UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

Formale Ableitung und Implementierung paralleler MPI-Programme aus Homomorphismen

Diplomarbeit

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

Betreuer: Dr. Sergei Gorlatch

Bearbeiter: Holger Bischof

4. Januar 1998

Eidesstattliche Erklärung

Hiermit erkläre ich, Holger Bischof, eidesstattlich, diese Arbeit ohne Verwendung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und alle Ausführungen, die wörtlich oder sinngemäß übernommen sind, als solche gekennzeichnet zu haben. Weiter erkläre ich, daß diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den 4. Januar 1998

Holger Bischof

Danksagung

Ich möchte allen herzlich danken, die mir direkt oder indirekt bei der Erstellung dieser Arbeit beigestanden haben. Ganz besonders danke ich Herrn Prof. Christian Lengauer, Ph.D., insbesondere für seine zügige Korrektur. Zu großem Dank verpflichtet bin ich auch meinem Betreuer Dr. Sergei Gorlatch für seine fachliche Betreuung und tatkräftige Unterstützung.

Zusammenfassung

Die Arbeit beschreibt die Entwicklung paralleler Programme mithilfe des Bird-Meertens Formalismus. Es wird eine Unterklasse der Divide-and-Conquer Algorithmen betrachtet, die verteilten Homomorphismen (DH), und eine parametrisierte Familie paralleler Implementierungen dafür hergeleitet. Weiterhin wird anhand der Fast Fourier Transformation die Anpassung an das DH-Format beschrieben, wobei hierfür noch spezielle Optimierungen vorgestellt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Formale Methoden sind eine Möglichkeit, die Probleme bei der Programmierung paralleler Maschinen zu meistern. Sie sind eine Alternative zur gegenwärtigen Praxis der low-level-Programmierung, bei der die Korrektheit und Performance vom Programmierer abhängt und meist mit viel Testen, Fehlersuche und architekturabhängigem Tunen verbunden ist.

In dieser Arbeit werden mithilfe von Transformationen aus einer intuitiv korrekten Spezifikation Programme abgeleitet. Dazu wird der Bird-Meertens-Formalismus (BMF) benutzt, wobei Funktionen höherer Ordnung auf Listen die Spezifikation und "equational reasoning" die Programmtransformation darstellen.

Genauer gesagt, wird ein skelettbasierter Ansatz wie in [Col88] benutzt, d.h. es werden Implementierungen für eine Klasse von Problemen mit gleichen Mustern von Parallelität hergeleitet. Dieser Ansatz verteilt das Erstellen paralleler Programme auf zwei Phasen bzw. zwei Arten von Programmierern.

- Bei der *Systemprogrammierung* müssen erst parallele Skelette für entsprechende nichttriviale Anwendungen gefunden werden. Diese müssen dann noch beweisbar korrekt, portabel und mit einer guten Performance implementiert werden. Diese Phase erfordert viel Wissen über den Formalismus und der Software der parallelen Maschine.

- Bei der *Anwendungsprogrammierung* sollte man vom Formalismus und Details der Maschine nichts mehr sehen. Die Aufgabe besteht darin, die Anwendung an ein Skelett anzupassen. Dazu müssen Änderungen und Optimierungen in der Spezifikation vorgenommen werden.

In dieser Arbeit werden beide Phasen der Skelettprogrammierung beschrieben. In der Systemphase werden Divide-and-Conquer-Algorithmen klassifiziert, wobei eine Unterklasse untersucht wird, sogenannte DH (Distributable Homomorphism). Für diese Klasse wird eine Familie paralleler Implementierungen formal hergeleitet und als SPMD-Programm (Single Program Multiple Data) dargestellt. Als Programmiersprache wird MPI (Message Passing Interface) benutzt.

Als Beispiel für die Anwendungsphase wird die mathematische Spezifikation der FFT (Fast Fourier Transformation) betrachtet, die mithilfe von Transformationen unter Benutzung von mathematischen Eigenschaften an das DH-Format angepaßt wird.

Die Mächtigkeit dieses Ansatzes wird dadurch demonstriert, daß die hergeleitete parametrisierte Familie paralleler Implementierungen im Fall der FFT zu effizienten Implementierungen führt, die in der Praxis verwendet werden.

Diese Arbeit ist folgendermaßen aufgebaut. Kapitel 2 ist eine Einführung in BMF. Es werden Listen und grundlegende Funktionen, sowie einfache Beziehungen dargestellt. Kapitel 3 erweitert den BMF auf geschachtelte Listen, die nötig sind, um den Hauptsatz für DHs in Kapitel 4 formulieren und beweisen zu können. Der Hauptsatz beschreibt eine parametrisierte Familie von Implementierungen, die in Kapitel 5 nach MPI umgesetzt und optimiert werden.

Während bis Kapitel 5 die Systemphase der Programmentwicklung beschrieben wird, zeigt Kapitel 6 die Anwendungsphase. Hier wird gezeigt, wie die FFT systematisch an das DH-Format angepaßt wird.

Kapitel 7 faßt die Arbeit zusammen und gibt einen Ausblick für künftige Arbeiten an. Im Anhang ist das DH-Skelett in Form eines MPI-Programms zu finden.

Kapitel 2

Bird-Meertens Formalismus

Der Bird-Meertens Formalismus (BMF) [Bir88] ist eine funktionale Notation, die auf den Programmentwurf ausgerichtet ist [Col88]. Genauso wie in der Mathematik Ausdrücke nach den dort geltenden Regeln umgeformt werden können, so können auch in BMF Algorithmen umgeformt, bzw. transformiert werden. Die Vorgehensweise bei der Programmentwicklung ist die, daß man mit einer intuitiven, offensichtlich korrekten Spezifikation beginnt. Diese kann dann durch sukzessive Anwendung der durch die Theorie gegebenen Identitäten in eine effiziente Spezifikation umgewandelt werden.

Einen Überblick über den BMF, seine Anwendung und seine Möglichkeiten bietet Skillicorn in [Ski92]. Er propagiert den Formalismus als Möglichkeit zur architekturunabhängigen Programmierung von Parallelrechnern [Ski90].

In dieser Arbeit wird die Theorie der Listen, die hauptsächlich von Bird [Bir87] entwickelt wurde, benutzt. In diesem Kapitel werden erst Listen vorgestellt und danach Funktionen, die in späteren Kapiteln benötigt werden. Diese bilden die Grundlage für das Hauptergebnis dieser Arbeit, dem Hauptsatz für DHs, in Kapitel 4.

2.1 Listen

Eine Liste ist eine Folge von Werten eines bestimmten Typs oder eine leere Liste. Als Konstruktor wird üblicherweise ":" benutzt, der ein Element an eine Liste anfügt.

Definition 1. $[\alpha]$ *ist die Menge der Listen vom Typ* α*. Folgende Listen sind darin enthalten:*

- *Die leere Liste ist immer in* $[\alpha]$*:* $[\,] \in [\alpha]$*.*

- $xs \in [\alpha]$ *und* $x \in \alpha$ *genau dann, wenn* $x : xs \in [\alpha]$,
  wobei $a : [x_1, \ldots, x_n] = [a, x_1, \ldots, x_i]$.

Da mit dem Konstruktor “:” eine Liste sequentiell ausgewertet wird, wird im folgenden die Listenkonkatenation $+\!\!+$ als Listenkonstruktor verwendet. Weiterhin werden nur noch nichtleere Listen betrachtet.

Definition 2. *Seien* $xs, ys \in [\alpha]$, $x \in \alpha$ *dann ist:*

$$[x] +\!\!+ xs = x : xs$$
$$x : xs +\!\!+ ys = x : (xs +\!\!+ ys)$$

Als Beweismittel für induktiv aufgebaute Datenstrukturen wird die Induktion über den Strukturaufbau benutzt. Bei den hier verwendeten Datenstrukturen, den Listen, wird über die Länge der Liste induziert.

In BMF wird Induktion für grundlegende Beziehungen benutzt, aufwendige Aussagen lassen sich oft durch Gleichheiten von Funktionen, also ohne Listenargumente beweisen.

Definition 3. *Die Länge einer Liste wird mit Hilfe folgender Funktion berechnet:*

$$|.| : [\alpha] \rightarrow nat$$
$$|[a]| = 1$$
$$|x +\!\!+ y| = |x| + |y|$$

Mithilfe von Definition 3 ist es möglich, Listen einer festen Länge n einzuführen.

$$[\alpha]_n = \{x | x \in [\alpha] \text{ und } |x| = n\}$$

Auf diesen Listen wird die *Projektion* folgendermaßen definiert.

$$\pi_i : [\alpha]_n \rightarrow \alpha \cup \{\text{undef.}\}$$
$$\pi_i\,[x_0, \ldots, x_{n-1}] = \begin{cases} x_i & \text{falls } 0 \leq i < n \\ \text{undef.} & \text{sonst} \end{cases}$$

Notation 1. *Zur besseren Handhabung der Projektion benutzt man eine einfachere Schreibweise. Sei* $x \in [\alpha]_n$ *und* $0 \leq i < n$*, dann ist:*

$$x_{(i)} = \pi_i\ x$$

Die Projektion einer durch den Listenkonstruktor “$+\!\!+$” zusammengesetzten Liste sieht folgendermaßen aus:

$$(x +\!\!+ y)_{(i)} = \begin{cases} x_{(i)} & \text{falls } 0 \leq i < |x| \\ x_{(i-|x|)} & \text{sonst} \end{cases}$$

2.2 Homomorphismen

Um parallele Berechnungen beschreiben zu können wird eine Klasse von Funktionen eingeführt, die sogenannten *Homomorphismen.*

Definition 4. *Eine Funktion* h *auf Listen heißt Homomorphismus genau dann, wenn ein binärer Operator* $\odot$ *existiert, so daß für alle Listen* x *und* y *gilt:*

$$h(x +\!\!+\, y) = h\ x\ \odot\ h\ y$$

Intuitiv wird Homomorphismus h so berechnet, daß zuerst h auf Teillisten x und y rekursiv berechnet wird und danach die Teilergebnisse mithilfe eines *combine*-Operators $\odot$ wieder verschmolzen werden.

Homomorphismen sind Divide-and-Conquer-Funktionen und können wie diese implementiert werden (Abbildung 2.1, linke Seite). Zuerst werden die Daten mithilfe der *divide*-Funktion verteilt, danach wird eine *base*-Funktion auf die einzelnen Elemente angewendet und abschließend werden die Daten gemäß der *conquer*-Funktion wieder verschmolzen. Bei den Homomorphismen entspricht die *divide*-Funktion der Konkatenation $+\!\!+$ und die *conquer*-Funktion dem binären Operator $\odot$.

Während des Verschmelzen der Daten können bei den Homomorphismen auch noch zusätzliche Berechnungen anfallen. Diese Berechnungen müssen auf immer größer werdenden Listen durchgeführt werden. Dadurch entstehen große Kommunikationskosten und eine schlechte Lastverteilung in dieser Phase.

In [Gor96a] wurde ein alternatives Implementierungsschema vorgeschlagen, das auf der rechten Seite von Abbildung 2.1 skizziert ist. Hier werden die Daten ohne Berechnungen verteilt, danach werden mehrere verteilte Berechnungen auf diesen Daten durchgeführt und am Ende werden die Daten wieder verschmolzen ohne zusätzliche Berechnungen.

Diese Implementierung hat weiterhin den Vorteil, daß bei der Komposition solcher Homomorphismen das Verschmelzen und erneute Aufteilen weggelassen werden kann. Außerdem kommt es in der Praxis häufig vor, daß Daten auf mehreren Leitungen schon verteilt an die entsprechenden Prozessoren geschickt werden.

Das Problem dieser verteilten Implementierung ist jedoch, daß sich nicht alle Homomorphismen auf diese Weise implementieren lassen. Im Kapitel 4 wird eine Unterklasse der Divide-and-Conquer-Algorithmen vorgestellt, für die dies möglich ist.

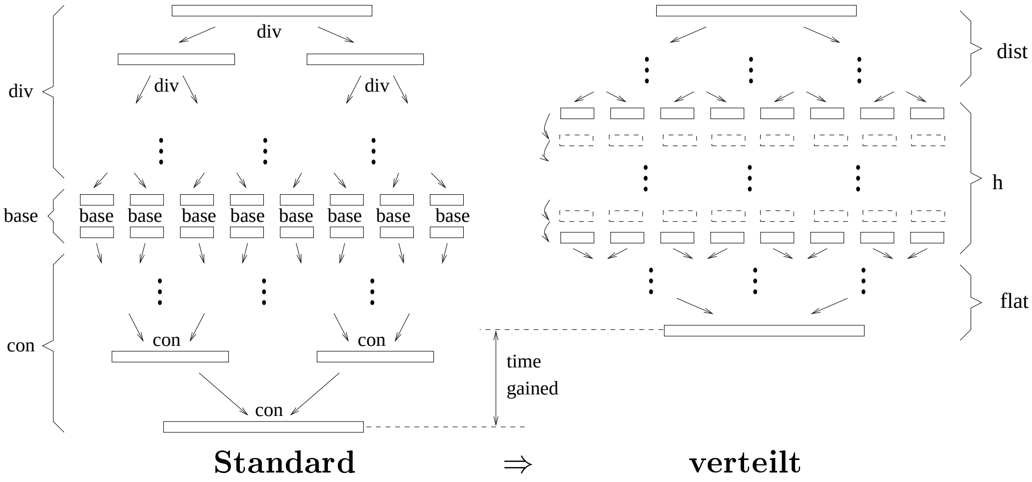

Abbildung 2.1: Implementierung für Homomorphismen

Beispiele von Homomorphismen:

- *map* wendet eine unäre Funktion auf alle Elemente einer Liste an:

$$\begin{aligned}&map : (\alpha \to \alpha) \to [\alpha] \to [\alpha]\\&map\ f\ [a] = [f\ a]\\&map\ f\ (x +\!\!\!+\, y) = (map\ f\ x) +\!\!\!+\, (map\ f\ y)\end{aligned}$$

  oder auch mit Hilfe der Projektion (i beliebig):

$$(map\ f\ x)_{(i)} = f\ x_{(i)}$$

- *red* steht für Reduktion und verknüpft alle Elemente einer Liste mit einem binären Operator:

$$\begin{aligned}&red : (\alpha \times \alpha \to \alpha) \to [\alpha] \to \alpha\\&red(\oplus)\ [a] = a\\&red(\oplus)\ (x +\!\!\!+\, y) = (red(\oplus)\ x) \oplus (red(\oplus)\ y)\end{aligned}$$

- *scan* berechnet die Präfixsummen einer Liste, d.h. das i-te Element der Ergebnisliste ist gleich der Reduktion aller vorherigen Elemente der Ausgangsliste:

$$\begin{aligned}
&scan : (\alpha \times \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&scan\,(\oplus)\ [a] = [a] \\
&scan\,(\oplus)\ (x +\!\!+\, y) = scan\,(\oplus)\ x +\!\!+\ map\,(last(scan\,(\oplus)\ x)\oplus)(scan\,(\oplus)\ y)
\end{aligned}$$

wobei *last* das letzte Element einer Liste liefert. Mit Hilfe der Projektion:

$$(scan\,(\oplus)\ [x_0, \ldots, x_{n-1}])_{(i)} = red\,(\oplus)\ [x_0, \ldots, x_i]$$

BMF-Ausdrücke werden aus Funktionen mithilfe der Funktionskomposition aufgebaut.

Sei $f : \beta \to \gamma, \quad g : \alpha \to \beta$, dann gilt für alle $x \in \alpha$:
$(f \circ g)\, x = f(g\, x)$

2.3 Powerlisten

Homomorphismen wurden in Kapitel 2.2 auf Listen beliebiger Länge definiert. Nach dieser Definition können Listen in zwei beliebig lange Teillisten geteilt werden. Dies hat den Nachteil, daß die Berechnungen auf den Teillisten je nach Länge unterschiedlich lange dauern. Damit immer in zwei gleich lange Listen geteilt werden kann, beschränken wir uns auf Listen der Länge einer Zweierpotenz. Diese Listen werden *Powerlisten* genannt [Mis94].

Im Nachfolgenden wird davon ausgegangen, daß alle Listen Powerlisten sind. Auf Powerlisten definieren wir zwei neue Funktionen:

Definition 5. *Funktion zip:*

$$\begin{aligned}
&zip : (\alpha \times \alpha \to \alpha) \to [\alpha]_n \to [\alpha]_n \to [\alpha]_n \\
&zip\,(\odot)([a],[b]) = [a \odot b] \\
&zip\,(\odot)(x_1 +\!\!+\, x_2, y_1 +\!\!+\, y_2) = zip\,(\odot)(x_1, y_1) +\!\!+\ zip\,(\odot)(x_2, y_2)
\end{aligned}$$

oder anschaulich:

$$zip\,(\odot)([x_1, \ldots, x_n], [y_1, \ldots y_n]) = [x_1 \odot y_1, \ldots, x_n \odot y_n]$$

zip ist eine Funktion, die aus zwei Listen gleichen Typs und gleicher Länge eine Liste erzeugt, indem die jeweiligen Elemente mit einer binären Operation verknüpft werden.

Definition 6. *Sei n eine Zweierpotenz, dann ist $dist_n$ auf Powerlisten definiert, falls n kleiner oder gleich der Länge der Argumentliste ist.*

$$\begin{aligned}
&dist_n : [\alpha] \to [[\alpha]]_n \\
&dist_1\ x = [x] \\
&dist_n\ (x +\!\!+ y) = (dist_{n/2}\ x) +\!\!+ (dist_{n/2}\ y)
\end{aligned}$$

Die Funktion $dist_n$ macht aus einer einfachen Liste eine Liste von Listen, die wir als zweidimensionale Liste bezeichnen werden, wobei die äußere Liste n Listen enthält.

Hierzu ein einfaches Beispiel:

$$dist_4\ [x_0, \ldots x_7] = [[x_0, x_1], [x_2, x_3], [x_4, x_5], [x_6, x_7]]$$

2.4 Einfache Beziehungen

In diesem Kapitel werden Beziehungen zwischen den in den letzten Kapiteln vorgestellten Funktionen bewiesen.

Lemma 1. *Für $x, y \in [\alpha]_n$ gilt:*

$$\begin{aligned}
&dist_n \circ zip\,(\oplus)(x, y) = zip^2(\oplus)(dist_n\ x, dist_n\ y) \qquad (2.1) \\
&wobei \quad zip^2(\oplus) = zip\,(zip\,(\oplus)
\end{aligned}$$

Beweis. Induktion über n.
Induktionsanfang mit $a, b \in \alpha$:

$$\begin{aligned}
&dist_1 \circ zip\,(\oplus)([a], [b]) = dist_1[a \oplus b] = [[a \oplus b]] \\
=&zip^2(\oplus)([[a]], [[b]]) = zip^2(\oplus)(dist_1[a], dist_1[b])
\end{aligned}$$

Induktionsschritt mit $x_1, x_2, y_1, y_2 \in [\alpha]_{n/2}$:

$$\begin{aligned}
&dist_n \circ zip\,(\oplus)(x_1 +\!\!+ x_2, y_1 +\!\!+ y_2) \\
= &\{\ \text{Definition } zip\ \} \\
&dist_n(zip\,(\oplus)(x_1, y_1) +\!\!+ zip\,(\oplus)(x_2, y_2)) \\
= &\{\ \text{Definition } dist_n\ \} \\
&(dist_{n/2} \circ zip\,(\oplus)(x_1, y_1)) +\!\!+ (dist_{n/2} \circ zip\,(\oplus)(x_1, y_1)) \\
= &\{\ \text{Induktionsannahme}\ \} \\
&(zip^2(\oplus)(dist_{n/2}\ x_1, dist_{n/2}\ x_2)) +\!\!+ (zip^2(\oplus)(dist_{n/2}\ y_1, dist_{n/2}\ y_2)) \\
= &\{\ \text{Definition } zip\ \} \\
&zip^2(\oplus)((dist_{n/2}\ x_1) +\!\!+ (dist_{n/2}\ x_2), (dist_{n/2}\ y_1) +\!\!+ (dist_{n/2}\ y_2)) \\
= &\{\ \text{Definition } dist\ \} \\
&zip^2(\oplus)(dist_n(x_1 +\!\!+ x_2), dist_n(y_1 +\!\!+ y_2)
\end{aligned}$$

Lemma 2.

$$map\,(f \circ g) = (map\;\; f) \circ (map\;\; g) \tag{2.2}$$

Beweis. Induktion über die Länge der Liste. Induktionsanfang ist trivial. Induktionsschritt:

$$\begin{aligned} map\,(f \circ g)(x \mathbin{+\!\!+} y) &= (map\,(f \circ g)\; x) \mathbin{+\!\!+} (map\,(f \circ g)\; y) \\ &= ((map\;\; f\; x) \circ (map\;\; g\; x)) \mathbin{+\!\!+} ((map\;\; f\; y) \circ (map\;\; g\; y)) \\ &= map\,(f)(x \mathbin{+\!\!+} y) \circ map\,(g)(x \mathbin{+\!\!+} y) \end{aligned}$$

□

Lemma 3. *Auf zweidimensionalen Listen gilt für eine beliebige Funktion f und $map^2\; f = map\,(map\;\; f)$:*

$$(map\;\; f) \circ red\,(\mathbin{+\!\!+}) = red\,(\mathbin{+\!\!+}) \circ (map^2\;\; f) \tag{2.3}$$

Beweis. Bewiesen wird wieder durch Induktion über die Länge der Liste: Induktionsanfang:($x \in [\alpha]$)

$$\begin{aligned} ((map\;\; f) \circ red\,(\mathbin{+\!\!+}))[x] = map\;\; f\;\; x &= red\,(\mathbin{+\!\!+})[map^2\; f\; x] \\ &= (red\,(\mathbin{+\!\!+}) \circ (map^2\;\; f))\; x \end{aligned}$$

Induktionsschritt:

$$\begin{aligned} &((map\;\; f) \circ red\,(\mathbin{+\!\!+}))(x \mathbin{+\!\!+} y) \\ &= (((map\;\; f) \circ red\,(\mathbin{+\!\!+}))\; x) \mathbin{+\!\!+} (((map\;\; f) \circ red\,(\mathbin{+\!\!+}))\; y) \\ &= (red\,(\mathbin{+\!\!+}) \circ (map^2\;\; f)\; x) \mathbin{+\!\!+} (red\,(\mathbin{+\!\!+}) \circ (map^2\;\; f)\; y) \\ &= red\,(\mathbin{+\!\!+})((map^2\;\; f)\; x) \mathbin{+\!\!+} ((map^2\;\; f)\; y) \\ &= (red\,(\mathbin{+\!\!+}) \circ (map^2\;\; f))(x \mathbin{+\!\!+} y) \end{aligned}$$

□

Lemma 4. *Sei $\odot$ ein beliebiger assoziativer Operator, dann gilt:*

$$red\,(\odot) \circ red\,(\mathbin{+\!\!+}) = red\,(\odot) \circ map\,(red\,(\odot)) \tag{2.4}$$

Beweis. Induktionsanfang:($x \in [\alpha]$)

$$\begin{aligned} (red\,(\odot) \circ red\,(\mathbin{+\!\!+}))[x] &= red\,(\odot)\; x = map\,(red\,(\odot))[x] \\ &= red\,(\odot) \circ map\,(red\,(\odot))[x] \end{aligned}$$

Induktionsschritt:

$$
\begin{aligned}
(red\,(\odot) \circ red\,(+\!\!\!+))(x +\!\!\!+ y) \\
&= red\,(\odot)((red\,(+\!\!\!+)\ x) +\!\!\!+ (red\,(+\!\!\!+)\ y)) \\
&= ((red\,(\odot) \circ red\,(+\!\!\!+))\ x) +\!\!\!+ ((red\,(\odot) \circ red\,(+\!\!\!+))\ y) \\
&= ((red\,(\odot) \circ map\,(red\,(\odot)))\ x) +\!\!\!+ ((red\,(\odot) \circ map\,(red\,(\odot)))\ y) \\
&= red\,(\odot)((map\,(red\,(\odot)x)) +\!\!\!+ (map\,(red\,(\odot)x))) \\
&= (red\,(\odot) \circ map\,(red\,(\odot)))(x +\!\!\!+ y)
\end{aligned}
$$

$\square$

Kapitel 3

Erweiterung auf geschachtelte Listen

3.1 Geschachtelte Listen

In diesem Kapitel wird BMF auf geschachtelte Listen endlicher Länge erweitert. Dies geschieht auf natürliche Weise, da der Typ der Listenelemente beliebig ist, also auch wieder eine Liste sein kann. Geschachtelte Listen beschränkter Länge sind definiert durch:

$$[\alpha]_{(n_1 \times \cdots \times n_d)} = [[\alpha]_{(n_2 \times \cdots \times n_d)}]_{n_1}$$

Diese geschachtelten Listen werden im folgenden als d-dimensionale Listen oder auch als Listen der Schachtelungstiefe d bezeichnet. Sie besitzen folgende Eigenschaften:

- Sie sind endlich.
- Alle einfachen Elemente (das sind die Elemente der d-ten Schachtelungstiefe) sind vom Typ α.
- Alle Teillisten auf der gleichen Schachtelungstiefe haben die gleiche Länge. D.h. alle i-fach geschachtelten Teillisten haben die Länge n_i.

Die *Projektion* wird für geschachtelte Listen induktiv erweitert.

$$\pi_{(i_1,\dots,i_d)} : [\alpha]_{(n_1 \times \cdots \times n_d)} \to \alpha \cup \{\text{undef.}\}$$

$$\pi_{(i_1,\dots,i_d)}\, x = \begin{cases} \pi_{(i_2,\dots,i_d)}\,(\pi_{i_1}\, x) & \text{falls } 0 \le i_1 < n_1 \\ \text{undef.} & \text{sonst} \end{cases}$$

Notation 2. *Auch bei der erweiterten Projektion benutzen wir eine einfachere Notation. Sei* $y \in [\alpha]_{(n_1 \times \ldots \times n_d)}$ *und* $\forall j : 1 \leq j \leq d : 0 \leq i_j < n_j$, *dann:*

$$y_{(i_1,\ldots,i_d)} = \pi_{(i_1,\ldots,i_d)}\ y$$

Beispiel 1. *Ein paar kleine Beispiele höherdimensionaler Listen:*

$$\begin{aligned}
&x \stackrel{def}{=} [[a_{00}, a_{01}, a_{02}], [a_{10}, a_{11}, a_{12}]] \in [\alpha]_{(2\times 3)} \\
&y \stackrel{def}{=} [\ [[0,1],[2,3]]\ ,\ [[4,5],[6,7]]\ ] \in [nat]_{(2\times 2\times 2)} \\
&\pi_2(\pi_1\ x) = x_{(1,2)} = a_{12} \\
&\pi_0(\pi_1(\pi_0\ y)) = y_{(0,1,0)} = 2
\end{aligned}$$

3.2 Erweiterung bekannter Funktionen

In diesem Kapitel werden die schon bekannten Funktionen induktiv erweitert, so daß sie auch auf geschachtelten Listen benutzt werden können

$$\begin{aligned}
map^d\ f &= map\,(map^{d-1}\ f) \\
zip^d(\odot) &= zip\,(zip^{d-1}(\odot)) \\
red^d(\odot) &= red\,(\odot) \circ map^{d-1}(red\,(\odot))
\end{aligned}$$

Um höherdimensionale Listen zu erzeugen, wird *dist* ebenfalls auf natürliche Weise erweitert.

$$dist_{(n_1,\ldots,n_d)} = map\,(dist_{(n_2,\ldots,n_d)}) \circ dist_{n_1}$$

Beispiel:

$$dist_{(2,3)}\ [x_0, \ldots x_5] = [\ [[x_0],[x_1],[x_2]], [[x_3],[x_4],[x_5]]\ ]$$

Die inverse Operation, die aus mehrdimensionalen Listen wieder einfache Listen macht ist eine Reduktion mit der Konkatenation als Operator:

$$\begin{aligned}
red\,(+\!\!+) \circ dist_{n_1} &= id \\
red^d(+\!\!+) \circ dist_{(n_1,\ldots,n_d)} &= id
\end{aligned}$$

Man beachte, daß *red* linksinvers ist, nicht jedoch rechtsinvers:

$$dist_{n_1} \circ red\,(+\!\!+) \neq id$$

3.3 Funktionen auf geschachtelten Listen

In diesem Kapitel werden neue Funktionen auf geschachtelten Listen definiert. Betrachten wir den zweidimensionalen Fall. Für eine Funktion $f : [\alpha] \to [\alpha]$ und eine zweidimensionale Liste $x \in [\alpha]_{(n_1 \times n_2)}$ wendet *map f x* die Funktion f auf alle inneren Listen von x an:($\mathrm{x}_i \in [\alpha]_m$)

$$map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

Sieht man zweidimensionale Listen als Matrizen an, wendet *map f* eine Funktion f auf alle Zeilen der Matrix an. Will man ein Funktion f auf alle Spalten einer Matrix anwenden, so geht das mit den bisher definierten Funktionen nicht. Dafür brauchen wir die *Transposition.*

Definition 7. *Die Definition der Transposition:*

$$\begin{aligned} &tr : [\alpha]_{n_1 \times n_2} \to [\alpha]_{n_2 \times n_1} \\ &(tr\ x)_{(i_1,i_2)} = x_{(i_2,i_1)} \end{aligned}$$

Es wird hier bewußt auf eine rekursive Definition verzichtet, da diese zu aufwendig wäre. Eine übliche rekursive Definition kann in Form eines Lemmas mithilfe von Definition 7 bewiesen werden.

Lemma 5. *Seien $x, y \in [\alpha]_{(n_1 \times n_2)}$, dann gilt:*

$$zip\,(+\!\!+)(tr\ x, tr\ y) = tr\,(x +\!\!+ y)$$

Beweis. Es wird bewiesen, daß alle Elemente der linken und der rechten Seite gleich sind.

$$\begin{aligned} &(zip\,(+\!\!+)(tr\ x, tr\ y))_{(i_1,i_2)} \\ &= ((tr\ x)_{i_1} +\!\!+ (tr\ y)_{i_1})_{i_2} \\ &= \begin{cases} (tr\ x)_{(i_1,i_2)} & \text{falls } j \leq l, l = |(tr\ x)_{i_1}| \\ (tr\ y)_{(i_1,i_2-l)} & \text{sonst} \end{cases} \\ &= \begin{cases} x_{(i_2,i_1)} & \text{falls } j \leq l, l = |x| \\ y_{(i_2-l,i_1)} & \text{sonst} \end{cases} \\ &= (tr\,(x +\!\!+ y))_{(i_1,i_2)} \end{aligned}$$

□

Jetzt können wir eine Funktion definieren, die $f : [\alpha] \to [\alpha]$ auf alle Spalten einer Matrix anwendet:

$$\begin{aligned} mapdim^{(1,2)} &: ([\alpha] \to [\alpha]) \to [\alpha]_{(n_1 \times n_2)} \to [\alpha]_{(n_1 \times n_2)} \\ mapdim^{(1,2)}\ f &= tr \circ (map\ f) \circ tr \end{aligned}$$

Die Matrix wird transponiert, danach wird f auf alle Zeilen gemappt und anschließend wird nochmals transponiert. In Abbildung 3.1 sind die beiden Möglichkeiten zu mappen dargestellt.

$$mapdim^{(2,2)}\ f\ x = \begin{array}{l} \big[\, f[a_{11} \quad \ldots \quad a_{1m}], \\ \quad f[a_{21} \quad \ldots \quad a_{2m}], \\ \quad \vdots \\ \quad f[a_{m1} \quad \ldots \quad a_{mm}] \,\big] \end{array}$$

$$mapdim^{(1,2)}\ f\ x = tr \left(\begin{array}{l} \big[\, f[a_{11} \quad \ldots \quad a_{m1}], \\ \quad f[a_{12} \quad \ldots \quad a_{m2}], \\ \quad \vdots \\ \quad f[a_{1m} \quad \ldots \quad a_{mm}] \,\big] \end{array} \right)$$

Abbildung 3.1: Die beiden Möglichkeiten zu mappen

Als nächstes werden die Funktionen tr und $mapdim^{(1,2)}$ auf d-dimensionale Listen erweitert.

Definition 8. *Für $1 \leq l_1 \leq l_2 \leq d$ ist die allgemeine Transposition definiert. Dabei steht chdim für "change dimension".*

$$\begin{aligned} chdim^{(l_1,l_2)} &: [\alpha]_{(n_1 \times \cdots \times n_d)} \to [\alpha]_{(m_1 \times \cdots \times m_d)} \\ &\quad wobei\ m_{l_1} = n_{l_2}, m_{l_2} = n_{l_1}, \forall k \notin \{l_1, l_2\} : m_k = n_k \\ (chdim^{(l_1,l_2)}\ x)_{(i_1,\ldots,i_d)} &= x_{(j_1,\ldots,j_d)} \\ &\quad wobei\ i_{l_1} = j_{l_2}, i_{l_2} = j_{l_1}, \forall k \notin \{l_1, l_2\} : i_k = j_k \end{aligned}$$

$mapdim^{(i,d)}$ wendet im allgemeinen Fall eine Funktion, definiert auf Listen, auf alle Listen der i-ten Dimension an.

Definition 9. *Für $1 \leq i \leq d$ ist folgendes definiert:*

$$mapdim^{(i,d)} : ([\alpha]_{n_i} \to [\alpha]_{n_i}) \to [\alpha]_{(n_1 \times \cdots \times n_d)} \to [\alpha]_{(n_1 \times \cdots \times n_d)}$$
$$mapdim^{(i,d)}\ f = chdim^{(i,d)} \circ (map^{\,d-1}\ f) \circ chdim^{(i,d)}$$

$mapdim^{(i,d)}$ wendet im allgemeinen Fall eine Funktion, die auf Listen definiert ist, auf alle Listen der i-ten Dimension an.

3.4 Beziehungen höherdimensionaler Funktionen

Wir beweisen hier zwei Eigenschaften von eben eingeführten Funktionen. Diese werden im folgenden Kapitel benutzt.

Lemma 6. *Sei $x \in [\alpha]_{(n_1 \times \cdots \times n_d)}$ und $1 \leq n \leq m < d$, dann gilt:*

$$map\,(chdim^{(n,m)})\ x = chdim^{(n+1,m+1)}\ x \tag{3.1}$$

Beweis.

$$
\begin{aligned}
& (map\,(chdim^{(n,m)})\ x)_{(i_1,\dots,i_d)} \\
=\ & \{\ \text{Definition } map\ \} \\
& (chdim^{(n,m)}\ x_{(i_1)})_{(i_2,\dots,i_d)} \\
=\ & \{\ \text{Sei } j_{n+1} = i_{m+1}, j_{m+1} = i_{n+1}, \forall k \notin \{n+1, m+1\} : j_k = i_k\ \} \\
& x_{(j_1,\dots,j_d)} \\
=\ & \{\ \text{Definition } chdim^{(n+1,m+1)}\ \} \\
& (chdim^{(n+1,m+1)}\ x)_{(i_1,\dots,i_d)}
\end{aligned}
$$

□

Lemma 7. *Sei $x \in [\alpha]_{(n_1 \times \cdots \times n_{d+1})}, f : [\alpha] \to [\alpha]$ und $a \leq d$, dann gilt:*

$$map\,(mapdim^{(a,d)}\ f)\ x = mapdim^{(a+1,d+1)}\ f\ x \tag{3.2}$$

Beweis.

$$
\begin{aligned}
& map\,(mapdim^{(a,d)}\ f) \\
= & \ \{ \text{ Definition von } mapdim^{(a,d)} \ \} \\
& map\,(chdim^{(a,d)} \circ (map^{\,d-1}\ f) \circ chdim^{(a,d)}) \\
= & \ \{ \ (2.2) \ \} \\
& map\,(chdim^{(a,d)}) \circ (map^{\,d}\ f) \circ map\,(chdim^{(a,d)}) \\
= & \ \{ \ (3.1) \ \} \\
& chdim^{(a+1,d+1)} \circ (map^{\,d}\ f) \circ chdim^{(a+1,d+1)} \\
= & \ \{ \text{ Definition von } mapdim^{(a+1,d+1)} \ \} \\
& = mapdim^{(a+1,d+1)}\ f
\end{aligned}
$$

$\square$

Kapitel 4

Verteilbare Homomorphismen

4.1 Definition

In diesem Kapitel definieren wir eine spezielle Divide-and-Conquer-Klasse die, wie wir sehen werden, verteilt implementiert werden kann (siehe Abbildung 2.1). Diese neue Klasse, die erstmals in [Gor96a] eingeführt wurde, ist eine Unterklasse der Homomorphismen und wird als DH (Distributable Homomorphisms) bezeichnet. DHs zeichnen sich durch folgende Eigenschaften aus:

- Man betrachtet nur Powerlisten (siehe Kapitel 2.3)
- In der divide-Phase wird in genau zwei gleich große Listen geteilt. Ansonsten bleibt die Liste unverändert.
- Die conquer-Phase besteht aus komponentenweisen Operationen
- Die Basisfunktion ist die identische Abbildung.

Oder formal ausgedrückt:

Definition 10. *Seien $\oplus, \otimes : \alpha \to \alpha \to \alpha$ beliebige binäre Operationen, so ist $\oplus \updownarrow \otimes : [\alpha] \to [\alpha]$ ein DH, falls für beliebige Listen x und y gleicher Länge gilt:*

$$
\begin{aligned}
(\oplus \updownarrow \otimes)[a] &= [a] \\
(\oplus \updownarrow \otimes)(x +\!\!+\, y) &= \mathit{zip}\,(\oplus)(u, v) +\!\!+\, \mathit{zip}\,(\otimes)(u, v) \\
\mathit{wobei}\ u &= (\oplus \updownarrow \otimes)x, \quad v = (\oplus \updownarrow \otimes)y
\end{aligned}
$$

Beispiel 2. *redd ist eine verteilte Reduktion, ersetzt also alle Elemente der Liste durch die Reduktion der Liste:*

$$redd(\oplus)[a_1, \ldots, a_n] = [b, \ldots, b], \quad wobei\ b = a_1 \oplus \ldots \oplus a_n$$

Diese Funktion ist ein DH:

$$\begin{aligned} redd(\oplus)(x +\!\!+\, y) &= zip\,(\oplus)(u, v) +\!\!+\, zip\,(\oplus)(u, v) \\ wobei\ u = redd(\oplus)\ x, &\quad v = redd(\oplus)\ y \end{aligned}$$

Es ist folglich $redd(\oplus) = \oplus \updownarrow \oplus$

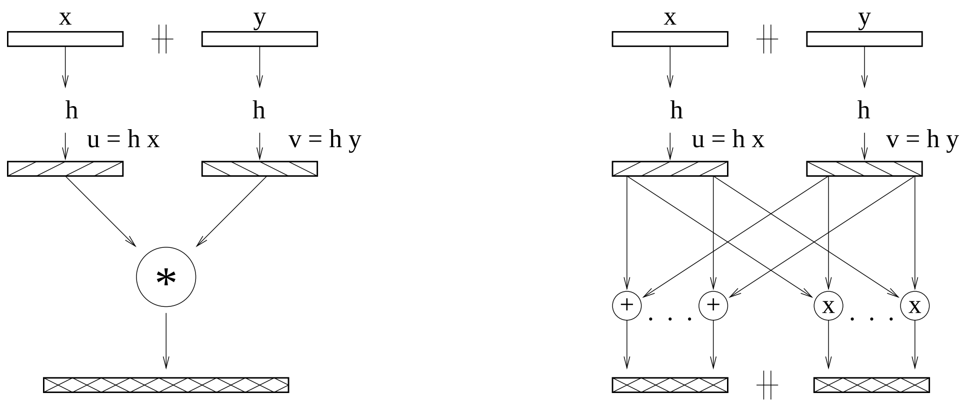

Abbildung 4.1: Allgemeine Homomorphismen ⇔ DH

In Abbildung 4.1 ist der Unterschied zwischen der allgemeinen Implementierung von Homomorphismen und der verteilten Implementierung von DHs dargestellt. Beide Varianten wenden den Homomorphismus h auf die Teillisten parallel an. Der Vorteil der verteilten Implementierung ist die verteilte *conquer*-Funktion. Auch hier muß bei einer konkreten Implementierung kommuniziert werden, denn beide Teillisten benötigen die Elemente der anderen Teilliste. Allerdings ist die Länge der zu kommunizierenden Blöcke fest und die Berechnung wird parallel ausgeführt.

4.2 Hauptsatz für DHs

Nachdem DHs eingeführt wurden, wird jetzt ein Theorem bewiesen, das eine ganze Familie von Implementierungen für DHs beschreibt. Darunter sind die von Gorlatch in [Gor96a] vorgeschlagenen Implementierungen für DHs sowie die in der Praxis benutzten Implementierungen zur Berechnung der Fast Fourier Transformation (siehe [Qui94]) enthalten.

Theorem 1. *Sei h ein DH, der auf Listen der Länge 2^l operiert, dann gilt für alle $d \in \{0, \ldots, l-1\}$:*

$$h = red^d(+\!\!+) \circ (\bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\, h) \circ dist_{(n_1,\ldots,n_d)} \tag{4.1}$$

Bevor dieses Theorem bewiesen wird, sollte man sich über dessen Bedeutung klar werden. Dazu betrachten wir einige Spezialfälle:

- $d = 0$: In diesem Fall steht im Hauptsatz einfach $h = h$. Dies ist der sequentielle Algorithmus zur Berechnung von h.

- $d = 1$: Dieser Fall ist in Abbildung 4.2 skizziert. Dieser Algorithmus mappt den DH auf Zeilen, transponiert danach die zweidimensionale Liste, mappt den DH nochmals und endet mit einer weiteren Transposition. Man beachte, daß dieser Algorithmus bei der Berechnung von FFT oft benutzt wird (siehe [Kum94]), wobei eine Dimension die Prozessordimension und die andere die lokale Dimension darstellt. In der Praxis wird oft das abschließende Transponieren weggelassen, weil beim Weiterarbeiten die Daten nicht immer zwingend in gleicher Reihenfolge wie vor der Berechnung vorliegen müssen.

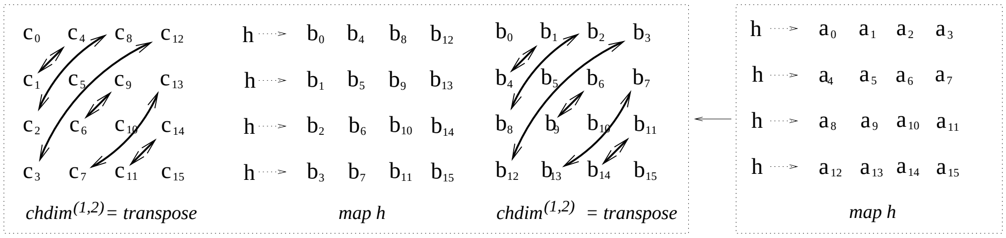

Abbildung 4.2: Der 2-dimensionale Fall

- $d = 2$: In der Praxis wird auch dieser Fall implementiert. Hier hat man zwei Prozessordimensionen und eine lokale Dimension. Die ersten beiden Schritte in diesem Spezialfall sind in Abbildung 4.3 gezeigt. Nach dem Ausführen eines lokalen *map* müssen die Dimensionen 2 und 3 vertauscht werden. In Abbildung 4.3 auf der rechten Seite ist skizziert, was beim $chdim^{(2,3)}$ passiert. Anschaulich werden alle Ebenen, die durch Koordinatenrichtung y und z aufgespannt sind, transponiert.

- $d = l - 1$: In diesem Fall sind die Daten in einem l-dimensionalen Hypercube angeordnet. Der Algorithmus besteht hier aus l Schritten,

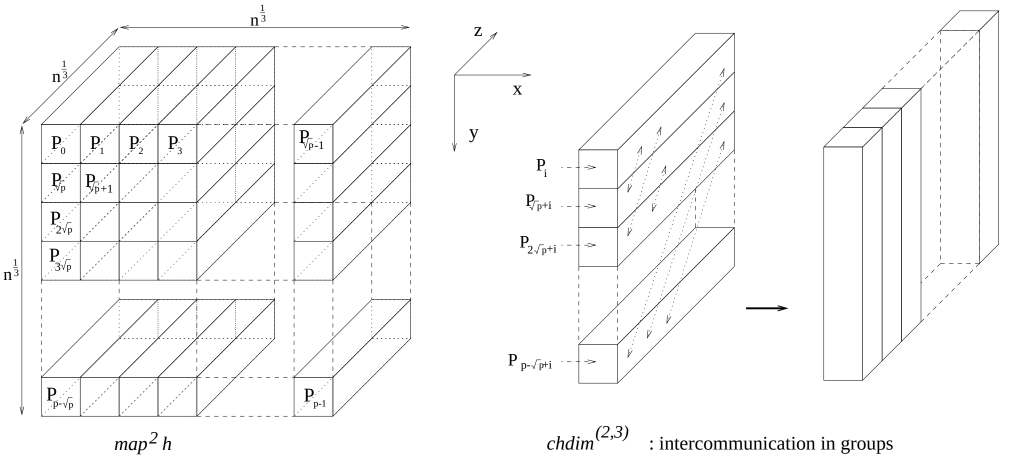

Abbildung 4.3: Der Fall $d = 2$

wobei im k-ten Schritt $mapdim^{(k,l)}\,(h)$ berechnet wird. Betrachten wir den k-ten Schritt näher:

$$
\begin{aligned}
& (chdim^{(k,l)} \circ map^{\,l-1}h \circ chdim^{(k,l)}\,x)_{(i_1,\dots,i_l)} \\
= & \ \{ \text{ Sei } i'_k = i_l, i'_l = i_k \text{ und } i'_j = i_j \text{ für alle anderen } j \ \} \\
& (map^{\,l-1}h \circ chdim^{(k,l)}\,x)_{(i'_1,\dots,i'_l)} \\
= & \ \{ \text{ Definition } map \text{ mehrmals anwenden } \} \\
& (h(chdim^{(k,l)}\,x)_{(i'_1,\dots,i'_{l-1})})_{(i'_l)} \\
= & \ \{ \text{ Sei } u = (chdim^{(k,l)}\,x)_{(i'_1,\dots,i'_{l-1},0)} \text{ und } v = (chdim^{(k,l)}\,x)_{(i'_1,\dots,i'_{l-1},1)} \ \} \\
& \begin{cases} u \oplus v & \text{falls } i_k = 0 \\ u \otimes v & \text{falls } i_k = 1 \end{cases}
\end{aligned}
$$

Im k-ten Schritt werden genau die Elemente mit $\oplus$ bzw. $\otimes$ verknüpft, die im Hypercube in Dimensionsrichtung k benachbart sind. Dieser Fall entspricht dem Hypercubealgorithmus(siehe [Gor96a]), wobei alle Dimensionen auf Prozessordimensionen abgebildet werden und wir genügend viele Prozessoren (nämlich genau so viele wie Elemente) haben. Dieser Fall ist in Abbildung 4.4 für $l = 2$ skizziert.

- $d = \log_2(p)$: Wenn p die Anzahl der Prozessoren ist und alle Dimensionen außer der innersten auf Prozessoren gemappt werden, dann entspricht dieser Fall genau dem Hypercubealgorithmus für eine feste Anzahl von Prozessoren [Gor96a]. Als Beispiel nehmen wir 8 Prozessoren und mehrere Elemente je Prozessor. Dann sieht der Algorithmus dem

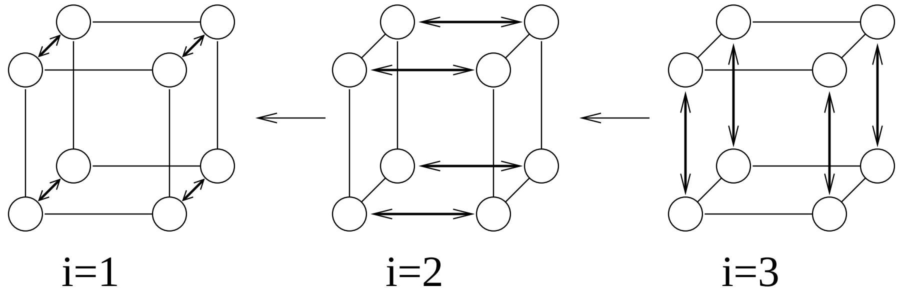

Abbildung 4.4: Hypercube-Fall mit einem Element je Prozessor

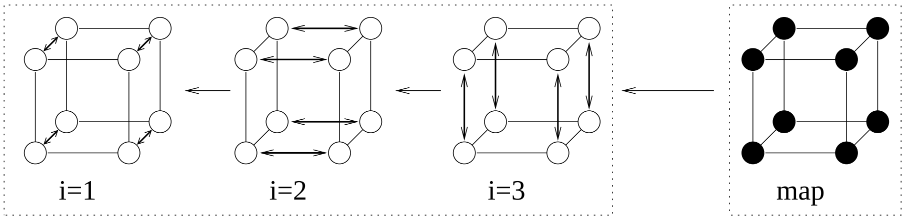

Abbildung 4.5: Hypercube-Fall mit mehreren Elementen je Prozessor

vorigen Fall $d = l - 1$ sehr ähnlich, hier wird vor den Berechnungen der $mapdim^{(i,d)}$ entlang der Prozessordimensionen ein *map* auf die lokale Dimension ausgeführt. Abbildung 4.5 verdeutlicht diesen Fall.

4.3 Beweis des Hauptsatzes

Um den Hauptsatz zu beweisen, werden wir erst vier kleinere Aussagen beweisen.

Lemma 8. *Sei* $\oplus\updownarrow\otimes$ *ein beliebiger DH und* $\odot$ *ein Operator, so daß*

$$(\oplus\updownarrow\otimes)(x +\!\!+\, y) = (\oplus\updownarrow\otimes\, x) \odot (\oplus\updownarrow\otimes\, y),$$

dann gilt für $x \in [\alpha]_{n_1 \times n_2}$*:*

$$(\oplus\updownarrow\otimes \circ \mathit{red}\,(+\!\!+))\; x = (\mathit{red}\,(\odot) \circ \mathit{map}\,(\oplus\updownarrow\otimes))\; x \qquad (4.2)$$

Beweis. Induktionsbasis: $(a \in [\alpha]_{n_2})$

$$\begin{aligned}(\oplus\updownarrow\otimes \circ \mathit{red}\,(+\!\!+))\;[a] &= (\oplus\updownarrow\otimes)\; a = \mathit{map}\,(\oplus\updownarrow\otimes))\;[a] \\ &= (\mathit{red}\,(\odot) \circ \mathit{map}\,(\oplus\updownarrow\otimes))\;[a]\end{aligned}$$

Induktionsschritt:

$$
\begin{aligned}
& (\mathit{red}\,(\odot) \circ \mathit{map}\,(\oplus\updownarrow\otimes))\ (x +\!\!+\, y) \\
=\ & \{\ \text{Definition } \mathit{map}\ \} \\
& \mathit{red}\,(\odot)((\mathit{map}\,(\oplus\updownarrow\otimes)\ x) +\!\!+\, \mathit{map}\,(\oplus\updownarrow\otimes)\ y)) \\
=\ & \{\ \text{Definition } \mathit{red}\ \} \\
& ((\mathit{red}\,(\odot) \circ \mathit{map}\,(\oplus\updownarrow\otimes))\ x) \odot ((\mathit{red}\,(\odot) \circ \mathit{map}\,(\oplus\updownarrow\otimes))\ y) \\
=\ & \{\ \text{Induktionsannahme}\ \} \\
& ((\oplus\updownarrow\otimes \circ \mathit{red}\,(+\!\!+))\ x) \odot ((\oplus\updownarrow\otimes \circ \mathit{red}\,(+\!\!+))\ y) \\
=\ & \{\ \text{Definition } \odot\ \} \\
& (\oplus\updownarrow\otimes)((\mathit{red}\,(+\!\!+)\ x) +\!\!+\, (\mathit{red}\,(+\!\!+)\ y)) \\
=\ & \{\ \text{Definition } \mathit{red}\ \} \\
& (\oplus\updownarrow\otimes \circ \mathit{red}\,(+\!\!+))\ (x +\!\!+\, y)
\end{aligned}
$$

□

Lemma 9. *Sei* $\oplus\updownarrow\otimes$ *wieder ein beliebiger DH und* $\odot$ *ein Operator wie im letzten Lemma. Dann gilt für* $x \in [\alpha]_{n_1 \times n_2}$*:*

$$
(\mathit{zip}\ \oplus)\updownarrow(\mathit{zip}\ \otimes) = \mathit{dist}_{n_1} \circ \mathit{red}\,(\odot) \tag{4.3}
$$

Beweis. Bewiesen wird wieder mithilfe von Induktion. Zuerst der Induktionsanfang: $(a \in [\alpha]_{n_2})$

$$
(\mathit{zip}\ \oplus)\updownarrow(\mathit{zip}\ \otimes)\ [a] = [a] = \mathit{dist}_1[a] = (\mathit{dist}_1 \circ \mathit{red}\,(\odot))[a]
$$

Induktionsschritt ($m = |x| = |y|$):

$$
\begin{aligned}
& (dist_{2m} \circ red\,(\odot))(x \mathbin{+\!\!+} y) \\
= & \ \{ \text{ Definition } red \ \} \\
& dist_{2m}((red\,(\odot)\ x) \odot (red\,(\odot)\ y)) \\
= & \ \{ \text{ Definition } \odot \ \} \\
& dist_{2m}(zip\,(\oplus)(red\,(\odot)\ x, red\,(\odot)\ y) \mathbin{+\!\!+} zip\,(\otimes)(red\,(\odot)\ x, red\,(\odot)\ y)) \\
= & \ \{ \text{ Definition } dist_{2m} \ \} \\
& dist_m(zip\,(\oplus)(red\,(\odot)\ x, red\,(\odot)\ y)) \mathbin{+\!\!+} dist_m(zip\,(\otimes)(red\,(\odot)\ x, red\,(\odot)\ y)) \\
= & \ \{ \ (2.1) \ \} \\
& zip^2(\oplus)(dist_m \circ red\,(\oplus)\ x, dist_m \circ red\,(\oplus)\ y) \mathbin{+\!\!+} \\
& \qquad zip^2(\otimes)(dist_m \circ red\,(\oplus)\ x, dist_m \circ red\,(\oplus)\ y) \\
= & \ \{ \text{ Induktionsannahme } \} \\
& (zip^2(\oplus)((zip\ \oplus) \updownarrow (zip\ \otimes)\ x, (zip\ \oplus) \updownarrow (zip\ \otimes)\ y)) \mathbin{+\!\!+} \\
& \qquad (zip^2(\otimes)((zip\ \oplus) \updownarrow (zip\ \otimes)\ x, (zip\ \oplus) \updownarrow (zip\ \otimes)\ y)) \\
= & \ \{ \text{ Definition DH } \} \\
& (zip\ \oplus) \updownarrow (zip\ \otimes)(x \mathbin{+\!\!+} y)
\end{aligned}
$$

□

Lemma 10. *Sei $h = \oplus \updownarrow \otimes$, dann gilt für alle $x \in [\alpha]_{n\times m}$:*

$$(zip\ \oplus) \updownarrow (zip\ \otimes)\ x = (tr \circ map\ \ h \circ tr)\ x \tag{4.4}$$

Beweis. Beweis mit Induktion über die Länge der Liste x.
Induktionsanfang: (eine einelementige Liste hat die Form $[[a_1, \ldots, a_m]]$)

$$
\begin{aligned}
(zip\ \oplus) \updownarrow (zip\ \otimes)[[a_1, \ldots, a_m]] &= [[a_1, \ldots, a_m]] = map\ \ h\ [[a_1], \ldots, [a_m]] \\
&= (tr \circ map\ \ h)\ [[a_1], \ldots, [a_m]] = (tr \circ map\ \ h \circ tr)\ [[a_1, \ldots, a_m]]
\end{aligned}
$$

Der Induktionsschritt wird bewiesen, indem die Gleichheit für alle Elemente

(i,j) gezeigt wird: (g=$(zip\ \oplus) \updownarrow (zip\ \otimes)$)

$$
\begin{aligned}
& g((x +\!\!\!+ y))_{(i,j)} \\
= & \ \{ \text{ Definition DH } \} \\
& ((zip^2(\oplus)(g\ x, g\ y)) +\!\!\!+ (zip^2(\otimes)(g\ x, g\ y)))_{(i,j)} \\
= & \ \{ \text{ Induktionsannahme } \} \\
& (zip^2(\oplus)((tr \circ map\ \ h \circ tr)\ x, (tr \circ map\ \ h \circ tr)\ y) +\!\!\!+ \\
& \quad zip^2(\otimes)((tr \circ map\ \ h \circ tr)\ x, (tr \circ map\ \ h \circ tr)\ y))_{(i,j)} \\
= & \ \{ \text{ Definition } +\!\!\!+,\ l = |(tr \circ map\ \ h \circ tr)\ x|\ \} \\
& \begin{cases} (zip^2(\oplus)((tr \circ map\ \ h \circ tr)\ x, (tr \circ map\ \ h \circ tr)\ y))_{(i,j)} & \text{falls } i \leq l \\ (zip^2(\otimes)((tr \circ map\ \ h \circ tr)\ x, (tr \circ map\ \ h \circ tr)\ y)_{(i-l,j)} & \text{sonst} \end{cases} \\
= & \ \{ \text{ Def. von } zip, tr,\ l = |((map\ \ h \circ tr)\ x)_1|\ \} \\
& \begin{cases} ((map\ \ h \circ tr)\ x)_{(j,i)} \oplus ((map\ \ h \circ tr)\ y)_{(j,i)} & \text{falls } i \leq l \\ ((map\ \ h \circ tr)\ x)_{(j,i-l)} \otimes ((map\ \ h \circ tr)\ y)_{(j,i-l)} & \text{sonst} \end{cases} \\
= & \ \{ \text{ Definition } map,\ l = |h(tr\ \ x)_1|\ \} \\
& \begin{cases} (h(tr\ \ x)_j)_i \oplus (h(tr\ \ y)_j)_i & \text{falls } i \leq l \\ (h(tr\ \ x)_j)_{i-l} \otimes (h(tr\ \ y)_j)_{i-l} & \text{sonst} \end{cases} \\
= & \ \{ \text{ Definition DH } \} \\
& (h((tr\ \ x)_j +\!\!\!+ (tr\ \ y)_j))_i \\
= & \ \{ \text{ Definition } zip\ \} \\
& (h(zip(+\!\!\!+)(tr\ \ x, tr\ \ y))_j)_i \\
= & \ \{ \text{ Lemma 5 } \} \\
& (h(tr(x +\!\!\!+ y))_j)_i \\
= & \ \{ \text{ Definition } map\ \} \\
& (map\ \ h \circ tr(x +\!\!\!+ y))_{(j,i)} \\
= & \ \{ \text{ Definition } tr\ \} \\
& (tr \circ map\ \ h \circ tr(x +\!\!\!+ y))_{(i,j)}
\end{aligned}
$$

□

Lemma 11. *Sei $a \leq (d-1), d \geq 2$, dann gilt für jeden DH h:*

$$map^{d-1}(red(+\!\!\!+)) \circ mapdim^{(a,d+1)}\ h = mapdim^{(a,d)}\ h \circ map^{d-1}(red(+\!\!\!+)) \tag{4.5}$$

Beweis.

$$
\begin{aligned}
& map^{\,d-1}(red\,(+\!\!\!+)) \circ mapdim^{(a,d+1)}\,h \\
= & \ \{\ mapdim^{(a,d+1)}\,h = chdim^{(a,d-1)} \circ mapdim^{(d-1,d+1)}\,h \circ chdim^{(a,d-1)}\ \} \\
& map^{\,d-1}(red\,(+\!\!\!+)) \circ chdim^{(a,d-1)} \circ mapdim^{(d-1,d+1)}\,h \circ chdim^{(a,d-1)} \\
= & \ \{\ map^{\,d-1}(red\,(+\!\!\!+)) \circ chdim^{(a,d-1)} = chdim^{(a,d-1)} \circ map^{\,d-1}(red\,(+\!\!\!+))\ \} \\
& chdim^{(a,d-1)} \circ map^{\,d-1}(red\,(+\!\!\!+)) \circ mapdim^{(d-1,d+1)}\,h \circ chdim^{(a,d-1)} \\
= & \ \{\ (3.2),(2.2)\ \} \\
& chdim^{(a,d-1)} \circ map^{\,d-2}(map\,red\,(+\!\!\!+) \circ mapdim^{(1,3)})h \circ chdim^{(a,d-1)} \\
= & \ \{\ map\,red\,(+\!\!\!+) \circ mapdim^{(1,3)})h = mapdim12)h \circ map\,red\,(+\!\!\!+)\ \} \\
& chdim^{(a,d-1)} \circ map^{\,d-2}(mapdim^{(1,2)})h \circ map\,red\,(+\!\!\!+)) \circ chdim^{(a,d-1)} \\
= & \ \{\ (3.2)(2.2)\ \} \\
& chdim^{(a,d-1)} \circ mapdim^{(d-1,d)}\,h \circ map^{\,d-1}(red\,(+\!\!\!+)) \circ chdim^{(a,d-1)} \\
= & \ \{\ map^{\,d-1}(red\,(+\!\!\!+)) \circ chdim^{(a,d-1)} = chdim^{(a,d-1)} \circ map^{\,d-1}(red\,(+\!\!\!+))\ \} \\
& chdim^{(a,d-1)} \circ mapdim^{(d-1,d)}\,h \circ chdim^{(a,d-1)} \circ map^{\,d-1}(red\,(+\!\!\!+)) \\
= & \ \{\ \text{siehe Erklärung zur ersten Gleichheit}\ \} \\
& mapdim^{(a,d)}\,h \circ map^{\,d-1}(red\,(+\!\!\!+))
\end{aligned}
$$

□

Beweis des Hauptsatzes

Auch hier wird wieder Induktion verwendet, und zwar über d. Der DH bestehe aus den beiden Operatoren $\oplus, \otimes$, es ist also $h = \oplus\updownarrow\otimes$. Der Fall $d = 0$ ist offensichtlich:

$$h = red^{0}(+\!\!\!+) \circ (mapdim^{(1,1)}\ h) = h$$

Basisfall $d = 1$. Diesen Fall brauchen wir, um den Induktionsschritt auf die Induktionsannahme zu reduzieren:

$$
\begin{aligned}
& \mathit{red}\,(+\!\!+) \circ (\mathit{mapdim}^{(1,2)}\ h) \circ (\mathit{mapdim}^{(2,2)}\ h) \circ \mathit{dist}_{n_1} \\
= & \ \{\ \text{Definition } \mathit{mapdim}^{(2,2)}\ \} \\
& \mathit{red}\,(+\!\!+) \circ (\mathit{mapdim}^{(1,2)}\ h) \circ (\mathit{map}\ h) \circ \mathit{dist}_{n_1} \\
= & \ \{\ (4.4)\ \} \\
& \mathit{red}\,(+\!\!+) \circ (\mathit{zip}\ \oplus) \updownarrow (\mathit{zip}\ \otimes) \circ (\mathit{map}\ h) \circ \mathit{dist}_{n_1} \\
= & \ \{\ (4.3),\ \text{wobei } \odot \text{ die Operation des DHs}\ \} \\
& \mathit{red}\,(+\!\!+) \circ \mathit{dist}_{n_1} \circ \mathit{red}\,(\odot) \circ (\mathit{map}\ h) \circ \mathit{dist}_{n_1} \\
= & \ \{\ (4.2)\ \} \\
& \mathit{red}\,(+\!\!+) \circ \mathit{dist}_{n_1} \circ \oplus \updownarrow \otimes \circ \mathit{red}\,(+\!\!+) \circ \mathit{dist}_{n_1} \\
= & \ \{\ \mathit{red}\,(+\!\!+) \circ \mathit{dist}_{n_1} = \mathit{id}\ \} \\
& h
\end{aligned}
$$

Mithilfe des zweiten Basisfalls ($d = 1$) kann nun das Theorem für die restlichen $d > 1$ bewiesen werden:

$$
\begin{aligned}
& red^{d}(+\!\!+) \circ \bigodot_{i=1}^{d+1} (mapdim^{(i,d+1)}\ h) \circ dist_{(n_1,\ldots,n_{d-1})} \\
= & \ \{ \text{ Definition } red^{d} \ \} \\
& red^{d-1}(+\!\!+) \circ map^{d-1}(red\,(+\!\!+)) \circ \bigodot_{i=1}^{d+1} (mapdim^{(i,d+1)}\ h) \circ dist_{(n_1,\ldots,n_{d-1})} \\
= & \ \{ \text{ (4.5) mehrmals, } A = red^{d-1}(+\!\!+) \circ \bigodot_{i=1}^{d-1} (mapdim^{(i,d)}\ h) \ \} \\
& A \circ (map^{d-1}(red\,(+\!\!+))) \circ mapdim^{(d,d+1)}\, h \circ mapdim^{(d+1,d+1)}\, h \circ dist_{(n_1,\ldots,n_{d-1})} \\
= & \ \{ \text{ (2.2) und (3.2) mehrmals } \} \\
& A \circ (map^{d-1}(red\,(+\!\!+))) \circ map^{d-1}(mapdim^{(1,2)}\, h \circ mapdim^{(2,2)}\, h) \circ dist_{(n_1,\ldots,n_{d-1})} \\
= & \ \{ \text{ (2.2) und Definition von } dist \ \} \\
& A \circ map^{d-1}(red\,(+\!\!+) \circ mapdim^{(1,2)}\, h \circ mapdim^{(2,2)}\, h \circ dist_{n_{d-1}}) \circ dist_{(n_1,\ldots,n_{d-2})} \\
= & \ \{ \text{ Basisfall } d = 1 \ \} \\
& A \circ map^{d-1}\ h \circ dist_{(n_1,\ldots,n_{d-2})} \\
= & \ \{ \text{ Definition } mapdim^{(d,d)} \ \} \\
& A \circ mapdim^{(d,d)}\ h \circ dist_{(n_1,\ldots,n_{d-2})} \\
= & \ \{ \text{ Notation } \bigcirc \ \} \\
& red^{d-1}(+\!\!+) \circ \bigodot_{i=1}^{d} (mapdim^{(i,d)}\, h) \circ dist_{(n_1,\ldots,n_{d-2})} \\
= & \ \{ \text{ Induktionsannahme } \} \\
& h
\end{aligned}
$$

Kapitel 5

Implementierung und Optimierung

Das Ziel in diesem Kapitel ist es, das in den letzten Kapiteln erarbeitete Implementierungsschema für verteilte Homomorphismen in einer konkreten Programmiersprache zu implementieren.

5.1 MPI

Die Wahl der Programmiersprache fiel auf C in Verbindung mit MPI (Message Passing Interface). Diese Kombination hat folgende Vorteile:

- MPI ist anerkannter Standard [For95].
- Da MPI noch weiterentwickelt wird, ist davon auszugehen, daß in Zukunft dieser Standard noch populärer wird. (MPI-2.0 wurde gerade erst vorgestellt und muß noch implementiert werden).
- MPI bietet kollektive Kommunikation. Da wir uns auf diese beschränken, und keine expliziten Sends oder Receives benutzen, bleiben die Programme übersichtlich.
- MPI bietet Topologien an, wie z.B. kartesische Koordinatensysteme.
- MPI ist auf vielen Maschinen verfügbar, vor allem auch auf der Parsytec GCel 64, die uns zur Verfügung stand. Als Betriebssystem wird Parix 1.2 [Par93] verwendet.
- C ist praktisch auf jeder Maschine vorhanden.

- C/MPI sind zwar keine sehr hohen Programmiersprachen, dafür bleibt eine Performanceeinbuße wegen der MPI-Implementierung überschaubar.

Um den MPI-Standard zu definieren und zu modifizieren wurde ein internationales Forum gebildet, das MPI-Forum (der MPI-1.1 Standard ist in [For95] nachzulesen). Gute Bücher, um in MPI einzusteigen sind [GLS94] und [Pac97].

Als Datenstruktur benutzen wir Arrays. Wir werden die innersten Dimensionen im Prozessor als eindimensionales Array und alle anderen Dimensionen auf Prozessordimensionen mappen. Beispiel: Angenommen wir haben dreidimensionale Listen ($m \times n \times o$) zu implementieren, dann gibt es folgende Möglichkeiten, diese abzuspeichern:

- Auf einem Prozessor. In diesem Prozessor speichern wir die zweidimensionale Liste in einem Array ab, und zwar so, daß das Element (i, j, k) im Array mit Index $[i * n * o + j * o + k]$ vorliegt.

- Auf n Prozessoren, die als ein kartesisches Koordinatensystem der Dimension 1 aufgebaut sind. Das Element (i, j, k) ist dann im Prozessor i im Arrayindex $[j * o + k]$ abgespeichert.

- Werden die Prozessoren auf ein zweidimensionales kartesisches Koordinatensystem verteilt, dann ist das Element (i, j, k) im Prozessor mit den Koordinaten (i, j) abgespeichert im Array an der k-ten Stelle.

Im folgenden werden die innersten, lokal vorliegenden Dimensionen "lokale Dimensionen" und die anderen "Prozessordimensionen" genannt. Erinnern wir uns an den Hauptsatz für DHs (Theorem 1). Die Funktionen *dist* und das $red\,(+\!\!+)$ brauchen wir nicht zu implementieren, da wir von verteilten Daten ausgehen und diese auch verteilt lassen. Fehlen also nur noch die Funktionen $map^d\ h$ und $chdim^{(i,d)}$.

Beim *map* wird DH h auf alle Listen der innersten Dimension angewendet, und jede solche Liste ist lokal in einem Prozessor in Form eines Arrays abgespeichert. In C wird die Berechnung von h als Aufruf der Funktion h implementiert, wobei die Adresse des ersten Elements und die Länge der Dimension übergeben wird. Sind mehr als eine Dimension lokal, dann wird es eine Schleife von solchen Aufrufen sein.

Es bleibt noch die Implementierung von $chdim^{(i,d)}$. Sind i und d lokal, dann wird $chdim^{(i,d)}$ durch entsprechendes lokales Umkopieren der Arrayelemente in den Prozessoren implementiert. Problematischer wird die Implementierung, wenn i eine Prozessordimension ist. Um zu verstehen, was zu tun ist, schauen wir uns zuerst an, wie $chdim^{(1,2)}$ auf einem eindimensionalen Prozessorfeld aussieht:

Prozessor 1 : $a_{11}\ a_{12}\ \ldots\ a_{1n}$
Prozessor 2 : $a_{21}\ a_{22}\ \ldots\ a_{2n}$
...
Prozessor n : $a_{n1}\ a_{n2}\ \ldots\ a_{nn}$

Nach der Anwendung von $chdim^{(1,2)}$ sieht die Verteilung der Daten folgendermaßen aus:

Prozessor 1 : $a_{11}\ a_{21}\ \ldots\ a_{n1}$
Prozessor 2 : $a_{12}\ a_{22}\ \ldots\ a_{n2}$
...
Prozessor n : $a_{1n}\ a_{2n}\ \ldots\ a_{nn}$

Diese Umverteilung der Daten entspricht dem Aufruf der MPI-Funktion `MPI_Alltoall` (siehe MPI-Standard [For95]).

Mithilfe des Konzepts der Kommunikatoren in MPI kann man Prozessoren, die in einem kartesischen Koordinatensystem angeordnet sind so gruppieren, daß alle Prozessoren, die sich nur in einer Prozessorkoordinate unterscheiden, zur gleichen Gruppe gehören. Diese Menge von Gruppen betrachten wir als neuen Kommunikator. Hat man nun d Prozessordimensionen, erstellt man d solcher Kommunikatoren, einen für jede Dimension.

Wenn wir unsere Dimensionen alle bis auf die innerste auf Prozessordimensionen aufteilen und alle Dimensionen gleich groß sind, dann entspricht $chdim^{(i,d)}$ genau einem `MPI_Alltoall(comm[i])`, wobei `comm[i]` der Kommunikator für die i-te Dimension ist.

Es bleibt noch die Einschränkung, daß alle Dimensionen die gleiche Größe haben müssen. Da wir mit Powerlisten arbeiten, haben alle Dimensionen die Größe einer Zweierpotenz. Weiterhin gehen wir davon aus, daß lokal mehr Elemente gespeichert sind als in der größten Dimension. Diese Einschränkung wird auch in der Praxis vorgenommen und verbessert die Performance.

Betrachten wir den i-ten Schritt des Hauptsatzes:

$$
\begin{aligned}
& mapdim^{(i,d+1)} \\
= \{\ & map^{d}(red(+\!\!+)) \circ map^{d}(dist_{n_i}) = id\ \} \\
& mapdim^{(i,d+1)} \circ map^{d}(red(+\!\!+)) \circ map^{d}(dist_{n_i}) \\
= \{\ & (4.5)\ \} \\
& map^{d}(red(+\!\!+)) \circ mapdim^{(i,d+2)}\, h \circ map^{d}(dist_{n_i})
\end{aligned}
$$

Der i-te Schritt kann in drei Stufen ausgeführt werden, wobei die erste und dritte Stufe – $map^d(red(+\!\!+))$ und $map^d(dist_{n_i})$ – nicht implementiert werden müssen, denn der lokale Array bleibt wie oben beschrieben immer eindimensional. Für die zweite Stufe ergibt sich:

$$mapdim^{(i,d+2)}\, h \;=\; chdim^{(i,d+1)} \circ mapdim^{(d+1,d+2)}\, h \circ chdim^{(i,d+1)}$$

$mapdim^{(d+1,d+2)}\, h$ eine rein lokale Funktion: lokales Transponieren, Schleife von Aufrufen von h und nochmaliges lokales Transponieren (siehe hierzu die Definition von $mapdim^{(d+1,d+2)}\, h$ in Kapitel 3). Bleibt noch die Funktion $chdim^{(i,d+1)}$. Durch das weitere Aufsplitten der innersten Liste wurde erreicht, daß die i-te und $(d+1)$-te Dimension gleiche Länge haben, folglich kann die MPI-Funktion `MPI_Alltoall` benutzt werden. Damit ergibt sich ein SPMD-Programm, das in MPI folgendermaßen implementiert wird:

```
Kommunikatoren erzeugen
map(h)
FOR i = d DOWNTO 1
    MPI_Alltotall(comm[i])
    local_transpose
    map(h)
    local_transpose
    MPI_Alltoall(comm[i])
```

`map(h)` ist der Aufruf der sequentiellen Implementierung des DHs auf die entsprechenden lokalen Teillisten. Nach dem Erzeugen der Kommunikatoren besteht diese Implementierung aus $d+1$ Berechnungsschritten. Im ersten Schritt wird der DH auf die lokalen Daten angewendet. Die restlichen d Iterationen der Schleife bestehen jeweils aus drei Stufen:

- Umverteilen der Daten.
- Anwenden des DHs auf die lokalen Daten.
- Erneutes Umverteilen der Daten.

Durch diese Schritte und Stufen bleibt die Implementierung übersichtlich. Weiterhin wird in der Berechnung nur eine Funktion `MPI_Alltoall` zur Kommunikation benutzt und für die Implementierung des DHs kann die schnellste sequentielle Implementierung gewählt werden.

5.2 Reorganisation von Kommunikation

Die Funktion $mapdim^{(i,d)}$ besteht aus Kommunikation, gefolgt von lokalen Berechnungen und wiederum Kommunikation. Wenn man den Hauptsatz (Theorem 1) betrachtet, fällt auf, daß aufgrund aufeinanderfolgender $mapdim^{(i,d)}$ immer zwei Kommunikationsschritte hintereinander ausgeführt werden, $chdim^{(i,d)}$ und $chdim^{(i+1,d)}$. Das Austauschen von Dimensionen nach der lokalen Berechnung beeinflußt die lokalen Berechnungen des nächsten Schritts nicht. Man kann also die Hälfte aller Kommunikationen an das Ende des Ausdrucks schieben. Dies zeigt das folgende Theorem:

Theorem 2. *Sei h ein DH, dann gilt für alle $1 \leq k \leq d$:*

$$\bigcirc_{i=k}^{d+1} (mapdim^{(i,d+1)}\ h) = \bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k}^{d+1} (map^{d}\ h \circ chdim^{(i,d+1)})$$

Beweis. Beweis durch Induktion über k.
Basisfall $k = d$:

$$\begin{aligned}
& chdim^{(d,d+1)} \circ map^{d}\, h \circ chdim^{(d,d+1)} \circ chdim^{(d+1,d+1)} \circ map^{d}\, h \circ chdim^{(d+1,d+1)} \\
= & \ \{\ \textit{Definition } chdim^{(d+1,d+1)}\ \} \\
& chdim^{(d,d+1)} \circ map^{d}\ h \circ chdim^{(d,d+1)} \circ map^{d}\ h \\
= & \ \{\ \textit{nochmals Definition von } chdim^{(d+1,d+1)}\ \} \\
& chdim^{(d,d+1)} \circ map^{d}\ h \circ chdim^{(d,d+1)} \circ map^{d}\ h \circ chdim^{(d+1,d+1)}
\end{aligned}$$

Induktionsschritt.
Annahme: Die Aussage gilt für ein $k \geq 2$. Dann gilt sie auch für $k - 1$.

$$\bigcirc_{i=k-1}^{d+1} (mapdim^{(i,d+1)}\ h)$$

$= \{$ *Definition* $\bigcirc$ *und* $mapdim^{(k-1,d+1)}$ $\}$

$$chdim^{(k-1,d+1,\circ)}\ map^{\,d}\ h \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k-1}^{d+1} (mapdim^{(i,d+1)}\ h)$$

$= \{$ *Induktionsannahme* $\}$

$$chdim^{(k-1,d+1,\circ)}\ map^{\,d}\ h \circ chdim^{(k-1,d+1)} \circ$$
$$\bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$$

$= \{$ *Sei* $A = \bigcirc_{i=k}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$ $\}$

$$chdim^{(k-1,d+1)} \circ map^{\,d}\ h \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(d,d+1)} \circ A$$

$= \{$ $\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)}$ *nach links durch unabhängige Dimensionen schieben* $\}$

$$\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(k-1,d+1)} \circ map^{\,d}\ h \circ chdim^{(k-1,d+1)} \circ chdim^{(d,d+1)} \circ A$$

$= \{$ $chdim^{(k-1,d+1)} \circ chdim^{(d,d+1)} = chdim^{(k-1,d)} \circ chdim^{(k-1,d+1)}$ $\}$

$$\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(k-1,d+1)} \circ map^{\,d}\ h \circ chdim^{(k-1,d)} \circ chdim^{(k-1,d+1)} \circ A$$

$= \{$ $map^{\,d}\ h$ *nach rechts schieben* $\}$

$$\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} \circ map^{\,d}\ h \circ chdim^{(k-1,d+1)} \circ A$$

$= \{$ $map^{\,d}\ h \circ chdim^{(k-1,d+1)}$ *in* $\bigcirc$ *aufnehmen* $\}$

$$\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} \circ \bigcirc_{i=k-1}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$$

$= \{$ $chdim^{(k-1,d+1)} \circ chdim^{(k-1,d)} = chdim^{(d,d+1)} \circ chdim^{(k-1,d+1)}$ $\}$

$$\bigcirc_{i=k}^{d-1} chdim^{(i,i+1)} \circ chdim^{(d,d+1)} \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k-1}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$$

$= \{$ $chdim^{(d,d+1)}$ *in* $\bigcirc$ *aufnehmen* $\}$

$$\bigcirc_{i=k}^{d} chdim^{(i,i+1)} \circ chdim^{(k-1,d+1)} \circ \bigcirc_{i=k-1}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$$

$= \{$ $\forall s = k, \ldots, d : chdim^{(s,s+1)} \circ chdim^{(k-1,s+1)} = chdim^{(k-1,s)} \circ chdim^{(s,s+1)}$ $\}$

$$\bigcirc_{i=k-1}^{d} chdim^{(i,i+1)} \circ \bigcirc_{i=k-1}^{d+1} (map^{\,d}\ h \circ chdim^{(i,d+1)})$$

Die in Theorem 2 an das Ende geschobenen Transpositionen fallen aus der Berechnung heraus und können wie auch in MPI mit einem Kommunikationsschritt implementiert werden. Falls die Verteilung der Daten nach der Berechnung nicht gleich der Verteilung vor der Berechnung sein muß, kann auf diesen Kommunikationsschritt sogar ganz verzichtet werden. Man spart sich folglich ca. 50% der Kommunikation.

5.3 Beinahe-DHs

Es gibt Funktionen, die keine DHs sind, sich aber trotzdem verteilt implementieren lassen. Diese tupeln die Eingabeliste, wenden dann einen DH auf diese Tupel an und projizieren wieder auf die erste Komponente: Für das Tupeln wird eine neue Funktion definiert:

$$
\begin{aligned}
&\mathit{addcomp} :: \mathit{nat} \to [\beta]_{(n_1\times\cdots\times n_d)} \to [\alpha]_{(n_1\times\cdots\times n_d)} \to [(\alpha,\beta)]_{(n_1\times\cdots\times n_d)} \\
&(\mathit{addcomp}^{(d)}\, c)\, x = \mathit{zip}^{\,d}(\mathit{pair})(x, c) \qquad (5.1)\\
\text{wobei } &\mathit{pair}\; a\, b = (a, b) \qquad \text{für } a \in \alpha, b \in \beta
\end{aligned}
$$

Jetzt können Beinahe-DHs formal definiert werden:

Definition 11. *Sei h' ein DH, $c \in [\beta]_{(n_1\times\cdots\times n_d)}$, dann ist h ein Beinahe-DH, falls gilt:*

$$h = (\mathit{map}\ \pi_1) \circ h' \circ (\mathit{addcomp}^{(1)}\, c)$$

Ein Beispiel für einen Beinahe-DH ist die *scan*-Funktion, die in [Gor96a] oder auch [Gor96b] näher untersucht wird. Ein etwas abstrakteres Beispiel ist ein DH, der in seinen Berechnungen die Rekursionstiefe benötigt:

$$
\begin{aligned}
&(\mathit{map}\ \pi_1) \circ \oplus\!\updownarrow\!\otimes \circ (\mathit{addcomp}^{(1)}\, c)\\
\text{wobei}&\\
&c_{(i)} = \log_2(|c|) \quad \text{für alle } i\\
&\begin{pmatrix} a \\ b \end{pmatrix} \oplus \begin{pmatrix} a' \\ b \end{pmatrix} = \begin{pmatrix} \oplus'(a, a', b) \\ b-1 \end{pmatrix} \quad \begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} a' \\ b \end{pmatrix} = \begin{pmatrix} \otimes'(a, a', b) \\ b-1 \end{pmatrix}\\
&\oplus', \otimes' \text{ beliebige Operatoren}
\end{aligned}
$$

Kapitel 6

Fallstudie: FFT

In den vorangegangenen Kapiteln wurden Implementierungen für DHs hergeleitet. Um die Ergebnisse für praktische Probleme zu benutzen, müssen die Probleme systematisch an das DH-Format angepaßt werden.

Als Fallstudie wird die in vielen Bereichen der Informatik eingesetzte Fast Fourier Transformation (FFT) betrachtet. Für diese Funktion wurden mehrere Algorithmen für unterschiedliche parallele Plattformen entwickelt. Die bekanntesten Implementierungen sind der 2D, der 3D und der Hypercube Algorithmus, wie sie in [Kum94] beschrieben sind. Diese sehen auf den ersten Blick aus wie die Fälle $d = 1$, $d = 2$ und $d = \log p$ des Hauptsatzes (Theorem 1). Die Vermutung liegt also nahe, daß es sich bei der FFT um einen DH handelt. Das Ziel ist ein paralleles Programm für die FFT durch eine systematische Anpassung an das DH-Format aus der mathematische Spezifikation der FFT abzuleiten.

6.1 Spezifikation

Angenommen wir haben eine Liste $x = [x_0, \ldots, x_{n-1}]$ der Länge 2^l. Die FFT erzeugt aus x eine neue Liste gleicher Länge, dessen i-tes Element folgendermaßen definiert ist:

$$(FFT\ x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

wobei ω_n die n-te primitive Einheitswurzel der komplexen Ebene ist, also

$$\omega_n = e^{2\pi\sqrt{-1}/n}, \text{ und } i = 0, \ldots, n-1$$

Diese Spezifikation wird üblicherweise in eine Divide-and-Conquer-Form umgeschrieben (siehe z.B. [Qui94]).

$$\begin{aligned}
(FFT\ x)_i &= \sum_{k=0}^{n-1} x_k \omega_n^{ki} \\
&= \sum_{k=0}^{n/2-1} x_{2k}\omega_{n/2}^{ki} + \omega_n^i \sum_{k=0}^{n/2-1} x_{2k+1}\omega_{n/2}^{ki} \\
&= (FFT\ u)_{i \bmod (n/2)} + \omega_n^i (FFT\ v)_{i \bmod (n/2)} \\
&= \begin{cases} (FFT\ u)_i + \omega_n^i (FFT\ v)_i & \text{falls } i < n/2 \\ (FFT\ u)_{i-n/2} + \omega_n^{i-n/2}(FFT\ v)_{i-n/2} & \text{sonst} \end{cases} \\
&\text{wobei } u = [x_0, x_2, \ldots, x_{n-2}], v = [x_1, x_3, \ldots, x_{n-1}]
\end{aligned}$$

Diese Divide-and-Conquer-Form der FFT stellen wir nun in BMF dar. Dazu betrachten wir die Basis-, Divide- und Conquer-Funktion:

$$\begin{aligned}
&\mathit{base} = \mathit{id} \\
&\mathit{divide}\,[x_0, \ldots x_{n-1}] = [[x_0, x_2, \ldots, x_{n-2}], [x_1, x_3, \ldots, x_{n-1}]] \\
&\mathit{conquer}\,[u, v] = \mathit{zip}\,(+)(u, \mathit{zip}\,(\cdot)(\Omega, v)) +\!\!+\ \mathit{zip}\,(-)(u, \mathit{zip}\,(\cdot)(\Omega, v)) \qquad (6.1) \\
&\Omega = [\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2-1}] \\
&\omega_n : n\text{-te primitive Einheitswurzel}
\end{aligned}$$

6.2 Anpassung

Um die Implementierung aus Kapitel 5 benutzen zu können, muß die FFT an das DH-Format angepaßt werden. Dies wird in mehreren Schritten geschehen.

Schritt 1

Als erstes muß (6.1) umgeformt werden, sodaß man folgende Divide-Funktion bekommt.

$$\mathit{divide}[x_0, \ldots, x_{n-1}] = [[x_0, \ldots, x_{n/2-1}], [x_{n/2}, \ldots, x_{n-1}]]$$

Dazu permutiert man erst die Ausgangsliste, und zwar in der Weise, daß man die Elemente x_i und x_j vertauscht, wobei j die umgekehrte Bitdarstellung von i hat [AS95]. Zum besseren Verständnis gleich ein Beispiel: Angenommen wir haben eine Ausgangsliste mit 8 Elementen $[x_0, \ldots, x_7]$, dann müssen folgende Paare getauscht werden:

- Binärdarstellung:

$$(000, 000), (001, 100), (010, 010), (011, 110), (101, 101), (111, 111)$$

- Dezimaldarstellung:

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

Die FFT berechnet sich nun folgendermaßen:

$$FFT = FFT' \circ permute$$

FFT' ist jetzt eine Divide-and-Conquer-Funktion, dessen *divide*-Funktion den Forderungen des DHs entspricht.

$$\begin{aligned}
&divide\,[x_0, \ldots x_{n-1}] = [[x_o, \ldots , x_{n/2-1}], [x_{n/2}, \ldots , x_{n-1}]] \\
&\Omega, \mathit{base}, \mathit{conquer} : \text{wie oben}
\end{aligned}$$

Schritt 2

Als nächstes soll die Funktion FFT' so umgeformt werden, daß sie dem DH-Format entspricht (Definition 10). Dazu betrachtet man die einzelnen Elemente der Liste nach der Anwendung von FFT':
$u = FFT'\ x, \quad v = FFT'\ y, \quad m = |u| = |x|$

$$\begin{aligned}
&(FFT'(x +\!\!+\, y))_i \\
=&(zip\,(+)(u, zip\,(\cdot)(\Omega, v)) +\!\!+\, zip\,(-)(u, zip\,(\cdot)(\Omega, v)))_{(i)} \\
=&\begin{cases} u_{(i)} + \omega_{2m}^{i} \cdot v_{(i)} & \text{falls } i < m \\ u_{(i-m)} - \omega_{2m}^{i-m} \cdot v_{(i-m)} & \text{sonst} \end{cases}
\end{aligned}$$

Wir haben damit eine komponentenweise Berechnung auf u und v erreicht, das einzige Problem sind noch die Koeffizienten ω_{2m}^{i} und ω_{2m}^{i-m}. FFT' ist also kein reiner DH, sondern ein Beinahe-DH. Es wird nötig sein, mehrere Werte zu tupeln mit deren Hilfe die ω_{2m}^{i} und ω_{2m}^{i-m} berechnet werden können. Weiterhin braucht man einen neuen DH auf Tupel. Es bleibt allerdings noch die Frage welchen DH und welche Ausgangstupel wir brauchen. Die Berechnung von m ist kein Problem, denn im Basisfall ist $m = 1$ und bei jedem *conquer*-Schritt verdoppelt sich m. Bleibt also noch die Berechnung von i. Wenn man die Elemente der Ausgangsliste vor der Berechnung von FFT' fortlaufend durchnumeriert, dann ist $i = (j \mod m)$, wobei sich j während der Berechnung von FFT' nicht ändert. Damit ergibt sich:

$$FFT' = (map\ \pi_1) \circ FFT3 \circ (addcomp\ c) \tag{6.2}$$

wobei

$$c = \left[\begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdots \begin{pmatrix} n-1 \\ 1 \end{pmatrix} \right]$$

und $FFT3 = \oplus \updownarrow \otimes$ mit

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1 + \omega_{2c_1}^{(b_1 \bmod c_1)\cdot a_2} \\ b_1 \\ 2c_1 \end{pmatrix}$$

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1 - \omega_{2c_2}^{(b_2 \bmod c_2)\cdot a_2} \\ b_2 \\ 2c_2 \end{pmatrix}$$

Schritt 3

Jetzt kann man die allgemeine Implementierung für DH verwenden. Setzen wir also (6.2) in den Hauptsatz ein. Dann ergibt sich für die Berechnung der FFT:

$$\begin{aligned} FFT = map\ \pi_1 \circ red^d(+\!\!+) \circ (\bigcirc_{i=1}^{d+1} mapdim^{(i,d+1)}\, h) \circ \\ dist_{(n_1,\ldots,n_d)} \circ (addcomp\ c) \circ permute \end{aligned}$$

Dieses Programm besteht aus drei Teilen.

- Permutieren, tupeln und verteilen der Daten kann als ein Schritt betrachtet werden. Wie dies konkret implementiert wird, hängt davon ab, wie die Daten an das parallele System geschickt werden bzw. wie die Daten auf dem parallelen System schon verteilt sind. Liegen die Daten schon verteilt vor, müssen sie noch entsprechend permutiert werden. Soll ein Prozessor die Daten an die restlichen Prozessoren verteilen, so wird er sie lokal permutieren und dann verteilen. Das Tupeln können alle Prozessoren lokal vornehmen.

- Im zweiten Schritt erfolgt die eigentliche FFT-Berechnung, wie sie in Kapitel 4 allgemein für DHs hergeleitet wurde.

- Die Implementierung des letzten Schritts hängt ähnlich wie im ersten Schritt davon ab, was mit den Daten geschehen soll. Werden die Daten weiterhin verteilt benötigt, dann braucht dieser Schritt nicht implementiert zu werden. Denkbar wäre auch ein Sammeln der Daten auf einem Prozessor.

6.3 Tupel-DHs

Die im letzten Kapitel vorgestellte Implementierung entspricht nicht ganz den in der Praxis benutzten Implementierungen. Anstatt Tripel zu kommunizieren, berechnen diese die benötigten Werte innerhalb der sequentiellen FFT-Funktion. Um näher zu dieser Implementierung zu kommen, werden in diesem Kapitel spezielle Beinahe-DHs eingeführt.

Definition 12. *Sei $f : \beta \to \beta$ eine beliebige Funktion und $\oplus', \otimes' : (\alpha, \beta) \to (\alpha, \beta) \to (\alpha, \beta)$ beliebige Operatoren. Dann ist h ein Tupel-DH, falls $h = map\ \pi_1 \circ \oplus \updownarrow \otimes \circ addcomp^{(1)}$ ein Beinahe-DH mit folgenden Operatoren $\oplus, \otimes$ ist:*

$$\begin{pmatrix} a \\ b \end{pmatrix} \oplus \begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{pmatrix} \oplus'(a, b, a', b') \\ f(b) \end{pmatrix}, \quad \begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{pmatrix} \otimes'(a, b, a', b') \\ f(b') \end{pmatrix}$$

Tupel-DHs sind also Beinahe-DHs, deren DH auf spezielle Operatoren eingeschränkt wird. Ziel ist es, das Tupeln am Anfang und Projizieren am Ende der Berechnung direkt vor bzw. hinter die lokalen Berechnungen der Implementierung zu schieben. Dann müssen bei $chdim^{(i,d)}$, der einzigen Funktion unserer Implementierung mit Kommunikation, nicht mehr Tripel kommuniziert werden. Dazu erst zwei einfache Beziehungen:

$$addcomp^{(d)}(dist_{(n_1,\ldots,n_d)}\ c) \circ dist_{(n_1,\ldots,n_d)} = dist_{(n_1,\ldots,n_d)} \circ (addcomp^{(1)}\ c) \tag{6.3}$$

$$addcomp^{(d)}(chdim^{(i,j)}\ c) \circ chdim^{(i,j)} = chdim^{(i,j)} \circ (addcomp^{(d)}\ c) \tag{6.4}$$

Außerdem wird noch eine Notation eingeführt:

Notation 3. *Sei h ein DH, $x \in [\alpha]_{(n_1 \times \cdots \times n_d)}$, $c \in [\beta]_{(n_1 \times \cdots \times n_d)}$ und $i \in \{1, 2\}$, dann ist:*

$$h^{(d)}_{(i)}\ c\ x = (map^{\,d}\ \pi_i) \circ (map^{\,d-1}\ h) \circ (addcomp^{(d)}\ c)\ x$$

$(h^{(1)}_{(1)}\ c)$ ist eine Notation für einen Beinahe-DH. Außerdem beachte man, daß im Falle eines Tupel-DHs $(h^{(d)}_{(2)}\ c\ x)$ unabhängig von x ist. Es gilt:

$$h^{(d)}_{(2)}\ c = map^{\,d}(f^{\log_2(n_d)})\ c \tag{6.5}$$

Theorem 3. *Sei h ein Tupel-DH, $c \in [\beta]_{(n_1 \times \cdots \times n_d)}$, dann gilt:*

$$h^{(1)}_{(1)}\ c = red^{\,d-1}(+\!\!+) \circ (\bigcirc_{i=1}^{d} chdim^{(i,d)} \circ (h^{(d)}_{(1)}\ c''') \circ chdim^{(i,d)}) \circ dist_{(n_1,\ldots,n_{d-1})}$$

wobei

$$c''' = chdim^{(i,d)} \circ map^{\,d}(f^{\log_2(\prod_{j=1}^{i-1} n_j)}) \circ dist_{(n_1,\ldots,n_{d-1})}\ c \tag{6.6}$$

Vor dem Beweis dieses Theorems wird erst noch ein Lemma bewiesen, das zeigt, wie ein $mapdim^{(i,d)}\ h$ die zweite Komponente verändert.

Lemma 12. *Sei h ein Tupel-DH, $c \in [\beta]_{(n_1 \times \cdots \times n_d)}$, $i \leq d$ und $A = mapdim^{(i,d)}\ h \circ (addcomp^{(d)}\ c)$, dann gilt:*

$$A = (addcomp^{(d)}\ (map^{\,d}(f^{\log_2(n_i)})\ c)) \circ map^{\,d}\pi_1 \circ A$$

Beweis.

$$\begin{aligned}
& mapdim^{(i,d)}\ h \circ (addcomp^{(d)}\ c) \\
= & \ \{ \text{ Def. } mapdim^{(i,d)} \ \} \\
& chdim^{(i,d)} \circ (map^{\,d-1}\ h) \circ chdim^{(i,d)} \circ (addcomp^{(d)}\ c) \\
= & \ \{ \ (6.4), \text{ sei } c'_i = c_d, c'_d = c_i \text{ und } c'_j = c_j \text{ für alle anderen } j \ \} \\
& chdim^{(i,d)} \circ (map^{\,d-1}\ h) \circ (addcomp^{(d)}\ c') \circ chdim^{(i,d)} \\
= & \ \{ \text{ Da } (addcomp^{(d)}\ c) \circ \pi_1 = id, \text{ falls } c \text{ zweite Komponente vor } \pi_1 \text{ war.} \\
& \quad \text{Außerdem sei } B = (map^{\,d-1}\ h) \circ (addcomp^{(d)}\ c') \circ chdim^{(i,d)} \ \} \\
& chdim^{(i,d)} \circ (addcomp^{(d)}\ (h^{(d)}_{(2)}\ c')) \circ (map^{\,d}\ \pi_1) \circ B \\
= & \ \{ \ (6.5) \ \} \\
& chdim^{(i,d)} \circ (addcomp^{(d)}\ (map^{\,d}(f^{\log_2(n_d)})\ c')) \circ (map^{\,d}\ \pi_1) \circ B \\
= & \ \{ \ (6.4) \ \} \\
& (addcomp^{(d)}\ (map^{\,d}(f^{\log_2(n_i)})\ c)) \circ map^{\,d}\pi_1 \circ A
\end{aligned}$$

$\square$

Mithilfe dieses Lemmas kann jetzt das Theorem bewiesen werden:

Beweis von Theorem 3.

$$h^{(1)}_{(1)} = (map\ \pi_1) \circ h \circ (addcomp^{(1)}\ c)$$
$$= \{ \text{ Haupttheorem } \}$$
$$(map\ \pi_1) \circ red^{d-1}(+\!\!+) \circ (\bigodot_{i=1}^{d} mapdim^{(i,d)}\ h) \circ dist_{(n_1,\ldots,n_{d-1})} \circ (addcomp^{(1)}\ c)$$
$$= \{ \text{ (6.3), } c' = dist_{(n_1,\ldots,n_{d-1})}\ c \ \}$$
$$red^{d-1}(+\!\!+) \circ (map^{d}\ \pi_1) \circ (\bigodot_{i=1}^{d} mapdim^{(i,d)}\ h) \circ (addcomp^{(d)}\ c') \circ dist_{(n_1,\ldots,n_{d-1})}$$
$$= \{ \text{ mehrmaliges Anwenden des Lemmas, } c'' = map^{d}(f^{\log_2(\prod_{j=1}^{i-1} n_j)})\ c' \ \}$$
$$red^{d-1}(+\!\!+) \circ (\bigodot_{i=1}^{d}(map^{d}\ \pi_1) \circ mapdim^{(i,d)}\ h \circ (addcomp^{(d)}\ c'')) \circ dist_{(n_1,\ldots,n_{d-1})}$$
$$= \{ \text{ Def. } mapdim^{(i,d)}, \text{ (6.4), } c''' = chdim^{(i,d)}\,(c''), \ \}$$
$$red^{d-1}(+\!\!+) \circ (\bigodot_{i=1}^{d} chdim^{(i,d)} \circ (h^{(d)}_{(1)}\ c''') \circ chdim^{(i,d)}\,) \circ dist_{(n_1,\ldots,n_{d-1})}$$

□

Die Elemente der d-dimensionalen Listen, auf die $chdim^{(i,d)}$ angewendet wird, sind nun keine Tupel mehr, sondern einfache Werte. Dafür ist der Ausdruck zwischen den beiden $chdim^{(i,d)}$ aufwendiger geworden. Dieser Ausdruck wird nun näher betrachtet:

$$((map^{d}\ \pi_1) \circ map^{d-1}\ h \circ (addcomp^{(d)}\ c''')\ x)_{(i_1,\ldots,i_{d-1})}$$
$$= \{ \text{ (5.1), Beziehung } map\text{-}\ zip \ \}$$
$$(zip^{d-1}((map\ \pi_1) \circ h \circ zip\,(pair))(x, c'''))_{(i_1,\ldots,i_{d-1})}$$
$$= \{ \text{ Beziehung } zip\text{- Projektion } \}$$
$$((map\ \pi_1) \circ h \circ zip\,(pair)(x_{(i_1,\ldots,i_{d-1})}, c'''_{(i_1,\ldots,i_{d-1})}))$$
$$= \{ \text{ (5.1) } \}$$
$$((map\ \pi_1) \circ h \circ (addcomp^{(1)}\ c'''_{(i_1,\ldots,i_{d-1})}))\ x_{(i_1,\ldots,i_{d-1})}$$

Jede Liste $x_{(i_1,\ldots,i_{d-1})}$ der d-ten Dimension muß mit einer jeweils neu generierten Liste $c'''_{(i_1,\ldots,i_{d-1})}$ getupelt werden. Nach Anwendung von h auf diese Tupellisten werden die zweiten Komponenten wieder verworfen.

Die Effizienz des Algorithmus, der in Theorem 3 beschrieben wurde, hängt also hauptsächlich von der Berechnung der $c'''_{(i_1,\ldots,i_{d-1})}$ ab. Deswegen muß c weiter eingeschränkt werden.

$$\begin{aligned}\odot &:: \beta \to \beta \to \beta \\ \mathit{init}, \mathit{step} &:: \beta \\ c_{(i)} &= \mathit{init} \odot (\bigodot_{j=0}^{i-1} \mathit{step}) \\ \forall x, y \in \beta &: f(x \odot y) = f(x) \odot f(y)\end{aligned}$$

Dann ergibt sich für c'''

$$\begin{aligned}
&c'''_{(i_1,\ldots,i_{d-1})} \\
=&((\mathit{chdim}^{(i,d)} \circ \mathit{map}^d(f^{\log_2(\prod_{j=1}^{i-1} n_j)}) \circ \mathit{dist}_{(n_1,\ldots,n_{d-1})})\ c)_{(i_1,\ldots,i_{d-1})} \\
=&\ \{ \text{ Definition } \mathit{chdim}^{(i,d)},\ j_i = i_d \quad j_d = i_i \quad \forall k \notin \{i,d\} : j_k = i_k \ \} \\
&(\mathit{map}^d(f^{\log_2(\prod_{j=1}^{i-1} n_j)}) \circ \mathit{dist}_{(n_1,\ldots,n_{d-1})}\ c)_{(j_1,\ldots,j_{d-1})} \\
=&\mathit{map}\,(f^{\log_2(\prod_{j=1}^{i-1} n_j)})\ (\mathit{dist}_{(n_1,\ldots,n_{d-1})}\ c)_{(j_1,\ldots,j_{d-1})} \\
=&\mathit{map}\,(f^{\log_2(\prod_{j=1}^{i-1} n_j)})\ [\mathit{init}', \mathit{init}' \odot \mathit{step}', \ldots, \mathit{init}' \odot \bigodot_{k=0}^{n_d-1} \mathit{step}']
\end{aligned}$$

Anstatt $c'''_{(i_1,\ldots,i_{d-1})}$ komplett vor dem Anwenden von h zu berechnen, überlassen wir einen Teil der Berechnung h selbst. Es wird init' und step' berechnet und an h übergeben. Wenn g die Umkehrfunktion von f ist, ergibt sich für h:

```
h(init, step, z) =
      (x,y) = halbieren(z)
      temp1 = init
      temp2 = init ⊙ ⨀_{i=0}^{|x|-1} step
      u = h(g(temp1), g(step), x)
      v = h(g(temp2), g(step), y)
      FOR i=0 TO |x|-1
           r[i] = ⊕′ (u[i], v[i], temp1, temp2)
           r[i+|x|] = ⊗′ (u[i], v[i], temp1, temp2)
           temp1 = temp1 ⊙ step
           temp2 = temp2 ⊙ step
      RETURN r
```

Bei dieser Implementierung ist der Programmierer nicht mehr frei in der Wahl der sequentiellen Implementierung des DHs h, er muß sich genau an das beschriebene Format halten. Weiterhin kommen einige neue Berechnungen hinzu, die je nach Definition von $\odot$ sehr aufwendig werden können.

An dieser Stelle muß auf zukünftige Arbeiten verwiesen werden. Diese Klasse der Tupel-DHs beinhaltet mit der FFT ein nichttriviales Beispiel, wobei die in diesem Kapitel angedeutete Optimierung als Motivation für eine genauere Betrachtung diese Unterklasse der Beinahe-DHs zu sehen ist.

6.4 Experimente

Aufgrund der Anpasung von der FFT an das DH-Format erhält man eine Familie von Implementierungen. Das MPI-Programm für die Berechnung der FFT ist variabel in der Anzahl der Elemente, der Anzahl der Prozessoren und der Anzahl der Prozessordimensionen. Die folgenden Zeitmessungen wurden mit dem MPI-Programm, das im Anhang abgedruckt ist, auf einem transputerbasierten Parallelrechner der Firma Parsytec durchgeführt.

Abbildung 6.1 zeigt den Speedup auf 64 Prozessoren gegenüber der sequentiellen Implementierung auf einem Prozessor der gleichen Maschine bei d-dimensionalem Prozessorgitter für verschiedene Längen der Eingabeliste. Man sieht hier, daß selbst auf einer Maschine mit fester Prozessoranzahl es von der Eingabegröße abhängt, wie viele Prozessordimensionen für den besten Speedup zu wählen sind.

Abbildung 6.2 zeigt den Speedup bei fester Länge der Eingabelisten und variabler Anzahl an Prozessoren. Hier wird der Speedup des 2D-Algorithmus mit zuhnehmender Prozessorzahl sogar schlechter, da bei 256 Prozessoren nur noch 65536/256=256 Elemente auf einem Prozessor abgespeichert sind und die Funktion `MPI_Alltoall` auf allen 256 Prozessoren aufgerufen wird, während beim Hypercubealgorithmus die Prozessoren innnerhalb der Kommunikatoren in Gruppen zu je 2 Prozessoren aufgteilt werden.

Welcher Algorithmus den besten Speedup erzeugt, hängt von der Performance der Funktion `MPI_Alltoall` ab. Folgende zwei Trends lassen sich jedoch aufgrund der hier skizzierten Messergebnisse sagen:

- Für zunehmende Prozessoranzahl bringt eine größere Anzahl von Prozessordimensionen einen größeren Speedup.
- Je größer die Eingabeliste ist, umso kleiner sollte die Anzahl der Prozessordimensionen sein.

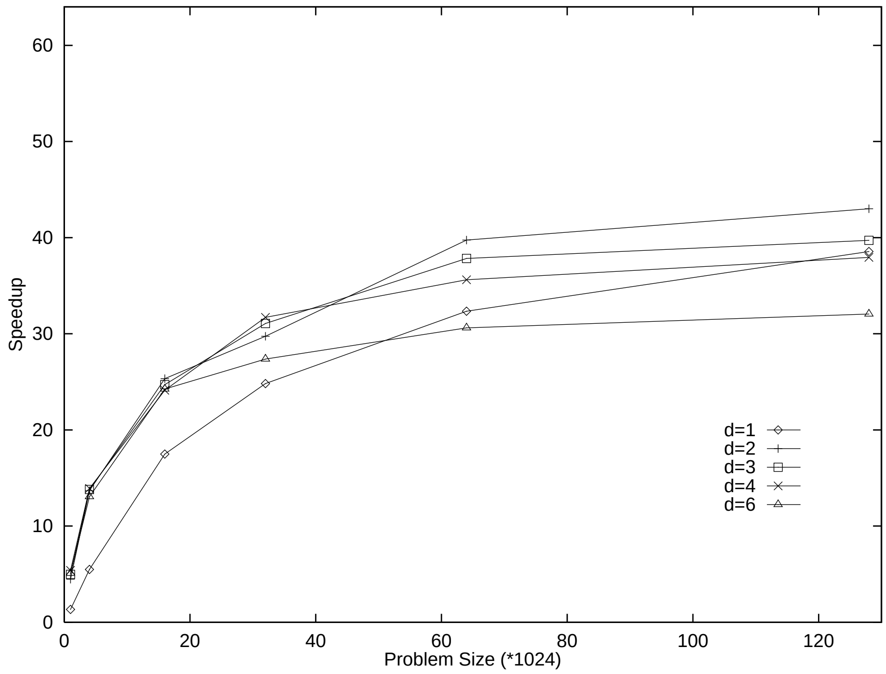

Abbildung 6.1: Speedup auf 64 Prozessoren

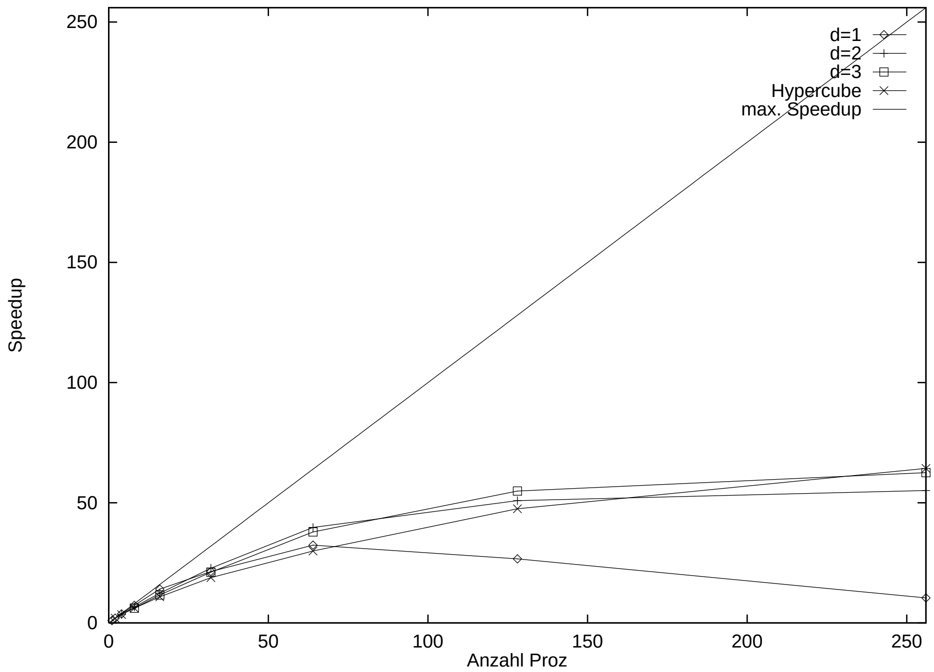

Abbildung 6.2: Speedup bei 65536 Elementen

Kapitel 7

Zusammenfassung und Ausblick

Die funktionale Notation und Transformationen in BMF wurden benutzt um effiziente parallele Programme für Divide-and-ConquerAlgorithmen herzuleiten.

Diese Arbeit demonstriert einen systematischen, formalen Ansatz zur Parallelprogrammierung, wobei die Programmentwicklung in zwei Phasen aufgeteilt wird.

- In der Systemphase werden Divide-and-Conquer-Algorithmen mithilfe der BMF Notation spezifiziert und klassifiziert. Für eine spezielle Unterklasse, den DHs, wird eine Familie effizienter paralleler Implementierungen mithilfe von semantikerhaltenden Transformationen abgeleitet. Das Ergebnis ist eine parametrisierte verteilte Implementierung. Diese Implementierung ist beweisbar korrekt, architekturunabhängig und hat eine vorhersagbare Performance, da das Zielprogramm ein SPMD-Programm mit nur einem Konstrukt zur kollektiven Kommunikation ist.

- In der Anwendungsphase wird in einer Fallstudie gezeigt, wie eine spezielle Funktion, die FFT, systematisch an das DH Skelett angepaßt wird. Dadurch erhält man schließlich eine allgemeine parallele Implementierung für die FFT.

Mit diesem Ansatz werden Implementierungen abgeleitet, deren Spezialfälle (Hypercube, 2D und 3D) den in der Praxis benutzten Lösungen entsprechen.

Im Gegensatz zur Verifikation paralleler Programme wird in dieser Arbeit eine effiziente parallele Implementierung durch Transformationsregeln in BMF hergeleitet. Als Datenstruktur werden Powerlisten benutzt [Mis94], aber im Gegensatz zu J. Misra wird hier eine iterative Lösung erzeugt, die direkt als MPI-Programm implementiert werden kann.

Die abgeleiteten Programme bestehen aus mehreren Schritten, wobei jeder Schritt aus genau drei Phasen besteht: Kommunikation, Berechnung, Kommunikation. In der optimierten Version sind es nur zwei Phasen, da die dritte Phase wegfällt. Solche Programme sind einfach zu verstehen und deren Performance kann gut abgeschätzt werden.

Ein weiterer Vorteil ist die Architekturunabhängigkeit der Lösungen und die Möglichkeit, die Zielprogramme direkt in MPI zu implementieren. Für MPI gibt es effiziente Implementierungen für die meisten parallelen und verteilten Plattformen.

Die größte Einschränkung dieses Ansatzes ist das DH-Format. Allerdings hat erst diese Spezialisierung die verteilte Implementierung ermöglicht, die für die FFT eher den in der Praxis benutzten Lösungen entspricht als der allgemeinere Ansatz in [GL95].

Für die Zukunft bleiben noch ein paar offene Fragen:

- Tupel-DHs: Es wäre wünschenswert, einen besseren Überblick über diese Unterklasse der Beinahe-DHs zu bekommen. Die Optimierung dieser Klasse wurde in Kapitel 6.3 nicht zu Ende geführt. Es ist aber davon auszugehen, daß man auf diesem Weg näher an die Implementierungen herankommt, die für die FFT in der Praxis benutzt werden.

- Neben der Konkatenation $+\!\!+$ wird bei Powerlisten noch ein zweiter Konstruktor $\bowtie$ benutzt. Dieser Konstruktor entspricht einer "odd-even" Verkettung zweier Listen wie sie auch bei der FFT benötigt wird (siehe Schritt 1 der Anpassung in Kapitel 6.2). Es ist davon auszugehen, daß DHs mit diesem Konstruktor ähnlich implementiert werden können.

- Es wäre weiterhin wünschenswert neben der FFT noch weitere Funktionen an das DH-Format anzupassen.

Literaturverzeichnis

[AS95] K. Achatz and W. Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In Moeller B, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 97–127, 1995.

[Bir87] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer Verlag, 1987.

[Bir88] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.

[Col88] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[For95] MPI Forum. *MPI: A Message-Passing Interface Standard.* http://www.mpi-forum.org, 1995.

[GL95] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* MIT Press, 1994.

[Gor96a] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer Verlag, 1996.

[Gor96b] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming languages: Implementation, Logics and Programs. PLILP'96*, Lecture Notes in Computer Science 1140, pages 274–288. Springer Verlag, 1996.

[Kum94] V. Kumar. *Introduction to Parallel Computing.* Benjamin/Cummings, 1994.

[Mis94] J. Misra. Powerlists: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.

[Pac97] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, 1997.

[Par93] Parsytec Computer GmbH. *PARIX 1.2 Software Documentation*, March 1993.

[Qui94] M. J. Quinn. *Parallel Computing.* McGraw-Hill, Inc., 1994.

[Ski90] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.

[Ski92] D. B. Skillicorn. *Parallelism and the Bird-Meertens Formalism.* ftp://ftp.qucis.queensu.ca/pub/skill, April 1992.

Anhang A

Programmcode

```
#include<stdio.h>
#include<stdlib.h>
#include <mpi.h>
#include "zeitm.h"
#include "complex.h"

typedef struct {
  Complex x;
} Datentyp;

typedef struct {
  Complex omega_ni;
  Complex omega_factor;
} Datentyp2;


/* self=Prozessornummer, np=Anzahl Prozessoren,   */
/* global_laenge=Gesamtanzahl der Eingabeelemente */
/* local_laenge=global_laenge/np                  */
int self, np, global_laenge=0, local_laenge=0;
MPI_Datatype MPI_COMPLEX, MPI_DATENTYP;
MPI_Comm *split_comms;
MPI_Comm cart_comm;


void h(Datentyp a[], int laenge, Datentyp2 b) {
  int i,laenge2=laenge>>1;
  Complex omega_ni;
```

```
  Datentyp temp;
  Datentyp2 b_new;
  Complex lambda;

  if(laenge!=1) {
    b_new.omega_ni=Complex_times(b.omega_ni,b.omega_ni);
    b_new.omega_factor=Complex_times(b.omega_factor,
     b.omega_factor);
    h(a, laenge2, b_new);
    h(&(a[laenge2]), laenge2, b_new);

    omega_ni=b.omega_ni;

    for(i=0; i<laenge2; i++) {
      temp=a[i];

      lambda=Complex_times(omega_ni, a[laenge2+i].x);
      a[i].x         = Complex_plus(temp.x, lambda);
      a[laenge2+i].x = Complex_minus(temp.x, lambda);

      omega_ni=Complex_times(omega_ni, b.omega_factor);
    }
  }
}


/* testet, ob n der Form 2^k ist */
int pot2(int n) {
  if(n<1) return 0;
  while(n>1) {
    if(n%2==1) return 0;
    n/=2;
  }
  return 1;
}


void PrintSeq(Datentyp a[], int laenge, char *sss) {
  int i,j;
  for(j=0; j<np; j++) {
    if(self==j) {
```

```
      for(i=0; i<laenge; i++) {
printf("%s Datum (self*local_laenge+i) %i : ",
       sss, self*laenge+i);
Complex_print(a[i].x);
printf("\n");
fflush(0);
      }
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

Datentyp *init_random(int anzahl) {
  int i;
  Datentyp *a;

  global_laenge=anzahl;
  local_laenge=anzahl/np;
  a=(Datentyp *) malloc (anzahl*sizeof(Datentyp));
  for(i=0;i<anzahl;i++) {
    a[i].x.imag=(double)rand()/1000;
    a[i].x.real=(double)rand()/1000;
  }
  return a;
}

Datentyp *init_daten() {
  int i;
  char filename[256];
  FILE *datei;
  Datentyp indata;
  Datentyp *a;

  sprintf(filename, "proc%i",self);
  if ( (datei=fopen(filename,"r")) == 0) {
    printf("fehler beim oeffnen der datei %s\n", filename);
    exit(-1);
  }
  fscanf(datei,"%i\n", &global_laenge);

  local_laenge=global_laenge/np;
```

```
  if((!pot2(np)) || (!pot2(global_laenge)) ||
     (np*np>global_laenge)) {
    if(self==0) {
      printf("Daten muessen folgende form haben: ");
      printf("np=2^n laenge=2^m\n");
      printf("wobei m>=2*n\n");
    }
    fclose(datei);
    MPI_Finalize();
    exit(-1);
  }
  a=(Datentyp *) malloc (local_laenge*sizeof(Datentyp));

  for(i=0; i<local_laenge; i++) {
    fscanf(datei,"%lf,", &(indata.x.real));
    fscanf(datei,"%lf\n", &(indata.x.imag));
    a[i]=indata;
  }
  fclose(datei);
  return a;
}

void init_newtypes(void) {
  int blocklens[1];
  MPI_Datatype oldtypes[1];
  MPI_Aint indices[1];

  /* Definition des neuen Typs MPI_COMPLEX */
  MPI_Type_contiguous(2, MPI_DOUBLE, &MPI_COMPLEX);
  MPI_Type_commit(&MPI_COMPLEX);

  /* Definition des neuen eigenen Datentyps */
  oldtypes[0]=MPI_COMPLEX;
  indices[0]=0;
  blocklens[0]=1;
  MPI_Type_struct(1, blocklens, indices,
  oldtypes, &MPI_DATENTYP);
  MPI_Type_commit(&MPI_DATENTYP);
}

MPI_Comm *init_dimcomms(int proc_ndims, int *dims,
```

```
int *coords) {
  MPI_Comm *comms;
  MPI_Comm cart_comm;
  int i, j, factor;
  int *periods, color;

  comms=(MPI_Comm *) malloc(proc_ndims*sizeof(MPI_Comm));
  periods=(int *) malloc(proc_ndims*sizeof(int));

  for(i=0; i<proc_ndims; i++) {
    periods[i]=0;
  }
  MPI_Cart_create(MPI_COMM_WORLD, proc_ndims, dims ,
  periods, 0, &cart_comm);
  MPI_Cart_get(cart_comm, proc_ndims, dims, periods, coords);
  for(i=0; i<proc_ndims; i++) {
    color=0;
    factor=1;
    for(j=0; j<proc_ndims; j++) {
      if(j!=i) color+=factor*coords[j];
      factor*=dims[j];
    }
    MPI_Comm_split(cart_comm, color, 0, &comms[i]);
  }

  free(periods);

  return comms;
}

/* transponiert array [a]^(l1 X l2) */
void local_transpose(Datentyp a[], Datentyp b[],
     int l1, int l2) {
  int i;
  for(i=0; i<(l1*l2); i++)
    b[i]=a[(int) (i/l1)+(i%l1)*(l2)];
}

void Map_Dim(Datentyp *a, int laenge,
     int *coords, int *dims, int ndims, int mapdim)
{
```

```
int factor, position;
Complex omega;
Datentyp2 b;
Datentyp *temp;
int *new_dims;
int *new_coords;
int i,j;

temp=(Datentyp *) malloc (local_laenge*sizeof(Datentyp));

MPI_Alltoall(a, local_laenge/dims[mapdim], MPI_DATENTYP,
             temp, local_laenge/dims[mapdim], MPI_DATENTYP,
     split_comms[mapdim]);
local_transpose(temp, a, dims[mapdim],
local_laenge/dims[mapdim]);

/*******/
new_coords=(int *) malloc ((ndims+2)*sizeof(int));
new_dims=(int *) malloc ((ndims+2)*sizeof(int));

for(i=0; i<ndims; i++) {
  new_dims[i]=dims[i];
  new_coords[i]=coords[i];
}
new_dims[ndims]=dims[mapdim];
new_dims[ndims+1]=local_laenge/dims[mapdim];
new_coords[ndims]=coords[mapdim];
new_coords[mapdim]=0;

omega=Complex_exp((double) (2.0/(double) laenge));
new_coords[ndims+1]=0;
factor=1;position=0;
for(j=ndims+1; j>=0; j--) {
  position+=new_coords[j]*factor;
  if(j==mapdim)
    b.omega_factor=Complex_pow(omega,factor);
  factor*=new_dims[j];
}

b.omega_ni=Complex_pow(omega, position);
```

```
  for(i=0; i<local_laenge; i+=dims[mapdim]) {
    h(&(a[i]),dims[mapdim], b);
    b.omega_ni=Complex_times(b.omega_ni, omega);
  }

  free(new_dims);
  free(new_coords);
  /********/

  local_transpose(a, temp, local_laenge/dims[mapdim],
  dims[mapdim]);
  MPI_Alltoall(temp, local_laenge/dims[mapdim], MPI_DATENTYP,
               a, local_laenge/dims[mapdim], MPI_DATENTYP,
      split_comms[mapdim]);
  free(temp);
}


int main(int argc, char *argv[]) {
  Datentyp *a;
  Datentyp2 b;
  double start=0.0;
  int i, proc_ndims=0;
  int *dims;
  int *coords;
  int laenge;
  Complex omega;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &self);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if(argc<2) {
    if(self==0) printf("Usage: ... <anz_dims> [<anz_elem>]\n");
    MPI_Finalize();
    exit(-1);
  }

  proc_ndims=atoi(argv[1]);
  /* MPI_COMPLEX und MPI_DATENTYP erstellen */
  init_newtypes();
```

```
/* Kartesisches Koordinatensystem der
   dimension proc_ndims erstellen */
dims=(int *) malloc(proc_ndims*sizeof(int));
for(i=0; i<proc_ndims; i++) dims[i]=0;
MPI_Dims_create(np, proc_ndims, dims);

/* Kommunikatoren fuer dimensionsweise
   Kommunikation erstellen */
coords=(int *) malloc(proc_ndims*sizeof(int));
split_comms=init_dimcomms(proc_ndims, dims, coords);

/****************************************/
if(argc<3){
  if(self==0)
    printf("Usage: ... <anz_dims> <anz_elem>\n");
  MPI_Finalize();
  exit(-1);
}
a=init_random(atoi(argv[2]));
/***************************************
a=init_daten();
****************************************/
/*PrintSeq(a, local_laenge, "vorher");*/

start=timer_start(MPI_COMM_WORLD);

laenge=local_laenge;

omega=Complex_exp((double) (2/(double) laenge));
b.omega_ni=Complex_pow(omega,self*local_laenge);
b.omega_factor=omega;

h(a, local_laenge, b);
for(i=proc_ndims-1; i>=0; i--) {
  laenge*=dims[i];
  Map_Dim(a, laenge, coords, dims, proc_ndims, i);
}
if(self==0)
  printf("%i %i ",np,global_laenge);
timer_end(start, "", MPI_COMM_WORLD);
```

```
  /*PrintSeq(a, local_laenge, "nachher");*/

  free(a);
  for(i=0; i<proc_ndims; i++)
    MPI_Comm_free(&split_comms[i]);
  free(coords);
  free(dims);
  free(split_comms);
  MPI_Finalize();
  return 0;
}
```