

Der Kern $\mathrm{Ke}(\Phi_{A,lin} \circ \varphi_{lin})$ teilt sich auf in $\mathrm{Ke}(\varphi_{lin})$ und einen „Restkern". Dabei benötigen alle Prozessoren, die aufgrund des Kerns von φ_{lin} aufgezählt werden, das gleiche Datum. Es kann also sinnvoll sein, für diese Mengen jeweils `broadcast`-Anweisungen zu verwenden statt `send-/receive`-Paare. Allerdings müßten diese Mengen dann im voraus durch Gruppierung ermittelt werden.

Eine weitere Möglichkeit ist, im Falle eines nichttrivialen Kerns von $\Phi_{A,lin} \circ \varphi_{lin}$ alle von einem Prozessor aufgrund eines Arrayzugriffs benötigten Daten an alle anderen Prozessoren zu broadcasten. Da der Indexraum schon entsprechend erweitert ist, könnte dies mit mehreren `broadcast`-Anweisungen bzw. einer `all-to-all`-Kommunikation (wie sie `MPI` zur Verfügung stellt) realisiert werden. Jeder Prozessor, der tatsächlich ein Datum braucht, müßte dann aus der zusammengefaßten Botschaft den für ihn relevanten Teil herausfiltern und den Rest ignorieren.

6.6 Kommunikation in *LooPo*

Wie wir in den vorangegangenen Abschnitten gesehen haben, spielen bei der Plazierung von Synchronisations- und Kommunikationsanweisungen die Abhängigkeiten eines *Projekts* eine große Rolle. Die Datenabhängigkeitsanalyse ist im *LooPo*-Projekt ein eigenes Modul, das zu Beginn der Programmtransformation durch *LooPo* aufgerufen wird. Durch das `normloops`-Modul [Gei97] und die Codegenerierung [Wet95] selbst können jedoch zusätzliche Anweisungen im Programm eingeführt

[15] Basiert die Abhängigkeitsanalyse auf Zugriffen statt auf Anweisungen (wie in [Kei97] beschrieben), ist hier eine differenziertere Sicht möglich, so daß nicht jeder Zugriff einer Anweisung dieselbe Bedingung zu erfüllen hat.

[16] In der Implementation des `targout`-Moduls ist lediglich dieses Kriterium zum Löschen redundanter `barrier`-Anweisungen realisiert (für jedes von *LooPo* erzeugte *Projekt* genügt es, wenn alle `barrier`-Anweisungen am Ende einer sequentiellen Schleife stehen, was das zweite Kriterium für diese *Projekte* nicht anwendbar und damit überflüssig macht).

[17] Auch hier ist durch zugriffsbasierte Abhängigkeitsanalyse eine differenziertere Sicht möglich.

werden, die von keiner Schleife getragene Abhängigkeiten im Programmtext erzeugen. Da wir für die Programmtransformation die Abhängigkeiten aus der Abhängigkeitsanalyse verwenden, werden diese neu hinzugekommenen Abhängigkeiten ignoriert. Es ist jedoch nach [Gei97] und [Wet95] garantiert, daß diese Anweisungen in einem Block vor der ersten Schleife stehen, in deren Grenzen oder Rumpf diese Werte zum ersten Mal gebraucht werden können. Es genügt in diesem Fall also, eine `barrier`-Anweisung vor der ersten Schleife eines *Projekts* zu plazieren, um die korrekte Ausführung zu gewährleisten.

Außerdem werden durch Tiling [Sch97] sequentielle Schleifen eingeführt, die von der Abhängigkeitsanalyse ebenfalls nicht betrachtet werden. Für unsere Zwecke genügt jedoch die nach [Sch97] korrekte Annahme, daß keine dieser Schleifen eine Abhängigkeit trägt oder in ihrer sequentiellen Komponente trägt.

Die Transformation von `get`- und `put`-Anweisungen zu `LOAD` und `STORE` – bzw. `send`/`receive`-Paaren – erzeugt, so wie sie in Abschnitt 6.5.1 beschrieben wurde, ineffizienten Code, falls der Kern des linearen Anteils von $\Phi \circ \varphi$ für die Datenallokation Φ und die Indexfunktion φ nichttrivial ist; denn in diesem Fall werden alle Prozessorkoordinaten der entsprechenden Dimensionen sequentiell aufgezählt. Darum ist es in der Implementation möglich, diese Umwandlung gar nicht oder nur für solche Funktionen durchführen zu lassen, bei denen der Kern trivial ist.

Das folgende Beispiel zeigt die lauffähige Ausgabe eines durch *LooPo* parallelisierten Programmstücks:

Beispiel 6.13 *Betrachten wir nochmals das Programm aus Beispiel 5.16. Die Ausgabe des Codegenerierungsmoduls ist (in* LooPo-`C`*-Notation):*

```
/* LooPo program
**      C
*/

for(loopo_targgen_0=0;loopo_targgen_0<=n-1;loopo_targgen_0++)
 {
  parfor(loopo_targgen_1=0;loopo_targgen_1<=n-1;loopo_targgen_1++)
   {
    parfor(loopo_targgen_2=0;loopo_targgen_2<=n-1;loopo_targgen_2++)
     {
      C[loopo_targgen_1][loopo_targgen_2]=
C[loopo_targgen_1][loopo_targgen_2]+
A[loopo_targgen_1][loopo_targgen_0]*B[loopo_targgen_0][loopo_targgen_2];
     }
   }
 }

/*    end of
** LooPo program
*/
```

Als komplettes `C`*-Programm mit Aufrufen an die* LooPo*-Library erhalten wir folgende Ausgabe:*

```
#include <stdlib.h>
#include <limits.h>
#include "LPOall.h"
#include "comm_par.h"
#include "comm_all.h"
#include "ploop.h"
```

```
#include "pitfalls.h"
#include "gen_par.h"
#include "symtab.h"

/* LooPo program
**      C
*/

int n;
int ****A=NULL;
int ****B=NULL;
int ****C=NULL;
void LooPo_PRG0(void)
 {
  int t1;
  int p1;
  int p2;
  int array1;
  int array2;
  int array3;
  LooPo_getConstFromMaster(-1,0,1,0);
  LooPo_beginSection(2,n,n);
  if(LOOPO_SECTION_OK)
   {
    LooPo_initializeRTS(LOOPO_TRUE,LOOPO_FALSE);
    LooPo_setSourceToZero(2);
    LooPo_redistributeArray(LOOPO_TRUE,LOOPO_FALSE,2,
LooPo_constructSPITFALLS(1,0,1,1,
1+((int)floor(((double)(2))/LooPo_lcm(1,n))),n,0,1,LooPo_lcm(1,n),0),
LooPo_constructSPITFALLS(1,0,1,1,
1+((int)floor(((double)(2))/LooPo_lcm(1,n))),n,0,1,LooPo_lcm(1,n),1));
    LooPo_setSourceToZero(1);
    LooPo_redistributeArray(LOOPO_TRUE,LOOPO_FALSE,1,
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1),
LooPo_constructSPITFALLS(1,0,1,1,
1+((int)floor(((double)(2))/LooPo_lcm(1,n))),n,0,1,LooPo_lcm(1,n),0));
    LooPo_setSourceToZero(0);
    LooPo_redistributeArray(LOOPO_TRUE,LOOPO_FALSE,0,
LooPo_constructSPITFALLS(1,0,1,1,
1+((int)floor(((double)(2))/LooPo_lcm(1,n))),n,0,1,LooPo_lcm(1,n),0),
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1));
    LOOPO_SYNC_GRID();
    for(t1=0;t1<=n-1;t1++)
     {
      /* DOALL */
      p1=LooPo_parBegin(0,n-1);
      if(LOOPO_PARFOR_OK)
       {
        /* DOALL */
        p2=LooPo_parBegin(0,n-1);
        if(LOOPO_PARFOR_OK)
         {
          if(LOOPO_COMPUTE_OK)
           {
```

```
            LooPo_RTSSRecvCoord(&(array1),2,LooPo_mod(p1,(1+n-1)),
LooPo_mod(p2,(1+n-1)),((int)floor((p1-(1+0-1))/LooPo_lcm(1,n))),
LooPo_mod((p1-(1+0-1)),1),((int)floor((p2-(1+0-1))/LooPo_lcm(1,n))),
LooPo_mod((p2-(1+0-1)),1));
            LooPo_RTSSRecvCoord(&(array3),1,LooPo_mod(0,(1+n-1)),
LooPo_mod(p2,(1+n-1)),((int)floor((p2-(1+0-1))/LooPo_lcm(1,n))),
LooPo_mod((p2-(1+0-1)),1),0,t1);
            array1=array1+array2*array3;
            LooPo_RTSSSendCoord(&(array1),2,LooPo_mod(p1,(1+n-1)),
LooPo_mod(p2,(1+n-1)),((int)floor((p1-(1+0-1))/LooPo_lcm(1,n))),
LooPo_mod((p1-(1+0-1)),1),((int)floor((p2-(1+0-1))/LooPo_lcm(1,n))),
LooPo_mod((p2-(1+0-1)),1));
           }
         }
        LooPo_parEnd();
       }
      LooPo_parEnd();
      LOOPO_SYNC_GRID();
     }
    LOOPO_SYNC_GRID();
    LooPo_redistributeArray(LOOPO_FALSE,LOOPO_TRUE,0,
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1),
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1));
    LooPo_redistributeArray(LOOPO_FALSE,LOOPO_TRUE,1,
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1),
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1));
    LooPo_redistributeArray(LOOPO_FALSE,LOOPO_TRUE,2,
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1),
LooPo_constructSPITFALLS(0,0,1,1,1,n,0,1,n,-1));
    LooPo_finalizeRTS();
   }
  LooPo_endSection();
 }

/*    end of
** LooPo program
*/


int main(int argc,char **argv)
 {
  LooPo_initSymtabs(3-1,1-1);
  LooPo_declareConstant(&n,0,LOOPO_INT);
  LooPo_declareArray(&A,0,LOOPO_INT,2,0,2,0,2);
  LooPo_setTrafo(0,LooPo_constructMatrix(2,2,1,1,0,0,1));
  LooPo_setProcTrafo(0,LooPo_constructMatrix(2,2,1,1,0,0,1));
  LooPo_declareArray(&B,1,LOOPO_INT,2,0,2,0,2);
  LooPo_setTrafo(1,LooPo_constructMatrix(2,2,1,0,1,1,0));
  LooPo_setProcTrafo(1,LooPo_constructMatrix(2,2,1,1,0,0,1));
  LooPo_declareArray(&C,2,LOOPO_INT,2,0,2,0,2);
  LooPo_setTrafo(2,LooPo_constructMatrix(2,2,1,1,0,0,1));
  LooPo_setProcTrafo(2,LooPo_constructMatrix(2,2,1,1,0,0,1));
  LooPo_initializeComm(&argc,&argv);
  LooPo_connectHost(0);
```

```
  while(LOOPO_LOOP_OK)
   {
    switch(LooPo_waitCommand())
     {
      case 0:
       LooPo_PRG0();
       LooPo_shutdownMainServer();
       break;
      default:
       break;
     }
   }
  LooPo_disconnectHost();
  LooPo_finalizeComm();
  LooPo_closeSymtabs();
  exit(0);
 }
```

Im Hauptprogramm wird zunächst wie beim Hostprogramm in Abschnitt 5.5 eine Tabelle mit Informationen über die zu verarbeitenden Variablen angelegt. Hierzu gehören auch die zur SPITFALLS*-basierten Allokationsdarstellung nötigen Transformationsmatrizen. Nach Verbindungsaufbau mit dem Hostrechner (wobei der lokale Rechner „*`localhost`*“ voreingestellt ist), wird je nach dem vom Hostrechner empfangenen Befehl ein* Projekt *ausgeführt (oder das Programm beendet). Innerhalb des* Projektes *wird durch Aufruf an* `LooPo_redistributeArray` *das jeweilige Array in einer bestimmten Allokation auf den Parallelrechner aufgebracht, wobei* `LooPo_redistributeArray` *auch für die Allokation des (von der Datenallokation abhängigen) Speichers übernimmt. Die Kommunikation während des Schleifendurchlaufs geschieht über die Funktionen* `LooPo_RTSSRecvCoord` *bzw.* `LooPo_RTSSSendCoord`*, die mit einem auf dem* Owner*-Prozessor quasiparallel zum eigentlichen* Projekt *ablaufenden Thread kommunizieren, um* `get`*- und* `put`*-Anweisungen zu implementieren.*

7 Zusammenfassung

Das Ziel dieser Arbeit war die Generierung eines SPMD-Programms für asynchrone Parallelcomputer mit verteiltem Speicher aus einem *Projekt* mit expliziten Parallelschleifen mit `fork-join`-Semantik, die eine Anweisungsallokation vorgeben. Solche *Projekte* sind eine Verallgemeinerung der Ausgabe des automatischen Parallelisierers *LooPo*.

Die Transformation allgemeiner *Projekte* zu SPMD-Programmen für Distributed-Memory-Maschinen führt hierbei zu einer Reihe von Problemen, die teilweise durch die Synchronisation asynchroner Programme mittels `for`-Schleifen verursacht werden, teilweise auch durch die Komplexität der Datenallokation, selbst wenn diese statisch als eine *Modularabbildung* definiert ist, sowie durch komplizierte (nichtinvertierbare oder nichtaffine) Indexausdrücke.

Um die Anweisungsallokation und die `fork-join`-Semantik von parallelen Schleifen im SPMD-Programm garantieren zu können, haben wir zunächst drei Stufen ermittelt, für die sich ein Prozessor bei der Ausführung einer gegebenen Anweisung qualifizieren kann, und jeder dieser Stufen entsprechende Aktionen zugeordnet, die der Prozessor durchzuführen hat:

1. Stufe I: Der Prozessor gehört zum Prozessorfeld; als Datenbesitzer sind also Kommunikationen durchzuführen.

2. Stufe II: Der Prozessor erfüllt die Bedingungen aller umliegenden `if`-Anweisungen und `par`-Schleifen; es sind also auch Kontrollanweisungen auszuführen, um evtl. weitere Bedingungen abzutesten.

3. Stufe III: Zusätzlich zu den Bedingungen der Stufe II sind die Koordinaten des Prozessors in den noch nicht durch `par`-Schleifen aufgespannten Dimensionen 0; es sind also auch Zuweisungen und Prozeduraufrufe auszuführen.

Diese Unterscheidung in Datenbesitzer, potentiell aktive Prozessoren und aktive Prozessoren ist nötig, falls

- es Zuweisungen bzw. Prozeduraufrufe gibt, die nicht von der maximalen Anzahl von `par`-Schleifen umgeben sind (ansonsten sind die Bedingungen für die Stufen II und III äquivalent) und

- das erweiterte Prozessorfeld echt größer ist als das Prozessorfeld (ansonsten sind die Bedingungen für die Stufen I und II äquivalent).

Um das resultierende Programm auch auf einer asynchronen Maschine ausführen zu können, müssen die Prozessoren untereinander synchronisiert werden. Diese Synchronisation muß am Ende eines *Basisblocks* stattfinden, wenn sich dieser am Ende einer sequentiellen Schleife befindet, welche die sequentielle Komponente mindestens einer Abhängigkeit *trägt*, oder wenn von ihm Abhängigkeiten ausgehen, die nicht von einer umliegenden Schleife *getragen* werden.

Außerdem haben wir mögliche Implementationen für diese Synchronisation identifiziert, nämlich Regruppierung bzw. Indexraumerweiterung. Regruppierung kann bei Maschinen mit verteiltem Speicher durchaus die effizientere Implementation bieten; können jedoch `get`- und `put`-Anweisungen durch `LOAD`- bzw. `STORE`- Anweisungen ersetzt werden, ist es hierfür ohnehin erforderlich (zumindest bei einer statischen Datenallokation), den Indexraum zu erweitern, so daß in diesem Fall letztere Methode vorzuziehen ist.

Auf einer Maschine mit verteiltem Speicher sind die Daten, für die die Berechnungen stattfinden, auf dem Parallelrechner verteilt, d.h. jeder Prozessor hält einen Teil der Daten in seinem lokalen Speicher. Eine dynamische Datenallokation, bei der einem Datum kein eindeutiger *Owner*-Prozessor zugeordnet werden kann, kann bei der Feststellung des aktuellen *Owners* des Datums zu Schwierigkeiten führen. Wir haben uns darum für eine statische Datenallokation entschieden. Für bestimmte Zugriffsmuster, für die die Quelle einer Abhängigkeit eindeutig bestimmt werden kann und der Raum, in dem diese Abhängigkeit gilt, exakt beschrieben ist, ist es jedoch sicherlich auch auf effiziente Weise möglich, eine dynamische Allokation zu verwenden, die dann während der Berechnungsphase u.U. einen geringeren Kommunikationsaufwand aufweisen kann. Eine mögliche Implementation besteht hierbei aus drei Teilen:

1. Jede Variable wird vom ursprünglichen (statischen) *Owner* zu den Prozessoren geschickt, die auf diese Variable zugreifen, ohne daß dieser Zugriff Ziel einer Abhängigkeit ist.
2. Nach jeder Berechnung (Zuweisung) auf Prozessor P_S wird das Ergebnis zu Prozessor P_T geschickt, falls es für eine Berechnung auf P_T benötigt wird (was durch eine Abhängigkeit angezeigt wird).
3. Jede Variable wird von dem Prozessor, auf dem zuletzt ein neuer Wert für sie berechnet wurde, zu ihrem *Owner* geschickt.

Jedoch ist diese Lösung nur bei einer sehr eingeschränkten Klasse von *Projekten* problemlos realisierbar. Eine andere Möglichkeit wäre die tabellengespeicherte Verteilung einer Variablen, die aber bei jeder Änderung des *Owners* auf jedem Prozessor modifiziert werden müßte, oder dynamischere Verfahren wie sie bei der Implementation von Shared Virtual Memory angewandt werden.

Als einerseits statische und andererseits reguläre Allokationen haben wir *Modularabbildungen* betrachtet. Daten, die auf demselben Prozessor zu speichern sind, werden dabei auf jeden Fall durch den Kern des linearen Anteils dieser Abbildung definiert. Um die Vielfachen eines Spannvektors des Kerns lokal in einer Array Dimension zu speichern, ist eine Basistransformation des Datenraums nötig. Im allgemeinen wird hierdurch das rechteckige Array zu einem Polytop umgeformt, dessen Inhalt möglicherweise größer ist als der des Rechtecks. Diese Vergrößerung läßt sich u.U. nicht ganz vermeiden, da die Abbildung, die durch die Umformung in das Polytop gegeben wird und ganzzahlige Punkte im Rechteck auf ganzzahlige Punkte im Polytop abbildet, injektiv sein muß. Durch Anwendung einer Stauchung auf diese Umformung kann die Vergrößerung jedoch verringert werden. Arraydefinitionen lassen so allgemeine Formen eigentlich nicht zu (Abhilfe könnte jedoch z.B. in `C` durch Allokation entsprechender dynamischer Arrays geschaffen werden); außerdem würde eine exakte Allokation als Polytop die Berechnung eines lokalen Array-Elements, dessen Koordinaten im globalen Datenraum gegeben sind, verkomplizieren (es würden für jede Koordinate affine Zusammenhänge zu den Prozessorkoordinaten geschaffen). Wir vergrößern darum das umgeformte Array auf das kleinste umliegende Rechteck.

Weiterhin haben wir beobachtet, daß sich die Verteilung eines Arrays unter einer *Modularabbildung* (das Urbild eines bestimmten Prozessors unter dieser Abbildung) durch eine einfachere *Modularabbildung* und eine Sequenz von *SPITFALLS*, regulären Mengen von *PITFALLS* (regulären Mengen von Array-Indizes), ausdrücken läßt (für bijektive Abbildungen genügt die Sequenz von *SPITFALLS*). Soweit durch *Modularabbildungen* gegebene Verteilungen[18] durch `HPF`-Konstrukte ausgedrückt werden können, sind dies Abbildungen, deren linearer Anteil durch eine Matrix in einer Art „Diagonalform“ gegeben ist. Solche Verteilungen können direkt als Sequenz von *SPITFALLS* ausgedrückt werden; in einfacheren Fällen, in denen jede

[18] Im Gegensatz zu den hier betrachteten Verteilungen unterstützt `HPF` jedoch auch solche, die nicht durch *Modularabbildungen* ausgedrückt werden können (replizierte oder teilweise replizierte Speicherung).

Arraydimension blockzyklisch verteilt ist, könnte eigentlich die beschriebende *PITFALLS* direkt angegeben werden, was die Berechnung der Breite der *Line Segments* zur Laufzeit überflüssig machen würde. In komplexeren Fällen sind dennoch auch bei der Angabe der Verteilung mittels *SPITFALLS* Matrixmultiplikationen nötig: auf dem Hostrechner muß die Transformation des Datenraums für jedes Element rückgängig gemacht werden und die Zeilentransformation hat zumindest zur Folge, daß auf jedem Prozessor ein oder mehrere virtuelle Prozessoren simuliert werden müssen, für deren Ermittlung eine Matrixmultiplikation nötig ist.[19]

Um Berechnungen auf verteilten Daten ausführen zu können, müssen Kommunikationsanweisungen generiert werden. Wir haben das prinzipielle Vorgehen von `HPF`-Compilern hierfür beschrieben: Für Schleifen, die keine Abhängigkeit *tragen*, genügt es, wie bei einer `doall`-Schleife in `Kali` [MR91], vor und (falls die OCR nicht berücksichtigt wird) nach der Schleife Kommunikationsphasen einzufügen, in denen alle in der Schleife benötigten bzw. geschriebenen Daten von bzw. zu ihrem *Owner* kommuniziert werden. Dieses Problem entspricht der Redistribution eines entsprechenden Teils des jeweiligen Arrays. Bei allgemeineren Schleifen können die Kommunikationen für die verschiedenen Iterationen der Schleife nicht zusammengefaßt werden. In diesem Fall wird prinzipiell der gesamte Indexraum der Schleife auf jedem Prozessor aufgezählt werden, um für jede Iteration alle nötigen Kommunikationsanweisungen durchzuführen. Durch Optimierungen kann dann die Anzahl der Iterationen pro Prozessor wieder verringert werden, so daß nicht tatsächlich alle Iterationen auf jedem Prozessor durchzuführen sind. Für asynchrone Programme ist dieser Ansatz jedoch nicht ohne weiteres möglich, da eine bloße Ausführung jeder Iteration einer `par`-Schleife auf jedem Prozessor nicht der definierten Ausführungsreihenfolge entspricht und somit möglicherweise inkorrekte Ergebnisse liefert.

Wir haben umgekehrt den Ansatz verfolgt, auf jedem Prozessor immer nur **eine** Iteration einer Parallelschleife auszuführen, wofür die Kommunikation zunächst über `get`- und `put`-Anweisungen stattfinden mußte. Für affine Indexausdrücke haben wir dabei eine Methode beschrieben, diese allgemeineren jedoch weniger effizienten Kommunikationsanweisungen durch `LOAD`- und `STORE`-Anweisungen (und damit durch explizite `send-/receive`-Anweisungen) zu ersetzen, indem für jeden Arrayzugriff ein neuer Schleifensatz generiert wird, der die Iterationen der `par`-Schleife aufzählt, zu denen der Prozessor als Datenbesitzer beiträgt. Für bestimmte Indexausdrücke und Datenallokationen – beispielsweise einfache blockzyklische Allokation und genau einmaliges Auftreten jedes Indexes einer umliegenden `par`-Schleife im Indexausdruck – ist die Anzahl der hier aufgezählten Indexwerte für `par`-Schleifen minimal. Dieser Ansatz ließe sich noch um Vektorisierung erweitern, so daß nicht unbedingt immer nur ein Wert an einen Prozessor geschickt wird. Ein Schwachpunkt der beschriebenen Methode liegt in Indexfunktionen und Allokationen mit nichttrivialem Kern. Das Ziel, eine möglichst geringe Menge von Prozessorkoordinaten in einer sequentiellen Schleife aufzuzählen, wird für diesen Fall nicht erfüllt. Für nichtinvertierbare Indexfunktionen wäre die Erweiterung dieses Ansatzes um `broadcast`-Anweisungen sinnvoll.

Bessere Ergebnisse könnten hier evtl. auch erreicht werden, indem der Indexraum für die entsprechenden Anweisungen künstlich (durch Einführung sequentieller Schleifen für die Prozessorkoordinaten) prinzipiell auf jedem Prozessor wieder auf die volle Größe gebracht und nachträglich durch Optimierungen, wie sie von `HPF`-Compilern in großem Ausmaß angewandt werden, wieder verkleinert würde. Wir haben außerdem gesehen, daß in rein synchronen Programmen (wie sie von *LooPo* erzeugt werden können) die parallelen Schleifen einfach in sequentielle Schleifen umgewandelt werden können, ohne die Semantik des Programms zu verändern. Für

[19] Die Implementation der *LooPo*-Library geht bei der Spaltentransformation vom allgemeinen Fall aus, könnte in der Effizienz also durch Beachtung der Spezialfälle verbessert werden.

diesen Fall wäre es also möglich, die Vielzahl von Methoden zur Kommunikationsgenerierung (und Parallelisierung), die von `HPF`-Compilern verwendet werden, (fast) unverändert anzuwenden; um die durch `par`-Schleifen gegebene Anweisungsallokation beizubehalten, müßten die hier angewandten Methoden lediglich mit der `HPF-ON`-Konstruktion verträglich sein. Allgemein erscheint es für weitere Arbeiten in jedem Falle sinnvoll, weitere Methoden zur Kommunikationsoptimierung wie die Vektorisierung und das Ausnutzen von `broadcast`-Anweisungen auf die erzeugten Programme anzuwenden.

Literatur

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[BS91] I.N. Bronstein und K.A. Semendjajew. *Taschenbuch der Mathematik*, Band 1. B.G. Teubner Verlagsgesellschaft, Stuttgart · Leipzig und Verlag Nauka, Moskau, 25. Ausgabe, 1991.

[CFH$^+$92] Alok Choudhary, Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka und Chau-Wen Tseng. Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computing, McLean, VA, USA*, 1992.

[CK88] David Callahan und Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[DR96] Michèle Dion und Yves Robert. Mapping Affine Loop Nests: New Results. In *High Performance Computing and Networking*, Lecture Notes in Computer Science 919, Seiten 184–189. Springer-Verlag, 1996.

[DV96] A. Darte und F. Vivien. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Seiten 281–291. IEEE, Boston, MA, USA, Oktober 1996.

[Fea88] Paul Feautrier. Parametric Integer Programming. *Operations Research*, 22(3):243–268, 1988.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling Problem, Part I, One-dimensional Time. *Journal of Parallel Programming*, 21(5):313–347, 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling Problem, Part II, Multidimensional Time. *Journal of Parallel Programming*, 21(6):389–420, 1992.

[Gei97] Max Geigl. Parallelization of general loop nests in the polyhedron model. Diplomarbeit, Universität Passau, 1997. `http://www.fmi.uni-passau.de/loopo/doc/geigl-d.ps.gz`.

[Gün96] Robert Günz. The new LooPo scanner and parser. Unveröffentlichter Bericht, Februar 1996. `http://www.fmi.uni-passau.de/loopo/doc/guenz-p.ps.gz`.

[Hig97] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1997. `http://www.crpc.rice.edu/HPFF/hpf2/index.html`.

[HKT91] Seema Hiranandani, Ken Kennedy und Chau-Wen Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proceedings of Supercomputing '91*, November 1991.

[HKT92] Seema Hiranandani, Ken Kennedy und Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[Kei97] Harald Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Diplomarbeit, Universität Passau, 1997. `http://www.fmi.uni-passau.de/loopo/doc/keimer-d.ps.gz`.

[Lam74] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, Februar 1974.

[Len93] Christian Lengauer. Loop Parallelization in the Polytope Model. In E. Best, Herausgeber, *CONCUR'93*, Lecture Notes in Computer Science 715, Seiten 398–416. Springer-Verlag, 1993. `http://www.fmi.uni-passau.de/papers/Len93.html`.

[LF94] Hyuk J. Lee und José A. B. Fortes. Modular Mappings of rectangular algorithms. Technischer Bericht, School of Electrical Engineering, Purdue University, Juni 1994.

[Mei96] Wolfgang Meisl. Practical Methods for Scheduling and Allocation in the Polytope Model. Diplomarbeit, Universität Passau, 1996. `http://www.fmi.uni-passau.de/loopo/doc/meisl-d.ps.gz`.

[Mes95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[MR91] P. Mehrotra und J. Van Rosendale. *Programming Distributed Memory Architectures Using Kali*, Kapitel 19. Pitman Publishing, London, 1991.

[Pac95] Peter S. Pacheco. *A User's Guide to MPI*. Department of Mathematics, University of San Francisco, San Francisco, CA, USA, 1995.

[Par93] Parsytec Computer GmbH. *PARIX 1.2 Software Documentation*, März 1993.

[RB95] Shankar Ramaswamy und Prithviraj Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, Seiten 342–349. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[Sch97] Martina Schumergruber. Partitionierung paralleler Schleifensätze. Diplomarbeit, Universität Passau, 1997. `http://www.fmi.uni-passau.de/loopo/doc/\`
`schumergruber-d.ps.gz`.

[SL94] Daniel J. Scales und Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Seiten 101–114, 1994.

[SPB93] Ernesto Su, 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, Seiten 30–38. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[Su93] Ernesto Yi-Hsin Su. Automatic Parallelization of Regular Computations for Distributed-Memory Multicomputers. Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[Usm87] Riaz A. Usmani. *Applied Linear Algebra.* Marcel Dekker Inc., 1987.

[Wet95] Sabine Wetzel. Automatic Code Generation in the Polytope Model. Diplomarbeit, Universität Passau, November 1995. `http://www.fmi.uni-passau.de/loopo/doc/wetzel-d.ps.gz`.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Universität Passau, 1995. `http://www.fmi.uni-passau.de/loopo/doc/wieninger-d.ps.gz`.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.

[ZBG88] Hans P. Zima, Heinz-J. Bast und Michael Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC93] Hans P. Zima und Barbara Mary Chapman. Compiling for Distributed Memory Systems. *Proceedings of the IEEE*, 81(2):264–287, 1993.

Index

%, *siehe* Modulo-Operator
/, *siehe* Division, ganzzahlige

Abhängigkeit, **3**, 71
  enthalten einer ∼, **3**
  tragen
    der sequentiellen Komponente einer ∼, **3**, 88
  tragen einer ∼, **3**
Allokation, 19, 60
  Berechnung einer ∼, 35
  blockzyklische, 40
Anweisungsallokation, 18, 67

`barrier`, 16, 21, 62–65, 68, 85–89
Basis, 49
Basisblock, **29**, 63, 70, 88, 93
Basistransformation, 49
Berechnungsblock, 29, 31, 32, 64
`broadcast`, 16, 65, 88, 95, 96

`C`, 19, 66, 89, 94
Computer, 62, 66, 79

Datenallokation, **19**, *siehe* auch Allokation, 20, 39, 44, 48, 56, 58
Datenraum, **3**
  globaler, **19**
  lokaler, **20**
DIV, *siehe* Division, ganzzahlige
Division
  ganzzahlige, **4**
`do`, 4, 64, 72, 73, 76
`doacross`, 4, 23
`doall`, 4
Dreistufenmodell, 25

`else`, 81, 83, 87
extremize, **5**

FALLS, **40**, 41–45, 58
Family of Line Segments, *siehe* FALLS
`for`, 2, 4, 63, 64, 66, 70, 71, 74, 75, 79, 80, 82, 85–87, 93
`Fortran D`, 1
Fourier-Motzkin-Algorithmus, 55

`get`, 68–73, 81, 82, 84–87, 89, 92, 93, 95
Gitter, 16, 25
`group`, 64

`High Performance Fortran`, *siehe* `HPF`
Hostrechner, 16, 34, 58
`HPF`, 1, 29, 31, 46, 62, 65, 66, 74, 94–96

`if`, 24, 27, 29, 31, 32, 64, 66, 74, 77, 80–83, 85, 87, 93
Indexraum, **2**
  aufspannen eines ∼, **2**
  globaler, **20**
  lokaler, **20**
Indexraums, 87

`Kali`, 1, 95
`killgroup`, 64
Kontrollblock, 29

Line Segment, **40**, 41, 43–45, 47, 75, 95
`LOAD`, 65–70, 72–74, 76, 80–83, 85, 87–89, 93, 95
LooPo, 1, 3, 18, 22, 24, 31, 32, 34, 59, 60, 62, 70, 88, 89, 93, 95
LS, *siehe* Line Segment

Matrix
  inverse, 15, 37
    verallgemeinerte, **14**, 37
MOD, *siehe* Modulo-Operator
Modularabbildung, **5**, 7, 18, 20, 34, 36, 38–41, 43, 44, 46–48, 56–59, 73, 75, 93, 94
Modulo-Operator, **4**
`MPI`, 16, 85, 88

Node-Programm, 21, 74, 85
`normloops`, 88

OCR, 36, *siehe* auch Owner Computes Rule, 66
`ON`, 96
Owner, **19**, 31, 35, 47, 51, 59, 66–68, 72, 73, 80–83, 85–87, 92, 94, 95
Owner Computes Rule, 34, *siehe* auch OCR, 35, 66, 67
Ownerpart, **85**

`par`, 2, 4, 20, 27, 29, 31, 33, 63, 67, 73, 75–77, 81, 83, 85–87, 93, 95, 96
`PARADIGM`, 1
Parallelrechner, 20
`Parix`, 16

`Pascal`, 66
`PIP`, 55
PITFALLS, 40, **41**, 42, 43–45, 59, 94, 95
  verallgemeinerte, **42**
Processor Index Tagged Family of Line Segments , *siehe* PITFALLS
Projekt, **2**, 3, 4, 16–18, 20, 23–25, 27, 29, 32, 34, 35, 49, 58–63, 67, 69, 70, 74, 85, 88, 89, 92, 93, 94
Prozessorfeld, **17**, 18, 25, 56
  erweitertes, **25**
`put`, 68–73, 81, 85–87, 89, 92, 93, 95

`receive`, 16, 65, 69, 72, 81, 85, 88, 89, 95
Rechnerarchitektur, 16
`regroup`, 64

`SAM`, 34
Schedule, 74
Schleife, 18
  parallele, *siehe* auch `doall`, `doacross`, `par`, 16, **17**, 18, 21
  sequentielle, 4, *siehe* auch `for`
Schrittweite, 22
  ganzzahlige, 24
  rationale, 18, 21, 22
`send`, 16, 65, 69, 72, 75, 81, 85, 88, 89, 95
Set of Processor Index Tagged Fami-Lies of Line Segments, *siehe* SPITFALLS
Spaltenstufenform, 48, 78
Speicher
  gemeinsamer, 1, 20, 62
  lokaler, 44
  verteilter, 1, 20, 63, 65
SPITFALLS, **44**, 49, 55, 58–60, 92, 94, 95
SPMD, 1, 19, 20, 25, 29, 62, 66, 93
`STORE`, 65–70, 72–74, 76, 80–83, 85, 87–89, 93, 95
`SUPERB`, 1

`targout`, 61, 62, 88
Thread, 68, 72
Tiling, 89

`while`
  als Sonderfall für `LOAD`/`STORE`, 66
`while`, 2, 3, 17, 24, 64, 66, 70, 71, 86

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, daß alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind und daß diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 25. November 1997

Peter Faber