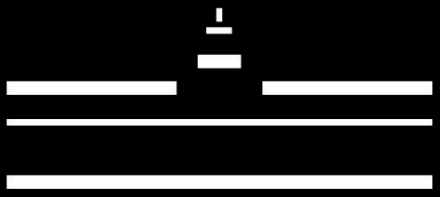


Westfälische
Wilhelms-Universität
Münster

Datenparallele algorithmische Skelette

Erweiterungen und Anwendungen der Münster Skelettbibliothek Muesli

Philipp Ciechanowicz

Datenparallele algorithmische Skelette

Erweiterungen und Anwendungen der Münster Skelettbibliothek Muesli

Inauguraldissertation
zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften durch die
Wirtschaftswissenschaftliche Fakultät der
Westfälischen Wilhelms-Universität Münster

vorgelegt von
Dipl.-Wirt.-Inf. Philipp Ciechanowicz
aus Dresden

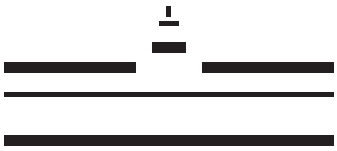
Münster, Dezember 2010

Dekan:	Prof. Dr. Thomas Apolte
Erstberichterstatter:	Prof. Dr. Herbert Kuchen
Zweitberichterstatter:	Prof. Dr. habil. Sergei Gorlatch
Tag der mündlichen Prüfung:	18. November 2010

Philipp Ciechanowicz

Datenparallele algorithmische Skelette

MV WISSENSCHAFT



Westfälische
Wilhelms-Universität
Münster

Wissenschaftliche Schriften der WWU Münster

Reihe IV

Band 2

Philipp Ciechanowicz

Datenparallele algorithmische Skelette

Erweiterungen und Anwendungen der Münster Skelettbibliothek Muesli

MV WISSENSCHAFT

Wissenschaftliche Schriften der WWU Münster
herausgegeben von der Universitäts- und Landesbibliothek Münster
http://www.ulb.uni-muenster.de

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Buch steht gleichzeitig in einer elektronischen Version über den Publikations- und Archivierungsserver der WWU Münster zur Verfügung.
http://www.ulb.uni-muenster.de/wissenschaftliche-schriften

Philipp Ciechanowicz
„Datenparallele algorithmische Skelette. Erweiterungen und Anwendungen der Münster Skelettbibliothek Muesli“
Wissenschaftliche Schriften der WWU Münster, Reihe IV, Band 2

Die Reihe „Wissenschaftliche Schriften der WWU Münster“ erscheint im Verlagshaus Monsenstein und Vannerdat OHG Münster
www.mv-wissenschaft.com

ISBN 978-3-8405-0029-9 (Druckausgabe)
URN urn:nbn:de:hbz:6-65489628556 (elektronische Version)

Satz: Philipp Ciechanowicz
Umschlag: MV-Verlag
Druck und Bindung: MV-Verlag

Vorwort

Die vorliegende Arbeit ist zum Abschluss meiner vierjährigen Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster verfasst und im September 2010 beim Dekanat der Wirtschaftswissenschaftlichen Fakultät eingereicht worden. Wie viele andere Werke ist auch diese Dissertation nicht im Alleingang, sondern mit Hilfe zahlreicher Unterstützer angefertigt worden, bei denen ich mich an dieser Stelle aufs Herzlichste bedanken möchte.

An vorderster Stelle möchte ich mich bei meinem Doktorvater Herrn Prof. Dr. Herbert Kuchen für die Möglichkeit bedanken, mich mit einem der spannendsten und zukunftsweisendsten Themen der Informatik, der parallelen Programmierung, auseinandersetzen zu können. Insbesondere für die Unterstützung beim Verfassen und Veröffentlichen meiner Forschungsarbeiten und der damit verbundenen Chance zur Teilnahme an zahlreichen internationalen Konferenzen bin ich sehr dankbar. Mein Dank gilt auch Herrn Prof. Dr. habil. Sergei Gorlatch für die Übernahme des Zweitgutachtens sowie Herrn Prof. Dr. Ulrich Müller-Funk für seine Mitwirkung in der Promotionskommission.

Des Weiteren möchte ich den Mitarbeitern unserer Forschungsgruppe für die kollegiale Zusammenarbeit sowie für die damit verbundene ungezwungene Arbeitsatmosphäre danken. Hervorheben möchte ich an dieser Stelle Herrn Dipl.-Wirt.-Inf. Christian Hermanns, aber auch bei Herrn Dipl.-Wirt.-Inf. Christian Arndt, Frau Dr. Susanne Gruttmann, Herrn Dipl.-Wirt.-Inf. Henning Heitkötter, Herrn MScIS Tim Majchrzak, Herrn Dr. Michael Poldner, Herrn Dipl.-Wirt.-Inf. Claus Usener, Herrn Dipl.-Wirt.-Inf. Ulrich Wolffgang sowie bei Frau Ursula Kortemeyer möchte ich mich bedanken. Ein herzliches Dankeschön für die tolle, unvergessliche Zeit!

Zu guter Letzt ist es mir ein besonderes Anliegen meiner Familie zu danken. Vor allem meinen Eltern möchte ich einerseits für ihre bedingungslose und immerwährende Unterstützung, andererseits für das Korrigieren der vorliegenden Arbeit sowie für zahlreiche Ergänzungen und Verbesserungsvorschläge einen ganz speziellen Dank aussprechen. Ihr seid einfach die Besten! Auch bei meiner Freundin möchte ich mich für ihren Rückhalt in guten wie in weniger guten Zeiten bedanken, was erheblich zum Gelingen dieser Arbeit beigetragen hat!

Münster, im Dezember 2010 Philipp Ciechanowicz

Abstract

Die Erstellung eines parallelen Programms ist für gewöhnlich eine anspruchsvolle und komplexe Aufgabe. In der Designphase muss die Dekomposition des Problems und die Aggregation der Teilergebnisse berücksichtigt, in der Implementierungsphase die Kommunikation zwischen den einzelnen Recheneinheiten koordiniert werden. Zwar existiert eine Vielzahl ausgereifter und leistungsfähiger Programmierschnittstellen zur parallelen Programmierung, deren Anwendung erfordert jedoch umfangreiche Kenntnisse und ist in Folge des geringen Abstraktionsniveaus vergleichsweise fehleranfällig. Um diese Problematik zu vermeiden, bietet sich daher die Verwendung sogenannter *algorithmischer Skelette* an. Diese kapseln typische parallele Rechen- bzw. Kommunikationsmuster und abstrahieren somit von den Details der Parallelisierung. Die Münsteraner Skelettbibliothek Muesli stellt diese Skelette in Form einer Programmbibliothek zur Verfügung, um so die Erstellung paralleler Programme zu vereinfachen.

Die vorliegende Arbeit thematisiert den datenparallelen Bestandteil der Münsteraner Skelettbibliothek, der neben den verteilten Datenstrukturen vor allem die datenparallelen Skelette beinhaltet. Das grundsätzliche Ziel besteht in der Weiterentwicklung dieses Teils der Bibliothek, so dass im Folgenden neben den implementierten Erweiterungen auch Anwendungen beschrieben werden, die mit Hilfe von Muesli parallelisiert wurden. Eine der gravierendsten Neuerungen ist die Unterstützung von Mehrkernprozessoren durch die Verwendung von OpenMP, infolgedessen mit Hilfe von Muesli entwickelte Programme auch auf Parallelrechnern mit hybrider Speicherarchitektur effizient skalieren. Durch die Implementierung einer speziellen Abstraktionsschicht besitzt dieses Feature darüber hinaus optionalen Charakter, d.h. mit Hilfe von Muesli entwickelte Programme skalieren weiterhin auch auf Einkernprozessoren bzw. Rechnern mit verteiltem Speicher. Eine zusätzliche elementare Erweiterung der Skelettbibliothek stellt die Neu-

entwicklung einer verteilten Datenstruktur für dünnbesetzte Matrizen dar. Letztere implementiert, neben diversen datenparallelen Skeletten, ein flexibles und erweiterbares Designkonzept, was die Verwendung benutzerdefinierter Kompressions- sowie Lastverteilungsmechanismen ermöglicht. Abschließend werden mit dem LM OSEM-Algorithmus, einem Verfahren zur medizinischen Bildrekonstruktion, sowie den ART 2-Netzen, einer speziellen Form künstlicher neuronaler Netze, zwei Anwendungen vorgestellt, die mit Hilfe von Muesli parallelisiert wurden. Neben einer ausführlichen Beschreibung der Funktionsweise sowie der grundlegenden Eigenschaften und Konzepte von MPI und OpenMP wird darüber hinaus der aktuelle Forschungsstand skizziert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

Abkürzungsverzeichnis

AMD ..... Advanced Micro Devices, Inc.

API ..... application programming interface

ART ..... adaptive Resonanztheorie

BSR ..... block sparse row

CAML ..... Categorical Abstract Machine Language

CORBA ..... Common Object Request Broker Architecture

CPU ..... central processing unit

CRS ..... compressed row storage

CUDA ..... Compute Unified Device Architecture

DSP ..... digital signal processor

EM-ML ..... Expectation Maximization–Maximum Likelihood

GB ..... Gigabyte

Gb ..... Gigabit

GCC ..... GNU Compiler Collection

GHz ..... Gigahertz

GPFS ..... General Parallel File System

GPGPU ..... General-Purpose Computing on Graphics Processing Units

HPC ..... high-performance computing

ICC ..... Intel C++ Compiler

ISD ..... image space decomposition

KNN ..... künstliches neuronales Netz

LM OSEM .. List-Mode Ordered Subset Expectation Maximization

MIMD ..... multiple instruction, multiple data

MISD ..... multiple instruction, single data

MPI ..... Message Passing Interface

OAL ..... OpenMP Abstraction Layer

OpenCL ..... Open Computing Language

OpenMP ..... Open Multi-Processing

OSEM ..... Ordered Subset Expectation Maximization

PALMA ..... Paralleles Linux-System für Münsteraner Anwender

PET ..... Positronen-Emissions-Tomographie

POSIX ..... Portable Operating System Interface for Unix

PSD ..... projection space desomposition

PU ..... processing unit

RAID ..... redundant array of independent (inexpensive) disks

RAM ..... random-access memory

RMI ..... remote method invocation

SATA ..... Serial Advanced Technology Attachment

SIMD ..... single instruction, multiple data

SISD ..... single instruction, single data

STL ..... Standard Template Library

TB ..... Terabyte

TBB ..... Threading Building Blocks

ZIV ..... Zentrum für Informationsverarbeitung

Symbolverzeichnis

ρ	Toleranzparameter eines ART-Netzes, $\rho \in \mathbb{R}^+, \rho \leqslant 1$
A	dünnbesetzte Matrix
a	(verteiltes) eindimensionales Feld
a_i	Argument i einer n-stelligen Funktion, $i, n \in \mathbb{N}_0, i < n$
B	dünnbesetzte Matrix
b	verteiltes Feld
BU	Bottom-Up-Matrix eines ART 2-Netzes
c	Spaltenanzahl einer Submatrix, $c \in \mathbb{N}$
$cols$	Anzahl Blöcke pro Spalte einer verteilten Matrix, $cols \in \mathbb{N}$
d_i	Teilbild, $d_i \in \mathbb{R}^w$
d_i^*	lokales Teilbild, $d_i^* \in \mathbb{R}^w$
e	Element einer Datenstruktur
f	benutzerdefinierte Argumentfunktion
F_0	Eingabeschicht eines ART 2-Netzes
F_1	Vergleichsschicht eines ART 2-Netzes
F_2	Erkennungsschicht eines ART 2-Netzes

g benutzerdefinierte Argumentfunktion

h zu rekonstruierendes Bild in Vektorform, $h \in \mathbb{R}^w$

i Indexvariable, $i \in \mathbb{N}_0$

ia eindimensionales Feld zur Zeilenseparation

j Indexvariable, $j \in \mathbb{N}_0$

ja eindimensionales Feld zur Speicherung von Spaltennummern

k Indexvariable, $k \in \mathbb{N}_0$

m Spaltenanzahl einer (Sub-)Matrix, $m \in \mathbb{N}$

max maximale Anzahl zu verteilender Submatrizen, $max \in \mathbb{N}$

mx zweidimensionales Feld

N Menge an Neuronen eines künstlichen neuronalen Netzes, $N \in \mathbb{N}$

n Zeilenanzahl einer (Sub-)Matrix, $n \in \mathbb{N}$

ne Anzahl von einer Submatrix gespeicherten Elemente, $ne \in \mathbb{N}_0$

nnz Anzahl von Null verschiedener Elemente einer Matrix, $nnz \in \mathbb{N}_0$

$nnzb$ Anzahl von Blöcken, die mindestens eine Zahl ungleich Null enthalten, $nnzb \in \mathbb{N}$

np Anzahl verwendeter Prozesse, $np \in \mathbb{N}$

ns Anzahl von einem Prozess gespeicherten Submatrizen, $ns \in \mathbb{N}_0$

nt Anzahl verwendeter Threads pro Rechenknoten, $nt \in \mathbb{N}$

op Operator einer **`reduction`**-Klausel, $op \in \{+, -, *, \hat{}, \&, |, \&\&, ||\}$

P Systemmatrix, $P \in \mathbb{R}^{v \times w}$

P_i Projektion, $P_i \in \mathbb{R}^w$

p_i Prozess i, $i \in \mathbb{N}_0$

pid Prozess-Nummer, $pid \in \mathbb{N}_0$

q Normalisierungsvektor, $q \in \mathbb{R}^w$

R Reset-Komponente eines ART-Netzes

r Zeilenanzahl einer Submatrix, $r \in \mathbb{N}$

$rows$ Anzahl Blöcke pro Zeile einer verteilten Matrix, $rows \in \mathbb{N}$

S_i Schicht eines künstlichen neuronalen Netzes, $i \in \mathbb{N}_0$

s_i Teilmenge aller Zerfallsereignisse, $i \in \mathbb{N}, i \leqslant u$

sid Submatrix-Nummer, $sid \in \mathbb{N}_0$

$size$ Anzahl Iterationen pro Block, $size \in \mathbb{N}$

t_i Thread i, $i \in \mathbb{N}_0$

TD Top-Down-Matrix eines ART 2-Netzes

tid Thread-Nummer, $tid \in \mathbb{N}_0$

$type$ Strategie zur Aufteilung des Iterationsraums, $type \in \{$**`auto`**, **`dynamic`**, **`guided`**, **`runtime`**, **`static`**$\}$

u Anzahl Teilmengen, $u \in \mathbb{N}$

V Menge an Verbindungen zwischen den Neuronen eines künstlichen neuronalen Netzes

v Ereignisse pro Teilmenge, $v \in \mathbb{N}$

var Variable einer OpenMP-Klausel

W Gewichtsmatrix eines neuronalen Netzes

w Anzahl Voxel des zu rekonstruierenden Bildes h, $w \in \mathbb{N}$

X n-elementiger Eingabevektor, $n \in \mathbb{N}$

x Anzahl sämtlicher Zerfallsereignisse, $x \in \mathbb{N}$

Y Ausgabevektor

Kapitel 1

Einführung

Inhalt

1.1 Algorithmische Skelette

Die Erstellung eines parallelen Programms ist für gewöhnlich eine anspruchsvolle und komplexe Aufgabe, bei der während des gesamten Softwareentwicklungsprozesses eine Vielzahl von Schwierigkeiten bewältigt werden müssen. Vor dem Hintergrund des klassischen Wasserfallmodells [42, S. 99 ff.] lassen sich den einzelnen Phasen folgende Problem- bzw. Fragestellungen zuordnen:

- In der Designphase muss die Dekomposition des Problems in mehrere Teilprobleme, die Verteilung der Teilprobleme auf die einzelnen Recheneinheiten sowie die Aggregation der Teilergebnisse berücksichtigt werden. Darüber hinaus muss ein Kommunikationsprotokoll konzipiert werden, das den effizienten Informationsaustausch zwischen den Recheneinheiten festlegt, bei dem es weder zu Verklemmungen (engl. deadlock) noch Verhungerungen (engl. starvation) kommen kann.

- In der Implementierungsphase muss, in Abhängigkeit von der Speicherarchitektur des zu verwendenden Parallelrechners, das oben erwähnte Kommunikationsprotokoll umgesetzt werden. Für den Fall eines verteilten Speichers muss der Nachrichtenaustausch implementiert werden, was i.d.R. die Speicherallokation für Nachrichtenpuffer sowie die (De-)Serialisierung von Objekten beinhaltet. Für den Fall eines gemeinsamen Speichers muss die Synchronisation kritischer Abschnitte gewährleistet werden, was für gewöhnlich die Verwendung eines Verfahrens zum wechselseitigen Ausschluss erfordert. Für den Fall eines hybriden Speichers müssen offensichtlich alle erwähnten Sachverhalte gleichzeitig berücksichtigt werden.
- In der Testphase müssen etwaige im Programm eingebaute Fehler identifiziert und behoben werden. Die Fehlersuche gestaltet sich hierbei jedoch wesentlich schwieriger als bei einem sequenziellen Programm, da das Verhalten eines parallelen Programms u.U. nicht deterministisch, d.h. von der konkreten Ausführungsreihenfolge der einzelnen Prozesse abhängig ist, so dass Fehlerzustände nicht reproduziert werden können.

Zwar existiert eine Vielzahl ausgereifter und leistungsfähiger Programmierschnittstellen zur parallelen Programmierung, wie z.B. MPI [231, 273] und OpenMP [237, 238] für die Programmierung von Parallelrechnern mit verteiltem bzw. gemeinsamem Speicher, deren Anwendung erfordert jedoch umfangreiche Kenntnisse und ist in Folge des geringen Abstraktionsniveaus vergleichsweise fehleranfällig. Um diese Problematik zu vermeiden und damit gleichzeitig die Erstellung paralleler Programme drastisch zu vereinfachen, bietet sich daher die Verwendung sogenannter *algorithmischer Skelette* an.

Das Konzept eines algorithmischen Skeletts, im Folgenden der Einfachheit halber kurz als Skelett bezeichnet, wurde erstmalig in [104] veröffentlicht und ist durch den Umstand motiviert, dass parallele Programme auf einem abstrakten Niveau häufig identische Strukturen aufweisen [229, 256]. Die Grundidee der skelettalen Programmierung besteht darin, diese abstrakten Muster zu identifizieren und als vorgefertigte Bausteine zur Verfügung zu stellen. In Analogie zu den bei der sequenziellen Programmierung bekannten Entwurfsmustern [150] lassen sich Skelette demnach als typische parallele

Rechen- bzw. Kommunikationsmuster charakterisieren, die die Details der parallelen Verarbeitung kapseln und somit vor dem Benutzer verbergen. Auf diese Weise kann die Erstellung eines parallelen Programms auf einem höheren Abstraktionsniveau erfolgen, so dass die oben beschriebenen Probleme hinfällig sind. Um aus dem abstrakten Baustein eine konkrete Anwendung zu erzeugen, muss jedes Skelett, ähnlich einer Schablone, unter Verwendung mindestens einer benutzerdefinierten Argumentfunktion spezialisiert werden.[1] Dieser Sachverhalt beschreibt das zentrale Charakteristikum aller Skelette und hat maßgeblich zur Namensgebung beigetragen, da es sich bei einem Skelett im Wesentlichen um eine Art Grundgerüst handelt, mit dessen Hilfe eine Vielzahl konkreter Anwendungen implementiert werden kann.

Grundsätzlich ist es sinnvoll, eine Differenzierung algorithmischer Skelette in task- bzw. datenparallele Skelette vorzunehmen. Zwar können die meisten Problemstellungen häufig sowohl mit Hilfe task- als auch datenparalleler Skelette gelöst werden, die Arbeitsweise der Letztgenannten ist jedoch grundverschieden und wird im Folgenden erörtert:

- Die Verwendung taskparalleler Skelette [103, 203, 248, 249, 250, 251, 252, 253] bietet sich an, wenn eine Aufgabe in mehrere Teilaufgaben zerlegt werden kann, die anschließend unabhängig voneinander gelöst werden können. Zu diesem Zweck wird mit der Konstruktion einer Skelett-Topologie ein Abbild der Problemstellung modelliert, indem entsprechende taskparallele Skelette miteinander kombiniert werden. Welche Aufgaben die einzelnen Skelette hierbei übernehmen, wird durch die benutzerdefinierten Argumentfunktionen festgelegt. Da taskparallele Skelette nicht Gegenstand dieser Arbeit sind, sei für einen umfassenden Überblick auf [248] verwiesen.
- Die Verwendung datenparalleler Skelette [99, 100, 101, 102, 103, 203] bietet sich an, wenn die Problemstellung mit Hilfe einer Datenstruktur modelliert werden kann, deren Elemente parallel manipuliert werden (siehe Abschn. 3.4). Die Art der verwendeten Datenstruktur, z.B. Feld, (dünnbesetzte) Matrix, Liste etc., hängt dabei von der konkreten Problemstellung ab. Konzeptionell kann eine Differenzierung datenparalleler Skelette in Rechen-, Kommunikations- und Hybridskelette

[1] Bei Skeletten handelt es sich folglich um Funktionen höherer Ordnung (siehe Abschn. 3.2.3).

vorgenommen werden. Während Rechenskelette ausschließlich Berechnungen auf den Elementen der Datenstruktur vornehmen, ohne dabei zu kommunizieren, wie z.B. *map* (siehe Abschn. 3.4.3), tauschen Kommunikationsskelette ausschließlich Elemente der Datenstruktur aus, ohne dabei Berechnungen durchzuführen, wie z.B. *permute* (siehe Abschn. 3.4.4). Hybridskelette, wie z.B. *fold* (siehe Abschn. 3.4.2), besitzen beide Eigenschaften.

Anzumerken bleibt, dass grundsätzlich auch die Kombination von task- und datenparallelen Skeletten möglich ist [205]. Da die meisten Problemstellungen jedoch i.d.R. entweder unter Verwendung task- oder datenparalleler Skelette abgebildet werden können, tritt dieser Anwendungsfall vergleichsweise selten ein.

Um die Erstellung paralleler Programme unter Verwendung algorithmischer Skelette zu ermöglichen und damit so stark wie möglich zu vereinfachen, wurde die Münster Skelettbibliothek Muesli (engl. Muenster Skeleton Library, siehe Kap. 3) entwickelt [203]. Die Bibliothek implementiert sowohl task- als auch datenparallele Skelette, wobei letztere stets auf einer verteilten Datenstruktur (siehe Abschn. 3.3) operieren und daher in Form von Elementfunktionen zur Verfügung gestellt werden. Konzeptionell kann Muesli folglich als zusätzliche Abstraktionsschicht zwischen dem Benutzer und dem zu verwendenden Parallelrechner betrachtet werden. Durch die Kapselung der Details der parallelen Verarbeitung bietet die Verwendung der Skelettbibliothek im Vergleich zur konventionellen Erstellung eines parallelen Programms daher folgende Vorteile:

- Vermeiden typischer Probleme bei der parallelen Programmierung. Da Skelette sämtliche Details der parallelen Verarbeitung bereits vordefinieren, können die anfangs erwähnten Schwierigkeiten, wie z.B. Verklemmungen, eine zu geringe Speicherallokation oder ein fehlerhaft implementierter wechselseitiger Ausschluss, nicht auftreten. Infolgedessen kann eine etwaige Fehlersuche auf die benutzerdefinierten Argumentfunktionen beschränkt werden.

- Paralleles Programmieren auf einem hohen Abstraktionsniveau. Skelette ermöglichen eine adäquate Modellierung der jeweiligen Problemstellung auf einem hohen Abstraktionsniveau, was die Erstellung des

parallelen Programms erheblich vereinfacht. Auf diese Weise erfolgt gleichzeitig eine Strukturierung des Quelltextes, was dessen Wartbarkeit deutlich erhöht.

- Keine parallelen Programmierkenntnisse notwendig. Da sämtliche Details der parallelen Verarbeitung innerhalb der Skelette gekapselt sind, müssen Benutzer keine Erfahrung im Erstellen paralleler Programme besitzen. Auch das Verständnis der von Muesli benutzten parallelen Programmierschnittstellen (MPI und OpenMP, siehe Abschn. 2.3 und 2.4) ist nicht erforderlich, da deren Verwendung für den Benutzer vollkommen transparent ist.

- Beibehalten eines sequenziellen Programmierstils. Durch das von Muesli verwendete SPMD-Programmiermodell (siehe Abschn. 3.2.1) ist die Erstellung eines parallelen Programms nicht komplizierter als die Erstellung eines sequenziellen Programms. Oberflächlich betrachtet erstellt der Benutzer daher ein rein sequenzielles Programm, die parallele Verarbeitung wird jedoch bei der Ausführung intern von der Skelettbibliothek umgesetzt.

- Verwenden einer effizienten, objektorientierten Programmiersprache. Die Skelettbibliothek verwendet mit C++ eine bewährte und ausgereifte Programmiersprache, so dass für die Erstellung eines parallelen Programms keine neue/parallele Programmiersprache erlernt werden muss.

Die vorliegende Arbeit basiert auf dem in [203] beschriebenen Prototypen von Muesli. Dessen datenparalleler Bestandteil umfasst zwei verteilte Datenstrukturen für Felder und Matrizen, die jeweils verschiedene Skelette inklusive diverser Varianten implementieren. Hierbei ist zu beachten, dass die Skelettbibliothek ursprünglich für die Verwendung auf Parallelrechnern mit verteiltem Speicher konzipiert wurde, abweichende Rechner- bzw. Speicherarchitekturen wurden nicht direkt bzw. nur unter Inkaufnahme von Effizienzverlusten unterstützt. Neben der Weiterentwicklung des datenparallelen Bestandteils der Skelettbibliothek im Allgemeinen ist die Unterstützung von hybriden Speicherarchitekturen bzw. Mehrkernprozessoren im Speziellen eine der wichtigsten Zielsetzungen dieser Arbeit. Letztere werden im folgenden Abschnitt ausführlicher thematisiert.

1.2 Zielsetzungen

Die vorliegende Arbeit basiert, wie bereits erwähnt, auf der in [203] beschriebenen Skelettbibliothek und verfolgt grundsätzlich das Ziel, den datenparallelen Teil der Bibliothek weiterzuentwickeln. Diese allgemeine Zielsetzung umfasst folgende konkrete Teilaspekte:

- Entwicklung einer verteilten Datenstruktur für dünnbesetzte Matrizen. Die in [203] beschriebene Skelettbibliothek implementiert zwar bereits verteilte Datenstrukturen für Felder und Matrizen, diese profitieren jedoch nicht von einer etwaigen Dünnbesetztheit. Die bei der Neuentwicklung der Datenstruktur zu berücksichtigenden zentralen Kriterien sind Flexibilität, Erweiterbarkeit und Performanz, weiterhin muss die Datenstruktur verschieden datenparallele Skelette inklusive diverser Varianten implementieren.

- Unterstützung von Mehrkernprozessoren bzw. hybriden Speicherarchitekturen. Die in [203] beschriebene Skelettbibliothek skaliert faktisch nur auf solchen Parallelrechnern effizient, die einen verteilten Speicher aufweisen und deren Rechenknoten jeweils über einen Einkernprozessor verfügen. Aktuelle Parallelrechner verfügen jedoch vermehrt über Mehrkernprozessoren und eine Kombination aus verteiltem und gemeinsamem Speicher, d.h. über eine hybride Speicherarchitektur. Die effiziente Unterstützung solcher Architekturen soll jedoch optionalen Charakter haben, d.h. die Skelettbibliothek muss weiterhin auch auf Parallelrechnern ausführbar sein, die lediglich über einen verteilten Speicher und Einkernprozessoren verfügen.

- Implementierung praktischer Anwendungen und Durchführung von Laufzeitmessungen. Die Effizienz der implementierten Erweiterungen der Skelettbibliothek soll nach Möglichkeit durch praktische Anwendungen demonstriert werden. Hierunter fällt vorrangig die Untersuchung des Skalierungsverhaltens, d.h. die Messung von Speedup und Effizienz.

- Überarbeitung bewährter und Entwicklung neuer datenparalleler Skelette. Im Zusammenhang mit der Unterstützung von Mehrkernprozessoren bzw. hybriden Speicherarchitekturen müssen sämtliche bereits

implementierten datenparallelen Skelette der verteilten Datenstrukturen für Felder und Matrizen überarbeitet werden. Des Weiteren kann im Rahmen der Entwicklung praktischer Anwendungen die Implementierung neuer datenparalleler Skelette erforderlich sein, um den spezifischen Anforderungen der Anwendung stärker Rechnung zu tragen.

Anzumerken bleibt, dass die Umsetzung der oben formulierten Ziele über die einzelnen Kapitel der Arbeit verteilt thematisiert wird. Die sachlogische Struktur der Arbeit kann dem folgenden Abschnitt entnommen werden.

1.3 Aufbau der Arbeit

Im Folgenden wird kurz auf die Struktur der vorliegenden Arbeit eingegangen, um die einzelnen Kapitel in einen inhaltlichen und sachlogischen Zusammenhang zu stellen. Ein entsprechender Ordnungsrahmen kann Abbildung 1.1 entnommen werden.

- Kapitel 2 erläutert wesentliche Grundlagen, Konzepte und Technologien, die für das Verständnis der folgenden Kapitel notwendig sind. Neben einer Taxonomie zur Klassifikation von Parallelrechnern werden vor allem die beiden Programmierschnittstellen MPI und OpenMP behandelt. Letztere werden jedoch nur in Auszügen erörtert, d.h. es erfolgt eine Beschränkung auf die für den weiteren Verlauf der Arbeit relevanten Aspekte. Abschließend werden die Eigenschaften und Merkmale der wichtigsten aktuell verfügbaren Skelettbibliotheken kurz skizziert, um den Beitrag dieser Arbeit in den Kontext der Forschung einzuordnen.

- Aufbauend auf Kapitel 2 widmet sich Kapitel 3 der Beschreibung der Münster Skelettbibliothek Muesli. Neben den konzeptionellen Grundlagen werden vor allem die implementierten Erweiterungen detailliert erläutert. Der Fokus der Betrachtung liegt hierbei auf den datenparallelen Skeletten sowie den damit verbundenen verteilten Datenstrukturen. Die von Muesli ebenfalls zur Verfügung gestellten taskparallelen Skelette werden hingegen nicht behandelt, da diese nicht Gegenstand der vorliegenden Arbeit sind. Hier sei erneut auf [248] verwiesen.

- Kapitel 4 stellt mit einer verteilten Datenstruktur für dünnbesetzte Matrizen eine wesentliche Erweiterung der in Kapitel 3 beschriebenen Skelettbibliothek vor. Neben einer Erörterung der zu Grunde liegenden Konzepte werden vor allem die verwendeten Klassen sowie die Implementierung der datenparallelen Skelette ausführlich erläutert. Darüber hinaus werden die Ergebnisse einiger Laufzeitmessungen diskutiert, um die Effizienz der Implementierung zu demonstrieren.
- Kapitel 5 stellt mit dem LM OSEM-Algorithmus einen Anwendungsfall für die in Kapitel 3 beschriebene Skelettbibliothek vor. Der Algorithmus selbst wird im Rahmen der Positronen-Emissions-Tomographie, einem bildgebenden Verfahren der Nuklearmedizin, zur dreidimensionalen Bildrekonstruktion verwendet. Neben einer Betrachtung der theoretischen Grundlagen und Konzepte werden vor allem mehrere parallele Implementierungen des Algorithmus im Detail erläutert. Die Implementierungen wurden einerseits unter Verwendung von Muesli, andererseits unter direktem Einsatz diverser paralleler Programmierschnittstellen entwickelt, was einen Vergleich der verschiedenen Abstraktionsniveaus ermöglicht. Darüber hinaus werden die Ergebnisse einiger Laufzeitmessungen diskutiert, um die Effizienz der Implementierung zu demonstrieren und vergleichen zu können.
- Kapitel 6 stellt mit einem Algorithmus, der das parallele Training eines ART 2-Netzes ermöglicht, einen weiteren Anwendungsfall für die in Kapitel 3 beschriebene Skelettbibliothek vor. Neben einer Betrachtung der theoretischen Grundlagen künstlicher neuronaler Netze im Allgemeinen sowie der adaptiven Resonanztheorie bzw. ART 2-Netzen im Speziellen wird vor allem der entwickelte Algorithmus detailliert erläutert. Zusätzlich werden zwei unter Verwendung von Muesli vorgenommene Implementierungen erörtert, wobei die erste Datenparallelität und die zweite Taskparallelität ausnutzt. Darüber hinaus werden die Ergebnisse einiger Laufzeitmessungen diskutiert, um die Effizienz der Implementierung zu demonstrieren und vergleichen zu können.
- Kapitel 7 bildet den Schlussteil der Arbeit und schließt diese mit einem Gesamtfazit ab. Zudem erfolgt ein Ausblick auf den weiteren Forschungsbedarf zum einen bezüglich der skelettalen Programmierung im Allgemeinen, zum anderen bezüglich Muesli im Speziellen.

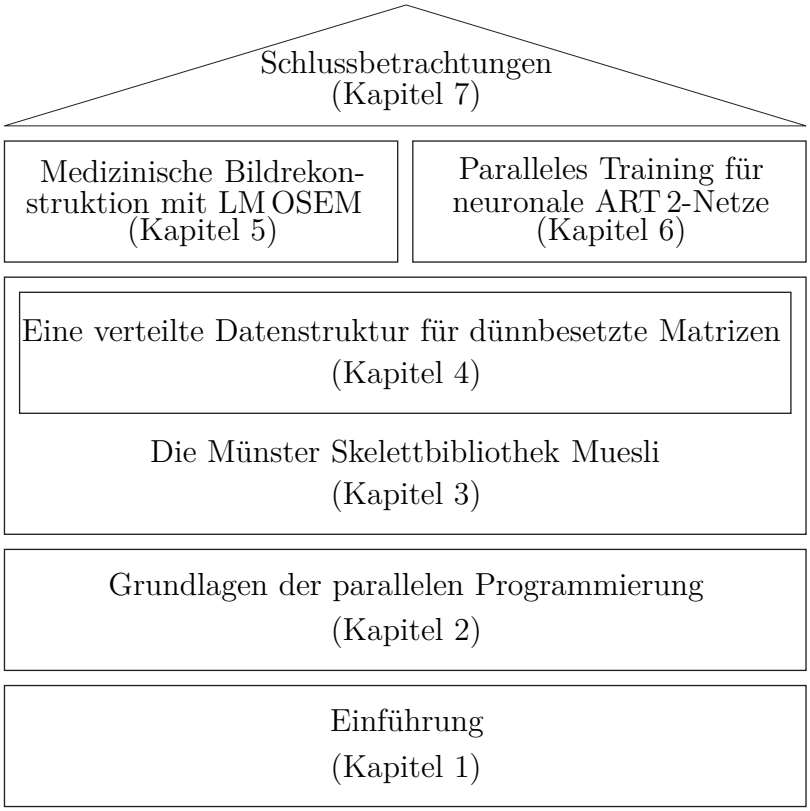


Abbildung 1.1: Ordnungsrahmen der Arbeit.

Kapitel 2

Grundlagen der parallelen Programmierung

Inhalt

2.1 Einleitung

In den folgenden Abschnitten 2.2 bis 2.6 werden grundlegende Konzepte und Technologien erörtert, die für das Verständnis der anschließenden Kapitel unabdingbar sind. Hierfür beschreibt Abschnitt 2.2 zunächst, welche Merkmale zur Klassifikation von Parallelrechnern herangezogen werden können und stellt mit der Rechner- bzw. der Speicherarchitektur zwei Kriterien vor, die wesentlichen Einfluss auf die zur Programmierung verwendeten Technologien haben. Letztere werden in den beiden folgenden Abschnitten beschrieben: Während Abschnitt 2.3 mit dem Message Passing Interface MPI eine

Schnittstelle für den Nachrichtenaustausch zwischen Prozessen erläutert, schildert Abschnitt 2.4 die zur Programmierung von Rechnern mit gemeinsamem Speicher entwickelte Open Multi-Processing Schnittstelle OpenMP. Anschließend geht Abschnitt 2.5 kurz auf den an der Universität Münster verfügbaren Parallelrechner ZIVHPC sowie die zur Laufzeitmessung verwendete Methodik ein. Abschließend wird in Abschnitt 2.6 eine Auswahl von Skelettbibliotheken anderer Forschungsgruppen vorgestellt, um den aktuellen Stand der Forschung zu skizzieren.

2.2 Klassifikation von Parallelrechnern

Der Begriff *Parallelrechner* wird von der einschlägigen Literatur weitgehend einheitlich verwendet, so dass sich diese Arbeit der folgenden Definition anschließt: Ein Parallelrechner ist ein Computer, der über mehrere Recheneinheiten (engl. processing unit, PU) verfügt und somit mehrere Instruktionen gleichzeitig, d.h. parallel, ausführen kann [255, S. 2] [257, S. 17]. Diese Begriffsbestimmung ist bewusst vage gehalten, um eine Vielzahl von unterschiedlichen Hardware-Architekturen zu berücksichtigen. Zu deren Entwicklung hat der Umstand beigetragen, dass die von einem Parallelrechner lösbaren Probleme häufig charakteristische Eigenschaften und Unterschiede aufweisen. Dies kann durch ein angepasstes Hardware-Design ausgenutzt werden, was i.d.R. eine effizientere Problemlösung ermöglicht. Trotz oder gerade wegen der Vielzahl an Unterschieden ist es daher sinnvoll, Parallelrechner in Klassen einzuteilen. Die bis heute geläufigste Einordnung findet mit Hilfe der nach ihrem Erfinder benannten *Flynn'schen Klassifikation* statt und wird in Abschnitt 2.2.1 vorgestellt. Anschließend wird in Abschnitt 2.2.2 mit der Speicherarchitektur ein weiteres wichtiges Unterscheidungsmerkmal von Parallelrechnern erläutert.

2.2.1 Rechnerarchitekturen

Die Flynn'sche Klassifikation wurde bereits im Jahr 1972 veröffentlicht, dient aber bis heute zur Einordnung von Rechnerarchitekturen, insbesondere denen von Parallelrechnern [147]. Je nachdem, wie viele Instruktions- und

Datenströme eine Rechnerarchitektur gleichzeitig verarbeiten kann, wird diese in eine der Klassen SISD, SIMD, MISD oder MIMD eingeteilt (siehe Tab. 2.1). Die folgenden Abschnitte erläutern die Semantik der jeweiligen Klasse.

Tabelle 2.1: Flynn'sche Klassifikation von Rechnerarchitekturen.

		Datenströme [#]	
		1	n
Instruktionsströme [#]	1	SISD	SIMD
	n	MISD	MIMD

2.2.1.1 Single instruction, single data

Rechnerarchitekturen dieser Klasse unterstützen lediglich einen Instruktions- bzw. Datenstrom, d.h. der Rechner kann zu jedem Zeitpunkt genau eine Instruktion auf genau einem Datum ausführen [255, S. 54]. SISD-Rechner, z.B. herkömmliche Einkernprozessor-Rechner, arbeiten rein sequenziell und weisen weder Instruktions- noch Datenparallelität auf (siehe Abb. 2.1a).

2.2.1.2 Single instruction, multiple data

Rechnerarchitekturen dieser Klasse unterstützen nur einen Instruktions-, aber mehrere Datenströme, d.h. der Rechner kann zu jedem Zeitpunkt zwar nur eine Instruktion, diese aber auf unterschiedlichen Daten gleichzeitig ausführen [255, S. 55]. SIMD-Rechner, z.B. Vektorrechner, werden unter anderem in der Bild- und Tonverarbeitung eingesetzt, da hier eine hochgradig parallele Verarbeitung der Daten möglich ist (siehe Abb. 2.1b).

2.2.1.3 Multiple instruction, single data

Rechnerarchitekturen dieser Klasse unterstützen mehrere Instruktions-, aber lediglich einen Datenstrom, d.h. der Rechner kann zu jedem Zeitpunkt zwar

unterschiedliche Instruktionen gleichzeitig, diese aber nur auf demselben Datum ausführen [255, S. 55 f.]. Die Klasse der MISD-Rechner ist umstritten, da zum einen der Nutzen eines solchen Rechners zweifelhaft erscheint und zum anderen nur wenige Rechner dieser Klasse zugeordnet werden können. Nichtsdestotrotz lassen sich fehlertolerante Systeme, die mehrfach redundant ausgelegt sind, um ein hohes Maß an Sicherheit zu gewährleisten, als MISD-Rechner klassifizieren, wie z.B. der Bordcomputer des NASA Space Shuttle [275] (siehe Abb. 2.1c).

2.2.1.4 Multiple instruction, multiple data

Rechnerarchitekturen dieser Klasse unterstützen mehrere Instruktions- bzw. Datenströme, d.h. der Rechner kann zu jedem Zeitpunkt gleichzeitig unterschiedliche Instruktionen auf unterschiedlichen Daten ausführen [255, S. 56]. Die Architektur von MIMD-Rechnern, z.B. von verteilten Systemen, wird derzeit beim Bau von Parallelrechnern am häufigsten verwendet, da diese, im Rahmen der Flynn'schen Klassifikation, am universellsten ist und alle anderen Rechnerarchitekturen emulieren kann (siehe Abb. 2.1d).

2.2.2 Speicherarchitekturen

Wie bereits erwähnt, verfügen die meisten modernen Parallelrechner über eine MIMD-Architektur, eine weitere Unterscheidung dieser Klasse ist mittels der Flynn'schen Taxonomie nicht vorgesehen. Neben Instruktions- und Datenströmen spielt für den Programmierer eines Parallelrechners jedoch ein weiteres Merkmal eine gravierende Rolle, nämlich die Speicherarchitektur. Diese beschreibt, wie der Hauptspeicher des Parallelrechners organisiert ist und auf welche Art und Weise die Recheneinheiten auf diesen zugreifen können. Da die Speicherarchitektur Auswirkungen auf die zur Programmierung eines Parallelrechners verwendeten Technologie(n) hat, z.B. MPI und/oder OpenMP, wird erstere als zusätzliches Unterscheidungsmerkmal zur Klassifikation von Parallelrechner vorgeschlagen und im weiteren Verlauf auch verwendet. Die folgenden Abschnitte 2.2.2.1 bis 2.2.2.3 behandeln die geläufigsten Speicherarchitekturen. Jede weist individuelle Vor- und Nachteile auf, so dass keine Architektur eine andere prinzipiell dominiert.

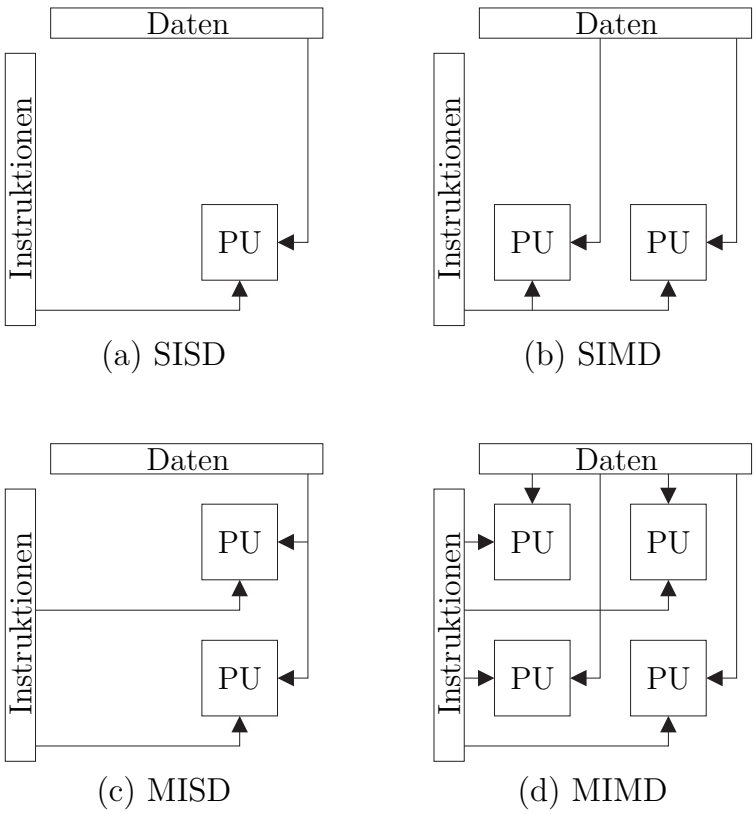
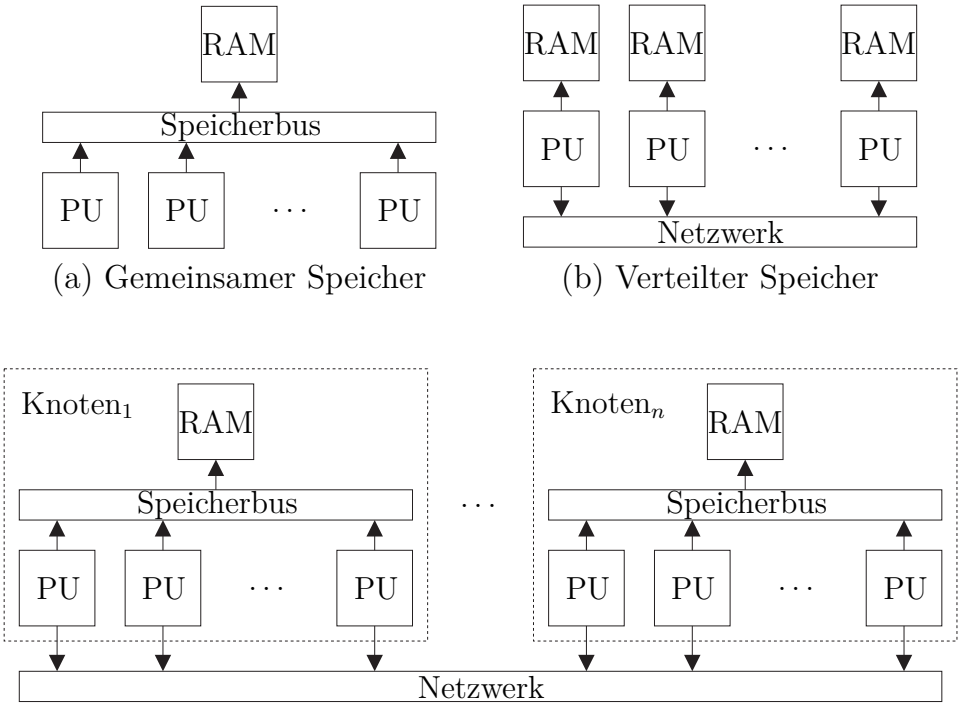


Abbildung 2.1: Hardware-Architekturen von Parallelrechnern.

2.2.2.1 Gemeinsamer Speicher

Parallelrechner mit gemeinsamem Speicher stellen einen singulären, einheitlichen Adressraum zur Verfügung, auf den alle Recheneinheiten gleichberechtigt zugreifen können (siehe Abb. 2.2a) [255, S. 43 ff.]. Die Vorteile dieser Architektur sind in der vergleichsweise einfachen Programmierung zu sehen, da der Adressraum als globale Einheit betrachtet werden kann. Weiterhin kann auf spezielle Hardware zur Abwicklung der Interprozess-Kommunikation verzichtet werden. Letztere kann über den gemeinsamen Speicher deutlich effizienter durchgeführt werden, als dies über ein separates Kommunikationsnetzwerk möglich wäre. Die Nachteile dieser Architektur sind vorrangig in der schlechten Skalierbarkeit zu sehen: Da sämtliche Recheneinheiten über den Speicher sowohl auf benötigte Daten zugreifen als auch kommunizieren, stellt die Speicheranbindung als knappe Ressource einen Flaschenhals dar. Weiterhin müssen bei einem gemeinsamen Speicher



(a) Gemeinsamer Speicher (b) Verteilter Speicher

(c) Hybrider Speicher

Abbildung 2.2: Speicherarchitekturen von Parallelrechnern.

Maßnahmen ergriffen werden, um die Caches der einzelnen Recheneinheit in einem kohärenten Zustand zu halten. Dies muss jedoch lediglich beim Design der Speicherarchitektur berücksichtigt werden, für die Programmierung eines entsprechenden Parallelrechners ist dieser Umstand irrelevant.

2.2.2.2 Verteilter Speicher

Bei Parallelrechnern mit verteiltem Speicher verfügt jede Recheneinheit exklusiv über einen eigenen Speicher, muss diesen also nicht mit anderen Recheneinheiten teilen (siehe Abb. 2.2b) [255, S. 45–49]. Im Gegensatz zu einer Architektur mit gemeinsamem Speicher ergibt sich daraus der Vorteil, dass die Speicheranbindung keinen Flaschenhals darstellt und somit auch nicht die Skalierbarkeit beschränkt. Weiterhin kann auf das Ergreifen von Maßnahmen zur Erhaltung eines kohärenten Zustands der jeweiligen Caches verzichtet werden. Die Nachteile dieser Architektur sind vorrangig in der Notwendigkeit zu sehen, spezielle Hardware für die Interprozess-

Kommunikation bereitzustellen, da die Recheneinheiten über keinen gemeinsamen Speicher zur Kommunikation verfügen. In diesem Fall stellt das Kommunikationsnetzwerk häufig den Flaschenhals dar und beschränkt die Skalierbarkeit. Insgesamt skaliert diese Architektur jedoch deutlich besser als eine Architektur mit gemeinsamem Speicher.

2.2.2.3 Hybrider Speicher

Jede der oben genannten Speicherarchitekturen weist für sich betrachtet ein Skalierungsproblem auf: Während bei einer Architektur mit gemeinsamem Speicher der Speicherbus den Flaschenhals darstellt, gilt dies bei einer Architektur mit verteiltem Speicher für das Kommunikationsnetzwerk. Diesen Einschränkungen begegnen aktuelle Parallelrechner durch die Verwendung einer hybriden Speicherarchitektur (siehe Abb. 2.2c) [255, S. 436 f.]. Die Recheneinheiten, in diesem Zusammenhang als sogenannte *Rechenkerne* (engl. cores) bezeichnet, werden zu logischen Knoten zusammengefasst. Innerhalb eines Knotens haben die Recheneinheiten Zugriff auf einen gemeinsamen Speicher. Für den Zugriff auf den verteilten Speicher, d.h. die Interprozess-Kommunikation, wird ein spezielles Netzwerk zur Verfügung gestellt. Der Vorteil der hybriden Speicherarchitektur liegt, verglichen mit den oben genannten Architekturen, in der deutlich besseren Skalierbarkeit. Der größte Nachteil ist sicherlich in der vergleichsweise komplexen Programmierung zu sehen, da entsprechende Software sowohl mit Technologien für gemeinsamen als auch verteilten Speicher implementiert werden muss, um maximale Effizienz zu erreichen.

2.3 Das Message Passing Interface MPI

Das Message Passing Interface MPI ist eine plattformunabhängige Programmierschnittstelle für die Programmiersprachen Fortran 77/90 und C/C++, die den Nachrichtenaustausch zwischen Prozessen ermöglicht [162, 273]. Seit der Veröffentlichung der ersten Version 1.0 im Mai 1994 hat sich MPI als de facto Standard für die Interprozess-Kommunikation auf Mehrprozessorsystemen etabliert, die aktuelle Version 2.2 [231] wurde im September 2009

verabschiedet. Bis zur Version 1.3 bot MPI im Wesentlichen eine Vielzahl von paarweisen und kollektiven Kommunikationsfunktionen sowie die Möglichkeit zur Erstellung von Prozesstopologien an. Mit der Version 2.0 wurde der Standard um Features erweitert, wie z.B. dynamische Prozessverwaltung [231, Kap. 10], einseitige Kommunikation [231, Kap. 11] und parallele Ein- und Ausgabe von Daten [231, Kap. 13].

Die folgenden Erläuterungen beziehen sich auf die Sprachanbindung zu C++. Zu beachten ist, dass die MPI-Bibliothek zunächst mit der Anweisung `#include "mpi.h"` eingebunden werden muss. Darüber hinaus muss durch den einmaligen Aufruf der Funktion `MPI_Init` die MPI-Umgebung initialisiert werden, bevor irgendeine andere MPI-Funktion aufgerufen werden kann [273, S. 291].[2] Bevor das Programm terminiert, sollte durch einen einmaligen Aufruf der Funktion `MPI_Finalize` die MPI-Umgebung ordnungsgemäß beendet werden. Anschließend sind keine weiteren Aufrufe von MPI-Funktionen möglich, auch nicht der von `MPI_Init` [273, S. 292].

Die folgenden Abschnitte 2.3.1 bis 2.3.3 behandeln zunächst grundlegende Konzepte bzw. Abstraktionsmechanismen von MPI, die für den Nachrichtenaustausch von zentraler Bedeutung sind (siehe Abschn. 2.3.1). Anschließend werden sowohl paarweise (siehe Abschn. 2.3.2) als auch kollektive Kommunikationsfunktionen (siehe Abschn. 2.3.3) vorgestellt, mit denen die eigentliche Kommunikation durchgeführt wird.

2.3.1 Konzepte

Die folgenden Abschnitte behandeln mit Prozessen (siehe Abschn. 2.3.1.1), Puffern (siehe Abschn. 2.3.1.2), Datentypen (siehe Abschn. 2.3.1.3), Gruppen (siehe Abschn. 2.3.1.4), Kommunikatoren (siehe Abschn. 2.3.1.5) sowie Fehlercodes und Fehlerklassen (siehe Abschn. 2.3.1.6) die wichtigsten Abstraktionsmechanismen von MPI und sind für das Verständnis der Schnittstelle unabdingbar.

[2] Die einzige Ausnahme von dieser Regel stellt die Funktion `MPI_Initialized` dar. Diese testet, ob die MPI-Umgebung bereits initialisiert wurde.

2.3.1.1 Prozesse

Wie bereits erwähnt, ermöglicht MPI den Nachrichtenaustausch zwischen mehreren Prozessen. Diese werden vom MPI-Laufzeitsystem erzeugt, deren Anzahl wird durch die Option `-np` beim Programmstart festgelegt.[3] So startet z.B der Aufruf `mpirun -np 4 ./foo` das Programm `foo`, welches parallel von vier Prozessen ausgeführt wird. Bemerkenswert ist hier, dass der MPI-Standard mit dem Konzept der Prozesse von der tatsächlichen Anzahl an verfügbaren Ressourcen, d.h. Prozessoren, abstrahiert [273, S. 8]. Das MPI-Laufzeitsystem startet immer genau $np \in \mathbb{N}$ Prozesse, die Zuteilung zu verfügbaren Prozessoren findet in einem zweiten Schritt statt. Offensichtlich ist hierbei jedoch eine 1:1-Zuteilung zwischen Prozessen und Prozessoren bzw. Prozessorkernen erstrebenswert, da nur so das maximale Parallelitätspotenzial ausgeschöpft werden kann. Jeder gestartete Prozess erhält vom MPI-Laufzeitsystem eine global eindeutige ganzzahlige Nummer (ID), den sogenannten *Rang* [273, S. 8]. Ränge beginnen bei 0 und sind fortlaufend nummeriert, d.h. bei np gestarteten Prozessen erhalten diese die Ränge 0 bis $np-1$. Der Rang eines Prozesses kann mit der Funktion `MPI_Comm_rank` bestimmt werden [273, S. 217]. Dieses Vorgehen ermöglicht die Verwendung beider bei der Programmierung von Mehrprozessorsystemen üblichen Programmiermodelle, SPMD (siehe Abschn. 3.2.1) und MPMD, da Prozesse, in Abhängigkeit vom eigenen Rang, unterschiedliche Programmteile ausführen können.

2.3.1.2 Puffer

Ein weiteres wichtiges Konzept für den Nachrichtenaustausch in MPI sind sogenannte *Puffer*. Ein Puffer ist im Wesentlichen ein zusammenhängender Speicherbereich, d.h. ein Feld, welches im Fall einer Sendeoperation die zu versendenden bzw. im Fall einer Empfangsoperation die erhaltenen Daten enthält. Um ein Höchstmaß an Flexibilität zu gewährleisten, sind sämtliche Puffer vom Typ `void*` [213, S. 154]. Auf diese Weise geht zwar die Typsicherheit verloren, was wiederum eine explizite Typkonvertierung erfordert,

[3] Wie bereits erwähnt, ermöglicht der MPI-Standard ab Version 2.0 Prozesse auch dynamisch zur Laufzeit zu starten und zu beenden.

andererseits können so beliebige benutzerdefinierte Typen ausgetauscht werden. Weiterhin ist zu beachten, dass in C++ die Größe eines Feld, anders als z.B. in der Programmiersprache Java [200, S. 102–106], nach dessen Erzeugung nicht mehr bestimmt werden kann. Für die Benutzung von MPI ergibt sich daraus die Notwendigkeit, dass jede Kommunikationsoperation neben dem Puffer `void* buf` mit `int cnt` die Anzahl der auszutauschenden Elemente sowie mit `MPI_Datatype typ` deren Datentyp und damit die Größe der Elemente als Parameter erwartet.

2.3.1.3 Datentypen

Wie bereits erwähnt, erwartet jede Kommunikationsoperation neben dem Puffer die Anzahl sowie den Datentyp der auszutauschenden Elemente als Argument (siehe Abschn. 2.3.1.2). Um letzteren zu spezifizieren, erscheint es naheliegend, einen in C++ definierten elementaren Datentyp als Argument zu übergeben. Dieses Vorgehen würde jedoch die Erstellung portabler Programme erschweren, da die Größe eines elementaren Datentyps in C++ von der konkreten Hardware abhängig ist, auf der das Programm kompiliert wurde [213, S. 58]. Stattdessen definiert der MPI-Standard selbst eine Reihe von Datentypen, die im Wesentlichen denen von C++ entsprechen, z.B. `MPI_INT` und `int`. Dieses Vorgehen stellt Portabilität bzw. Plattformunabhängigkeit sicher, da die Größe einer Nachricht aus der Anzahl der Elemente und deren Datentyp berechnet werden kann. Zusätzlich steht mit `MPI_BYTE` ein Datentyp zur Verfügung, der in Kombination mit dem in C++ definierten `sizeof`-Operator effektiv dazu benutzt werden kann, die Größe einer Nachricht in Bytes anzugeben. Ein weiterer Vorteil des hier beschriebenen Ansatzes besteht in der Möglichkeit, benutzerdefinierte Datentypen zu verwenden und so nicht auf die in C++ definierten Typen beschränkt zu sein. Eine vollständige Liste der vom MPI-Standard definierten Datentypen findet sich in [273, S. 19].

2.3.1.4 Gruppen

Gruppen dienen der Unterteilung der an einem Programm beteiligten Prozesse in logische Untermengen und werden zur Erzeugung von Kommuni-

katoren (siehe Abschn. 2.3.1.5) benötigt. Eine Gruppe ist formal als geordnete Menge von Prozessen definiert und wird durch ein Objekt vom Typ `MPI_Group` repräsentiert [273, S. 203]. Jeder Prozess einer Gruppe erhält einen bezüglich der Gruppe eindeutigen Rang, der dem bereits beschriebenen Nummerierungsschema folgt. Da ein Prozess zur selben Zeit mehreren Gruppen angehören kann, besitzt dieser in jeder Gruppe einen Rang, wobei sich in diesem Fall die Ränge bezüglich der verschiedenen Gruppen i.d.R. unterscheiden [273, S. 20]. Eine beispielhafte Unterteilung von Prozessen in Gruppen kann Abbildung 2.3 entnommen werden, die entsprechenden Ränge sind in Tabelle 2.2 aufgeführt.

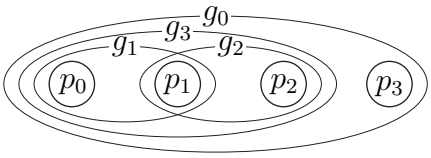


Abbildung 2.3: Beispielhafte Unterteilung von vier Prozessen in die Gruppen g_0 bis g_3.

Tabelle 2.2: Ränge der in Abbildung 2.3 definierten Gruppen.

		Rang bzgl.			
		g_0	g_1	g_2	g_3
Prozess	p_0	0	0	-	0
	p_1	1	1	0	1
	p_2	2	-	1	2
	p_3	3	-	-	-

Der MPI-Standard definiert eine Reihe von Funktionen, mit denen neue Gruppen erzeugt werden können. Zu beachten ist, dass eine neue Gruppe jedoch immer nur als Untermenge der als Parameter übergebenen Gruppe(n)

erzeugt werden kann und dass daran sämtliche Prozesse der ursprünglichen Gruppe(n) beteiligt sein müssen [273, S. 206]. Eine neue Gruppe kann aus der Vereinigung (`MPI_Group_union`), dem Schnitt (`MPI_Group_intersection`) oder der Differenz (`MPI_Group_difference`) von zwei bestehenden Gruppen gebildet werden. Steht für die Bildung der neuen Gruppe lediglich eine bereits bestehende Gruppe zur Verfügung, so können mit der Funktion `MPI_Group_incl` ausgewählte Prozesse der ursprünglichen Gruppe in die neue Gruppe aufgenommen werden. Mit der Funktion `MPI_Group_excl` werden ausgewählte Prozesse der ursprünglichen Gruppe von der neuen Gruppe ausgeschlossen. `MPI_Group_free` gibt eine erzeugte Gruppe wieder frei, d.h. zerstört diese. Listing 2.1 zeigt eine beispielhafte Anwendung, eine vollständige Übersicht über das Gruppenmanagement findet sich in [273, S. 207–216].

2.3.1.5 Kommunikatoren

In MPI erwarten sämtliche Kommunikationsfunktionen als Parameter einen Kommunikator. Dieser legt die sogenannte *Kommunikationsdomäne* fest, d.h. das Gebiet, innerhalb dessen Nachrichten zwischen den beteiligten Prozessen ausgetauscht werden sollen, und wird durch ein Objekt vom Typ `MPI_Comm` repräsentiert [273, S. 204–207]. M.a.W. isolieren Kommunikatoren den Nachrichtenaustausch zwischen beteiligten und unbeteiligten Prozessen, so dass verschiedene Gruppen ungestört parallel kommunizieren können. Hierbei ist wichtig, dass die Definition eines Kommunikators immer auf einer bereits bestehenden Gruppe basiert und, da letztere u.U. nicht kommunizieren muss, optional ist. Der MPI-Standard unterscheidet zwischen sogenannten *Intra-* und *Interkommunikatoren* [273, S. 203 f.]. Intrakommunikatoren dienen der kollektiven bzw. paarweisen Kommunikation innerhalb einer Gruppe, Interkommunikatoren hingegen können lediglich zur paarweisen Kommunikation zwischen zwei Gruppen verwendet werden. Im Folgenden wird lediglich die Funktionsweise der Intrakommunikatoren erläutert, da Interkommunikatoren bei datenparallelen Berechnungen keine Rolle spielen.

Wie bereits erwähnt, kann ein neuer Kommunikator lediglich auf Basis einer bereits bestehenden Gruppe erzeugt werden. Nach der Initialisierung

Listing 2.1: Quelltext zum Erstellen der in Abbildung 2.3 definierten Gruppen. Zusätzlich wird der Kommunikator c_3 für die Gruppe g_3 erstellt. g_0 muss nicht erstellt werden, da es sich hier um die zum Kommunikator MPI_COMM_WORLD gehörende Gruppe handelt.

```
1  MPI_Comm c3;                    // Kommunikator deklarieren
2  MPI_Group g0, g1, g2, g3;       // Gruppen deklarieren
3  int a[] = {0, 1};               // globale Ränge
4  int b[] = {0, 3};               // globale Ränge
5
6  MPI_Comm_group(MPI_COMM_WORLD, &g0);
7  MPI_Group_incl(g0, 2, a, &g1);
8  MPI_Group_excl(g0, 2, b, &g2);
9  MPI_Group_union(g1, g2, &g3);
10 MPI_Comm_create(MPI_COMM_WORLD, g3, &c3);
```

der MPI-Umgebung steht mit der Konstanten MPI_COMM_WORLD jedoch lediglich ein Kommunikator zur Verfügung, mit dem sämtliche Prozesse des Programms kommunizieren können, nicht aber die dazugehörige Gruppe. Abhilfe schafft hier die Funktion MPI_Comm_group, die zu einem bestehenden Kommunikator die entsprechende Gruppe liefert. Kommunikatoren können neu erzeugt (MPI_Comm_create), dupliziert (MPI_Comm_dup) oder aufgeteilt (MPI_Comm_split), mit MPI_Comm_free ein erzeugter Kommunikator wieder freigegeben werden. Eine vollständige Übersicht über das Kommunikatormanagement findet sich in [273, S. 216–223].

2.3.1.6 Fehlercodes und Fehlerklassen

Sämtliche Kommunikationsfunktionen geben einen Fehlercode vom Typ **int** zurück. Dieser signalisiert, ob während der Ausführung der Kommunikationsfunktion ein Fehler aufgetreten ist oder ob die Funktion ordnungsgemäß ausgeführt wurde. Zusätzlich kann mit der Funktion MPI_Error_string eine Fehlernachricht zu einem Fehlercode erzeugt werden. Um aufgetretene Fehler detaillierter beschreiben zu können, erlaubt der MPI-Standard implementierungsabhängige Fehlercodes. Diese können mit der Funktion MPI_Error_code auf vordefinierte Fehlerklassen abgebildet werden, damit Portabilität und Plattformunabhängigkeit gewährleistet bleiben. Eine voll-

ständige Liste der vom MPI-Standard definierten Fehlerklassen findet sich in [273, S. 299].

2.3.2 Paarweise Kommunikationsfunktionen

Paarweise Kommunikationsfunktionen [273, Kap. 2] ermöglichen den Datenaustausch zwischen genau zwei Prozessen. Dabei tritt ein Prozess als *Sender*, der andere Prozess als *Empfänger* auf, die auszutauschenden Daten werden als *Nachricht* bezeichnet. MPI stellt für diese Art der Kommunikation eine Vielzahl von verschiedenen Funktionen zur Verfügung, die sich in ihrer Semantik unterscheiden. Für deren Verständnis ist es sinnvoll, zunächst zwischen *blockierenden* und *nicht-blockierenden* Kommunikationsfunktionen zu differenzieren:

- Bei blockierenden Kommunikationsfunktionen kehrt die Programmkontrolle erst dann wieder zum aufrufenden Prozess zurück, wenn der für den Aufruf benötigte Sende- bzw. Empfangspuffer wiederverwendet werden kann [273, S. 32–35]. Für den Sender bedeutet dies, dass dieser den Sendepuffer überschreiben darf, für den Empfänger, dass die zu empfangende Nachricht vollständig in den Empfangspuffer kopiert wurde. Blockierende Kommunikationsfunktionen besitzen kein Präfix in der Bezeichnung, z.B. `MPI_Send`.
- Bei nicht-blockierenden Kommunikationsfunktionen kehrt die Programmkontrolle sofort zum aufrufenden Prozess zurück [273, S. 49 f.]. Für den Sender bedeutet dies, dass der für den Aufruf benötigte Puffer u.U. noch nicht wieder verwendet werden kann, für den Empfänger, dass die zu empfangende Nachricht u.U. noch nicht in den Puffer kopiert wurde. Um eine korrekte Funktionsweise zu gewährleisten, müssen nicht-blockierende Kommunikationsfunktionen immer in Kombination mit den Funktionen `MPI_Test` und/oder `MPI_Wait` benutzt werden [273, S. 52 ff.]. `MPI_Test` überprüft, ob die Kommunikationsoperation bereits abgeschlossen wurde. Alternativ kann mit `MPI_Wait` solange gewartet werden, bis die Kommunikationsoperation abgeschlossen wurde. Nicht-blockierende Kommunikationsfunktionen besitzen zur Unterscheidung das Präfix `I` (engl. immediate) in der Bezeichnung, z.B. `MPI_Isend`.

Parallelrechner besitzen i.d.R. spezielle Kommunikationshardware, die dediziert für den Nachrichtenaustausch zuständig ist. Dadurch kann die CPU entlastet werden, so dass Berechnungen und Kommunikationsoperationen parallel zueinander ausgeführt werden können. Dies erfordert jedoch, dass die zu versendenden Daten in einen speziellen Systempuffer kopiert werden. So kann beispielsweise die Programmkontrolle nach der blockierenden Sendeoperation `MPI_Send` bereits nach dem Kopieren der zu versenden Daten in den Systempuffer zurückkehren, statt darauf zu warten, dass der Empfänger die Nachricht erhalten hat. Um den Nachrichtenaustausch noch feingranularer steuern zu können, stellt MPI sowohl für die blockierenden als auch die nicht-blockierenden Kommunikationsfunktionen jeweils vier Modi zur Verfügung [273, S. 89 f.]:

- Standard (kein Präfix): Der Nachrichtenaustausch erfolgt, abhängig von der MPI-Implementierung, entweder synchron oder gepuffert. Häufig werden, da der Kopieraufwand gering ist, kleine Nachrichten gepuffert, große jedoch synchron übertragen. Der Schwellwert ist ebenfalls von der konkreten MPI-Implementierung abhängig.

- Synchron (engl. synchronous, Präfix S bzw. s): Der Sender wartet solange, bis der Empfänger die entsprechende Empfangsoperation gestartet hat.

- Gepuffert (engl. buffered, Präfix B bzw. b): Die zu versendende Nachricht wird, unabhängig von deren Größe, immer in einen Systempuffer kopiert.

- Bereit (engl. ready, Präfix R bzw. r): Die Sendeoperation darf nur gestartet werden, wenn die entsprechende Empfangsoperation bereits gestartet wurde.

Während MPI insgesamt acht verschiedene Sendeoperationen definiert, stehen für den Empfang einer Nachricht lediglich zwei Operationen zur Verfügung. Blockierende und nicht-blockierende Sende- und Empfangsoperationen können jedoch beliebig kombiniert werden [273, S. 50]. Einen Überblick über die vom MPI-Standard definierten paarweise Kommunikationsfunktionen sowie deren Semantik gibt Tabelle 2.3.

Da die Signaturen der verschiedenen Sende- und Empfangsoperationen kaum Unterschiede aufweisen, werden diese nicht im Einzelnen aufgeführt. Hierfür sei auf die MPI-Referenz verwiesen [273, S. 18, 22, 51 f., 90–93]. Stattdessen wird im Folgenden die Bedeutung der Parameter sowie deren Zusammenspiel erläutert:

- **void*** `buf`: Der für die Kommunikationsoperation bereitgestellte Puffer. Der Sender stellt in diesem die zu versendenden Daten bereit, der Empfänger einen Puffer, der nach Abschluss der Operation die empfangenen Daten enthält.
- **int** `cnt`: Die Anzahl der zu versendenden bzw. empfangenden Elemente.
- `MPI_Datatype typ`: Der Typ der zu versendenden bzw. empfangenden Elemente.
- **int** `src`: Der Rang des Senders. Dieser muss vom Empfänger angegeben werden.
- **int** `dst`: Der Rang des Empfängers. Dieser muss vom Sender angegeben werden.
- **int** `tag`: Das Tag der zu versendenden bzw. empfangenden Nachricht.
- `MPI_Comm com`: Der Kommunikator, der bei der Operation verwendet werden soll.
- `MPI_Status* sts`: Zeiger auf ein Status-Objekt, das Informationen über die abgeschlossene blockierende Empfangsoperation enthält.
- `MPI_Request* req`: Zeiger auf ein Request-Objekt, das Informationen über den Zustand der nicht-blockierenden Kommunikationsoperation enthält.

Zunächst wird das Zusammenspiel der Parameter `buf`, `cnt` und `typ` erläutert. Sei s die Größe von `buf` in Bytes. Der Sender verschickt die ersten $t = \texttt{cnt} \cdot \textbf{sizeof}\texttt{(typ)}$ Bytes aus `buf` an den Empfänger. Falls $t > s$, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen. Falls $t < s$, so werden die letzten $s - t$ Bytes aus `buf` nicht versendet. Im Regelfall sollte daher $s = t$ gelten. Der Empfänger wiederum gibt mit

t die maximale Größe der zu empfangenden Nachricht an. Sei u die Größe der empfangenen Nachricht. Falls $u > t$, wird eine Ausnahme vom Typ `MPI_ERR_TRUNCATE` ausgelöst und das Programm beendet. Falls $u \leq t$ und $s < u$, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen. Falls $u \leq t$ und $s > u$, werden die letzten $s - u$ Bytes aus `buf` nicht verändert. Im Regelfall sollte daher $u = t$ und damit $u = s$ gelten. Der Empfänger kann mit Hilfe von `sts` die tatsächliche Größe der empfangenen Nachricht abfragen.

Tabelle 2.3: Semantik der im MPI-Standard definierten paarweisen Kommunikationsfunktionen.

	blockierend	nicht-blockierend	synchron	gepuffert	bereit
`MPI_Send`	•				
`MPI_Recv`	•				
`MPI_Ssend`	•		•		
`MPI_Bsend`	•			•	
`MPI_Rsend`	•				•
`MPI_Isend`		•			
`MPI_Irecv`		•			
`MPI_Issend`		•	•		
`MPI_Ibsend`		•		•	
`MPI_Irsend`		•			•

Auch die Parameter `dst`, `src`, `tag` und `com` können nicht separat voneinander betrachtet werden. Der Sender gibt mit `dst` den Rang des Prozesses an, an den die Nachricht verschickt werden soll. Der Empfänger gibt mit `src` den Rang des Prozesses an, von dem eine Nachricht empfangen werden soll. Sowohl Sender als auch Empfänger müssen identische Tags und Kommunikatoren verwenden, anderenfalls wird der Empfänger die Nachricht nicht empfangen. Alternativ kann der Sender bzw. Empfänger auch das von MPI vordefinierte Tag `MPI_ANY_TAG` benutzen, das den Versand bzw. Empfang von Nachrichten unabhängig von ihrem Tag erlaubt. Tabelle 2.4 gibt einen

Überblick über die Argumentlisten der vom MPI-Standard definierten paarweisen Kommunikationsfunktionen.

Tabelle 2.4: Argumente der im MPI-Standard definierten paarweisen Kommunikationsfunktionen. Die Reihenfolge der Spalten entspricht dabei der Reihenfolge der Argumente in der Argumentliste der Funktion.

	buf	cnt	typ	src	dst	tag	com	sts	req
MPI_Send	•	•	•		•	•	•		
MPI_Recv	•	•	•	•		•	•	•	
MPI_Bsend	•	•	•		•	•	•		
MPI_Rsend	•	•	•		•	•	•		
MPI_Ssend	•	•	•		•	•	•		
MPI_Isend	•	•	•		•	•	•		•
MPI_Irecv	•	•	•	•		•	•		•
MPI_Ibsend	•	•	•		•	•	•		•
MPI_Irsend	•	•	•		•	•	•		•
MPI_Issend	•	•	•		•	•	•		•

2.3.3 Kollektive Kommunikationsfunktionen

Gegenüber paarweisen Kommunikationsoperationen ermöglichen kollektive Kommunikationsoperationen den Nachrichtenaustausch innerhalb einer Gruppe von Prozessen. Hierbei wird jedoch nicht zwischen Sender und Empfänger unterschieden, vielmehr tritt bei den meisten Operationen ein sogenannter *Wurzelprozess* als Quelle bzw. Senke des Nachrichtenaustauschs in Erscheinung [273, S. 147]. Zusätzlich definiert der MPI-Standard kollektive Kommunikationsfunktionen, bei denen die beteiligten Prozesse als gleichberechtigte Partner auftreten und daher keinen Wurzelprozess benötigen. Da kollektive Kommunikationsoperationen nicht zwischen Sender und Empfänger unterscheiden, existieren auch keine dedizierten Sende- bzw. Empfangsoperationen. Vielmehr müssen alle beteiligten Prozesse dieselbe Funk-

tion aufrufen, übergeben dabei aber evtl. unterschiedliche Parameter [273, S. 150]. Welche Prozesse an dem Nachrichtenaustausch beteiligt sein sollen, wird über den Intrakommunikator festgelegt, der von sämtlichen kollektiven Kommunikationsfunktionen als Argument erwartet wird [273, S. 151]. Dies ist von entscheidender Bedeutung, da der Wurzelprozess durch seinen Rang bezüglich der Gruppe bestimmt wird, die dem übergebenen Kommunikator zugeordnet wird.

Tabelle 2.5: Argumente der im MPI-Standard definierten kollektiven Kommunikationsfunktionen. Die Reihenfolge der Spalten entspricht dabei der Reihenfolge der Argumente in der Argumentliste der Funktion.

	sbuf	scnt	styp	rbuf	rcnt	rtyp	op	root	com
MPI_Allgather	•	•	•	•	•	•			•
MPI_Allreduce	•			•	•	•	•		•
MPI_Bcast	•	•	•					•	•
MPI_Gather	•	•	•	•	•	•		•	•
MPI_Reduce	•			•	•	•	•	•	•

Sämtliche kollektiven Kommunikationsfunktionen sind blockierend und werden im Standard-Modus ausgeführt, der MPI-Standard sieht keine nichtblockierenden Operationen bzw. keine weiteren Modi vor [273, S. 149]. Mit Ausnahme von `MPI_Bcast` existiert für jede Operation eine Variante, bei der jeder beteiligte Prozess eine unterschiedliche Anzahl an Elementen zum Nachrichtenaustausch beisteuern kann [273, S. 147]. Diese Varianten werden mit dem Suffix `v` (engl. vector) von ihren einfachen Pendants unterschieden. Im Folgenden werden die wichtigsten kollektiven Kommunikationsoperationen in ihrer jeweils einfachen Variante vorgestellt. Einen Überblick über die Parameterlisten der vom MPI-Standard definierten kollektiven Kommunikationsfunktionen gibt Tabelle 2.5, eine vollständige Übersicht findet sich in [273, Kap. 4]. Allen Beispielen ist gemein, dass jeweils drei Prozesse an der kollektiven Kommunikationsoperation beteiligt sind und jeder Prozess ein Element vom Typ **`int`** im Sendepuffer `buf` bzw. `sbuf` zur Verfügung stellt.

2.3.3.1 MPI_Bcast

Mit der Funktion `MPI_Bcast` kann ein Prozess dieselben Elemente an alle anderen Prozesse einer Gruppe senden [273, S. 152 f.]:

```
int MPI_Bcast(void* buf, int cnt, MPI_Datatype typ, int root,
              MPI_Comm com)
```

Der Wurzelprozess mit dem Rang `root` sendet `cnt` Elemente vom Typ `typ` an alle anderen Prozesse der Gruppe, die durch den Kommunikator `com` festgelegt ist. Die Daten werden vom Wurzelprozess im Puffer `buf` zur Verfügung gestellt, nach Abschluss der Operation besitzen alle beteiligten Prozesse dieselben Daten in `buf`. Zu beachten ist, dass kein separater Sende- und Empfangspuffer bereitgestellt werden muss, vielmehr ist ein einziger Puffer ausreichend, dessen Inhalt allerdings überschrieben wird. Die beispielhafte Durchführung einer broadcast-Operation kann Abbildung 2.4 entnommen werden.

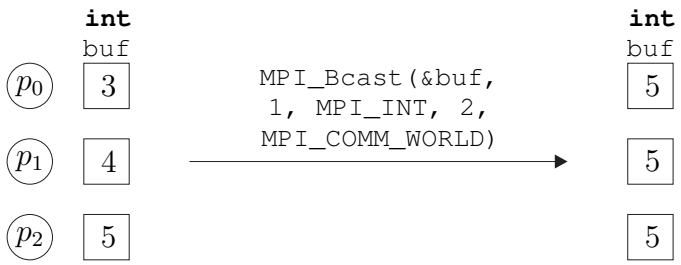


Abbildung 2.4: Durchführung einer broadcast-Operation mit drei Prozessen. p_2 ist der Wurzelprozess und sendet ein Element an alle anderen Prozesse.

2.3.3.2 MPI_Gather

Mit der Funktion `MPI_Gather` kann ein Prozess Elemente von allen anderen Prozessen einer Gruppe sammeln [273, S. 154 ff.]:

```
int MPI_Gather(void* sbuf, int scnt, MPI_Datatype styp,
               void* rbuf, int rcnt, MPI_Datatype rtyp,
               int root, MPI_Comm com)
```

Der Wurzelprozess mit dem Rang `root` empfängt von jedem anderen Prozess der Gruppe, die durch den Kommunikator `com` festgelegt ist, `rcnt` Elemente vom Typ `rtyp` und speichert diese im Puffer `rbuf`. Die zu versendenden Daten werden im Puffer `sbuf` zur Verfügung gestellt, jeder Prozess sendet `scnt` Element vom Typ `styp`. Zu beachten ist, dass auch der Wurzelprozess Daten zur Verfügung stellt und dass diese entsprechend den Rängen der beteiligten Prozesse aufsteigend im Empfangspuffer `rbuf` abgelegt werden. Die beispielhafte Durchführung einer gather-Operation kann Abbildung 2.5 entnommen werden.

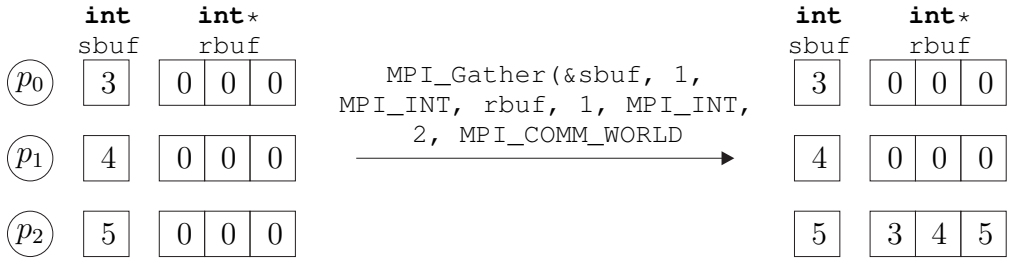


Abbildung 2.5: Durchführung einer gather-Operation mit drei Prozessen. p_2 ist der Wurzelprozess und speichert nach Abschluss der Operation drei Elemente in `rbuf`.

2.3.3.3 MPI_Allgather

Mit der Funktion `MPI_Allgather` können alle Prozesse einer Gruppe Elemente von allen anderen Prozessen der Gruppe sammeln [273, S. 170 f.]:

```
int MPI_Allgather(void* sbuf, int scnt, MPI_Datatype styp,
                  void* rbuf, int rcnt, MPI_Datatype rtyp,
                  MPI_Comm com)
```

Jeder Prozess der Gruppe, die durch den Kommunikator `com` festgelegt ist, stellt `scnt` Elemente vom Typ `styp` im Puffer `sbuf` zur Verfügung und versendet diese an alle anderen beteiligten Prozesse. Die zu empfangenden Elemente werden im Empfangspuffer `rbuf` entsprechend den Rängen der beteiligten Prozesse aufsteigend abgelegt, jeder Prozess empfängt `rcnt`

Elemente vom Typ `rtyp`. `MPI_Allgather` kann als Abfolge der Kommunikationsfunktionen `MPI_Gather` und `MPI_Bcast` betrachtet werden, wobei jeweils derselbe Wurzelprozess verwendet werden muss. Die beispielhafte Durchführung einer allgather-Operation kann Abbildung 2.6 entnommen werden.

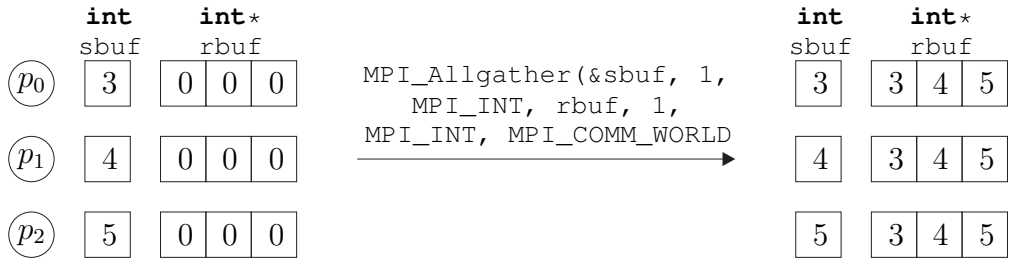


Abbildung 2.6: Durchführung einer allgather-Operation mit drei Prozessen. Jeder Prozess speichert nach Abschluss der Operation drei Elemente in `rbuf`.

2.3.3.4 MPI_Reduce

Mit der Funktion `MPI_Reduce` kann ein Prozess Elemente von allen anderen Prozessen einer Gruppe sammeln und diese mit einer binären Reduktionsoperation zu einem Element verdichten [273, S. 175–178]:

```
int MPI_Reduce(void* sbuf, void* rbuf, int cnt,
               MPI_Datatype typ, MPI_Op op,
               int root, MPI_Comm com)
```

Jeder Prozess der Gruppe, die durch den Kommunikator `com` festgelegt ist, stellt im Sendepuffer `sbuf` insgesamt `cnt` Elemente vom Typ `typ` bereit und versendet diese an den Prozess mit dem Rang `root`. Der Wurzelprozess benutzt anschließend die binäre Reduktionsoperation `op`, um die empfangenen Elemente sukzessive zu einem Wert zu verdichten. Dieser wird im Empfangspuffer `rbuf` gespeichert. Der MPI-Standard definiert eine Reihe von Reduktionsoperationen, z.B. `MPI_MAX` (Maximum) oder `MPI_SUM` (Summe). Eine vollständige Liste aller zur Verfügung gestellten Reduktionsoperationen findet sich in [273, S. 178]. Darüber hinaus besteht die Möglichkeit,

benutzerdefinierte Reduktionsoperationen zu verwenden [273, S. 189–195]. Die beispielhafte Durchführung einer reduce-Operation kann Abbildung 2.7 entnommen werden.

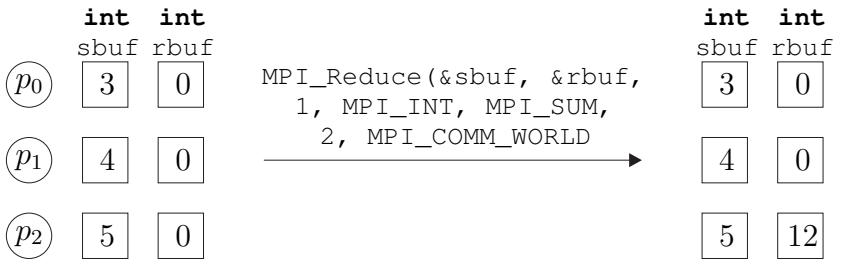


Abbildung 2.7: Durchführung einer reduce-Operation mit drei Prozessen. p_2 ist der Wurzelprozess und speichert nach Abschluss der Operation die Summe der Elemente in `rbuf`.

2.3.3.5 MPI_Allreduce

Mit der Funktion `MPI_Allreduce` können alle Prozess einer Gruppe Elemente von allen anderen Prozessen der Gruppe sammeln und diese mit einer binären Reduktionsoperation zu einem Element verdichten [273, S. 175–178]:

```
int MPI_Allreduce(void* sbuf, void* rbuf, int cnt,
                  MPI_Datatype typ, MPI_Op op, MPI_Comm com)
```

Jeder Prozess der Gruppe, die durch den Kommunikator `com` festgelegt ist, stellt im Sendepuffer `sbuf` insgesamt `cnt` Elemente vom Typ `typ` bereit

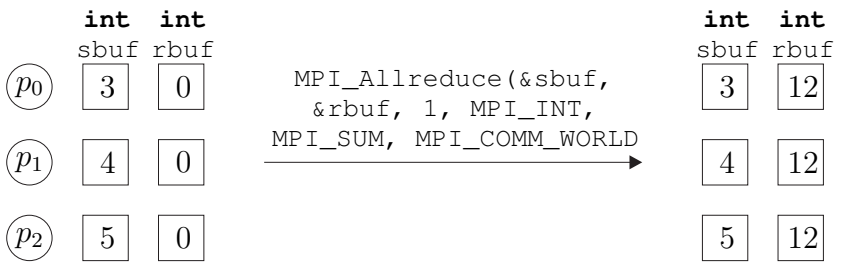


Abbildung 2.8: Durchführung einer allreduce-Operation mit drei Prozessen. Nach Abschluss der Operation speichert jeder Prozess die Summe der Elemente in `rbuf`.

und versendet diese an alle anderen Prozesse der Gruppe. Jeder Prozess benutzt anschließend die binäre Reduktionsoperation `op`, um die empfangenen Elemente sukzessive zu einem Wert zu verdichten. Dieser wird im Empfangspuffer `rbuf` gespeichert. `MPI_Allreduce` kann als Abfolge der Kommunikationsfunktionen `MPI_Reduce` und `MPI_Bcast` betrachtet werden, wobei jeweils derselbe Wurzelprozess verwendet werden muss. Die beispielhafte Durchführung einer allreduce-Operation kann Abbildung 2.8 entnommen werden.

2.4 Die Open Multi-Processing API OpenMP

Die Open Multi-Processing API OpenMP ist eine plattformunabhängige Programmierschnittstelle für die Programmiersprachen Fortran 77/90 und C/C++, die die Programmierung von Rechnern mit gemeinsamem Speicher ermöglicht [98, 179, 257]. Seit der Veröffentlichung der ersten Version für C/C++ im Oktober 1998 hat sich OpenMP als de facto Standard für die Programmierung von Mehrkernprozessoren etabliert, die aktuelle Version 3.0 [238] wurde im Mai 2008 verabschiedet. OpenMP besteht im Wesentlichen aus einer Menge von Direktiven und Klauseln [237, Kap. 2], Bibliotheksfunktionen [237, Kap. 3] und Umgebungsvariablen [237, Kap. 4]. Im Vordergrund steht insbesondere die Ausnutzung von Datenparallelität bei **`for`**-Schleifen, die keine Datenabhängigkeiten aufweisen, Taskparallelität wird aber ebenfalls unterstützt.

Die folgenden Erläuterungen beziehen sich auf die Sprachanbindung von OpenMP zu C++. Zu beachten ist, dass die OpenMP-Bibliothek zunächst mit der Präprozessor-Anweisung **`#include`** `"omp.h"` eingebunden werden muss. Weiterhin muss bei den gängigen Compilern eine Option eingeschaltet werden, um die OpenMP-Erweiterungen zu aktivieren (siehe Tab. 2.6). Mit dem Makro `_OPENMP` kann außerdem festgestellt werden, ob die OpenMP-Erweiterungen aktiviert wurden [237, S. 21]. Unterstützt ein Compiler die OpenMP-Erweiterungen nicht oder sind diese deaktiviert, ist auch das Makro nicht definiert. Dies kann zur bedingten Kompilierung genutzt werden.

Tabelle 2.6: Compileroptionen zur Aktivierung der OpenMP-Erweiterungen.

Hersteller	Version	Option	Quelle
GCC	4.4.2	-fopenmp	[282, S. 30]
IBM	XL C/C++ 10.1	-qsmp	[184, S. 222]
Intel	ICC 11.1	-openmp	[186, S. 519]
Microsoft	Visual Studio 2008	/openmp	[179, S. 6]
Sun	Studio 12, Update 1	-xopenmp	[285, S. 13]

Der folgende Abschnitt erläutert zunächst drei wesentliche Konzepte von OpenMP (siehe Abschn. 2.4.1). Danach werden in den Abschnitten 2.4.2 und 2.4.3 die wichtigsten Direktiven und Klauseln vorgestellt. Abschnitt 2.4.4 erörtert zentrale Bibliotheksfunktionen, bevor in Abschnitt 2.4.5 Hinweise für eine effiziente Parallelisierung mit OpenMP gegeben werden.

2.4.1 Konzepte

Die folgenden Abschnitte 2.4.1.1 bis 2.4.1.3 behandeln mit Threads (siehe Abschn. 2.4.1.1), der inkrementellen Parallelisierung (siehe Abschn. 2.4.1.2) sowie dem Fork-Join-Prinzip (siehe Abschn. 2.4.1.3) die wichtigsten Konzepte von OpenMP und sind für das Verständnis der Schnittstelle unabdingbar.

2.4.1.1 Threads

Im Gegensatz zu MPI findet bei OpenMP die Parallelisierung nicht auf Prozess-, sondern auf Thread-Ebene statt. Dieses Vorgehen bietet sich in einer Umgebung mit gemeinsamem Speicher an, da alle von einem Prozess erzeugten Threads gemeinsam auf dessen Ressourcen zugreifen können [280, S. 187]. Außerdem sind Threads leichtgewichtiger als Prozesse, d.h. Threads können schneller erzeugt bzw. zerstört werden als Prozesse [280, S. 188]. Die Thread-Verwaltung wird vollständig vom OpenMP-Laufzeitsystem übernommen, ist für den Benutzer also transparent. Dieser markiert durch das Einfügen von OpenMP-Direktiven lediglich Bereiche des Quelltextes, die parallel ausgeführt werden sollen. Dem Hersteller eines OpenMP-fähigen

Compilers ist es jedoch freigestellt, wie dieser Threads intern umsetzt, z.B. auf Basis von POSIX Threads (Pthreads) [80] oder sogar als vollwertige Prozesse [179, S. 24].

2.4.1.2 Inkrementelle Parallelisierung

Das grundlegende Vorgehensmodell bei der Erstellung eines parallelen Programms mit OpenMP ist das der sogenannten *inkrementellen Parallelisierung*. Dieses Konzept setzt immer eine sequenzielle Version des zu parallelisierenden Programms voraus und sieht vor, dass in den Quelltext sukzessive OpenMP-Direktiven eingefügt werden. Auf diese Weise wird das ursprüngliche Programm lediglich ergänzt, der Quelltext muss i.d.R. kaum umgestaltet werden, so dass die Wahrscheinlichkeit, neue Fehler einzubauen, relativ gering ist. Da es sich bei den eingefügten Direktiven um sogenannte *Pragmas*, d.h. Präprozessor-Anweisungen, handelt, werden diese von einem Compiler, der die OpenMP-Erweiterungen nicht implementiert, als Kommentare behandelt und somit ignoriert. Solange der ursprüngliche Quelltext lediglich mit OpenMP-Direktiven ergänzt wird, lässt sich dieser folglich auch mit einem Compiler übersetzen, der die OpenMP-Erweiterungen nicht implementiert bzw. bei dem die Compileroption zur Aktivierung der OpenMP-Erweiterungen nicht eingeschaltet wurde. Dieses Vorgehen ist z.B. bei Einkernprozessoren sinnvoll, um den durch das OpenMP-Laufzeitsystem erzeugten Mehraufwand zu vermeiden. Sobald jedoch Bibliotheksfunktionen, wie z.B. `omp_get_thread_num` (siehe Abschn. 2.4.4.2), benutzt werden, muss der Quelltext von einem OpenMP-fähigen Compiler mit eingeschalteter OpenMP-Option übersetzt werden. Eine Möglichkeit, diese Einschränkung zu umgehen, wird in Abschnitt 3.5.2 vorgestellt.

2.4.1.3 Fork-Join-Prinzip

Ein Programm mit OpenMP-Erweiterungen wird zunächst lediglich von einem Thread ausgeführt, dem sogenannten *Master* (siehe Abb. 2.9). Trifft der Master-Thread im Laufe der Programmausführung auf eine **`parallel`**-Direktive (siehe Abschn. 2.4.2.1), so erzeugt dieser ein sogenanntes *Thread-Team* (fork) [98, S. 23 f.]. Dieses Team besteht aus insgesamt *nt* Threads,

wobei der Master-Thread ebenfalls Bestandteil des Teams ist, so dass dieser lediglich $nt-1$ zusätzliche Slave-Threads erzeugen muss. Sämtliche Threads des Teams führen den folgenden Codeblock[4] anschließend parallel aus. Falls dieser keine Direktiven zur Arbeitsaufteilung beinhaltet, wird der Codeblock von dem Thread-Team parallel, d.h. redundant, ausgeführt. Anderenfalls wird die Arbeit zwischen den Threads entsprechend der verwendeten Direktive(n) aufgeteilt. Am Ende des parallelen Bereichs zerstört der Master-Thread das vorher erzeugte Thread-Team (join) und führt die Abarbeitung des Programms allein fort.

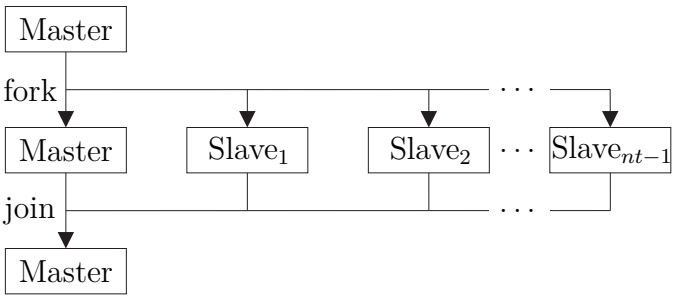


Abbildung 2.9: Erzeugen und Zerstören eines Thread-Teams nach dem Fork-Join-Prinzip.

2.4.2 Direktiven

Wie bereits erwähnt, erfordert das Vorgehensmodell der inkrementellen Parallelisierung, den Quelltext eines sequenziellen Programms sukzessive um OpenMP-Direktiven zu ergänzen. Diese bestehen immer aus einem Namen sowie optionalen Klauseln (siehe Abschn. 2.4.3) und müssen folgende Syntax aufweisen, um korrekt verarbeitet zu werden:

`#pragma omp` $Direktive\ [Klausel[,\ Klausel]^*]$

[4] Ein Codeblock wird, analog zu C/C++, von geschweiften Klammern umschlossen, d.h. { und }. Zu beachten ist, dass die öffnende Klammer nicht in derselben Zeile wie das Pragma stehen darf.

Anzumerken bleibt, dass das Auffinden von fehlerhaften Direktiven und Klauseln ein schwieriges Unterfangen ist, da diese vom Präprozessor ignoriert werden. Die folgenden Abschnitte 2.4.2.1 bis 2.4.2.3 erläutern die Funktionsweise der zentralen Direktiven **`parallel`** (siehe Abschn. 2.4.2.1), **`for`** (siehe Abschn. 2.4.2.2) und **`critical`** (siehe Abschn. 2.4.2.3).

2.4.2.1 parallel

Die **`parallel`**-Direktive ist das grundlegende Parallelisierungskonstrukt von OpenMP, da mit dieser Bereiche des Quelltextes markiert werden, die parallel von einem Thread-Team ausgeführt werden sollen [237, S. 26–29]. Sämtliche andere Direktiven müssen von einer **`parallel`**-Direktive umschlossen sein, um korrekt ausgeführt zu werden. Jeder Thread, der während der Ausführung des Programms auf eine **`parallel`**-Direktive trifft, erzeugt nach dem Fork-Join-Prinzip (siehe Abschn. 2.4.1.3) ein neues Thread-Team. Innerhalb eines parallelen Bereichs dürfen weitere **`parallel`**-Direktiven verwendet werden, d.h. diese dürfen beliebig geschachtelt werden [98, S. 111 ff.].[5] Trifft ein Slave-Thread auf eine **`parallel`**-Direktive, so wird dieser der Master-Thread des neu erzeugten Thread-Teams. Anzumerken bleibt, dass am Ende jedes parallelen Bereichs eine implizite Synchronisationsbarriere eingebaut ist, d.h. der Master-Thread zerstört das Thread-Team erst dann, wenn alle anderen Threads des Teams ihre Arbeit beendet haben.

2.4.2.2 for

Die **`for`**-Direktive ist das grundlegende Konstrukt zur Ausnutzung von Datenparallelität und dient der Parallelisierung von **`for`**-Schleifen [237, S. 33–38]. Trifft ein Thread-Team auf eine solche Direktive, werden die Schleifeniterationen auf die einzelnen Threads verteilt, d.h. jeder Thread durchläuft nur einen Teil des gesamten Iterationsraums. Die Strategie zur Aufteilung des Iterationsraums wird mit der optionalen **`schedule`**-Klausel festgelegt

[5] I.d.R. ist dieses Vorgehen jedoch nicht sinnvoll. Da der OpenMP-Standard die Umsetzung dieses Features nicht erzwingt, erzeugen viele Compiler bei verschachtelten Bereichen ein Thread-Team, das lediglich aus dem Thread besteht, der auf die Direktive getroffen ist.

(siehe Abschn. 2.4.3). Ist die Klausel nicht spezifiziert, bestimmt die interne Kontrollvariable `def-sched-var` die Aufteilungsstrategie [237, S. 38 f.]. Zu beachten ist, dass die Benutzung der **`for`**-Direktive einer Reihe von Einschränkungen unterliegt:

- Die **`for`**-Direktive kann ausschließlich zur Parallelisierung von **`for`**-Schleifen verwendet werden. Andere Schleifen, wie z.B. **`while`** oder **`do-while`**, können nicht parallelisiert werden, es existiert auch keine andere Direktive, die diese Möglichkeit bietet. Der einzige Weg, diese Einschränkung zu umgehen, besteht in der Umformulierung der Schleife, d.h. jede (**`do`**-)**`while`**-Schleife muss, falls diese parallelisiert werden soll, in eine **`for`**-Schleife umgewandelt werden. Dieses Vorgehen birgt natürlich das Risiko, beim nachträglichen Parallelisieren Fehler in das Programm einzubauen und zeigt, dass das Konzept der inkrementellen Parallelisierung unter gewissen Umständen nicht angewendet werden kann.

- Der Kopf der zu parallelisierenden **`for`**-Schleife muss in sogenannter *kanonischer* Form formuliert sein. Die daraus resultierenden Einschränkungen betreffen die Initialisierung und das Inkrement/Dekrement der Schleifenvariable sowie die Schleifenbedingung. Die Restriktionen werden hier nicht im Einzelnen aufgeführt, da diese vergleichsweise umfangreich sind. Für eine vollständige Auflistung sei auf die OpenMP API verwiesen [237, S. 34]. Der Zweck dieser Einschränkungen besteht darin, dem OpenMP-Laufzeitsystem die Berechnung der Anzahl der Schleifeniterationen im Voraus zu ermöglichen, was wiederum die Aufteilung des Iterationsraums erheblich vereinfacht bzw. erst sinnvoll ermöglicht.

- Aus der Anforderung, den Schleifenkopf in kanonischer Form zu formulieren, resultiert die Einschränkung, dass die **`for`**-Direktive nicht zum Traversieren von Container-Objekten der C++-Standardbibliothek mit Hilfe der Iterator-Schnittstelle benutzt werden kann. Eine Möglichkeit, diese Einschränkung effektiv zu umgehen, wird in Abschnitt 4.3.4.2 vorgestellt. An dieser Stelle sei vorweggenommen, dass auch hier der Kopf der **`for`**-Schleife umformuliert werden muss.

Anzumerken bleibt, dass mit der **`parallel for`**-Direktive eine Kurzschreibweise für den Fall existiert, dass lediglich eine einzelne **`for`**-Schleife parallelisiert wird [237, S. 47 f.]. Die oben beschriebenen Eigenschaften und Restriktionen bleiben jedoch erhalten.

2.4.2.3 critical

Die **`critical`**-Direktive dient der Synchronisation eines Thread-Teams und stellt sicher, dass der assoziierte Codeblock, der sogenannte *kritische Abschnitt*, zu jedem Zeitpunkt von maximal einem Thread ausgeführt wird [237, S. 52 ff.]. Kritische Abschnitte ermöglichen folglich die Parallelisierung von Codeabschnitten, die Datenabhängigkeiten und/oder Wettlaufsituationen (engl. race condition) aufweisen. Eine Datenabhängigkeit liegt vor, wenn das Ergebnis einer Berechnung vom Ergebnis einer vorherigen Berechnung abhängt [179, S. 74]. Eine Wettlaufsituation liegt vor, wenn mehrere Threads parallel auf demselben Datum arbeiten und das Ergebnis der Berechnung von der zeitlichen Ausführungsreihenfolge der Instruktionen abhängig ist [179, S. 67]. Dies gilt insbesondere für die parallele Ausführung von Funktionen, die Seiteneffekte aufweisen.[6]

Der **`critical`**-Direktive kann optional ein Bezeichner zugeordnet werden. Auf diese Möglichkeit sollte grundsätzlich zurückgegriffen werde, da das OpenMP-Laufzeitsystem den Zugriff auf kritische Abschnitte mit *demselben Bezeichner* synchronisiert, wobei kritische Abschnitte ohne Bezeichner per Definition denselben unspezifizierten Bezeichner besitzen [237, S. 53]. Dies hat zur Folge, dass zwei Thread-Teams, die parallel unterschiedliche Teile eines Programms ausführen, miteinander synchronisiert werden, wenn in beiden Programmabschnitten eine **`critical`**-Direktive mit identischem Bezeichner vorkommt. Wird hingegen kein Bezeichner vergeben, besteht die Gefahr, beim Einfügen einer neuen **`critical`**-Direktive nach dem Prinzip der inkrementellen Parallelisierung (siehe Abschn. 2.4.1.2) den Bezeichner versehentlich wegzulassen, was u.U. eine ungewollte Synchronisation nach

[6] In C++ können Funktionen, die keine Seiteneffekte aufweisen, mit dem Schlüsselwort **`const`** deklariert werden [213, S. 513]. Im Umkehrschluss bedeutet dies, dass, eine saubere Implementierung vorausgesetzt, Funktionen, die nicht mit dem Schlüsselwort **`const`** deklariert sind, Seiteneffekte erzeugen.

sich zieht. Anzumerken bleibt, dass der Verwaltungsaufwand bei der Verwendung der **critical**-Direktive durch die Verwendung der **atomic**-Direktive reduziert werden kann [237, S. 55–58]. Letztere dient zwar ebenfalls der Synchronisation eines Thread-Teams, ermöglicht dies aber, statt für einen vollständigen Codeblock, lediglich für eine singuläre atomare Anweisungen der Form `x++`, `++x`, `x--`, `--x` und `x op= e`, wobei `x` und `e` skalare Typen sein müssen und `op` $\in \{$`+`, `-`, `*`, `/`, `&`, `|`, `^`, `<<`, `>>`$\}$.

2.4.3 Klauseln

Wie bereits erwähnt, können Direktiven mit optionalen Klauseln versehen werden, wobei jede Klausel nur mit bestimmten Direktiven kombiniert werden kann. Die folgenden Abschnitte 2.4.3.1 bis 2.4.3.3 erläutern die Funktionsweise der zentralen Klauseln **private** (siehe Abschn. 2.4.3.1), **reduction** (siehe Abschn. 2.4.3.2) und **schedule** (siehe Abschn. 2.4.3.3).

2.4.3.1 private

Die **private**-Klausel kann mit der **parallel**-, der **for**- sowie der **parallel for**-Direktive verwendet werden und weist folgende Syntax auf [237, S. 26, 33 und 47]:

$$\textbf{private}(var[,\ var]^*)$$

Die Klausel erwartet eine Liste von Variablen als Argument und bewirkt, dass jeder Thread des Thread-Teams eine private, d.h. eigene Kopie der Variable(n) erhält [237, S. 19 und 73 ff.]. Bei einem Thread-Team mit nt Threads existieren folglich jeweils nt Kopien der Variable(n) im Speicher. Wird die Klausel weggelassen, greifen sämtliche Threads auf dieselbe Variable zu, d.h. diese wird von allen Threads gemeinsam genutzt [237, S. 65]. Jede Variable in der Liste wird so initialisiert, als ob diese lediglich lokal deklariert wurde [237, S. 73]. Dies bedeutet insbesondere, dass Variablen vom Typ Zeiger auf keine gültigen Adressen verweisen und Klassen einen Standardkonstruktor zur Verfügung stellen müssen, falls entsprechende Objekte

in der Argumentliste verwendet werden. Soll eine Variable hingegen mit dem Wert initialisiert werden, den diese vor der Ausführung der entsprechenden Direktive hatte, muss die **firstprivate**-Klausel verwendet werden [237, S. 75 ff.]. Anzumerken bleibt, dass Indexvariablen von **for**-Schleifen stets privat sind [237, S. 64].

2.4.3.2 reduction

Die **reduction**-Klausel kann mit der **parallel**-, der **for**- sowie der **parallel for**-Direktive verwendet werden und weist folgende Syntax auf [237, S. 27, 33 und 47]:

$$\texttt{reduction}(op\text{: } var[,\ var]^*)$$

Die Klausel erwartet einen Reduktionsoperator $op \in \{+, -, *, \hat{}, \&, |, \&\&, ||\}$ sowie eine Liste von Variablen als Argument und bewirkt zunächst, dass jeder Thread eine private Kopie der Variablen erhält [237, S. 79–83]. Für jede Variable werden, nachdem das Thread-Team die entsprechende Direktive abgearbeitet hat, die Werte der privaten Variablen mit Hilfe der Reduktionsoperation *op* zu einem Wert verdichtet und anschließend der ursprünglichen Variable zugewiesen. Anzumerken bleibt, dass die Mächtigkeit der **reduction**-Klausel insofern beschränkt ist, als dass der Reduktionsoperator *op* keine benutzerdefinierten Funktionen als Argument akzeptiert.

2.4.3.3 schedule

Die **schedule**-Klausel kann sowohl mit der **for**- als auch der **parallel for**-Direktive verwendet werden und weist folgende Syntax auf [237, S. 33 und 47]:

$$\texttt{schedule}(type[, size])$$

Die Klausel legt fest, wie der Iterationsraum der **`for`**-Schleife unter den Threads aufgeteilt wird. Falls keine **`schedule`**-Klausel spezifiziert ist, wird der Iterationsraum in Abhängigkeit von der internen Kontrollvariable `def-sched-var` aufgeteilt [237, S. 38 f.]. Der Parameter *size* muss, falls angegeben, zu einer natürlichen Zahl auswertbar sein, d.h. $size \in \mathbb{N}$. Für den Parameter *type* stehen folgende Aufteilungsstrategien zur Verfügung [237, S. 36 f.]:

`static` legt fest, dass der Iterationsraum in Blöcke der Größe *size* aufgeteilt wird und diese im Voraus, d.h. vor der Ausführung der Schleife, reihum den einzelnen Threads zugewiesen werden. Die Verteilung erfolgt aufsteigend anhand der Thread-Nummer, d.h. Thread t_0 erhält den ersten Block. Wird *size* nicht spezifiziert, wird der Iterationsraum derart aufgeteilt, dass die einzelnen Blöcke in etwa dieselbe Größe aufweisen und dass jeder Thread maximal einen Block zugeteilt bekommt, die genaue Blockgröße ist von der jeweiligen OpenMP-Implementierung abhängig.

`dynamic` legt fest, dass der Iterationsraum in Blöcke der Größe *size* aufgeteilt wird und diese nicht im Voraus zugewiesen, sondern zur Laufzeit angefordert werden. Hat ein Thread einen Block abgearbeitet, fordert dieser einen weiteren Block an, bis keine weiteren Blöcke verteilt werden müssen. Wird *size* nicht spezifiziert, wird der Iterationsraum in Blöcke der Größe 1 aufgeteilt.

`guided` legt fest, dass der Iterationsraum in Blöcke aufgeteilt wird und diese nicht im Voraus zugewiesen, sondern zur Laufzeit angefordert werden. Die Größe eines angeforderten Blocks ist jedoch nicht fix, sondern proportional zur Anzahl der noch nicht zugewiesenen Iterationen dividiert durch die Anzahl an Threads *nt*. Falls $size > 1$, umfasst ein Block, bis auf den letzten, mindestens *size* Iterationen. Wird *size* nicht spezifiziert, gilt $size = 1$.

`auto` legt fest, dass der Iterationsraum in Abhängigkeit vom Compiler und/oder OpenMP-Laufzeitsystem aufgeteilt wird.

`runtime` legt fest, dass der Iterationsraum in Abhängigkeit von der internen Kontrollvariable `run-sched-var` aufgeteilt wird.

2.4.4 Laufzeitbibliothek

OpenMP deklariert eine Reihe von Bibliotheksfunktionen, mit denen der Zustand der OpenMP-Laufzeitumgebung manipuliert bzw. abgefragt werden kann, um so das Verhalten des parallelen Programms dynamisch zu beeinflussen. Die folgenden Abschnitte 2.4.4.1 bis 2.4.4.3 erläutern die Semantik der zentralen Funktionen `omp_get_max_threads` (siehe Abschn. 2.4.4.1), `omp_get_thread_num` (siehe Abschn. 2.4.4.2) und `omp_set_num_threads` (siehe Abschn. 2.4.4.3).

2.4.4.1 omp_get_max_threads

Die Funktion **`int`** `omp_get_max_threads()` gibt die maximale Anzahl an Threads zurück, die einen parallelen Bereich ausführen [237, S. 194 f.]. Falls keine parallelen Bereiche geschachtelt werden, gibt die Funktion folglich die Größe des Thread-Teams zurück.

2.4.4.2 omp_get_thread_num

Die Funktion **`int`** `omp_get_thread_num()` liefert die Thread-Nummer *tid* bezüglich des Thread-Teams zurück, das den umgebenden parallelen Bereich ausführt [237, S. 95 f.]. Falls die Funktion außerhalb eines parallelen Bereichs aufgerufen wird, ist der Rückgabewert 0. Die Thread-Nummer liegt stets zwischen 0 und `omp_get_max_threads` − 1. Anzumerken bleibt, dass der Master-Thread immer die Nummer 0 zugewiesen bekommt.

2.4.4.3 omp_set_num_threads

Die Funktion **`void`** `omp_set_num_threads(`**`int`**` nt)` beeinflusst die Anzahl zu erzeugender Threads in sämtlichen nachfolgenden parallelen Bereichen, die die `num_threads`-Klausel nicht spezifizieren [237, S. 93]. Wie bereits erwähnt, hängt die tatsächliche Anzahl zu erzeugender Threads von einer Reihe weiterer Faktoren ab, so dass auch der Aufruf dieser Funktion eher als Anfrage denn als strikte Forderung interpretiert werden muss

(siehe Abschn. 2.4.3). Faktisch setzt die Funktion lediglich den Wert der internen Kontrollvariablen `nthreads-var` auf den übergebenen Wert `nt`, wobei `nt` $\in \mathbb{N}$. Derselbe Effekt kann durch das Setzen der Umgebungsvariablen `OMP_NUM_THREADS` erreicht werden. Alternativ kann, falls die Anzahl zu erzeugender Threads lediglich für einen bestimmten parallelen Bereich festgelegt werden soll, die `num_threads`-Klausel spezifiziert werden.

2.4.5 Effiziente Parallelisierung

Bei der Verwendung der in den vorherigen Abschnitten erläuterten Direktiven, Klauseln und Bibliotheksfunktionen können wiederkehrende Muster identifiziert werden, die eine effektive und effiziente Parallelisierung ermöglichen. Die folgende Auflistung stellt einen Leitfaden dar, der, ohne Anspruch auf Vollständigkeit zu erheben, bei der Programmierung mit OpenMP von praktischer Relevanz ist und auf Erfahrungen basiert, die bei der Implementierung der datenparallelen Skelette gesammelt wurden (siehe Abschn. 4.3.5):

- **`for`**-Schleifen, die mit Hilfe der Iterator-Schnittstelle der STL über die Elemente eines Containerobjekts iterieren, können nicht ohne weiteres parallelisiert werden, da OpenMP lediglich ganzzahlige, vorzeichenbehaftete Typen für den Schleifenindex erlaubt.[7] Diese Einschränkung kann umgangen werden, indem sämtliche Adressen der in dem Container enthaltenen Objekte in ein temporäres Feld kopiert werden. Anschließend muss der Kopf der **`for`**-Schleife umgeschrieben werden, so dass über das temporäre Feld iteriert wird. Dieser Ansatz verursacht zwar einen gewissen Mehraufwand, letzterer sollte sich jedoch durch die parallele Ausführung der **`for`**-Schleife amortisieren (siehe Abschn. 4.3.4.2).
- Bei der Parallelisierung von geschachtelten **`for`**-Schleifen sollte stets ausschließlich die äußere Schleife parallelisiert werden. Dieses Vorgehen maximiert zum einen die Anzahl der parallel auszuführenden Instruktionen und minimiert zum anderen den Verwaltungsaufwand, der

[7] Dies gilt nur für OpenMP 2.5, OpenMP 3.0 erlaubt die Verwendung von Iteratoren [238, S. 39].

bei der Erzeugung bzw. Zusammenführung eines Thread-Teams nach dem Fork-Join-Prinzip 2.4.1.3 entsteht. Würde hingegen zusätzlich die innere Schleife parallelisiert, so würde in jeder Iteration der äußeren Schleife ein neues Thread-Team erzeugt und zusammengeführt (siehe Abschn. 4.3.5.1).

- Bei der Parallelisierung von geschachtelten **`for`**-Schleifen sollte stets überprüft werden, ob durch das Vertauschen der Schleifen auf etwaige Direktiven zur Synchronisierung des Thread-Teams verzichtet werden kann. Während eine Reihenfolge zur korrekten Parallelisierung u.U. die Verwendung der **`critical`**-Direktive erfordert, ist letztere nach dem Vertauschen möglicherweise überflüssig (siehe Abschn. 4.3.5.2).
- Direktiven zur Synchronisierung des Thread-Teams, z.B. **`critical`** und **`atomic`**, sind nach Möglichkeit zu vermeiden, da diese einen hohen Verwaltungsaufwand erfordern und die Skalierbarkeit negativ beeinflussen. Dies kann durch die Einrichtung eines Thread-privaten Speicherbereichs in Form eines Feldes von der Größe des Thread-Teams vermieden werden. Anschließend kann jeder Thread mit Hilfe seiner ID exklusiv auf ein Element des Feldes zugreifen, um dort z.B. Zwischenergebnisse zu speichern. Der entscheidende Vorteil, verglichen mit der Verwendung der **`private`**-Direktive, liegt darin, dass die Zwischenergebnisse nach der Abarbeitung des parallelen Bereichs nicht verloren gehen, sondern dem Master-Thread weiterhin zur Verfügung stehen, der diese abschließend nur noch kombinieren muss. Im Vergleich zu einer synchronisierten Variante sind zum einen der höhere Speicherplatzverbrauch, zum anderen der Mehraufwand, der durch den zusätzlichen Kombinationsschritt entsteht, von Nachteil (siehe Abschn. 4.3.5.2).
- Innerhalb eines parallelen Bereichs muss jeder Aufruf einer nichtkonstanten Elementfunktion[8] bezüglich der Notwendigkeit einer Syn-

[8] Funktion eines Objekts, die *nicht* mit dem Modifikator **`const`** deklariert ist. Diese darf, im Gegensatz zu konstanten Elementfunktionen, den Zustand des entsprechenden Objekts verändern, d.h. Seiteneffekte erzeugen. Im Gegensatz dazu sind konstante Elementfunktionen solche, die mit dem Modifikator **`const`** deklariert sind. Diese dürfen den Zustand des entsprechenden Objekts *nicht* verändern und wiederum nur konstante Elementfunktionen oder statische Funktionen aufrufen, d.h. konstante Elementfunktionen erzeugen *keine* Seiteneffekte.

chronisation überprüft werden. Dieses Vorgehen ist erforderlich, da eine nicht-konstante Elementfunktion den Zustand des entsprechenden Objekts verändert und es in einer parallelen Umgebung bei einem nicht synchronisierten Aufruf dazu kommen kann, dass sich das Objekt in einem inkonsistenten Zustand befindet, was offensichtlich verhindert werden muss. Hierbei ist wichtig, dass der Aufruf von nicht-konstanten Elementfunktionen nicht generell, sondern nur im Einzelfall synchronisiert werden muss. Der Aufruf konstanter Elementfunktionen ist hingegen ohne weiteres möglich.

- Bei der Markierung eines parallelen Bereichs muss jede Variable, die innerhalb des Bereichs verwendet wird, bezüglich der Notwendigkeit von Datenzugriffsklauseln, wie z.B. **`private`** und **`shared`**, überprüft werden.[9] Dieses Vorgehen ist erforderlich, da jeder Thread standardmäßig auf dieselbe Variable zugreift und es somit zu ungewollten Wettlaufsituationen kommen kann. Anzumerken bleibt, dass das Feld zur Einrichtung eines Thread-privaten Speicherbereichs nicht als **`private`** deklariert werden muss, da jeder Thread exklusiv mit Hilfe seiner ID auf einen separaten Bereich des Feldes zugreift (siehe Abschn. 4.3.5).

2.5 Laufzeitmessung

Sämtliche im Rahmen dieser Arbeit durchgeführten Laufzeitmessungen wurden auf einem Parallelrechner der Universität Münster vorgenommen, dem ZIVHPC [303]. Dieser lässt sich nach Flynn der MIMD-Kategorie zuordnen und besitzt eine hybride Speicherarchitektur (siehe Abschn. 2.2). Der Rechner besteht aus 20 Rechenknoten, wobei jeder dieser Knoten über zwei Quad-Core AMD Opteron CPUs @ 2.1 GHz, 32 GB Hauptspeicher und eine 250 GB SATA Festplatte verfügt. Insgesamt können folglich 160 Rechenkerne benutzt werden. Unterstützt wird das System von drei GPFS Dateiservern, die jeweils über zwei Quad-Core AMD Opteron CPUs @ 2.1 GHz, 24 GB Hauptspeicher, zwei 80 GB SATA-II RAID Level 1 Festplatten und zwei QLogic 4 Gb Fibre Channel Adapter verfügen und insgesamt 3 TB

[9] Eine Ausnahme stellen hier Indexvariablen von **`for`**-Schleifen dar. Diese werden, wie bereits erwähnt, standardmäßig als **`private`** deklariert.

Speicherplatz bereitstellen. Alle Rechenknoten sind über ein Infiniband-Netzwerk miteinander verbunden, zusätzlich besteht eine redundante Vernetzung über Gigabit Ethernet. Als Betriebssystem wird Scientific Linux 5.2 verwendet. Im Rahmen dieser Arbeit wurde für die Übersetzung von Programmen der Intel C++-Compiler 10.1.008 verwendet, der OpenMP 2.5 unterstützt [185].[10] Mit OpenMPI 1.2.6 [291] wurde eine quelloffene Implementierung des MPI-2 Standards [231] verwendet, die von vielen Parallelrechnern in den TOP500 [293] verwendet wird.

Um die Leistungsfähigkeit paralleler Programme einfach und schnell beurteilen und miteinander vergleichen zu können, werden aus den gemessenen absoluten Laufzeiten folgende Kennzahlen abgeleitet:

Speedup Sei t^{seq} die Ausführungszeit eines sequenziellen Programms auf einem Einkernprozessor und $t^{par}_{np,nt}$ die Ausführungszeit der parallelisierten Variante desselben Programms auf einem Parallelrechner mit np Rechenknoten, die jeweils über nt Rechenkerne verfügen. Der Speedup ist derjenige Faktor, um den die Ausführungszeit t^{seq} im Vergleich zu $t^{par}_{np,nt}$ reduziert wird und ist wie folgt definiert:

$$Sp_{np,nt} = t^{seq} \,/\, t^{par}_{np,nt} \tag{2.1}$$

Effizienz Faktor, der die Leistungsfähigkeit der parallelen im Vergleich zur sequenziellen Variante eines Programms beschreibt. Die Effizienz wird aus dem Speedup hergeleitet und ist wie folgt definiert:

$$Eff_{np,nt} = Sp_{np,nt} \,/\, (np \cdot nt) = t^{seq} \,/\, (t^{par}_{np,nt} \cdot np \cdot nt) \tag{2.2}$$

Anzumerken bleibt, dass zur Berechnung des Speedups bzw. der Effizienz im Folgenden nicht die Ausführungszeit des sequenziellen Programms t^{seq}, sondern die des parallelen Programms unter Verwendung von $np = nt = 1$ verwendet wird, d.h. $t^{seq} = t^{par}_{1,1}$.

[10] Der aktuelle Intel C++-Compiler 11.1 unterstützt OpenMP 3.0 [186, S. 1121], ist jedoch auf dem ZIVHPC nicht installiert. Das Nachfolgesystem PALMA (Paralleles Linux-System für Münsteraner Anwender) [304] unterstützt zwar OpenMP 3.0, stand aber noch nicht zur Verfügung.

2.6 Stand der Forschung

Die folgenden Abschnitte 2.6.1 bis 2.6.24 geben eine Überblick über die wichtigsten Skelettbibliotheken anderer Forschergruppen, um einerseits den aktuellen Stand der Forschung zu skizzieren und andererseits einen Vergleich mit Muesli zu ermöglichen. Die Auflistung erhebt keinen Anspruch auf Vollständigkeit, alternative Darstellungen können [209, 299] entnommen werden. Die Gegenüberstellung findet mit Hilfe folgender Kriterien statt:

Programmiersprache Gibt an, in welcher Programmiersprache der Benutzer das parallele Programm erstellt.

Taskparallelität Gibt an, ob die Skelettbibliothek taskparallele Skelette zur Verfügung stellt. Das Kriterium gilt eingeschränkt, falls nicht mindestens die Skelette *Pipeline* und *Farm* implementiert sind.

Datenparallelität Gibt an, ob die Skelettbibliothek datenparallele Skelette zur Verfügung stellt. Das Kriterium gilt eingeschränkt, falls nicht mindestens die Skelette *fold* (siehe Abschn. 3.4.2) und *map* (siehe Abschn. 3.4.3) implementiert sind.

Schachtelbarkeit Falls die Skelettbibliothek ausschließlich taskparallele Skelette unterstützt, gibt das Kriterium an, ob letztere beliebig komponiert werden können. Falls die Skelettbibliothek zusätzlich datenparallele Skelette unterstützt, gibt das Kriterium an, ob taskparallele Skelette beliebig komponiert werden können und auf der untersten Ebene darüber hinaus datenparallele Skelette verwendet werden können. Falls die Skelettbibliothek ausschließlich datenparallele Skelette unterstützt, kann das Kriterium nicht erfüllt werden, da die Komposition datenparalleler Skelette keinen Sinn ergibt.

Verteilter/gemeinsamer/hybrider Speicher Gibt an, ob das unter Verwendung der Skelettbibliothek erstellte parallele Programm effizient auf Rechnerarchitekturen mit verteiltem, gemeinsamem und/oder hybridem Speicher skaliert (siehe Abschn. 2.2.2). Das jeweilige Kriterium ist eingeschränkt erfüllt, falls die Skalierbarkeit nur für einen Teil der Skelettbibliothek gewährleistet ist.

Benutzerdefinierte Typen Gibt an, ob die Skelettbibliothek neben elementaren auch beliebige benutzerdefinierte Typen unterstützt. Das Kriterium ist eingeschränkt erfüllt, falls eine C/C++-basierte Bibliothek zwar das Versenden von Strukturen ermöglicht, letztere aber keine Attribute vom Typ Zeiger enthalten dürfen.

Kommunikation Gibt an, welche Technologien zur Durchführung der Kommunikation bzw. Synchronisation verwendet werden.

Tabelle 2.7 auf Seite 74 fasst die Funktionalität der betrachteten Skelettbibliotheken anhand der oben erwähnten Kriterien übersichtlich zusammen. Anzumerken bleibt, dass Muesli der Vergleichbarkeit halber ebenfalls in der Tabelle aufgeführt ist, obwohl eine ausführliche Beschreibung erst in Kapitel 3 erfolgt. Abschnitt 2.6.25 schließt die Betrachtung des Forschungsstands mit einem Fazit ab.

2.6.1 ASSIST

ASSIST (A software development system based upon integrated skeleton technology) ist eine Programmierumgebung zur Erstellung paralleler Programme [11, 13, 14, 15, 16, 17, 297] und basiert auf den mit SkIE (siehe Abschn. 2.6.22) gewonnenen Erfahrungen. Im Gegensatz zu klassischen Skelettbibliotheken implementiert ASSIST mit *parmod* (engl. parallel module) allerdings nur ein einziges generisches Skelett (Modul), welches jedoch spezialisiert werden kann, um traditionelle Skelette, wie z.B. *farm* oder *map*, zu emulieren. Parallele Module besitzen einen internen Zustand und können, im Gegensatz zu sequenziellen Modulen, externe Objekte einbinden, z.B. über CORBA. Ein paralleles Programm wird in ASSIST durch die Konstruktion eines Graphen erstellt, der mit Hilfe von ASSISTcl (ASSIST coordination language) beschrieben wird. Der Graph kann sowohl parallele als auch sequenzielle Module enthalten und diese uneingeschränkt durch Ströme miteinander verbinden, d.h. der Graph kann Zyklen, $1:n$- sowie $n:1$-Verbindungen enthalten. Benutzerdefinierte Funktionen können in C, C++ oder F77 geschrieben werden. ASSIST-Programme skalieren sowohl auf dedizierten Parallelrechnern als auch in Grid-Umgebungen und verwenden zur Kommunikation TCP/IP-Sockets.

2.6.2 Calcium

Calcium ist eine von Lithium (siehe Abschn. 2.6.11) und Muskel (siehe Abschn. 2.6.13) inspirierte, Java-basierte Skelettbibliothek [82, 83, 84, 209]. Die Bibliothek implementiert mit *d&c*, *farm*, *for*, *fork*, *if*, *map*, *pipe*, *seq* und *while* sowohl task- als auch datenparallele Skelette, die darüber hinaus beliebig verschachtelt werden können. Neben einem auf Java Generics basierendem Typsystem sowie einem integrierten Modell zur Identifikation von Performanz-Engpässen in benutzerdefinierten Argumentfunktionen implementiert Calcium ein transparentes Modell für den Dateizugriff. Letzteres ermöglicht Skeletten den Zugriff auf das Dateisystem, d.h. jedes Skelett kann als Quelle eines Datenstroms in Erscheinung treten, was den Kommunikationsaufwand bei datenintensiven Anwendungen erheblich reduziert. Mit Calcium geschriebene Programme skalieren unter Verwendung von Java Threads bzw. ProActive [81, 85, 233], einer Java-basierten Middleware für Grids, auf Rechnerarchitekturen mit gemeinsamem bzw. verteiltem Speicher, hybride Speicherarchitekturen werden hingegen nicht unterstützt.

2.6.3 CO_2P_3S

CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) ist ein Java-basiertes Framework zur Erstellung paralleler Programme für Rechnerarchitekturen mit verteiltem oder gemeinsamem Speicher [34, 219, 220, 221, 222, 287]. Das Framework basiert auf dem sogenannten *parallelen Entwicklungsprozess* (engl. parallel design process, PDP) und stellt mit MetaCO_2P_3S eine grafische Benutzeroberfläche zur Verfügung, die durch ein integriertes Schichten-Konzept drei verschiedene Sichtweisen mit unterschiedlichem Abstraktionsgrad auf das parallele Programm ermöglicht. Der Entwicklungsprozess sieht vor, in einem ersten Schritt unter Verwendung der *Pattern Layer* zunächst die Skelett-Topologie zu konstruieren. Hierfür stehen die Skelette *Distributor*, *Mesh*, *Method Sequence*, *Phases*, *Pipeline* und *Wavefront* zur Verfügung, die beliebig miteinander kombiniert werden können. Unter Verwendung von MetaCO_2P_3S können darüber hinaus neue Skelette erstellt bzw. bestehende angepasst werden, um so die Erweiterbarkeit des Frameworks zu gewährleisten. Die Spezialisierung

der Skelette erfolgt einerseits durch die Implementierung benutzerdefinierter Methoden, andererseits durch die Konfigurierung der Skelette, wobei letztere jeweils eine Menge an charakteristischen Eigenschaften aufweisen, wie z.B. die Länge eines *Pipeline*- oder die Dimension eines *Mesh*-Skeletts. Aus der modellierten Skelett-Topologie und den benutzerdefinierten Methoden erzeugt CO_2P_3S Quelltext, der grundsätzlich funktionsfähig ist. Unter Verwendung der *Intermediate Code Layer* bzw. der *Native Code Layer* kann jedoch anschließend optional eine Feinabstimmung durchgeführt werden, um das Synchronisations- und Kommunikationsverhalten zu optimieren bzw. plattformspezifische Anpassungen vorzunehmen. Mit CO_2P_3S erstellte parallele Programme skalieren unter Verwendung von Java Threads bzw. Java RMI auf Parallelrechnern mit gemeinsamem bzw. verteiltem Speicher, hybride Speicherarchitekturen werden jedoch nicht unterstützt.

2.6.4 DatTeL

DatTeL (Data-parallel Template Library) wurde erstmalig in [65] veröffentlicht und erlaubt die Erstellung paralleler Programme unter Beibehaltung eines STL-konformen Programmierstils. Zu diesem Zweck definiert die Bibliothek einen Datentyp `par::vector<T>`, so dass bereits implementierte Programme lediglich den Namensraum `std` durch `par` ersetzen müssen, um den parallelen Datentyp zu benutzen [66]. Weitere Datentypen, wie z.B. `std::set` oder `std::map`, werden derzeit nicht implementiert, können aber flexibel erweitert werden. Die Kompatibilität mit den von der STL definierten Funktionen höherer Ordnung, wie z.B. `accumulate`, `for_each`, `inner_product` oder `partial_sum`, wird mit Hilfe von Iteratoren [213, S. 378–383] gewährleistet, die von `par::vector<T>` durch die Implementierung der Funktionen `begin` und `end` entsprechend zurückgegeben werden. Die parallele Verarbeitung wird, analog zu dem von Muesli implementierten Ansatz (siehe Abschn. 3.2.1), durch eine Partitionierung der Datenstruktur bewerkstelligt. Die Kommunikation findet mit Hilfe von MPI oder Pthreads statt, wobei das Backend der Skelettbibliothek hierfür eine eigene Abstraktionsschicht definiert. Dadurch skaliert ein mit Hilfe von DatTeL geschriebenes Programm auf Parallelrechnern mit verteiltem oder gemeinsamem Speicher, von einer hybriden Speicherarchitektur kann jedoch nicht profitiert werden [64]. Anzumerken bleibt, dass die Verwendung von benutzerde-

finierten Datentypen mit DatTeL nicht möglich ist, da kein entsprechender Mechanismus zur Serialisierung von Objekten implementiert ist.

2.6.5 Eden

Eden ist eine auf Haskell [181, 188, 292] basierende funktionale, parallele Programmiersprache [51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 73, 74, 75, 76, 77, 78, 128, 149, 163, 165, 175, 176, 177, 194, 195, 196, 197, 198, 214, 215, 216, 217, 218, 240, 242, 243, 247, 254, 260]. Eden erweitert Haskell um ein syntaktisches Konstrukt zur expliziten Definition von Prozessen, welches die direkte Kontrolle über die Prozessgranularität, die Verteilung der Daten sowie das Kommunikationsverhalten ermöglicht. Prozesse kommunizieren über unidirektionale Kanäle, wobei letztere stets zwei Prozesse miteinander verbinden. Während die Definition von Prozessen explizit ist, d.h. vom Benutzer vorgenommen werden muss, sind die Instanziierung von Prozessen sowie die Durchführung der Kommunikation bzw. Synchronisation implizit, d.h. diese werden vom Laufzeitsystem durchgeführt. Auch die Verteilung der Daten erfolgt implizit, indem der Benutzer bei der Prozessdefinition angibt, von welchen Daten ein Prozess abhängig ist. Eden verfügt über ein integriertes Kostenmodell, das unter Verwendung verschiedener Parameter, wie z.B. Problemgröße oder Latenzzeiten, die Vorhersage der Ausführungszeit ermöglicht, wobei die verwendeten Parameter teilweise vom Benutzer übergeben werden müssen bzw. teilweise automatisch vom Laufzeitsystem ermittelt werden können. In Eden sind Skelette, analog zu Skil (siehe Abschn. 2.6.23), kein integraler Bestandteil der Sprache, sondern werden unter Verwendung derselben definiert, so dass neue Skelette vergleichsweise einfach hinzugefügt werden können. Neben taskparallelen Skeletten, wie z.B. *pipeline* und *divide-and-conquer*, und datenparallelen Skeletten, wie z.B. *map* und *fold*, werden auch sogenannte *systolische Skelette*[11] definiert, wie z.B. *iterUntil*, *torus* und *ring*. Die Definition eines Skeletts besteht einerseits aus einer parallelen Implementierung, andererseits aus einer formalen Spezifikation, wobei letztere für formale Korrektheitsbeweise verwendet werden kann. Mit Eden erstellte parallele Programme skalieren unter Verwendung von PVM [151] oder MPI ausschließlich auf Parallelrechnern mit

[11] Systolische Skelette sind solche, die wiederholt abwechselnd parallele Berechnungen und eine globale Synchronisation durchführen.

verteiltem Speicher. Das Ausführungsmodell von Eden sieht vor, dass das parallele Programm, in Analogie zum Fork-Join-Prinzip von OpenMP (siehe Abschn. 2.4.1.3), zunächst von einem singulären Prozess ausgeführt wird, der bei Bedarf neue Prozesse erzeugt, die wiederum neue Prozesse erzeugen können usw. Die erzeugten Prozesse werden anschließend jedoch wieder zerstört, so dass das Programm am Ende erneut von nur einem Prozess ausgeführt wird. Anzumerken bleibt, dass die Definition und das Versenden benutzerdefinierter Typen möglich ist, solange diese von der Klasse `Trans` erben.

2.6.6 eSkel

eSkel (Edinburgh Skeleton Library) ist eine auf C und MPI basierende Skelettbibliothek, die mit *Butterfly*, *Deal*, *Farm*, *Haloswap* und *Pipeline* ausschließlich taskparallele Skelette zur Verfügung stellt [44, 45, 46, 47, 48, 49, 50, 105, 153, 154, 155, 156, 301]. Die Funktionalität der Skelette *Farm* und *Pipeline* entspricht der in Muesli implementierten Pendants, *Butterfly* implementiert Divide & Conquer-Funktionalität. *Deal* ist eine spezielle Form der *Farm*, die eine zyklische Aufgabenverteilung und keinen dedizierten Farmer-Prozess besitzt, mit *Haloswap* können Verfahren zur iterativen Approximation umgesetzt werden. Sämtliche Skelette können beliebig verschachtelt werden. Bei der Konstruktion der Skelett-Topologie kann darüber hinaus Einfluss auf den sogenannten *nesting mode* und *interaction mode* genommen werden. Der *nesting mode* beeinflusst das Verhalten von verschachtelten Skeletten und kann entweder auf *transient* oder *persistent* gesetzt werden. *Transient* bewirkt, dass verschachtelte Skelette vor jedem Aufruf neu erzeugt und anschließend wieder zerstört werden, *persistent*, dass verschachtelte Skelette lediglich einmalig erzeugt und anschließend wiederverwendet werden. Der *interaction mode* hat Auswirkungen auf den Datenfluss zwischen den einzelnen Skeletten und kann entweder auf *implicit* oder *explicit* gesetzt werden. *Implicit* bewirkt, dass der Datenfluss ausschließlich durch die Skelett-Topologie festgelegt wird, *explicit*, dass der Datenfluss manuell beeinflusst werden kann. Auf diese Weise kann z.B. die Stufe einer *Pipeline* pro Eingabe 0, 1 oder n Ausgabe(n) erzeugen. Zu diesem Zweck stehen dem Benutzer mit `Give` und `Take` zwei Funktionen zur

Verfügung, deren Funktionalität mit der von Muesli implementierten Funktionen `MSL_put` und `MSL_get` (siehe Abschn. 5.3.2) übereinstimmt. Das Datenmodell von eSkel (eSkel Data Model, eDM) sieht ausschließlich das Versenden und Empfangen von sogenannten *eDM atoms* vor, die neben den eigentlichen Daten ein sogenanntes *data spread tag* enthalten. Das Tag kann entweder auf *local* oder *global* gesetzt werden und gibt an, ob ein Datum Teil eines großen Ganzen ist (global) oder nicht (local). Darüber hinaus besteht mit der *eDM collection* die Möglichkeit, mehrere eDM atoms mit identischem Tag zu bündeln. Aktuelle Forschungsergebnisse zeigen diverse Optimierungen speziell für das *Pipeline*-Skelett, wie z.B. ein automatisiertes Lastverteilungsverfahren, welches für heterogene Systeme gedacht ist und in Abhängigkeit von der Auslastung des Parallelrechners einzelne Stufen des Skeletts anderen Prozessoren zuordnet.

2.6.7 FastFlow

FastFlow ist eine C++-basierte Template-Bibliothek zur Programmierung Cache-kohärenter Mehrkernprozessoren und eignet sich insbesondere zur Erstellung Datenstrom-basierter Anwendungen [12, 24, 25, 27, 28, 29]. Zentrales Charakteristikum ist die Implementierung eines parallelen, nicht-blockierenden Speicherallokators. Dieser vermeidet Architektur-bedingte Speicherbarrieren (engl. memory fences), die normalerweise benötigt werden, um die Caches der einzelnen Rechenkerne kohärent zu halten. Die Architektur des Frameworks besteht aus den drei aufeinander aufbauenden Schichten *runtime support*, *low-level programming* und *high-level programming*, die sukzessiv von der unterliegenden Hardware abstrahieren. Auf unterster Ebene können durch nicht-blockierende Erzeuger-Verbraucher-Warteschlangen (engl. producer-consumer-queues) einfache, Datenstrom-basierte Netzwerke erstellt werden. Darauf aufbauend ermöglicht die mittlere Ebene durch die Implementierung von $1:m$-, $n:1$- und $n:m$-Warteschlangen die Erstellung beliebiger Datenstrom-basierter Netzwerke. Die oberste Ebene stellt mit *Farm*, *Pipeline* und *D&C* drei taskparallele Skelette zur Verfügung, die beliebig verschachtelt werden können. Anzumerken bleibt, dass der Datenaustausch innerhalb der Warteschlangen mit Hilfe von unidirektionalen, asynchronen Kommunikationskanälen stattfindet.

2.6.8 HDC

HDC (Higher-Order Divide-and-Conquer) ist eine Teilmenge der funktionalen Programmiersprache Haskell [181, 188, 292], implementiert jedoch im Unterschied zu letzterer eine strikte Semantik [166, 167, 168, 169, 170, 171, 172, 173, 174]. HDC-Programme werden von einem Compiler in gültige C-Programme inklusive MPI-Kommunikationsroutinen übersetzt, die Ausführung erfolgt nach dem SPMD-Modell (siehe Abschn. 3.2.1). Das HDC-Laufzeitsystem übernimmt dabei Aufgaben, wie z.B. die Speicherverwaltung oder die (De-)Serialisierung von Daten, so dass beliebige benutzerdefinierte Typen ausgetauscht werden können. Analog zu Muesli sind Skelette als Funktionen höherer Ordnung (siehe Abschn. 3.2.3) implementiert, wenngleich, entgegen der Benennung der Bibliothek, sowohl task- als auch datenparallele Skelette unterstützt werden. Datenparallelität wird durch die Skelette *filter*, *map*, *red* (*fold*) und *scan* unterstützt, die ausschließlich auf Listen operieren, wobei letztere beim Aufruf des Skeletts in mehrere Partitionen zerlegt und unter den zur Verfügung stehenden Prozessen verteilt werden. Taskparallelität wird durch die Skelette *dc* (Divide & Conquer), Γ (Gamma) und *while* zur Verfügung gestellt. Das *dc*-Skelett nimmt hierbei eine Sonderstellung ein, da mit den Skeletten *dcA*, *dcB*, *dcC*, *dcD*, *dcE* und *dcF* mehrere Spezialisierungen implementiert wurden. Diese stehen über eine Hierarchie miteinander in Verbindung und nehmen verschiedene Einschränkungen des allgemeinen Divide & Conquer-Verfahrens vor, um zur Laufzeit diverse Optimierungen zu erzielen. Die vorgenommenen Optimierungen sind eine feste Rekursionstiefe (*dcB*), ein fester Aufteilungsgrad (*dcC*), mehrfache Blockrekursion (*dcD*), elementweise Operationen (*dcE*) sowie der Austausch korrespondierender Elemente (*dcF*). Anzumerken bleibt, dass das Hinzufügen benutzerdefinierter Skelette problemlos möglich ist und somit die Erweiterbarkeit gewährleistet ist.

2.6.9 HOC-SA

HOC-SA (Higher-Order Components-Service Architecture) ist ein Java-basiertes Framework zur Entwicklung paralleler Programme für Grids [32, 130, 131, 132, 133, 134, 135, 157]. Das Framework baut auf dem Globus Toolkit auf, einer weit verbreiteten, OGSA-kompatiblen Grid-Middleware, und

besitzt den Status eines Globus Incubator Projekts [289]. Skelette werden als sogenannte *Higher-Order Components* (HOCs) zur Verfügung gestellt, die von der unterliegenden Middleware abstrahieren. Mit *Alignment*, *Deformation*, *Divide & Conquer*, *Farm*, *Lifting*, *LooPo*, *Pipeline*, *Reduce* und *Wavefront* werden sowohl task- als auch datenparallele Skelette implementiert, deren Komposition ist jedoch nicht möglich. Die Kommunikation innerhalb eines Skeletts findet mit Hilfe von Web Services bzw. SOAP-Nachrichten statt und wird vom Globus Toolkit bereitgestellt. Als besonderes Feature implementiert HOC-SA einen sogenannten *code mobility*-Mechanismus in Form eines *remote class loaders*. Der Mechanismus sieht vor, dass benutzerdefinierte Java-Funktionen als Bytecode in Datenbanken abgelegt werden können. Bei Bedarf wird dieser ausgelesen und per SOAP-Nachricht an das entsprechende Skelett versendet, welches das Objekt per Reflection wiederherstellt und die benutzerdefinierte Funktion ausführt. Auf diese Weise können nicht nur Skelette, sondern auch benutzerdefinierte Funktionen wiederverwendet werden und sind global verfügbar. Anzumerken bleibt, dass benutzerdefinierte Funktionen neben Java auch mit C, C++, Fortran, Python oder Ruby geschrieben werden können, das Framework implementiert hierfür ein entsprechendes Gateway.

2.6.10 JaSkel und YaSkel

JaSkel (Java Skeleton-based Framework) ist eine Java-basiert Skelettbibliothek, die Konzepte der aspektorientierten Programmierung implementiert, um auf einer Vielzahl unterschiedlicher Rechner- bzw. Speicherarchitekturen zu skalieren [31, 146, 274]. Neben einem dynamischen Last- und Datenverteilungsmechanismus besteht die Bibliothek im Wesentlichen aus zwei Komponenten:

- Taskparallele Skelette, wie z.B. *Pipeline*, *Farm* oder *Heartbeat*, werden in Form von abstrakten Klassen zur Verfügung gestellt, die wiederum von der abstrakten Oberklasse *Compute* (inspiriert von Lithium, siehe Abschn. 2.6.11) erben. Diese Vererbungshierarchie ermöglicht einerseits die uneingeschränkte Schachtelbarkeit bestehender, andererseits die unkomplizierte Definition neuer Skelette und garantiert gleichzeitig deren Interoperabilität. Das Einfügen benutzerdefinierter

Funktionalität findet statt, indem von dem jeweiligen Skelett eine konkrete Unterklasse erzeugt wird, die sämtliche abstrakten Methoden überschreibt. Neben den parallelen Varianten (*concurrent* und *dynamic*) existiert für jedes Skelett auch eine sequenzielle Variante, die das Testen vereinfachen soll. Anzumerken bleibt, dass JaSkel keine verteilten Datenstrukturen implementiert, so dass Datenparallelität durch taskparallele Konstrukte simuliert werden muss.

- Die Skalierbarkeit der Skelettbibliothek wird durch externe Werkzeuge gewährleistet. Grundsätzlich skalieren mit Hilfe von JaSkel geschriebene parallele Programme durch die Verwendung von Java Threads nur auf Rechnerarchitekturen mit gemeinsamem Speicher. Die Bibliothek stellt jedoch mit dem sogenannten *Cluster* bzw. *Grid Enabling Tool* zwei Werkzeuge zur Verfügung, mit denen die zur Kommunikation benötigte Funktionalität per Dependency Injection in die Skelette eingefügt werden kann. Auf diese Weise kann JaSkel unter Verwendung von UkaRMI, einer Implementierung von Java RMI, bzw. des Dateisystems auch auf Clustern bzw. Grids skalieren. Darüber hinaus können die Werkzeuge kombiniert werden, so dass JaSkel auch auf hybriden Speicherarchitekturen skaliert. Dies setzt jedoch eine entsprechende Verschachtelung taskparalleler Skelette voraus, d.h. die Skelett-Topologie muss in Abhängigkeit von der Zielplattform konstruiert werden, auf der das Programm ausgeführt werden soll, was den Gedanken der aspektorientierten Programmierung ad absurdum führt.

Mit YaSkel (Yet another Skeleton Library) existiert darüber hinaus eine neuere Version von JaSkel, die die Entkopplung der benutzerdefinierten Funktionalität und der vom Skelett implizit definierten Parallelität noch stärker in den Vordergrund rückt [234]. Die Bibliothek implementiert die taskparallelen Skelette *Direct Acyclic Graph*, *Divide & Conquer*, *Farm*, *Map* und *Pipeline* und ermöglicht die nahtlose Parallelisierung von bereits bestehenden sequenziellen Programmen.

2.6.11 Lithium

Lithium ist eine Java-basierte Skelettbibliothek [10, 23, 26, 109, 117]. Mit *comp*, *d&c*, *farm*, *for*, *if*, *map*, *pipe*, *seq* und *while* werden sowohl task- als auch datenparallele Skelette zur Verfügung gestellt. Diese können darüber hinaus beliebig verschachtelt werden und operieren stets auf einem Datenstrom, verteilte Datenstrukturen werden nicht implementiert. Lithium basiert auf dem sogenannten *macro data flow* (MDF) Ausführungsmodell [108, 110]. Dieses erzeugt, ausgehend von der konstruierten Skelett-Topologie, einen MDF-Graphen, dessen Kanten dem Datenfluss und dessen Knoten den sequenziellen Berechnungen (MDF instructions, MDFi) entsprechen. Anschließend wird unter Verwendung des MDF-Graphen eine entsprechende Anzahl an Prozessen erzeugt, die die zu bearbeitenden Aufgaben aus einem zentral verwalteten Pool entnehmen können. Als weiteres Charakteristikum implementiert Lithium ein Verfahren zur Skelett-basierten Termersetzung [18, 19]. Auf diese Weise können Skelett-Topologien, die ausschließlich die Skelette *farm*, *pipe* und *seq* verwenden, in eine sogenannte *Normalform* transformiert werden, um unnötigen Kommunikationsaufwand zu vermeiden und die Verschachtelungstiefe zu verringern. Mit Lithium erstellte parallele Programme skalieren ausschließlich auf Rechnern mit verteiltem Speicher und verwenden zur Kommunikation Java RMI.

2.6.12 MALLBA

MALLBA (Malaga, La Laguna, Barcelona) ist eine C++-basierte Skelettbibliothek zur Lösung kombinatorischer Optimierungsprobleme [7, 8, 9]. Die Bibliothek unterstützt die folgenden drei Familien generischer Optimierungstechniken:

- Zur exakten Lösung eines Optimierungsproblems können die Skelette *Branch & Bound* (BnB), *Divide & Conquer* und *Dynamic Programming* verwendet werden.

- Die heuristische Lösung eines Optimierungsproblems wird von den Skeletten *Espectral Methods*, *Genetic Algorithm* (GA), *Hill Clim-*

bing, *Memetic Algorithm*, *Metropolis*, *Simulated Annealing* (SA) und *Tabu Search* (TS) implementiert.

- Darüber hinaus besteht die Möglichkeit, exakte und heuristische Techniken zu kombinieren. Solche hybriden Verfahren umfassen z.B. BnB + GA, BnB + SA, GA + SA und GA + TS.

Von jedem Skelett existieren drei verschiedene Implementierungen, die für die sequenzielle und die parallele Ausführung in LANs bzw. WANs optimiert ist. Unabhängig von der Implementierung verfügt jedes Skelett über eine Menge an Zustandsvariablen, die den aktuellen Zustand des Optimierungsproblems beschreiben und z.B. Zwischenergebnisse oder die bisher beste Lösung speichern. Die Variablen können zur Laufzeit abgefragt werden, um den Fortschritt der Optimierung zu beurteilen und ggf. korrigierend einzugreifen. Zur Kommunikation verwendet MALLBA eine selbstentwickelte, auf MPI basierende Kommunikationsschicht mit der Bezeichnung *NetStream*, wobei letztere im Wesentlichen die Benutzung der von MPI zur Verfügung gestellten Kommunikationsfunktionen vereinfacht.

2.6.13 muskel

muskel ist eine von Lithium (siehe Abschn. 2.6.11) inspirierte, Java-basierte Skelettbibliothek [20, 21, 22, 111, 113]. Mit *Farm*, *Pipeline* und *Seq* werden ausschließlich taskparallele Skelette implementiert, diese können jedoch beliebig verschachtelt werden. muskel basiert auf dem bereits von Lithium verwendeten *macro data flow* Ausführungsmodell, ermöglicht aber mit Hilfe einer grafischen Benutzeroberfläche die Manipulation des erzeugten MDF-Graphen. Auf diese Weise kann der Benutzer einerseits die von den Skeletten implementierten Kommunikationsmuster anpassen bzw. erweitern, andererseits sogenannte *Ad-hoc-Parallelität* ausdrücken. Als weitere charakteristische Eigenschaften implementiert muskel eine Art Quality of Service und verwendet für die Kommunikation Java RMI in Verbindung mit Java Secure Socket Extension (JSSE). Auf diese Weise ist eine Authentifizierung der Prozesse gegenüber dem weiterhin zentralisierten Aufgabenpool sowie eine sichere Übertragung der Aufgaben bzw. Ergebnisse möglich.

2.6.14 P^{3}L

P^{3}L (Pisa Parallel Programming Language) ist eine imperative, parallele, Skelett-basierte Programmiersprache, die neben einem eigenen Compiler (Anacleto) ein ausgeklügeltes Kostenmodell zur Optimierung des P^{3}L-Programms bereitstellt [35, 36, 37, 97, 112, 114, 115, 244, 245]. P^{3}L definierte drei Arten von Skeletten: Neben den taskparallelen Skeletten *pipe* und *farm* sowie den datenparallelen Skeletten *map*, *reduce*, *scan*, *scanl*, *scanr* und *comp*, die ausschließlich auf mehrdimensionalen Feldern operieren können, werden mit *seq* und *loop* zusätzlich zwei sogenannte *Kontrollskelette* (engl. control skeleton) zur Verfügung gestellt. Task- und datenparallele Skelette können zwar verschachtelt werden, die Komposition beschränkt sich jedoch auf ein sogenanntes *Zwei-Schichten-Modell*, das als äußeres Skelett stets ein taskparalleles, als inneres stets ein datenparalleles Skelett vorsieht. Kontrollskelette unterliegen nicht diesen Einschränkungen, da erstere die parallele Struktur des P^{3}L-Programms nicht beeinflussen. Skelette werden in P^{3}L in Form sogenannter *Templates* definiert, die einerseits das sogenannte *Prozessnetzwerk* (engl. process network), d.h. das Skelett selbst inklusive Dekomposition, Lastverteilung und Kommunikation, andererseits das Kostenmodell beinhaltet. Auf diese Weise können für dasselbe Skelett mehrere Templates existieren, um z.B. plattformabhängige Optimierungen oder verschiedene Implementierungsstrategien zu ermöglichen. Der bereits erwähnte Compiler Anacleto ist einerseits für die Übersetzung des P^{3}L-Programms in nativen C-Quelltext, andererseits für die Erstellung einer Kostenabschätzung zuständig. Das kompilierte Programm verwendet zur Kommunikation ausschließlich MPI-Funktionen, so dass P^{3}L für die Verwendung auf Parallelrechnern mit verteiltem Speicher beschränkt ist. Die Kostenabschätzung findet mit Hilfe vom Benutzer zur Verfügung gestellter Testdaten statt. Falls die Schätzung nicht den Erwartungen entspricht, muss u.U. eine manuelle Korrektur der Skelett-Topologie vorgenommen werden. Eine automatisierte Anpassung bzw. die Identifikation von Engpässen findet hingegen nicht statt. Anzumerken bleibt, dass P^{3}L neben den Skeletten eine Reihe integrierter Typen definiert, z.B. `int`, `char` oder `bool`. Benutzerdefinierte Typen werden ausschließlich in Form von (mehrdimensionalen) statischen Feldern unterstützt, d.h. die Verwendung von Strukturen und Objekten ist nicht möglich.

2.6.15 PAS, SuperPAS und EPAS

PAS (Parallel Architectural Skeletons) ist eine C++-basierte Template-Bibliothek zur Erstellung paralleler Programme nach dem SPMD-Modell [158, 159, 160, 161]. Die Bibliothek implementiert mit *Compositional*, *DataParallel*, *DivideConquer*, *Replication*, *Singleton* und *PipeLine* sowohl task- als auch datenparallele Skelette, die in einem sogenannten *HTree* (engl. hierarchical tree) angeordnet werden. Die Erstellung eines Programms erfolgt entweder direkt mit C++ oder alternativ unter Verwendung einer C++-basierten Spezifikationssprache, wobei im letzten Fall anschließend ein Perl-Skript C++-kompatiblen Quelltext erstellt. Das Vorgehensmodell von PAS erfordert zunächst die Parametrisierung eines Skeletts. Das Ergebnis ist ein *abstraktes Modul*, das durch das Hinzufügen von benutzerdefinierter Funktionalität zum *(konkreten) Modul* wird. Die anschließende Instanziierung eines Moduls wird als *Repräsentation* (*Rep*) bezeichnet. Zur Kommunikation werden spezielle, auf MPI basierende Protokolle verwendet, z.B. *PROT_Net* oder *PROT_Repl*. Diese implementieren verschiedene Kommunikationsroutinen, wie z.B. *SendWork* oder *ReceiveWork*, und stellen Konstrukte zur Synchronisation und (De-)Serialisierung bereit. Unterschieden wird zwischen *internen* Protokollen P_{Int} und *externen* Protokollen P_{Ext}. Während P_{Int} die Kommunikation innerhalb eines Skeletts durchführt und das inhärente Kommunikationsverhalten eines Skeletts widerspiegelt, wird P_{Ext} zur Kommunikation zwischen den Skeletten der bereits oben erwähnten Baumstruktur verwendet. Mit EPAS (Extensible Parallel Architectural Skeletons), das in einer ersten Version unter der Bezeichnung SuperPAS veröffentlicht wurde, existiert darüber hinaus eine neuere Version von PAS, die einerseits die Verschachtelung der Skelette ermöglicht, andererseits mit SDL (skeleton definition language) eine eigene Beschreibungssprache zur Definition neuer Skelette bereitstellt [3, 4, 5, 6, 298]. Das Kommunikationsverhalten wird dabei mit Hilfe eines virtuellen Prozessorgitters (VPG) festgelegt, zusätzlich implementiert EPAS Werkzeuge zur Verifikation der SDL-Syntax, zur Verwaltung des Skelett-Repositoriums sowie einen SDL-C++-Compiler.

2.6.16 QUAFF

QUAFF (Quick Application from Flow-based Framework) ist eine C++-basiert Skelettbibliothek, die Techniken der Template-Metaprogrammierung verwendet [143, 144, 145]. Die Konstruktion einer Skelett-Topologie erfolgt daher ausschließlich unter Verwendung von Typdefinitionen, so dass der zur Laufzeit erzeugte Mehraufwand für den Aufruf virtueller Funktionen vermieden werden kann, da die Spezialisierung von Template-Parametern bereits während der Kompilierung stattfindet (siehe Abschn. 3.2.2 und 3.5.4.2). Die Bibliothek implementiert die taskparallelen Skelette *pipe* und *farm*, Datenparallelität wird durch das Skelett *scm* (engl. split, compute, merge) unterstützt. Darüber hinaus existiert mit dem Skelett *pardo* (engl. parallel do) eine Möglichkeit, sogenannte *Ad-hoc-Parallelität* auszudrücken: Das Skelett startet $n \in \mathbb{N}$ parallele Prozesse, die Synchronisation bzw. Kommunikation muss hierbei jedoch explizit vom Benutzer implementiert werden. Eine bereits implementierte, sequenzielle Funktion kann mit Hilfe des *seq*-Skeletts gekapselt werden, so dass diese einem taskparallelen Skelett als Argumentfunktion übergeben werden kann. Sämtliche Skelette können beliebig verschachtelt werden und können unter Verwendung von MPI neben elementaren auch benutzerdefinierte Typen in Form von Strukturen austauschen. Die Serialisierung bzw. das Versenden von Objekten wird jedoch ebenso wenig unterstützt wie Mehrkernprozessoren bzw. Parallelrechner mit hybrider Speicherarchitektur.

2.6.17 SBASCO

SBASCO (Skeleton-Based Scientific Components) ist eine hauptsächlich für numerische Anwendungen entwickelte, Komponenten-basierte Skelettbibliothek, die Konzepte der aspektorientierten Programmierung umsetzt [123, 124, 125, 126, 127]. Das Framework implementiert mit *farm*, *multi-block* und *pipe* lediglich drei taskparallele Skelette, die Verschachtelung kann darüber hinaus ausschließlich unter Verwendung des *pipe*-Skeletts erfolgen. Zur Erstellung eines parallelen Programms bzw. zur Definition neuer Skelette stellt SBASCO zwei verschiedene Perspektiven zur Verfügung: Während sich die *application view* an Anwendungsprogrammierer richtet und die

Erstellung von Komponenten, die Komposition von Skeletten und das Festlegen von Eingabe- und Ausgabetypen ermöglicht, erlaubt die *configuration view* zusätzlich die Manipulation der Datenverteilung, der Zuordnung zwischen Prozessen und Prozessoren sowie der internen Komponentenstruktur. Sowohl das parallele Programm als auch neue Skelette sowie deren Komposition müssen unter Verwendung der SBASCO *composition language* implementiert werden, wobei letztere vor der Ausführung in reinen C++-Quelltext übersetzt wird, benutzerdefinierte Funktionen hingegen müssen direkt in C++ geschrieben werden. SBASCO fasst Skelette zu Komponenten zusammen, wobei hier zwischen sogenannten *scientific components* (SC) und sogenannten *aspect components* (AC) unterschieden wird. Während erstere ausschließlich zur Durchführung der sequenziellen Berechnungen verwendet werden, dienen letztere zur Kapselung von einzelnen Aspekten. So kapseln z.B. die sogenannten *communication aspect components* (CAC) das Kommunikationsverhalten zwischen einzelnen SCs und ermöglichen somit den Austausch bzw. diverse Optimierungen zur Laufzeit, wie z.B. das Ändern der Zuordnung zwischen Prozessen und Prozessoren oder das Zuweisen zusätzlicher Prozessoren zu einer Komponente. Die Verbindung von SCs und ACs geschieht mit Hilfe sogenannter *aspect connectors* (ACN), die an vordefinierten *join points* angekoppelt werden können, um z.B. vor dem Senden oder nach dem Empfangen von Daten bestimmte Funktionalität auszuführen. Die Kommunikation zwischen den Skeletten findet mit Hilfe der Funktionen `get_data` und `put_data` statt. Diese bauen intern auf MPI auf, das Versenden benutzerdefinierter Objekte ist nicht möglich. Anzumerken bleibt, dass SBASCO über ein internes Kostenmodell verfügt, das sowohl statisch als auch dynamisch zur Optimierung der Komponenten verwendet werden kann.

2.6.18 SCL

SCL (Structured Coordination Language) ist eine Skelett-basierte Programmiersprache [118, 119, 120, 121, 122]. Die Erstellung eines parallelen Programms erfolgt unter Verwendung eines zweischichtigen Modells: In einem ersten Schritt wird das sogenannten *coordination model* festgelegt, indem mit Hilfe von SCL die Skelett-Topologie beschrieben wird. In einem zweiten Schritt wird das sogenannte *computational model* umgesetzt, indem mit

Hilfe einer imperativen Programmiersprache (momentan wird lediglich Fortran unterstützt) die benutzerdefinierte Funktionalität implementiert wird. Sämtliche von SCL implementierten Skelette operieren grundsätzlich auf der verteilten Datenstruktur *ParArray* (engl. parallel array), wobei letztere beliebig verschachtelt werden kann. Skelette werden als polymorphe Funktionen höherer Ordnung implementiert und in Konfigurations-, Elementar- und Rechenskelette unterteilt. Konfigurationsskelette, wie z.B. (*re-*)*distribution*, partitionieren und verteilen die Elemente einer Datenstruktur mit Hilfe der Funktionen `partition` und `align`. Elementarskelette, wie z.B. *fold*, *map*, *imap* und *scan*, führen datenparallele Berechnungen auf der verteilten Datenstruktur durch, darüber hinaus stehen Skelette wie z.B. *rotate* und *brdcast* zur Verfügung. Mit *dc*, *DMPA* (engl. dynamic message passing architecture), *farm*, *pipe*, *RaMP* (engl. reduce and map over pairs), *SPMD*, *iterUntil* und *iterFor* werden darüber hinaus Rechenskelette implementiert, die im Wesentlichen taskparallele Berechnungen ermöglichen. Ein internes Kostenmodell sowie ein Optimierungsmechanismus ermöglichen die Transformation bestimmter Aufrufe von Skeletten, z.B. konsekutive Aufrufe von *map*. Mit SCL erstellte parallele Programme skalieren auf Transputern unter Verwendung von CS Tools.

2.6.19 Skandium

Skandium ist eine Java-basierte Skelettbibliothek, die eine vollständige Neuimplementierung von Calcium (siehe Abschn. 2.6.2) vornimmt [210, 211]. Mit Skandium erstellte parallele Programme skalieren ausschließlich auf Mehrkernprozessoren bzw. Rechnerarchitekturen mit gemeinsamem Speicher, andere Architekturen werden nicht unterstützt. Analog zu Calcium werden mit *d&c*, *farm*, *for*, *fork*, *if*, *map*, *pipe*, *seq* und *while* dieselben task- bzw. datenparallelen Skelette implementiert, die ebenfalls beliebig verschachtelt werden können. Benutzerdefinierte Methoden werden von Skandium als *Muskeln* bezeichnet und in vier Kategorien unterteilt: *Execution* (f_e), *Split* (f_s), *Merge* (f_m) und *Condition* (f_c). Je nach Semantik des verwendeten Skeletts werden unterschiedliche Muskeln verwendet, z.B. f_s und f_m für *d&c* oder f_c für *if* und *while*. Hierbei ist wichtig, dass die Ausführung von Muskeln grundsätzlich nicht synchronisiert wird, so dass es bei zustandsbehafteten Muskeln, die z.B. auf eine globale Variable zugreifen,

zu Wettlaufsituationen kommen kann. Dies zu verhindern ist Aufgabe des Benutzers, z.B. mit Hilfe des Schlüsselworts **`synchronized`** oder ähnlicher Mechanismen. Mit Skandium modellierte Aufgabenstellungen lassen sich auf ein allgemeines Produzent-Konsument-Problem zurückführen: Die zu bearbeitenden Aufgaben werden in eine zentrale Warteschlange eingereiht und von den zur Verfügung stehenden Threads abgearbeitet, wobei letztere dynamisch neue Aufgaben erzeugen und in die Warteschlange einreihen können. Zu beachten ist, dass jedes Skelett genau eine Eingabe konsumiert und hierfür auch genau eine Ausgabe produziert. Falls die Erzeugung mehrerer Ausgaben zwingend erforderlich ist, müssen diese in einem Container-Objekt gekapselt werden.

2.6.20 SKElib

SKElib (skeleton library) ist eine C-basierte Skelettbibliothek zur Erstellung paralleler Programme für Rechner mit verteiltem Speicher [116]. Die Bibliothek implementiert mit *farm*, *map*, *pipe*, *seq* und *while* sowohl task- als auch datenparallele Skelette, die darüber hinaus beliebig verschachtelt werden können. Analog zu P^{3}L (siehe Abschn. 2.6.14) und SkIE (siehe Abschn. 2.6.22) werden Skelette mit Hilfe sogenannter *Templates* beschrieben, die in einer Template-Bibliothek gespeichert werden. Templates werden für eine bestimmte Zielplattform erstellt und definieren, wie ein Skelett effizient durch ein Netzwerk von Prozessen implementiert werden kann, d.h. die Anzahl der zu verwendenden Prozesse, die Zuordnung von Prozessen zu Prozessoren sowie die Kommunikationskanäle. Die Festlegung der Skelett-Topologie erfolgt mit Hilfe der vordefinierten Konstruktoren `SKE_FARM`, `SKE_MAP`, `SKE_PIPE`, `SKE_SEQ` und `SKE_WHILE`. Die Funktion `SKE_CALL` startet das erstellte Prozessnetzwerk und erwartet als Argument neben dem äußersten Skelett den Dateinamen der Ein- bzw. Ausgabedatei, die Anzahl der zu verwendenden Prozesse sowie die Bezeichnung der zu verwendenden Rechenknoten. Das Ausführungsmodell von SKElib sieht vor, dass das parallele Programm zunächst lediglich auf einem Rechenknoten ausgeführt wird. Zusätzliche Rechenknoten werden erst beim Aufruf der Funktion `SKE_CALL` in Anspruch genommen. Die Funktion analysiert unter Verwendung der Template-Bibliothek die vom Benutzer erzeugte Skelett-Topologie und startet per *rsh*-Befehl (engl. remote shell) weitere Prozesse,

so dass das Programm anschließend nach dem SPMD-Modell ausgeführt wird. Die Kommunikation findet unter direkter Verwendung von TCP/IP-Sockets statt. Anzumerken bleibt, dass SKElib geringfügige Optimierungen am Prozessnetzwerk vornehmen kann, um den Kommunikationsaufwand zu reduzieren, z.B. Sender- und Empfängerprozess demselben Prozessor zuordnet.

2.6.21 SkeTo

SkeTo (Skeletons in Tokyo) ist eine von Muesli (siehe Kap. 3) inspirierte, C++-basierte Skelettbibliothek, deren grundlegende Konzepte im Wesentlichen denen von Muesli entsprechen [138, 139, 140, 187, 189, 190, 191, 223, 224, 225, 226, 227, 228, 262, 288]. SkeTo implementiert verschiedene parallele (verteilte) Datenstrukturen, z.B. für Bäume, Listen und (dünnbesetzte) Matrizen, taskparallele Skelette stehen hingegen nicht zur Verfügung. Sämtliche Datenstrukturen deklarieren einen Template-Parameter, um vom Typ der zu speichernden Elemente zu abstrahieren. Hierbei ist jedoch zu beachten, dass lediglich elementare Typen, wie z.B. **`int`** oder **`double`**, verwendet werden können, benutzerdefinierte Typen werden nicht unterstützt. Grundsätzlich implementieren alle parallelen Datenstrukturen die datenparallelen Skelette *map*, *reduce*, *scan* und *zip*, Listen darüber hinaus die Skelette *gscanl* und *gscanr*[12] bzw. *shiftl* und *shiftr*[13]. SkeTo stellt für alle Skelette eine auf dem Bird-Meertens-Formalismus [63] beruhende einheitliche Schnittstelle zur Verfügung, die den Konsum bzw. die Produktion von Elementen beschreibt. Dadurch können neue Skelette benutzerdefiniert erstellt oder als Kombination bzw. Verfeinerung bestehender Skelette definiert werden. Als Alleinstellungsmerkmal stellt die Bibliothek einen Mechanismus zur Verfügung, mit dem aufeinanderfolgende Aufrufe datenparalleler Skelette zu einem singulären Aufruf fusioniert werden können. Auf diese Weise wird einerseits die Erzeugung unnötiger temporärer Datenstrukturen, andererseits der durch wiederholte Funktionsaufrufe erzeugte Mehraufwand

[12] *gscan* ist ein generalisiertes *scan*-Skelett. Je nach Suffix reduziert dieses entweder von links nach rechts (l) oder von rechts nach links (r)

[13] *shift* verschiebt sämtliche Elemente der Datenstruktur um einen konstanten Faktor. Der Suffix gibt an, ob die Elemente nach links (l) oder nach rechts (r) verschoben werden.

vermieden. Grundsätzlich skalieren mit Hilfe von SkeTo erstellte parallele Programm lediglich auf Parallelrechnern mit verteiltem Speicher effizient, die Kommunikation wird hierbei unter Verwendung von MPI durchgeführt. Aktuelle Forschungsergebnisse demonstrieren die Skalierbarkeit der parallelen Datenstruktur für Matrizen auf hybriden Speicherarchitekturen [191]. Darüber hinaus existiert eine Version von SkeTo, die auf CUDA-fähigen Grafikkarten skaliert [262].

2.6.22 SkIE

SkIE (Skeleton-based Integrated Environment) ist eine integrierte Entwicklungsumgebung, die die graphische Erstellung paralleler, Skelett-basierter Programme für eine Vielzahl verschiedener Hardware-Architekturen ermöglicht [38, 296]. Die Entwicklungsumgebung basiert auf den von P^{3}L (siehe Abschn. 2.6.14) prototypisch umgesetzten Konzepten, implementiert darüber hinaus aber verschiedene Erweiterungen. Analog zu P^{3}L definiert SkIE mit SkIECL (SkIE Coordination Language) eine parallele Programmiersprache und unterstützt taskparallele (*pipe* und *farm*), datenparallele (*map*, *reduce* und *comp*) sowie Kontrollskelette (*seq* und *loop*). Sämtliche Skelett können beliebig verschachtelt werden, d.h. die Komposition ist nicht auf ein Zwei-Schichten-Modell beschränkt. Unter Verwendung von VSkIE (VisualSkIE) kann die Modellierung der Skelett-Topologie darüber hinaus grafisch erfolgen. Die Definition eines Skeletts erfolgt, ebenso wie in P^{3}L, mit Hilfe eines Templates, welches neben einer maschinenabhängigen Implementierung ein Kostenmodell definiert. Zu beachten ist, dass die Modellierung der Skelett-Topologie zwar mit Hilfe der bereitgestellten Skelette erfolgt, der von SkIE implementierte Compiler jedoch während der Übersetzung in Abhängigkeit von der Zielplattform entscheidet, welches der für ein Skelett implementierten Templates verwendet wird. Auf diese Weise ist theoretisch auch die Definition von Skeletten möglich, die von hybriden Speicherarchitekturen profitieren, derzeit ist jedoch keine entsprechende Implementierung verfügbar. Das Kostenmodell wird von einem integrierten Analyseprogramm dazu verwendet, die Kosten für die erstellte Skelett-Topologie zu schätzen, woraufhin ein ebenfalls integrierter Optimierer unter Verwendung formaler Ersetzungsregeln die Komposition der Skelette automatisiert anpassen kann. Als weiteres Werkzeug wird ein integrierter Debugger zur Verfügung gestellt.

SkIE erlaubt das Einbinden bereits implementierter Funktionen in den Sprachen C, C++, F77, F90, HPF und Java und verwendet zur Durchführung der Kommunikation MPI. Neben elementaren Typen und mehrdimensionalen statischen Feldern können somit Strukturen, jedoch keine beliebigen benutzerdefinierten Typen zwischen den Skeletten ausgetauscht werden.

2.6.23 Skil

Skil (Skeleton Imperative Language) ist eine imperative, parallele, Skelett-basierte Programmiersprache [68, 69, 70, 71]. Skil basiert auf einer Teilmenge der von C angebotenen Spracheigenschaften, implementiert aber mit Funktionen höherer Ordnung, Currying, partiellen Applikationen sowie einem polymorphen Typsystem eine Reihe von Erweiterungen. Der Skil-Compiler übersetzt diese Erweiterungen jedoch in gewöhnlichen C-Quelltext, der anschließend in einem zweiten Schritt von einem herkömmlichen C-Compiler übersetzt werden kann. Die Sprache bietet mit dem `pardata`-Konstrukt die Möglichkeit, parallele (verteilte) Datenstrukturen zu definieren. Letztere können, analog zu dem von C++ bekannten Template-Mechanismus (siehe Abschn. 3.2.2), polymorph definiert und mit beliebigen elementaren sowie benutzerdefinierten Typen instanziiert werden. Auch die Verwendung Zeiger-basierter Typen ist möglich, der Benutzer muss die Funktionen zum Serialisieren (`pack_f`) bzw. Deserialisieren (`unpack_f`) jedoch selbst bereitstellen. Die Verschachtelung paralleler Datenstrukturen ist hingegen nicht möglich. Skil definierte eine Reihe von datenparallelen Skeletten, wie z.B. *map*, *fold*, *permute* und *broadcast*. Diese operieren auf den parallelen Datenstrukturen, sind jedoch, analog zu Eden (siehe Abschn. 2.6.5), kein integraler Bestandteil der Sprache, sondern vielmehr unter Verwendung von Skil implementiert. Auf diese Weise können neue, benutzerdefinierte Skelette problemlos hinzugefügt werden. Skil skaliert ausschließlich auf Transputer-basierten Parallelrechnern mit verteiltem Speicher [202] und verwendet zur Kommunikation keine spezielle Schnittstelle, sondern Routinen des Betriebssystems.

2.6.24 SKIPPER

SKIPPER (Skeleton-Based Parallel Programming Environment) ist eine Skelett-basierte Programmierumgebung, die die Erstellung von Anwendungen zur Echtzeit-Verarbeitung visueller Eingabedaten auf eingebetteten Systemen bzw. dedizierten Parallelrechnern ermöglicht [107, 152, 276, 277, 278, 279]. Die Umgebung besteht im Wesentlichen aus einer Menge an vordefinierten Skeletten, einem Übersetzungs- (engl. compile-time system, CTS) sowie einem Laufzeitsystem (engl. run-time system, RTS). Die implementierten Skelette sind *scm* (engl. split, compute, merge), *df* (engl. data-farming), *tf* (engl. task-farming) und *itermem* (engl. iteration with memory). *scm* eignet sich zur statischen Dekomposition von Bildern in Teilbilder, wird also für Bilder verwendet, deren Größe fix und im Voraus bekannt ist. *df* hingegen implementiert eine dynamische Dekomposition nach dem Farm-Prinzip: Ein Master-Prozess teilt die Teilbilder zur Laufzeit den Slave-Prozessen zu, die wiederum das Ergebnis berechnen und dieses an den Master-Prozess zurücksenden. *tf* ist eine generalisierte Variante von *df* und arbeitet nach dem Divide & Conquer-Prinzip, d.h. Slave-Prozesse können nicht nur Ergebnisse, sondern auch neue Teilaufgaben erzeugen und an den Master-Prozess senden. *itermem* wird zur Bearbeitung von Bildern verwendet, bei denen das Ergebnis der Berechnung von einem Bild abhängt, das in einer vorherigen Iteration berechnet wurde. Ab SKIPPER-II wird die Verschachtelung der Skelette unterstützt. Jedes Skelett besitzt in SKIPPER sowohl eine deklarative als auch eine operationale Semantik. Während erstere in CAML definiert wird und die Funktionalität auf einem abstrakten Niveau beschreibt, enthält letztere eine Implementierung für die jeweilige Zielplattform. Anzumerken bleibt, dass die deklarative Semantik einerseits zum sequenziellen Debugging, andererseits zur Verifikation der operationalen Semantik, d.h. der Implementierung, verwendet werden kann. Die Beschreibung der Skelett-Topologie findet in SKIPPER ebenfalls unter Verwendung von CAML statt, benutzerdefinierte Funktionen hingegen müssen in C geschrieben werden. Das CTS generiert aus der Skelett-Topologie intern eine plattformunabhängige Zwischenrepräsentation, wobei letztere, je nach SKIPPER-Version, unterschiedlich implementiert ist (SKIPPER-0: statische Datenfluss-Graphen, SKIPPER-I: parametrische Prozess-Netzwerke, SKIPPER-II: hierarchische Task-Graphen, SKIPPER-D: Token-basierte Datenflüsse). In einem zwei-

ten Schritt wird aus dieser Zwischenrepräsentation unter Verwendung der operationalen Semantik eines Skeletts plattformabhängiger Quelltext generiert, wobei die Kommunikation in SKIPPER-I unter Verwendung von Transputer-Links bzw. einer plattformspezifischen Software (SynDEx), in SKIPPER-II mit Hilfe von MPI durchgeführt wird. SKIPPER unterstützt weder benutzerdefinierte Typen noch Mehrkernprozessoren bzw. hybride Speicherarchitekturen.

2.6.25 Fazit

Tabelle 2.7 fasst die Funktionalität der in Abschnitt 2.6 aufgelisteten Kriterien für jede der vorgestellten Skelettbibliotheken übersichtlich zusammen. Im Folgenden wird kurz auf die hieraus gewonnenen, zentralen Erkenntnisse eingegangen:

Programmiersprache Bei den ersten verfügbaren Skelettbibliotheken, wie z.B. P^{3}L, SCL, SkIE oder Skil, erfolgt die Erstellung des parallelen Programms unter Verwendung eigens für diesen Zweck entwickelter Programmiersprachen bzw. Spracherweiterungen. Die große Mehrheit, darunter Muesli, verwendet jedoch mit C++ bzw. Java objektorientierte Sprachen, die sich mittlerweile offensichtlich auch im HPC-Bereich durchsetzen konnten. An dieser Entwicklung hat die gestiegene Popularität von MPI zweifelsohne maßgeblichen Anteil, da MPI den Nachrichtenaustausch erheblich vereinfacht und in seiner ursprünglichen Form lediglich mit C, C++ und Fortran verwendet werden kann. Inzwischen gibt es aber auch für Java entsprechende Bibliotheken, z.B. mpiJava [40], MPJ [39, 86] und MPJ Express [41, 270].

Task- und Datenparallelität Während fast alle Skelettbibliotheken taskparallele Skelette implementieren, werden datenparallele Skelette nur von elf der untersuchten Bibliotheken umfassend unterstützt, u.a. von Muesli. Die Unterstützung beider Arten der Parallelität ist jedoch wichtig, um einerseits Daten- nicht durch Taskparallelität simulieren und damit Effizienzverluste in Kauf nehmen zu müssen, andererseits dem Benutzer keine spezielle Sichtweise auf dessen Problemstellung aufzuzwingen.

Schachtelbarkeit Die Komposition von Skeletten ist eine der grundlegendsten Anforderungen an eine Skelettbibliothek, da nur so gewährleistet werden kann, dass eine möglichst große Zahl an Problemstellungen mit Hilfe der implementierten Skelette modelliert werden kann. Dementsprechend wird dieses Kriterium von der Mehrheit der untersuchten Bibliotheken unterstützt, auch von Muesli. Ausnahmen bilden solche Bibliotheken, die ausschließlich datenparallele Skelette implementieren, wie z.B. DatTeL, SkeTo und Skil, da die Komposition hier keinen Sinn ergibt.

Benutzerdefinierte Typen Die Verwendung benutzerdefinierter Datentypen wird hauptsächlich von Java-basierten Bibliotheken unterstützt, da Java über einen integrierten Mechanismus zur Serialisierung von Objekten verfügt [200, S. 957–980]. C++-basierte Bibliotheken ermöglichen dies i.d.R. nicht und erzwingen somit die Verwendung von elementaren Typen bzw. Strukturen. Erfordert eine Problemstellung jedoch die Verwendung von Objekten, muss in einer Umgebung mit verteiltem Speicher die entsprechende Skelettbibliothek, z.B. Muesli, einen Serialisierungsmechanismus bereitstellen, um den Austausch von Daten zu ermöglichen.

Verteilter/gemeinsamer/hybrider Speicher Mit Ausnahme von FastFlow und Skandium skalieren sämtliche der untersuchten Skelettbibliotheken auf Parallelrechnern mit verteiltem Speicher. Im Gegensatz dazu werden Parallelrechner mit gemeinsamem Speicher lediglich von neun, solche mit hybridem Speicher sogar nur von vier Bibliotheken unterstützt, darunter Muesli. Die effiziente Unterstützung von hybriden Speicherarchitekturen ist jedoch für zukünftige Parallelrechner ein zentrales Kriterium, da eine Leistungssteigerung nicht mehr ausschließlich über die Anzahl der Knoten, sondern zusätzlich auch über die Anzahl der Rechenkerne erfolgen wird.

Kommunikation Während Java-basierte Skelettbibliotheken mit Threads [200, S. 499–527] für gemeinsamen Speicher bzw. RMI [200, S. 1121–1135] für verteilten Speicher hauptsächlich bereits in der Sprache enthaltene Mechanismen zur Kommunikation und Synchronisation verwenden, kommt bei C++-basierten Bibliotheken fast ausnahmslos MPI zum Einsatz. Letzteres verdeutlicht den hohen Verbreitungsgrad und

das Renommee von MPI als de facto Standard für den Nachrichtenaustausch auf Parallelrechnern mit verteiltem Speicher. Ob Java-basierte Skelettbibliotheken in Zukunft vermehrt auf die bereits oben erwähnten Bibliotheken mpiJava, MPJ und MPJ Express zurückgreifen oder ein MPI-ähnlicher Standard für Java etabliert wird, bleibt abzuwarten.

Tabelle 2.7: Überblick über die Funktionalität verschiedener Skelettbibliotheken. Vollständig erfüllte Kriterien sind mit •, nur teilweise erfüllte Kriterien mit ∘ markiert.

	Prog.-sprache	Taskparallelität	Datenparallelität	Schachtelbarkeit	ben.-def. Typen	vert. Speicher	gem. Speicher	hybr. Speicher	Kommunikation
ASSIST	ASSISTcl	∘	∘	•		•			TCP
Calcium	Java	•	∘	•	•	•	•		ProActive / Threads
CO_2P_3S	Java	•		•	•	•	•		RMI / Threads
DatTeL	C++		•			•	•		MPI / Threads
Eden	Eden	•	•		•	•			MPI / PVM
EPAS	C++	•		•	•	•			MPI
eSkel	C	•		•	∘	•			MPI
FastFlow	C++	•		•	•		•		Threads
HDC	Haskell	•	•		•	•			MPI
HOC-SA	Java	•	•		•	•			Globus Toolkit
JaSkel	Java	•		•	•	•	•	•	RMI + Threads
Lithium	Java	•	•	•	•	•			RMI
MALLBA	C++	∘		∘		•			MPI
Muesli	C++	•	•	•	•	•	∘	∘	MPI + OpenMP
muskel	Java	•		•		•	•		RMI
P^{3}L	P^{3}L	•	•	∘	∘	•			MPI
PAS	C++	•			•	•			MPI
QUAFF	C++	•	∘	•	∘	•			MPI
SBASCO	SBASCOcl	•		∘		•			MPI
SCL	SCL	•	•	∘		•			CS Tools
Skandium	Java	•	∘	•	•		•		Threads
SKElib	C	•	∘	•		•			TCP
SkeTo	C++		•			•	∘	∘	MPI + Threads
SkIE	SkIECL	•	•	•	∘	•	∘	∘	MPI
Skil	Skil		•		•	•			Betriebssystem
SKIPPER	C, CAML	•		•		•			MPI / SynDEx

Kapitel 3

Die Münster Skelettbibliothek Muesli

Inhalt

3.1 Einleitung

Muesli ist eine in C++ geschriebene, Template-basierte Skelettbibliothek, die die Erstellung paralleler Programme unter Beibehaltung eines sequenziellen Programmierstils ermöglicht [99, 100, 101, 102, 103, 203, 204, 248, 249, 250, 251, 252, 253]. Die Bibliothek implementiert viele der bereits in Skil (siehe Abschn. 2.6.23) umgesetzten Konzepte und wurde in ihrer ursprünglichen Version 1.0 bereits ausführlich in [203] beschrieben. Im Vergleich dazu wurden in der aktuellen Version 2.0 jedoch eine Reihe von Erweiterungen

implementiert, so dass der Forschungsstand der Skelettbibliothek wie folgt beschrieben werden kann:

- Muesli implementiert, im Gegensatz zu vielen anderen Skelettbibliotheken (siehe Abschn. 2.6), sowohl task- als auch datenparallele Skelette, wobei letztere stets auf einer verteilten Datenstruktur (siehe Abschn. 3.3) operieren. Dieser Ansatz bietet den Vorteil, dass datenparallele Operationen nicht künstlich durch den Einsatz von taskparallelen Konstrukten emuliert werden müssen, was einerseits dem Benutzer der Skelettbibliothek eine spezifische Sichtweise aufzwingen, andererseits zur Laufzeit einen unnötigen Mehraufwand erzeugen würde. Taskparallele Skelette, wie z.B. *Pipe*, *Farm* [250, 251], *Branch & Bound* [249] und *Divide & Conquer* [252, 253], werden in Form von speziellen Template-Klassen zur Verfügung gestellt und sollen an dieser Stelle nur der Vollständigkeit halber erwähnt werden. Für einen umfassenden Überblick sei auf [248] verwiesen. Datenparallele Skelette, wie z.B. *fold*, *map* und *zip*, werden hingegen in Form von Elementfunktionen einer verteilten Datenstruktur zur Verfügung gestellt (siehe Abschn. 3.4). In der aktuellen Version 2.0 der Skelettbibliothek werden Datenstrukturen für verteilte Felder, verteilte Matrizen [103] sowie verteilte dünnbesetzte Matrizen (siehe Kap. 4) bereitgestellt. Anzumerken bleibt, dass auch in Muesli, analog zu dem von P^3L implementierten Zwei-Schichten-Modell (siehe Abschn. 2.6.14), innerhalb eines taskparallelen Skeletts wiederum verteilte Datenstrukturen verwendet werden können [205].

- Sämtliche datenparallelen Skelette skalieren durch die Verwendung von MPI und OpenMP auf Parallelrechnern mit verteiltem, gemeinsamem und hybridem Speicher [102] (siehe Abschn. 2.2.2). Während MPI die Skalierbarkeit auf Parallelrechnern mit verteiltem Speicher gewährleistet, ermöglicht OpenMP die Skalierbarkeit auf Parallelrechnern mit gemeinsamem Speicher, so dass Muesli durch die kombinierte Verwendung von MPI und OpenMP auch auf Parallelrechnern mit hybridem Speicher skaliert (siehe Abschn. 3.5.1). In Übereinstimmung mit dem Grundgedanken einer Skelettbibliothek, sämtliche Details der Parallelisierung zu verbergen, ist der Einsatz von MPI und OpenMP für den Benutzer vollkommen transparent.

- Alle taskparallelen Skelette sowie verteilten Datenstrukturen abstrahieren mit Hilfe des von C++ bereitgestellten Template-Mechanismus vom Typ der Objekte, die innerhalb der Skelette verwendet werden (siehe Abschn. 3.2.2). Der zu diesem Zweck deklarierte Template-Parameter kann sowohl mit elementaren Typen und Strukturen als auch mit benutzerdefinierten Typen instanziiert werden. Um letztere mit Hilfe von MPI versenden bzw. empfangen zu können, implementiert die Skelettbibliothek einen automatisierten Serialisierungsmechanismus (siehe Abschn. 3.2.5). Dieser basiert im Wesentlichen auf der Möglichkeit, zur Kompilierzeit Vererbungsbeziehungen zu erkennen, so dass zur Laufzeit kein Mehraufwand entsteht.

- Die Skelettbibliothek implementiert einen Mechanismus, mit dem partielle Applikationen und Currying ermöglicht werden (siehe Abschn. 3.2.4). Infolgedessen sind sämtliche Skelettfunktionen überladen, so dass diesen als Argumentfunktion sowohl ein gewöhnlicher Funktionszeiger als auch eine partielle Applikation übergeben werden kann. Auf diese Weise ist die Verwendung von partiellen Applikationen für den Benutzer ebenfalls vollkommen transparent.

Anzumerken bleibt, dass mit Hilfe der Skelettbibliothek erstellte Programme ausschließlich auf Parallelrechnern kompiliert und ausgeführt werden können, die eine zum MPI-Standard 1.2 kompatible Laufzeitumgebung bereitstellen. Soll darüber hinaus zusätzlich von etwaigen Mehrkernprozessoren profitiert werden, muss auf dem Parallelrechner ein zu OpenMP 2.5 kompatibler C++-Compiler installiert sein und mit aktivierter OpenMP-Option zur Kompilierung des parallelen Programms verwendet werden (siehe Tab. 2.6, S. 35). Weitere Einschränkungen sind nicht zu beachten.

Der Rest des Kapitels ist wie folgt aufgebaut: Zunächst werden in Abschnitt 3.2 die wesentlichen Konzepte erläutert, die der Implementierung der Skelettbibliothek zu Grunde liegen. Anschließend wird in Abschnitt 3.3 auf die von Muesli bereitgestellten verteilten Datenstrukturen eingegangen, um darauf aufbauend in Abschnitt 3.4 die auf den Datenstrukturen operierenden datenparallele Skelette zu erörtern. Abschließend werden in Abschnitt 3.5 die implementierten Erweiterungen beschrieben. Abschnitt 3.6 zeigt die Ergebnisse einiger durchgeführter Laufzeitmessungen, Abschnitt 3.7 schließt das Kapitel mit einem Fazit ab.

3.2 Grundlagen

Die folgenden Abschnitte 3.2.1 bis 3.2.5 behandeln die wesentlichen Konzepte und Mechanismen, die der Implementierung der Skelettbibliothek zu Grunde liegen. Zunächst erläutert Abschnitt 3.2.1, auf welche Weise ein unter Verwendung einer verteilten Datenstruktur entwickeltes Programm auf einem Parallelrechner überhaupt skaliert. Anschließend wird in Abschnitt 3.2.2 mit dem parametrischen Polymorphismus ein Schlüsselkonzept der Skelettbibliothek erörtert. Funktionen höherer Ordnung werden kurz in Abschnitt 3.2.3 beschrieben, so dass darauf aufbauend in Abschnitt 3.2.4 die Funktionsweise partieller Applikationen sowie des Currying-Mechanismus dargelegt werden kann. Abschließend wird in Abschnitt 3.2.5 auf den implementierten Mechanismus zur Serialisierung benutzerdefinierte Objekte eingegangen.

3.2.1 Parallelisierung durch Partitionierung

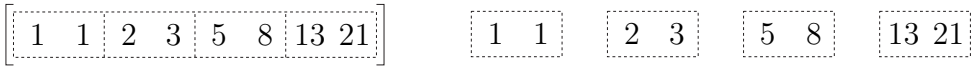
Wie bereits erwähnt, ermöglicht die Verwendung der Skelettbibliothek das Erstellen eines parallelen Programms unter Beibehaltung eines sequenziellen Programmierstils. Dies wirft jedoch unweigerlich die Frage auf, durch welchen Mechanismus dieses Programm überhaupt auf einem Parallelrechner skaliert. Die Antwort hierauf ergibt sich aus dem Zusammenspiel der beiden folgenden Aspekte:

- Verwendung des SPMD-Programmiermodells. Das Modell sieht vor, dass jeder zur Verfügung stehende Rechenknoten *dasselbe* parallele Programm ausführt, d.h. der Parallelrechner führt dasselbe parallele Programm *gleichzeitig* auf *np* Rechenknoten aus. Zu beachten ist, dass hierbei jedes Programm jedoch auf unterschiedlichen Daten operieren kann, so dass in Kombination mit einer Partitionierung der Daten (siehe unten) eine Beschleunigung der Abarbeitung stattfinden kann.

- Partitionierung der Daten. Die Elemente der verwendeten Datenstruktur werden unter den verfügbaren Prozessen aufgeteilt, so dass jeder

Prozess nur einen Teil der globalen Datenstruktur speichert und verarbeitet (siehe Abb. 3.1). Welcher Prozess für die Speicherung welcher Elemente zuständig ist, wird mit Hilfe der von MPI vergebenen Prozess-ID bestimmt (siehe Abschn. 2.3.1.1). Die Partitionierung erfolgt innerhalb des entsprechenden Konstruktors der verteilten Datenstruktur und erfordert keinerlei Kommunikation.

Durch die oben beschriebenen Aspekte kann nicht nur ein sequenzieller Programmierstil, sondern darüber hinaus auch eine globale Sichtweise auf die verteilte Datenstruktur beibehalten werden. Diese globale Sichtweise ist jedoch in Folge der Partitionierung einerseits rein virtueller Natur, andererseits eine logische Konsequenz, die sich aus dem Grundgedanken einer Skelettbibliothek ergibt, sämtliche Details der Parallelisierung vollständig vor dem Benutzer zu verbergen. Für den Benutzer entsteht so der Eindruck, als ob die Datenstruktur gar nicht verteilt ist. Infolgedessen kann dieser die Datenstruktur durch die Verwendung von Skeletten (siehe Abschn. 3.4) auch nur in Gänze manipulieren. Zwar ist auch ein dedizierter Zugriff auf die lokalen Partitionen möglich, hierbei verlässt der Benutzer jedoch die globale Sichtweise auf die Datenstruktur und muss sich der Partitionierung bewusst sein. Dieses Vorgehen kann zwar auf der einen Seite bei der Implementierung eines Algorithmus hilfreich sein, verletzt aber auf der anderen Seite den Grundgedanken einer Skelettbibliothek, da dem Benutzer Details der Parallelisierung offenbart werden.



(a) Globale Ansicht (b) Lokale Ansicht

Abbildung 3.1: Partitionierung eines Feldes der Größe $n = 8$ auf $np = 4$ Prozesse.

3.2.2 Parametrischer Polymorphismus

Bei der Implementierung einer Containerklasse ist es grundsätzlich erstrebenswert, vom Typ der zu speichernden Elemente zu abstrahieren. Dies ermöglicht der Klasse, Elemente unterschiedlichen Typs zu speichern, ohne

dass für jeden Typ eine eigenständige Implementierung vorgenommen werden muss. Dieses Konzept der generischen Programmierung wird in C++ durch sogenannte *Templates* ermöglicht [213, S. 731–802] [295]. Ein Template ist eine Art Schablone und definiert eine Familie von Klassen bzw. Funktionen, die unabhängig von konkreten Typen implementiert ist [295, S. 9]. Um ein Klassen- bzw. Funktionstemplate zu definieren, muss die entsprechende Klasse bzw. Funktion mindestens einen *Template-Parameter* deklarieren [295, S. 9 f. und 21]. Dieser abstrahiert vom konkreten Typ und kann im Gültigkeitsbereich der Klasse bzw. Funktion wie ein gewöhnlicher Typ verwendet werden [1, S. 236, §14.1.3]. In diesem Zusammenhang wird häufig auch von *parametrischem Polymorphismus* gesprochen, da Klassen bzw. Funktionen mit Hilfe eines Parameters generisch definiert werden können.

Sämtliche verteilte Datenstrukturen der Skelettbibliothek deklarieren einen Template-Parameter `T`, um vom konkreten Typ der zu speichernden Elemente zu abstrahieren. Auf diese Weise können neben elementaren Typen und Strukturen auch benutzerdefinierte Typen verwendet werden. Letzteres setzt jedoch voraus, dass die entsprechende Klasse von `msl::Serializable` erbt (siehe Abschn. 3.2.5). Problematisch ist in diesem Zusammenhang jedoch der Umstand, dass diese Vererbungsbeziehung in C++ mit sprachlichen Mitteln schlichtweg nicht ausgedrückt werden kann, d.h. der Typ von `T` kann nicht näher spezifiziert werden, wie dies z.B. in Java bei der Verwendung sogenannter *Generics* in Kombination mit gebundenen Wildcards möglich ist [200, S. 371–383]. Im Gegensatz zu Java wird in C++ bei der Instanziierung eines Templates, d.h. bei der Verwendung eines Templates mit konkreten Typen, eine typspezifische Version des Templates generiert, indem eine neue Klasse bzw. Funktion erzeugt wird, bei der jeder Template-Parameter durch den entsprechenden konkreten Typ ersetzt wurde [213, S. 745 f.]. Dieses Vorgehen weist auf der einen Seite den oben beschriebenen Nachteil auf, bietet auf der anderen Seite einen entscheidenden Vorteil, da der Template-Parameter in C++ auch mit elementaren Typen instanziiert werden kann, ohne zur Laufzeit einen Mehraufwand zu erzeugen. Zwar kann auch in Java der Typparameter mit elementaren Typen instanziiert werden, hierbei findet jedoch zur Laufzeit eine automatische Konvertierung zwischen elementaren Typen und einer entsprechenden Wrapper-Klassen statt, was im Vergleich zu C++ eine deutlich erhöhte Ausführungszeit zur Folge hat [200, S. 224 f.].

3.2.3 Funktionen höherer Ordnung

Eine Grundvoraussetzung für die Implementierung algorithmischer Skelette sind sogenannte *Funktionen höherer Ordnung*. Eine Funktion höherer Ordnung ist definiert als Funktion, die mindestens eine Funktion als Argument erwartet und/oder deren Ergebnis wiederum eine Funktion ist [206]. Konzeptionell betrachtet ist ein algorithmisches Skelett folglich nichts anderes als eine Funktion höherer Ordnung, da jedes Skelett mindestens eine Funktion als Argument erwartet, die dessen Verhalten näher spezifiziert (siehe Abschn. 3.4). Technisch umsetzen lassen sich Funktionen höherer Ordnung in C++ z.B. mit Hilfe des Adressoperators `&` und sogenannter *Funktionszeiger* [213, S. 336–339]. Ein Funktionszeiger ist ein Zeiger, der die Adresse einer Funktion speichert, wobei diese Adresse durch den Adressoperator bestimmt werden kann. Funktionszeiger können wie gewöhnliche Funktionen benutzt und als Argument und/oder Rückgabewert von Funktionen verwendet werden. Eine beispielhafte Anwendung von Funktionszeigern kann Listing 3.1 entnommen werden.

Listing 3.1: Beispielhafte Anwendung von Funktionszeigern. Das Ergebnis der `main`-Funktion ist `bar(g, 1) = bar(foo, 1) = foo(1) + 1 = 1 + 1 + 1 = 3`.

```
1  int main() {
2    int (*g)(int) = &foo;
3    return bar(g, 1);
4  }
5
6  int foo(int a) {
7    return a + a;
8  }
09 int bar(int (*f)(int), int b) {
10   return f(b) + b;
11 }
```

3.2.4 Partielle Applikation und Currying

Sei f eine n-stellige Funktion mit Argumenten a_1, a_2, …, a_n, $n \in \mathbb{N}$. Der Begriff *partielle Applikation* bezeichnet die Anwendung von f auf $k < n$ Argumente, $k \in \mathbb{N}$ [206]. Das Ergebnis dieser Anwendung ist eine $(n-k)$-stellige Funktion g mit Argumenten a_{k+1}, a_{k+2}, …, a_n, wobei die ersten k

Argumente von f an feste Werte gebunden sind. Der Prozess der Transformation einer n-stelligen Funktion in n Funktionen mit je einem Argument wird als *Currying* bezeichnet.

Listing 3.2: Anwendung des Curry-Mechanismus zur Erzeugung einer partiellen Applikation. Das Ergebnis der `main`-Funktion ist `g(1) = f(false)(1) = 1 + 1 = 2`.

```
1  int foo(bool b, int i) {
2    return b ? i : i + i;
3  }
4
5  int main(int argc, char** argv) {
6    typedef Fct2<bool, int, int, int (*)(bool, int)> F;
7    F f = curry(&foo);
8    Fct1<int, int, F::PartialAppl1> g = f(false);
9    return g(1);
10 }
```

Die Skelettbibliothek stellt mit den Klassen `Fct0`, `Fct1`, ..., `Fct6` sowie der überladenen Funktion `curry` einen Mechanismus zur Verfügung, mit dem die partielle Applikation bzw. Currying ermöglicht wird [284]. Dieser basiert im Wesentlichen darauf, einen Zeiger auf eine n-stellige Funktion f mit Hilfe einer `curry`-Funktion in ein Funktionsobjekt vom Typ `Fct`n umzuwandeln, wobei Funktionen mit bis zu $n = 6$ Argumenten unterstützt werden. Das Funktionsobjekt kapselt den Funktionszeiger und überlädt, wie der Name bereits andeutet, den Klammeroperator `()`, so dass dieses syntaktisch wie eine Funktion verwendet werden kann (siehe Lst. 3.2). Im Gegensatz zu einer gewöhnlichen Funktion kann ein Objekt vom Typ `Fct`n jedoch auf $k \leqslant n$ Argumente angewendet werden. Zu diesem Zweck überschreibt jedes Funktionsobjekt `Fct`n den Klammeroperator genau n-fach.[14] Der Klammeroperator ist derart überladen, dass an ein Funktionsobjekt `Fct`n bis zu n Argumente übergeben werden können, wobei die Reihenfolge der Argumente derjenigen von f entspricht. Falls n Argumente übergeben werden, wird die Funktion, auf die das Funktionsobjekt zeigt, unter Anwendung der übergebenen Argumente ausgewertet und das Ergebnis zurückgegeben. Falls $k < n$ Argumente übergeben werden, gibt der Klammeroperator ein

[14] Eine Ausnahme hiervon stellt die Klasse `Fct0` dar, die den Klammeroperator einmal überschreibt.

neues Funktionsobjekt `Fct`m zurück, wobei $m = n - k, m \in \mathbb{N}$. Dieses neue Funktionsobjekt kapselt nicht nur den Funktionszeiger, sondern speichert gleichzeitig auch die übergebenen k Argumente, so dass die ersten k Parameter der Funktion f an feste Werte gebunden werden.

Technisch betrachtet weisen die Funktionsobjekte eine vergleichsweise komplexe Syntax auf, da diese ausgiebig von dem in C++ verfügbaren Template-Mechanismus Gebrauch machen. Ein Funktionsobjekt vom Typ `Fct`n deklariert stets $n + 2$ Template-Parameter, wobei Parameter p_i den Typ von Argument a_i der gekapselten n-stelligen Funktion f, p_{n+1} den Rückgabetyp von f und p_{n+2} den Typ von f festlegt, $i \in \mathbb{N}$. Ein Benutzer der Skelettbibliothek kommt mit dieser Syntax für gewöhnlich jedoch nicht in Berührung, da sämtliche Skelette derart überladen sind, dass diese als Argumentfunktion sowohl einen gewöhnlichen Funktionszeiger als auch eine partielle Applikation entgegennehmen. Skelette, die mehr als eine Argumentfunktion erwarten, sind derart überladen, dass sämtliche Kombinationen von Funktionszeigern und partiellen Applikationen aufgerufen werden können. Auf diese Weise ist die Verwendung von Funktionsobjekten für den Benutzer vollständig transparent.

3.2.5 Serialisierung

Wie bereits erwähnt, deklarieren sämtliche verteilten Datenstrukturen einen Template-Parameter `T`, der den Typ der von der Datenstruktur zu speichernden Elemente festlegt (siehe Abschn. 3.2.2). Sofern für `T` nur Basistypen, wie z.B. **`int`** oder **`double`**, oder Strukturen verwendet werden, ist der Austausch mit Hilfe von MPI problemlos möglich. Benutzerdefinierte Objekte hingegen können nicht ohne weiteres kommuniziert werden, da diese u.U. nicht in einem zusammenhängenden Speicherbereich abgelegt sind, dies aber bei der Verwendung der von MPI definierten paarweisen und kollektiven Kommunikationsfunktion zwingend erforderlich ist (siehe Abschn. 2.3.2 und 2.3.3). Als Lösung bietet sich die Verwendung eines Serialisierungsmechanismus an, d.h. die Abbildung von Objekten in eine flache, sequenzielle Darstellungsform. Im Gegensatz zu Java [200, S. 957–980 und 1017–1042] verfügt C++ jedoch weder über einen integrierten Mechanismus zur Serialisierung von Objekten noch über eine Schnittstelle zur Reflexion.

Um das Senden und Empfangen von benutzerdefinierten Objekten dennoch zu ermöglichen, enthielt bereits die vorherige Version 1.8 der Skelettbibliothek einen entsprechenden Serialisierungsmechanismus [103, S. 58–61]. Dieser basiert auf der Möglichkeit, mit Hilfe des Makros `MSL_IS_SUPERCLASS` zur Kompilierzeit Vererbungsstrukturen zu erkennen und ist im Wesentlichen [30, S. 67–71] entnommen. Der Serialisierungsmechanismus[15] sieht vor, dass benutzerdefinierte Klassen von der abstrakten Oberklasse `msl::Serializable` erben und dabei die rein virtuellen Funktionen[16] `reduce`, `expand` und `getSize` überschreiben:[17]

`virtual void` `reduce(`**`void*`** `buf,` **`int`** `size) = 0`. Die Funktion serialisiert ein Objekt, indem es die Attribute des Objekts in den zusammenhängenden Speicherbereich `buf` kopiert. Zu beachten ist, dass `buf` vom Typ **`void*`** ist und `size` die Größe des Puffers in Byte angibt.

`virtual void` `expand(`**`void*`** `buf,` **`int`** `size) = 0`. Die Funktion deserialisiert ein Objekt, indem es dessen Zustand mit Hilfe von `buf` wiederherstellt. Zu beachten ist, dass `buf` vom Typ **`void*`** ist und `size` die Größe des Puffers in Byte angibt.

`virtual int` `getSize() = 0`. Die Funktion gibt die Größe des Puffers in Byte zurück, der notwendig ist, um das Objekt zu serialisieren.

Neben dem Mechanismus zur Serialisierung von Objekten stellt die Skelettbibliothek intern mit den überladenen Funktionen `MSL_Send` und `MSL_Receive` jeweils zwei Varianten von `MPI_Send` und `MPI_Recv` zur Verfügung: Während mit der einen Variante Typen versendet bzw. empfangen werden können, die nicht serialisiert werden müssen, wie z.B. elementare Typen oder Strukturen, dient die andere Variante zum Austausch von Typen,

[15] Die Klasse `Integer` implementiert den Serialisierungsmechanismus beispielhaft und kann Anhang A.3 entnommen werden.

[16] Eine rein virtuelle Funktion ist eine Funktion, die die Modifikatoren **`virtual`** und `= 0` aufweist. Das Schlüsselwort **`virtual`** setzt den Vererbungs-Polymorphismus um, sorgt also für das späte Binden. Der Modifikator `= 0` deklariert eine Funktion als abstrakt.

[17] Streng genommen ist dies nur dann erforderlich, wenn Skelette Kommunikation erfordern, wie z.B. *fold*. Da das Grundprinzip einer Skelettbibliothek jedoch darin besteht, sämtliche Details der Parallelisierung, d.h. auch die Kommunikation, zu verbergen, gilt diese Forderung pauschal.

die serialisiert werden müssen, wie z.B. benutzerdefinierte Klassen. Welche der beiden Varianten zum Einsatz kommt, wird mit Hilfe des Makros `MSL_IS_SUPERCLASS` in Abhängigkeit vom Template-Parameter `T` zur Kompilierzeit festgelegt. Auf diese Weise können sowohl elementare als auch benutzerdefinierte Typen ausgetauscht werden, ohne zur Laufzeit einen Mehraufwand zu erzeugen.

Die aktuelle Version 2.0 der Skelettbibliothek erweitert den oben beschriebenen Mechanismus, da dieser bisher nur den Austausch von genau einem Element ermöglicht hat. Für die Implementierung der kollektiven, serialisierten Kommunikationsfunktionen ist jedoch das Senden bzw. Empfangen von mehreren Elementen erforderlich, z.B. bei der Funktion `msl::allgather` (siehe Abschn. 3.5.3.2). Die erweiterten Funktionen wurden im Namensraum `msl` definiert, tragen die Bezeichnung `send` bzw. `receive` und gestatten den Austausch einer beliebigen Anzahl elementarer Typen, Strukturen und benutzerdefinierter Objekte. Die einzige Einschränkung, die an dieser Stelle hingenommen werden muss, ist der Umstand, dass alle benutzerdefinierten Objekte bei der Serialisierung eine identische Größe aufweisen müssen, d.h. das Ergebnis der Funktion `getSize` muss bei allen Objekten gleich sein.

3.3 Verteilte Datenstrukturen

Wie bereits erwähnt, stellt Muesli neben task- auch datenparallele Skelette zur Verfügung, wobei letztere als Elementfunktionen einer verteilten Datenstruktur implementiert werden. Die aktuelle Version der Skelettbibliothek stellt verteilte Datenstrukturen für Felder (siehe Abschn. 3.3.1), Matrizen (siehe Abschn. 3.3.2) sowie dünnbesetzte Matrizen (siehe Abschn. 3.3.3) bereit, die im Folgenden kurz beschrieben werden. Für eine umfassende Darstellung sei an dieser Stelle auf [103] verwiesen.

3.3.1 DistributedArray

Die Datenstruktur für verteilte Felder ist in der Klasse `DistributedArray` definiert und kann dazu benutzt werden, ein eindimensionales Feld der Länge n unter np Prozessen aufzuteilen [103, S. 16–23]. Hierbei ist zu beachten,

dass np als Zweierpotenz darstellbar sein muss und $n \bmod np = 0$. Diese Einschränkungen resultieren aus dem Umstand, dass die Kommunikation, konzeptionell betrachtet, entlang der Kanten eines Hyperwürfels stattfindet und dieser für eine korrekte Durchführung vollständig sein muss. Des Weiteren müssen sämtliche Partitionen eine identische Größe aufweisen, was die Rotation bzw. Permutation von Partitionen erheblich vereinfacht. Die Datenstruktur implementiert, neben den üblichen Skeletten *fold*, *map*, *permute*, *rotate*, *scan* und *zip*, auch spezielle Varianten von *map*. Diese erhöhen zwar nicht die Ausdrucksstärke der Skelettbibliothek, reduzieren aber den Mehraufwand, der durch aufeinanderfolgende Aufrufe von *map* und einem anderen Skelett, wie z.B. *fold*, entsteht [204].

3.3.2 DistributedMatrix

Die Datenstruktur für verteilte Matrizen ist in der Klasse `DistributedMatrix` definiert und kann dazu benutzt werden, ein zweidimensionales Feld der Größe $n \times m$ unter np Prozessen aufzuteilen [103, S. 23–31]. Zu diesem Zweck wird die Matrix in Blöcke der Größe $\frac{n}{rows} \times \frac{m}{cols}$ aufgeteilt, wobei $rows$ und $cols$ dem Konstruktor als Parameter übergeben werden müssen, $rows, cols \in \mathbb{N}$. Für die Parameter n, m, $rows$ und $cols$ gelten folgende Einschränkungen:

- $rows \leqslant n$ und $cols \leqslant m$, wobei $n \bmod rows = m \bmod cols = 0$.
- $rows \cdot cols = np$, wobei np als Zweierpotenz darstellbar sein muss.

Diese Restriktionen ergeben sich, da die Kommunikation, analog zur Klasse `DistributedArray`, entlang der Kanten eines Hyperwürfels stattfindet und dieser für eine korrekte Durchführung vollständig sein muss. Des Weiteren muss jedem Prozess genau ein Block zugewiesen sein, wobei sämtliche Blöcke eine identische Größe aufweisen müssen, was deren Rotation bzw. Permutation erheblich vereinfacht. Die Datenstruktur implementiert, neben den üblichen Skeletten *fold*, *map*, *permute* und *zip*, auch spezielle Varianten von *map*. Diese erhöhen zwar nicht die Ausdrucksstärke der Skelettbibliothek, reduzieren aber den Mehraufwand, der durch aufeinander folgende Aufrufe von *map* und einem anderen Skelett, wie z.B. *fold*, entsteht [204].

Darüber hinaus werden mit *permuteRows* und *permuteCols* Skelette zur zeilen- bzw. spaltenweisen Rotation der Blöcke bereitgestellt.

3.3.3 DistributedSparseMatrix

Die Datenstruktur für verteilte, dünnbesetzte Matrizen ist in der Klasse `DistributedSparseMatrix` definiert und kann dazu benutzt werden, ein zweidimensionales Feld der Größe $n \times m$ unter np Prozessen aufzuteilen [99]. Zu diesem Zweck wird die Matrix in Submatrizen der Größe $r \times c$ aufgeteilt, wobei r und c dem Konstruktor als Parameter übergeben werden müssen, $r, c \in \mathbb{N}$. Hierbei ist nur zu beachten, dass $r \leqslant n$ und $c \leqslant m$, weitere Restriktionen sind nicht zu berücksichtigen. Die Klasse implementiert einen zweistufigen Kompressionsmechanismus, der von dem Umstand profitiert, dass die Matrix nur wenige von Null verschiedene Elemente speichert, was zu einer deutlichen Reduktion des Speicherplatzverbrauchs sowie der Laufzeit führt. Darüber hinaus wurde ein flexibler Lastverteilungsmechanismus implementiert, der die Zuordnung zwischen Submatrizen und Prozessen vornimmt. Beide Mechanismen sind benutzerdefiniert erweiterbar. Für eine umfassende Darstellung sei an dieser Stelle auf Kapitel 4 verwiesen.

3.4 Datenparallele Skelette

Sämtliche datenparallelen Skelette werden von Muesli als Elementfunktionen einer verteilten Datenstruktur (siehe Abschn. 3.3) zur Verfügung gestellt und erwarten mindestens eine benutzerdefinierte Argumentfunktion f, die das Verhalten des Skeletts näher spezifiziert. Die Semantik eines Skeletts wird durch den Namen der Elementfunktion signalisiert, wobei dieser stets aus einem Präfix und einem optionalen Suffix zusammengesetzt ist. Das Präfix gibt an, welches Rechen- bzw. Kommunikationsmuster das Skelett implementiert, z.B. *fold*, *map* oder *zip*. Das Suffix wiederum legt fest, um welche Variante eines Skeletts es sich handelt:

Index signalisiert, dass f zusätzlich zum eigentlichen Element dessen Index übergeben bekommt. Hierbei ist wichtig, dass die Anzahl der Indizes

von der konkreten Datenstruktur abhängig ist. Während bei einem verteilten Feld lediglich ein Index übergeben wird, werden bei einer verteilten (dünnbesetzten) Matrix mit einem Zeilen- und einem Spaltenindex zwei Indizes an f übergeben.

InPlace signalisiert, dass das Ergebnis der Skelettoperation direkt in der Datenstruktur gespeichert wird, auf der das Skelett aufgerufen wurde.

IndexInPlace signalisiert, dass das Skelett beide der oben beschriebenen Eigenschaften aufweist, d.h. sowohl f zusätzlich zum eigentlichen Element dessen Index übergibt als auch das Ergebnis der Skelettoperation direkt in der Datenstruktur speichert, auf der das Skelett aufgerufen wurde.

Partitions signalisiert, dass nicht einzelne Elemente, sondern die vollständige Partition als Ganzes an die benutzerdefinierte Argumentfunktion f übergeben wird.

Ein Skelett, das kein Suffix aufweist, fertigt zunächst eine Kopie der Datenstruktur an, auf der dieses aufgerufen wurde, führt anschließend die Skelettoperation auf der erzeugten Kopie durch und gibt letztere abschließend zurück. Hierbei ist wichtig, dass der Zustand der Datenstruktur, auf der das Skelett aufgerufen wurde, nicht verändert wird. Anzumerken bleibt, dass die oben beschriebenen Varianten nicht von jedem Skelett verfügbar sind. So existiert z.B. auf Grund der Semantik von *fold* (siehe Abschn. 3.4.2) nur die Variante *foldIndex*. Die folgenden Abschnitte 3.4.1 bis 3.4.5 beschreiben die Semantik der Skelette *count*, *fold*, *map*, *permute* und *zip*.

3.4.1 count

Das Skelett *count* zählt die Elemente einer Datenstruktur in Abhängigkeit von der benutzerdefinierten Argumentfunktion *f*. Zu diesem Zweck muss f für jedes Element einen Wert vom Typ `boolean` zurückgeben. Dieser signalisiert, ob das entsprechende Element gezählt werden soll oder nicht. Eine beispielhafte Anwendung des Skeletts kann Abbildung 3.2 entnommen werden.

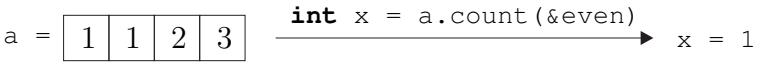


Abbildung 3.2: Anwendung des Skeletts *count* auf das verteilte Feld `a`. Die Funktion `even` erwartet ein Argument vom Typ **`int`** und gibt **`true`** zurück, falls dieses restlos durch zwei teilbar ist. Anderenfalls gibt die Funktion **`false`** zurück.

3.4.2 fold

Das Skelett *fold* reduziert sämtliche Elemente einer Datenstruktur durch sukzessives Anwenden der Argumentfunktion f zu einem singulären Element. Um zu gewährleisten, dass das Berechnungsergebnis unabhängig von der Aufteilung der verteilten Datenstruktur ist, muss f sowohl assoziativ als auch kommutativ sein. Eine beispielhafte Anwendung des Skeletts kann Abbildung 3.3 entnommen werden.

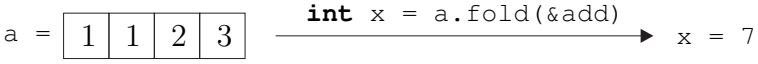


Abbildung 3.3: Anwendung des Skeletts *fold* auf das verteilte Feld `a`. Die Funktion `add` erwartet zwei Argumente vom Typ **`int`** und gibt deren Summe zurück.

3.4.3 map

Das Skelett *map* wendet auf jedes Element e einer Datenstruktur sukzessive die Argumentfunktion f an und ersetzt dieses durch $f(e)$. Eine beispielhafte Anwendung des Skeletts kann Abbildung 3.4 entnommen werden.

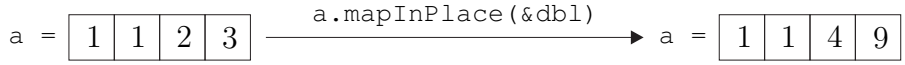


Abbildung 3.4: Anwendung des Skeletts *mapInPlace* auf das verteilte Feld `a`. Die Funktion `dbl` erwartet ein Argument vom Typ **`int`** und gibt den doppelten Wert zurück.

3.4.4 permute

Das Skelett *permute* vertauscht die Elemente einer Datenstruktur, indem mit Hilfe der Argumentfunktion f für jedes Element ein neuer Index berechnet wird. Hierbei ist zu beachten, dass f bijektiv sein muss, damit jede Position von genau einem Element besetzt wird. Eine beispielhafte Anwendung des Skeletts kann Abbildung 3.5 entnommen werden.

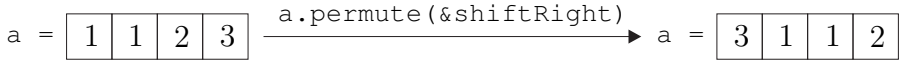


Abbildung 3.5: Anwendung des Skeletts *permute* auf das verteilte Feld `a`. Die Funktion `shiftRight` erwartet als Argument einen Index `i` und gibt als Ergebnis `(i + 1) % n` zurück, wobei `n` der Größe des verteilten Feldes entspricht.

3.4.5 zip

Das Skelett *zip* vereinigt zwei Datenstrukturen, indem auf jedes Paar korrespondierender Elemente sukzessive die Argumentfunktion f angewendet wird. Hierbei ist wichtig, dass die Datenstrukturen identisch dimensioniert sind. Eine beispielhafte Anwendung des Skeletts kann Abbildung 3.6 entnommen werden.

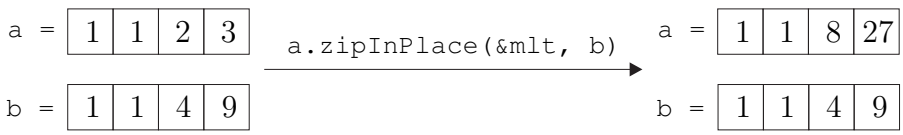


Abbildung 3.6: Anwendung des Skeletts *zipInPlace* auf das verteilte Feld `a`. Die Funktion `mlt` erwartet zwei Argumente vom Typ **`int`** und gibt deren Produkt zurück.

3.5 Erweiterungen

Die folgenden Abschnitte 3.5.1 bis 3.5.4 behandeln wesentliche Erweiterungen der Skelettbibliothek, die neben der Implementierung der verteilten Da-

tenstruktur für dünnbesetzte Matrizen vorgenommen wurden. Die wichtigste Neuerung stellt die Unterstützung von Mehrkernprozessoren bzw. hybriden Speicherarchitekturen dar und wird ausführlich in Abschnitt 3.5.1 beschrieben. Damit einhergehend wird in Abschnitt 3.5.2 die Funktionsweise der OpenMP-Abstraktionsschicht OAL erläutert. Anschließend werden in Abschnitt 3.5.3 kollektive Kommunikationsfunktionen vorgestellt, die den Austausch von serialisierbaren, benutzerdefinierten Objekten ermöglichen und gleichzeitig die Schachtelbarkeit von taskparallelen Skeletten und verteilten Datenstrukturen sicherstellen. Abschließend geht Abschnitt 3.5.4 auf diverse kleinere Ergänzungen ein.

3.5.1 Unterstützung von Mehrkernprozessoren

Die Unterstützung von Mehrkernprozessoren stellt die mit Abstand wichtigste Erweiterung der Skelettbibliothek dar. Im Gegensatz zur vorherigen Version 1.8, die durch die Verwendung des MPI-Standards (siehe Abschn. 2.3) auf Mehrprozessorsysteme mit Einkernprozessoren angepasst war, ermöglicht die aktuelle Version 2.0 der Skelettbibliothek durch den zusätzlichen Einsatz von OpenMP (siehe Abschn. 2.4) die effiziente Programmierung von Mehrprozessorsystemen mit Mehrkernprozessoren. M.a.W. erlaubt die neue Version der Skelettbibliothek die Programmierung von Parallelrechnern mit hybrider Speicherarchitektur (siehe Abschn. 2.2.2.3), während die bisherige Version auf die Programmierung von Parallelrechnern mit verteilter Speicherarchitektur (siehe Abschn. 2.2.2.2) angepasst war. Zwar können, indem pro Rechenkern ein Prozess gestartet wird, Mehrkernprozessoren auch unter alleiniger Verwendung von MPI programmiert werden, dieses Vorgehen vernachlässigt jedoch die spezifischen Vorteile einer Architektur mit gemeinsamem Speicher, so dass zwangsläufig Effizienzverluste auftreten. Das Feature zur Unterstützung von Mehrkernprozessoren besitzt mehrere wichtige Eigenschaften, auf die im Folgenden genauer eingegangen wird:

- Analog zum Konzept einer Skelettbibliothek ist die Verwendung von OpenMP für den Benutzer vollkommen transparent, da sämtliche Details der parallelen Verarbeitung innerhalb der Skelettfunktionen gekapselt werden. Der Benutzer muss folglich zur Erstellung eines pa-

rallelen Programms weiterhin weder MPI noch OpenMP beherrschen, so dass ein sequenzieller Programmierstil beibehalten werden kann.

- Aus der Transparenzeigenschaft ergibt sich, dass bestehende, unter Verwendung von Muesli implementierte parallele Programme lediglich neu kompiliert werden müssen, um auf Mehrkernsystemen zu skalieren, d.h. Änderungen am Quelltext sind nicht notwendig. Bei der Kompilierung muss lediglich darauf geachtet werden, die entsprechende Compileroption zu aktivieren (siehe Tab. 2.6, S. 35).

- Die Unterstützung von Mehrkernprozessoren hat *optionalen* Charakter, d.h. die Skelettbibliothek kann weiterhin auch auf Mehrprozessorsystemen mit Einkernprozessoren verwendet werden. Dies zu gewährleisten ist jedoch nicht ohne weiteres möglich, da der Aufruf von OpenMP-Bibliotheksfunktionen stellenweise unumgänglich ist und diese, im Gegensatz zu Pragmas, vom Compiler nicht ignoriert werden dürfen. Folglich können Compiler, die kein OpenMP unterstützen oder bei denen die OpenMP-Option nicht aktiviert wurde, entsprechende Quelltexte nicht übersetzen. Eine Möglichkeit, diese Einschränkung zu umgehen, wird in Abschnitt 3.5.2 erläutert.

Damit das vom Benutzer unter Verwendung der Skelettbibliothek geschriebene parallele Programm auf einem Mehrkernsystem skaliert, muss zum einen der Quelltext mit einem OpenMP-fähigen Compiler übersetzt, zum anderen auf dem Parallelrechner vor der Ausführung des Programms u.U. die sogenannte *CPU-Affinität* deaktiviert werden:

- Die Aktivierung der OpenMP-Erweiterungen der Skelettbibliothek ist denkbar einfach zu handhaben und erfordert keinerlei Änderungen am Quelltext. Dieser muss lediglich von einem OpenMP-fähigen Compiler mit aktivierter OpenMP-Option übersetzt werden (siehe Tab. 2.6, S. 35), weitere Maßnahmen sind nicht erforderlich. Soll die OpenMP-Unterstützung der Skelettbibliothek deaktiviert werden, so muss das entsprechende Programm erneut kompiliert werden, wobei die entsprechende Option zur Aktivierung der OpenMP-Erweiterung bei der Verwendung eines OpenMP-fähigen Compilers *nicht* verwendet werden darf. Wird hingegen ein Compiler verwendet, der grundsätzlich

kein OpenMP unterstützt, sind bei der Kompilierung keine weiteren Maßnahmen erforderlich.

- Die CPU-Affinität beeinflusst die Zuordnung von Prozessen zu Rechenknoten. Ist diese deaktiviert, wird pro Rechenknoten *ein* MPI-Prozess gestartet, d.h. auf jedem Rechenknoten wird zunächst nur ein Rechenkern in Anspruch genommen. Die weiteren Rechenkerne des Knotens kommen bei der Verwendung datenparalleler Skelette zum Einsatz, wenn nach dem Fork-Join-Prinzip (siehe Abschn. 2.4.1.3) zusätzliche Threads erzeugt werden. Auf diese Weise skaliert die Skelettbibliothek einerseits durch den Einsatz von MPI mit den verfügbaren Rechenknoten, andererseits durch die Verwendung von OpenMP mit den verfügbaren Rechenkernen.

Listing 3.3: Beispielhaftes Programm zur Verdeutlichung der Ressourcennutzung.

```
1  void main(int argc, char** argv) {
2    Muesli::InitSkeletons(argc, argv);
3    DistributedArray<int> da(32, &init);
4    da.set(0, 2);
5    da.mapInPlace(&sqr);
6    da.show();
7    Muesli::TerminateSkeletons();
8  }
9
10 int init(int i) {
11   return i;
12 }
13 int sqr(int x) {
14   return x * x;
15 }
```

Das folgende Beispiel verdeutlicht die Ressourcennutzung während der Ausführung eines mit Hilfe von Muesli implementierten parallelen Programms (siehe Lst. 3.3). Letzteres wird unter Verwendung von $np = 4$ Prozessen gestartet, wobei jeder Prozess in Folge der deaktivierten CPU-Affinität auf einem separaten Knoten des ZIVHPC (siehe Abschn. 2.5) ausgeführt wird und dabei zu Beginn lediglich einen Rechenkern beansprucht. Das Programm erzeugt zunächst ein 32-elementiges verteiltes Feld, d.h. jeder Prozess speichert lokal $32 \div 4 = 8$ Elemente, und initialisiert die einzelnen Elemente mit Hilfe der Funktion `init` (siehe Z. 3 und 10 ff.). Hierbei ist zu beachten,

dass die Ausführung des Konstruktors durch eine OpenMP-Direktive beschleunigt wird. Infolgedessen erzeugt das OpenMP-Laufzeitsystem für die Dauer der Ausführung des Konstruktors nach dem Fork-Join-Prinzip sieben zusätzliche Threads, so dass der Konstruktor auf jedem der $np = 4$ Knoten parallel von $nt = 8$ Threads bearbeitet wird (siehe Tab. 3.1). Nach der Abarbeitung des Konstruktors werden die zusätzlich erzeugten Threads zerstört, so dass die Ausführung des Programms lediglich vom Master-Thread fortgesetzt wird. Nach der Initialisierung des verteilten Feldes wird dessen erstes Element mit Hilfe der Funktion `set` auf den Wert 2 gesetzt (siehe Z. 4). Da es sich bei der Funktion um kein datenparalleles Skelett, sondern um eine rein sequenzielle Routine handelt, kann hier durch die Verwendung von OpenMP-Direktiven keine Beschleunigung erzielt werden. Folglich wird `set` auf jedem Knoten von einem Rechenkern ausgeführt, die weiteren sieben Rechenkerne jedes Knotens sind inaktiv. Anschließend werden sämtliche Elemente des Feldes mit Hilfe des Skeletts `mapInPlace` und der Funktion `sqr` quadriert (siehe Z. 5 und 13 ff.). Analog zum Konstruktor wird die Ausführung von `mapInPlace` durch die Verwendung von OpenMP-Direktiven beschleunigt, so dass auf jedem der vier Knoten jeweils acht Rechenkerne verwendet werden. Abschließend werden die Elemente des verteilten Feldes auf der Konsole ausgegeben (siehe Z. 6). Da es sich bei `show`, analog zu `set`, ebenfalls um kein datenparalleles Skelett handelt, wird auch hier pro Knoten lediglich ein Rechenkern verwendet.

Tabelle 3.1: Ressourcennutzung des in Listing 3.3 aufgeführten Programms.

Befehl	np	nt
`DistributedArray<int> da(32, &init);`	4	8
`da.set(0, 2);`	4	1
`da.mapInPlace(&sqr);`	4	8
`da.show();`	4	1

3.5.2 Die OpenMP-Abstraktionsschicht OAL

Wie bereits erwähnt, besitzt die Unterstützung von Parallelrechnern mit Mehrkernprozessoren optionalen Charakter, d.h. die Skelettbibliothek kann

weiterhin auch auf Einkernprozessoren ausgeführt werden. Um dies zu gewährleisten, wurde eine zusätzliche Abstraktionsschicht definiert. Dazu deklariert die Header-Datei `OAL.h` in einem Namensraum mit der Bezeichnung `oal` drei Hilfsfunktionen, die in der Datei `OAL.cpp` definiert sind (siehe Lst. 3.4). Hier wird zunächst auf das bereits erwähnte Makro `_OPENMP` zurückgegriffen (siehe Abschn. 2.4), um in Kombination mit den Präprozessor-Anweisungen **`#ifdef`** und **`#endif`** die Header-Datei `omp.h` bedingt zu inkludieren (siehe Z. 3 ff.). Das Makro wird von einem Compiler definiert, falls

Listing 3.4: Quelltext der Datei `OpenMP.cpp`. Zu erkennen sind die bedingte **`include`**-Anweisung sowie die Wrapper für die OpenMP-Bibliotheksfunktionen.

```
1  #include "OAL.h"
2
3  #ifdef _OPENMP
4  #include "omp.h"
5  #endif
6
7  int oal::getMaxThreads() {
8    #ifdef _OPENMP
9    return omp_get_max_threads();
10   #else
11   return 1;
12   #endif
13 }
14
15 void oal::setNumThreads(int nt) {
16   #ifdef _OPENMP
17   omp_set_num_threads(nt);
18   #endif
19 }
20
21 int oal::getThreadNum() {
22   #ifdef _OPENMP
23   return omp_get_thread_num();
24   #else
25   return 0;
26   #endif
27 }
```

dieser OpenMP unterstützt und die entsprechende Option aktiviert wurde. Falls ein Compiler OpenMP nicht unterstützt oder die entsprechende Option nicht aktiviert wurde, ist auch das Makro nicht definiert, so dass die Header-Datei `omp.h` nicht inkludiert wird.

Die bereits erwähnten Hilfsfunktionen `oal::getMaxThreads`, `oal::getThreadNum` und `oal::setNumThreads` machen von dem `_OPENMP`-Makro ebenfalls Gebrauch und dienen im Wesentlichen als Wrapper für die entsprechenden OpenMP-Bibliotheksaufrufe. Die Idee ist, stets einen der implementierten Wrapper statt der OpenMP-Funktion zu benutzen. Falls OpenMP unterstützt wird, ruft der Wrapper die entsprechende OpenMP-Funktion auf. Wird OpenMP hingegen nicht unterstützt oder ist deaktiviert, verhält sich jeder Wrapper so, als ob lediglich ein singulärer Thread ausgeführt werden kann. In diesem Fall besteht ein Thread-Team aus einem Thread, der stets die Nummer 0 erhält (siehe Z. 7–13 und 21–27). Das Festlegen der Größe eines Thread-Teams ergibt daher keinen Sinn, so dass ein entsprechender Aufruf ignoriert wird (siehe Z. 15–19).

Die vorgenommene Implementierung ist aus mehreren Gründen vorteilhaft. Zum einen muss die bedingte **`include`**-Anweisung für die Header-Datei `omp.h` nur einmalig in der Datei `OAL.cpp` erfolgen. Sämtliche anderen Dateien, die die implementierte OpenMP-Abstraktionsschicht nutzen wollen, müssen lediglich die Datei `OAL.h` inkludieren. Zum anderen erzeugt der gewählte Ansatz zur Laufzeit keinen Mehraufwand, da der Präprozessor zur Compilezeit überprüft, ob das Makro `_OPENMP` definiert ist und damit irrelevante Teile des Quelltextes überspringen kann [1, S. 302, §16.1.6]. Zu guter Letzt kann die Header-Datei `OAL.h` flexibel erweitert werden, falls weitere Wrapper für OpenMP-Bibliotheksfunktionen benötigt werden.

3.5.3 Kollektive, serialisierte Kommunikationsfunktionen

Bei der Verwendung von taskparallelen Skeletten findet der Nachrichtenaustausch i.d.R. ausschließlich mit Hilfe von paarweisen Kommunikationsfunktionen statt (siehe Abschn. 2.3.2). Im Gegensatz dazu benötigen datenparallele Skelette für gewöhnlich kollektive Kommunikationsfunktionen,

um Daten unter allen beteiligten Prozessen auszutauschen. Zwar definiert der MPI-Standard eine Vielzahl kollektiver Kommunikationsfunktionen (siehe Abschn. 2.3.3), diese können jedoch nur zum Senden bzw. Empfangen primitiver Datentypen verwendet werden. Um dennoch den Austausch benutzerdefinierter Objekte zu ermöglichen und gleichzeitig die Schachtelbarkeit von task- und datenparallelen Skeletten zu gewährleisten, wurden die Funktionen `MPI_Bcast`, `MPI_Allgather` und `MPI_Allreduce` auf Basis der serialisierten, paarweisen Kommunikationsfunktionen `msl::send` und `msl::receive` (siehe Abschn. 3.2.5) neu implementiert. Die entsprechenden Funktionen wurden im Namensraum `msl` definiert und tragen die Bezeichnung `broadcast`, `allgather` und `allreduce`. Allen ist gemein, dass diese einen Template-Parameter `T` deklarieren, der den Typ der zu versendenden bzw. zu empfangenden Daten festlegt. Anzumerken bleibt, dass die Verwendung von MPI-Gruppen und Kommunikatoren zur Gewährleistung der Schachtelbarkeit von task- und datenparallelen Skeletten nicht möglich ist, da an der Erzeugung einer neuen Gruppe *sämtliche* Prozesse beteiligt sein müssen, d.h. nicht nur diejenigen Prozesse, die Teil der neuen Gruppe sein sollen (siehe Abschn. 2.3.1.4 und 2.3.1.5). In der aktuellen Version 2.0 der Skelettbibliothek entscheidet aber jeder Prozess unabhängig von anderen Prozessen, welchen Teil des Programms dieser ausführt, d.h. ein Prozess besitzt keine Informationen über die Aktivitäten der anderen Prozesse, so dass eine gemeinsame Erzeugung einer neuen MPI-Gruppe nicht möglich ist. Die folgende Aufzählung listet die von den neu implementierten Kommunikationsfunktionen definierten Argumente auf und erläutert deren Semantik:

- Das Feld `T* sbuf` dient als Sendepuffer und enthält folglich die zu versendenden Daten. Die Anzahl der Elemente im Sendepuffer wird über den Parameter `count` festgelegt, der standardmäßig auf 1 gesetzt ist. Anzumerken bleibt, dass `sbuf` zum Versenden einer beliebigen Anzahl primitiver Datentypen oder serialisierbarer Objekte verwendet werden kann.

- Das Feld `T* rbuf` dient als Empfangspuffer und enthält nach Abschluss der Kommunikationsoperation folglich die empfangenen Daten. Die Anzahl der vom Empfangspuffer zu speichernden Elemente ist von der Semantik der Kommunikationsfunktion abhängig und wird im entsprechenden Abschnitt erläutert.

- Das Feld `int* ids` enthält die Ränge (IDs)der MPI-Prozesse, die an der kollektiven Kommunikationsoperation beteiligt sind. Die Anzahl der Elemente in `ids` wird über den Parameter `np` festgelegt. Hierbei ist wichtig, dass `np` nicht als Zweierpotenz darstellbar sein muss, d.h. der konzeptionell zur Abbildung der Kommunikation verwendete Hyperwürfel darf unvollständig sein.

- `idRoot` legt die ID des MPI-Prozesses fest, der als Wurzelprozess der Kommunikationsoperation agiert und wird lediglich in der Funktion `broadcast` verwendet. Hier wird vorausgesetzt, dass `idRoot` in dem übergebenen Feld `ids` enthalten ist.

- Die Argumentfunktion `f` dient zur Reduktion der empfangenen Elemente und wird lediglich in der Funktion `allreduce` verwendet.

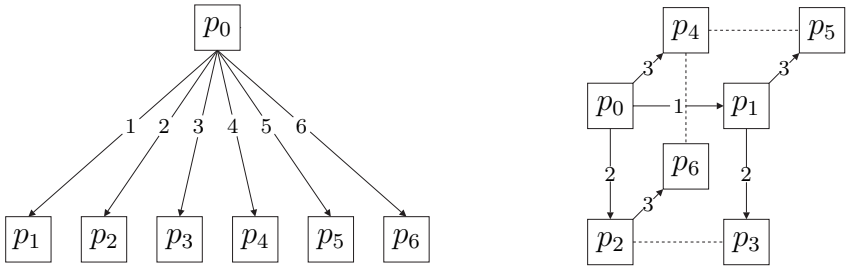
3.5.3.1 broadcast

Die Funktion `msl::broadcast` ist das semantische Pendant zu `MPI_Bcast` (siehe Abschn. 2.3.3.1), besitzt jedoch eine abweichende Signatur:

```
template<typename T>
static void broadcast(T* const sbuf, int* const ids,
                      int np, int idRoot, int count = 1)
```

Das Versenden von `sbuf` an alle beteiligten Prozesse findet, konzeptionell betrachtet, entlang der Kanten eines *Hyperwürfels* statt. Zu Beginn der Kommunikationsoperation besitzt lediglich der Wurzelprozess das zu versendende Element. Folglich müssen insgesamt $np-1$ Nachrichten versendet werden, damit jeder Prozess über das zu verteilende Datum verfügt. Bei der Verwendung eines sequenziellen Ansatzes würde der Wurzelprozess das zu versendende Element nacheinander an jeden Prozess senden, so dass in der Summe $np-1$ Kommunikationsschritte notwendig wären (siehe Abb. 3.7a). Diese Vorgehensweise ist jedoch vergleichsweise ineffizient, da hierbei lediglich der Wurzelprozess als Sender agiert und somit vernachlässigt wird, dass diejenigen Prozesse, die das zu versendende Element bereits empfangen haben, ebenfalls als Sender agieren können. Der implementierte Algorithmus

nutzt diesen Umstand aus, um so die Verbreitung des Datums zu beschleunigen. Zu diesem Zweck werden die beteiligten Prozesse gedanklich als Hyperwürfel angeordnet, die Kommunikation findet entlang der Kanten dieses Würfels statt (siehe Abb. 3.7b). In jedem Kommunikationsschritt $s \in \mathbb{N}$ können folglich 2^{s-1} Nachrichten versendet werden, so dass in der Summe $\lceil \log_2 np \rceil$ Kommunikationsschritte notwendig sind.



(a) Sequenzieller Ansatz (b) Hyperwürfel-Ansatz

Abbildung 3.7: Mögliche Kommunikationsschemata zur Durchführung eines Broadcast. Nachrichten werden entlang der gerichteten Kanten versendet, der Kommunikationsschritt ist auf der jeweiligen Kante vermerkt. p_0 ist der Wurzelprozess.

3.5.3.2 allgather

Die Funktion `msl::allgather` ist das semantische Pendant zu `MPI_Allgather` (siehe Abschn. 2.3.3.3), besitzt jedoch eine abweichende Signatur:

```
template<typename T>
static void allgather(T* const sbuf, T* const rbuf,
                      const int* const ids, int np,
                      int count = 1)
```

Analog zur Funktion `msl::broadcast` werden die beteiligten Prozesse gedanklich als Hyperwürfel angeordnet, entlang dessen Kanten die Kommunikation stattfindet (siehe Abb. 3.7b). Der Austausch der Elemente wird in zwei Phasen durchgeführt:

1. Durchführung einer Reduce-Operation. Konzeptionell wird dabei die Dimension des Hyperwürfels in jedem Kommunikationsschritt um eins

verringert, so dass nach $\lceil \log_2 np \rceil$ Kommunikationsschritten Prozess p_0 über sämtliche Elemente verfügt. Hierbei ist wichtig, dass p_0 die Elemente analog zur Reihenfolge der Prozess-IDs im Feld `ids` sammelt, was durch die Implementierung sichergestellt ist. Anzumerken bleibt, dass für die Durchführung der Reduce-Operation kein zusätzlicher Speicherplatz für etwaige Puffer benötigt wird.

2. Durchführung einer Broadcast-Operation. p_0 sendet mit Hilfe von `msl::broadcast` die gesammelten Elemente an die restlichen $np-1$ Prozesse, so dass nach $\lceil \log_2 np \rceil$ Kommunikationsschritten alle Prozesse über alle Elemente verfügen.

In Summe werden folglich $\lceil \log_2 np \rceil + \lceil \log_2 np \rceil = 2 \cdot \lceil \log_2 np \rceil$ Kommunikationsschritte zur Durchführung der Funktion benötigt.

3.5.3.3 allreduce

Die Funktion `msl::allreduce` ist das semantische Pendant zu `MPI_Allreduce` (siehe Abschn. 2.3.3.5), besitzt jedoch eine abweichende Signatur:

```
template<typename T, typename F>
static void allreduce(T* const sbuf, T* const rbuf,
                      int* const ids, int np,
                      Fct2<T, T, T, F> f, int count = 1)
```

Analog zur Funktion `msl::broadcast` werden die beteiligten Prozesse gedanklich als Hyperwürfel angeordnet, entlang dessen Kanten die Kommunikation stattfindet (siehe Abb. 3.7b). Der Austausch der Elemente wird in zwei Phasen durchgeführt:

1. Durchführung einer Reduce-Operation. Konzeptionell wird dabei die Dimension des Hyperwürfels in jedem Kommunikationsschritt um eins verringert, so dass nach $\lceil \log_2 np \rceil$ Kommunikationsschritten Prozess p_0 das globale Reduktionsergebnis besitzt. Hierbei ist wichtig, dass jeder Prozess nach jedem Kommunikationsschritt das erhaltene Datum und sein lokal verfügbares Element mit Hilfe der Argumentfunktion f faltet und dieses Ergebnis an den nächsten Prozess weiterleitet. Diese Vorgehensweise stellt einerseits sicher, dass p_0 letztlich das korrekte

Reduktionsergebnis berechnen kann, und ist andererseits effizient, da kein zusätzlicher Speicherplatz für etwaige Puffer benötigt wird.

2. Durchführung einer Broadcast-Operation. p_0 sendet mit Hilfe von `msl::broadcast` das globale Reduktionsergebnis an die restlichen $np-1$ Prozesse, so dass nach $\lceil \log_2 np \rceil$ Kommunikationsschritten alle Prozesse über das globale Reduktionsergebnis verfügen.

In Summe werden folglich, analog zur Funktion `msl::allgather`, $\lceil \log_2 np \rceil + \lceil \log_2 np \rceil = 2 \cdot \lceil \log_2 np \rceil$ Kommunikationsschritte zur Durchführung der Funktion benötigt.

3.5.4 Diverses

Im Folgenden werden zusätzliche kleinere Ergänzungen der Skelettbibliothek aufgeführt. Abschnitt 3.5.4.1 erläutert zunächst eine neu definierte Oberklasse für sämtliche verteilten Datenstrukturen. Abschnitt 3.5.4.2 beschreibt die Organisation des Quelltextes, der grundsätzlich in Header- und Quelldateien getrennt ist. Abschnitt 3.5.4.3 geht anschließend auf die neu definierte Klasse `Muesli` ein. Abschließend erörtert Abschnitt 3.5.4.4 einige Erweiterungen des Serialisierungsmechanismus.

3.5.4.1 DistributedDataStructure

Im Sinne eines objektorientierten Designs wird mit der Klasse `DistributedDataStructure` eine gemeinsame Oberklasse für sämtliche verteilten Datenstrukturen bereitgestellt (siehe Lst. 3.5). Die Klasse stellt im Wesentlichen drei Attribute zur Verfügung, die zur Speicherung der Prozess-ID, der (globalen) Größe der verteilten Datenstruktur sowie der Größe der lokalen Partition bestimmt sind (siehe Z. 5). Den Attributen entsprechende Zugriffsfunktionen sind ebenfalls definiert (siehe Z. 11 ff.). Anzumerken bleibt, dass der Konstruktor der Klasse mit dem Modifikator **`protected`** deklariert ist. Der Modifikator bewirkt, dass der Konstruktor lediglich in der Klasse selbst sowie in deren Unterklassen sichtbar ist und verhindert, dass von der Klasse `DistributedDataStructure` Objekte instanziiert werden können, da der Konstruktor ausschließlich von Unterklassen aufgerufen werden kann

[1, S. 175, §11.1]. Alternativ hätte auch der Destruktor der Klasse als rein virtuell deklariert werden können, so dass die Klasse abstrakt gewesen wäre.

Listing 3.5: Deklaration der Klasse `DistributedDataStructure`.

```
1  class DistributedDataStructure {
2
3  protected:
4
5    int pid;    int n;    int nLocal;
6
7    DistributedDataStructure(int size);
8
9  public:
10
11   inline int getId() const;
12   inline int getSize() const;
13   inline int getLocalSize() const;
14
15 };
```

3.5.4.2 Header- und Quelldateien

Generell ist die Deklaration einer Klasse in einer entsprechenden Header-Datei mit der Endung `.h` und die Definition in einer entsprechenden Quelldatei mit der Endung `.cpp` enthalten. Dies gilt sowohl für alle taskparallelen Skelette als auch für sämtliche verteilten Datenstrukturen. Einzige Ausnahme von dieser Regel bilden Klassen-Templates, da gängige Compiler bei deren Instanziierung neben der Template-Deklaration gleichzeitig auch die Template-Definition benötigen [213, S. 754 ff.]. Grundsätzlich kommen in Verbindung mit Templates folgende Übersetzungsmodelle zum Einsatz:

- Das *Explicit Instantiation Model* erlaubt zwar eine Trennung in Header- und Quelldateien, erfordert in der Quelldatei aber zusätzlich explizite Instanziierungen der definierten Templates [295, S. 65 ff.]. Als Konsequenz ergibt sich, dass die Templates lediglich mit den explizit instanziierten Typen verwendet werden können.

- Vom *Inclusion Model* existieren zwei Varianten [295, S. 61–64]: Bei der ersten Variante wird, statt zu Beginn der Quelldatei die entsprechende Header-Datei einzubinden, die Quelldatei in der Header-Datei eingebunden, und zwar nach sämtlichen Template-Deklarationen. Dieses Vorgehen erlaubt zwar eine Trennung in Header- und Quelldateien, ermöglicht aber das Auftreten von versteckten Namenskonflikten. Die zweite Variante vermeidet dieses Problem, indem der vollständige Quelltext in die Header-Datei verlagert wird, es findet also keine Trennung von Header- und Quelldateien statt.
- Das *Separation Model* markiert die in einer Header-Datei deklarierten Templates mit dem Schlüsselwort **`export`** [295, S. 68–72]. Dieses teilt dem Compiler mit, dass sich die Template-Definition in einer anderen Datei befindet und erlaubt somit die Trennung in Header- und Quelldateien. Bedauerlicherweise unterstützt keiner der gängigen Compiler dieses Schlüsselwort und somit das Übersetzungsmodell, einzige Ausnahme bildet hier der Comeau C/C++-Compiler [106].

Es bleibt festzuhalten, dass bei sämtlichen Klassen, die mindestens einen Template-Parameter deklarieren, die zweite Variante des Inclusion Model verwendet wird, da das Explicit Instantiation Model den Template-Gedanken ad absurdum führt, die erste Variante des Inclusion Model Namenskonflikte erzeugen kann und das Separation Model mangels eines entsprechenden Compilers nicht umsetzbar ist.

Listing 3.6: Header-Wächter der Datei `Muesli.h`.

```
1 #ifndef MUESLI_H
2 #define MUESLI_H
3 ...
4 #endif
```

Sämtliche Header-Dateien verwenden sogenannte *Header-Wächter* [213, S. 99 ff.]. Diese stellen sicher, dass Header-Dateien in einer Übersetzungseinheit[18] bei Bedarf mehrfach inkludiert werden können und verhindern beim wiederholten Einbinden derselben Header-Datei die Mehrfachdefinition der

[18] Eine Übersetzungseinheit ist eine Quelldatei, die vom Compiler in eine Objektdatei übersetzt wird.

in der Datei definierten Klassen und Funktionen. Header-Wächter sind im Wesentlichen Präprozessor-Makros und erzielen in Kombination mit den Präprozessor-Anweisungen **`#ifndef`**, **`#define`** und **`#endif`** den gewünschten Effekt (siehe Lst. 3.6). Hierbei ist wichtig, dass *sämtliche* Definitionen der Header-Datei zwischen den Anweisungen **`#define`** und **`#endif`** stehen. Wird die Datei `Muesli.h` inkludiert, überprüft der Präprozessor zunächst, ob das Makro `MUESLI_H` bereits definiert ist (siehe Z. 1). Ist dies nicht der Fall, so definiert der Präprozessor das Makro und fährt mit der Bearbeitung der Datei fort (siehe Z. 2). Ist das Makro hingegen bereits definiert, d.h. die Datei wurde in der aktuellen Übersetzungseinheit bereits an einer anderen Stelle inkludiert, springt der Präprozessor lediglich zur entsprechenden **`#endif`**-Anweisung, ohne die Datei erneut zu bearbeiten (siehe Z. 4).

3.5.4.3 Muesli

Bis zur vorherigen Version 1.8 der Skelettbibliothek führte der mehrfache Import der Header-Datei `Muesli.h` zu einem ungewollten Verhalten der in dieser definierten globalen, statischen Variablen, da diese nicht global, sondern pro Übersetzungseinheit einmalig sind. Infolgedessen wurden interne Zählvariablen, wie z.B. `MSL_myId`, nicht erwartungsgemäß inkrementiert, was für die ordnungsgemäße Funktionalität der Skelettbibliothek jedoch unabdingbar ist. Zur Behebung dieses Problems wurde eine neue Klasse `Muesli` definiert, die, neben den statischen Funktionen `InitSkeletons` und `TerminateSkeletons` zum Initialisieren und Beenden der Skelettbibliothek, sämtliche vormals globalen, statischen Variablen als statische Elemente definiert. Aus diesem Grund müssen die entsprechenden Variablen und Funktionen in der aktuellen Version 2.0 der Skelettbibliothek mit vorangestelltem Klassennamen verwendet werden, z.B. `Muesli::MSL_myId` oder `Muesli::InitSkeletons`.

3.5.4.4 Serialisierung

In der Datei `Serializable.h` werden im Namensraum `msl` einige Konstanten sowie die beiden Template-Funktionen `read` und `write` definiert (siehe Lst. 3.7). Während die Konstanten die Größe der elementaren Typen

in Byte speichern, vereinfachen `read` und `write` den Umgang mit Puffern vom Typ **`void*`** bei der Serialisierung benutzerdefinierter Objekte (siehe Abschn. 3.2.5), indem die Template-Funktionen die explizite Typkonvertierung für den Benutzer übernehmen. Zu beachten ist, dass die Funktionalität von `read` und `write` auf der Annahme basiert, dass **`sizeof(char)`** `= 1`, was bei gängigen Compilern, wie z.B. ICC oder GCC, der Fall ist.

Listing 3.7: Quelltext der Template-Funktionen `write` und `read`.

```
1  const int SOB = sizeof(bool);
2  const int SOC = sizeof(char);
3  const int SOD = sizeof(double);
4  const int SOF = sizeof(float);
5  const int SOI = sizeof(int);
6  const int SOL = sizeof(long);
7  const int SOS = sizeof(short);
8
9  template<typename T>
10 T read(void* buf, int index) {
11   return *((T*)(&((char*)buf)[index]));
12 }
13
14 template<typename T>
15 void write(void* buf, T val, int index) {
16   ((T*)(&(((char*)buf)[index])))[0] = val;
17 }
```

Zu diesem Zweck interpretieren die beiden Funktionen den gegebenen Puffer vom Typ **`void*`** intern als Puffer vom Typ **`char*`** und greifen mit Hilfe eines Index auf eine Position im Puffer zu, wobei der Index die Startposition des zu lesenden bzw. schreibenden Elements im Puffer in Byte angibt. Der Template-Parameter `T` darf ausschließlich mit elementaren Typen instanziiert werden und wird von den Funktionen zur Typkonvertierung benutzt. Im Fall von `write` muss `T` nicht explizit angegeben werden, da der Compiler den Typ mit Hilfe der Variablen `val` inferieren kann. Im Fall von `read` muss `T` hingegen explizit angegeben werden (siehe Lst. 3.8). Anzumerken bleibt, dass die Ausführung des Programms u.U. mit einem Pufferüberlauf abgebrochen wird, falls `index` auf eine Position außerhalb von `buf` zeigt. Da, wie bereits erwähnt, die Größe eines Feldes in C++ nachträglich nicht mehr festgestellt werden kann (siehe Abschn. 2.3.1.2), der Benutzer aber für die

Erzeugung von `buf` verantwortlich ist, muss dieser auch dafür Sorge tragen, dass `index` nur gültige Werte annimmt.

Listing 3.8: Beispielhafte Anwendung der Template-Funktionen `read` und `write`. Die Ausgabe der `main`-Funktion ist `'M, 1.123000, 58'`.

```
1  void main(int argc, char** argv) {
2    char c = 'M';    double d = 1.123;    int i = 58;
3    void* buf = new char[msl::SOC + msl::SOD + msl::SOI];
4    // alternative Speicherallokation:
5    // void* buf = malloc(msl::SOC + msl::SOD + msl::SOI);
6
7    msl::write(buf, c, 0);
8    msl::write(buf, d, msl::SOC);
9    msl::write(buf, i, msl::SOC + msl::SOD);
10
11   printf("'%c, %f, %d'\n",
12     msl::read<char>(buf, 0),
13     msl::read<double>(buf, msl::SOC),
14     msl::read<int>(buf, msl::SOC + msl::SOD));
15 }
```

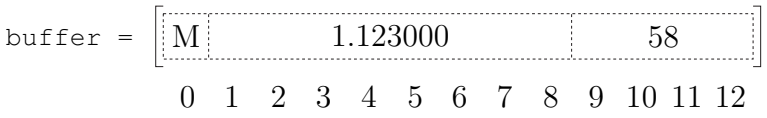


Abbildung 3.8: Beispielhafter Zustand des Puffers `buf` aus Listing 3.8 unmittelbar vor der `printf`-Ausgabe. Die Zahlen unter dem Puffer geben den Byte-basierten Index des jeweiligen Elements an.

3.6 Ergebnisse

Im Folgenden werden die Ergebnisse einiger Laufzeitmessungen erläutert. Die implementierten Testprogramme verwenden diverse Skelette der Klassen `DistributedArray` und `DistributedMatrix` und wurden auf dem ZIV-HPC (siehe Abschn. 2.5) durchgeführt. Anzumerken bleibt, dass jeder Prozess auf einem separaten Knoten ausgeführt wurde, wodurch jeder Prozess zusätzlich acht Threads starten konnte.

- Tabelle 3.2 zeigt die Laufzeiten einer schnellen Fourier-Transformation. Das Testprogramm speichert die zu transformierenden Elemente in einem verteilten Feld der Größe $n = 2^{26}$ und verwendet die Skelette *mapIndexInPlace* und *permutePartition*. Zu erkennen ist, dass das Programm mit steigender Anzahl an Prozessen geringfügig besser skaliert als mit steigender Anzahl an Threads. Dies lässt sich mit dem durch das OpenMP-Laufzeitsystem erzeugten Mehraufwand erklären, da das *mapIndexInPlace*-Skelett wiederholt aufgerufen wird, was das wiederholte Erzeugen und Zerstören von Thread-Teams zur Folge hat. Weiterhin fällt auf, dass der unter Verwendung von $np = 16$ Prozessen und $nt = 8$ Threads erzielte Speedup vergleichsweise gering ist. Hierfür ist im Wesentlichen das *permutePartition*-Skelett verantwortlich, welches durch die Verwendung von OpenMP-Direktiven nicht beschleunigt werden kann, da es sich um ein Kommunikationsskelett handelt (siehe Abschn. 7.2.2). Da ein verhältnismäßig hoher Anteil der Ausführungszeit auf das *permutePartition*-Skelett entfällt, dieses aber nicht beschleunigt werden kann, nimmt der relative Anteil der Ausführungszeit für das Skelett mit steigender Anzahl an Threads bzw. Prozessen zu. Infolgedessen liegt der Speedup mit 76 deutlich unterhalb des Optimums von 128.

Tabelle 3.2: Laufzeiten einer schnellen Fourier-Transformation in Sekunden. Der Speedup ist jeweils in Klammern angegeben.

$nt \setminus np$	1	2	4	8	16
1	207,69 (1,0)	106,26 (1,9)	54,01 (3,8)	27,33 (7,6)	13,94 (14,9)
2	107,27 (1,9)	55,18 (3,7)	28,34 (7,3)	14,51 (14,3)	7,46 (27,8)
4	56,16 (3,7)	28,62 (7,2)	16,08 (12,9)	8,34 (24,9)	4,37 (47,5)
8	30,24 (6,8)	16,46 (12,6)	8,97 (23,1)	4,85 (42,8)	2,71 (76,7)

- Tabelle 3.3 zeigt die Laufzeiten eines Gauss'schen Eliminationsverfahrens. Das Testprogramm speichert das zu lösende Gleichungssystem in einer verteilten Matrix der Größe $n = m = 2.048$ und verwendet die Skelette *mapIndexInPlace* und *broadcastPartition*. Zu erkennen ist, dass das Programm von einer höheren Anzahl an Prozessen genauso

stark profitiert wie von einer höheren Anzahl an Threads. Offensichtlich steigen die Kommunikationskosten bei der Verwendung zusätzlicher Prozesse genauso stark wie die Kosten für die Erzeugung und Zerstörung der Thread-Teams bei der Verwendung zusätzlicher Threads. Verglichen mit der schnellen Fourier-Transformation sind diese Mehraufwände jedoch signifikant niedriger, so dass der Speedup mit 104 deutlich höher ausfällt und näher am Optimum von 128 liegt.

Tabelle 3.3: Laufzeiten eines Gauss'schen Eliminationsverfahrens in Sekunden. Der Speedup ist jeweils in Klammern angegeben.

$nt \setminus np$	1	2	4	8	16
1	424,11 (1,0)	216,37 (1,9)	107,58 (3,9)	54,57 (7,7)	27,89 (15,2)
2	214,01 (1,9)	106,31 (3,9)	53,98 (7,8)	27,65 (15,3)	15,47 (27,4)
4	108,29 (3,9)	53,83 (7,8)	27,50 (15,4)	15,47 (27,4)	7,92 (53,5)
8	54,36 (7,8)	27,54 (15,5)	14,24 (29,7)	7,41 (57,2)	4,05 (104,6)

Tabelle 3.4: Laufzeiten einer Matrix-Matrix-Multiplikation in Sekunden. Der Speedup ist jeweils in Klammern angegeben. Auf Grund der in Abschnitt 3.3.2 erwähnten Einschränkungen kann der Test lediglich für $np = 1, 4$ und 16 ausgeführt werden.

$nt \setminus np$	1	4	16
1	216,77 (1,0)	54,40 (3,9)	13,55 (15,9)
2	111,81 (1,9)	26,44 (8,1)	6,83 (31,6)
4	58,70 (3,6)	14,21 (15,2)	3,44 (72,7)
8	28,45 (7,5)	7,05 (30,6)	1,96 (110,2)

- Tabelle 3.4 zeigt die Laufzeiten einer Matrix-Matrix-Multiplikation. Das Testprogramm speichert die zu multiplizierenden Matrizen jeweils in einer verteilten Matrix der Größe $n = m = 2.048$ und verwendet die Skelette *mapIndexInPlace* und *rotate*. Zu erkennen ist, dass das Programm, analog zur Gauss'schen Elimination, von einer steigenden Anzahl an Prozessen genauso stark profitiert wie von einer steigenden

Anzahl an Threads. Auch hier steigen die Kommunikationskosten bei der Verwendung zusätzlicher Prozesse offensichtlich genauso stark wie die Kosten für die Erzeugung und Zerstörung der Thread-Teams bei der Verwendung zusätzlicher Threads. Darüber hinaus ist der durch das *rotate*-Skelett verursachte Kommunikationsaufwand sowohl absolut als auch relativ betrachtet vergleichsweise gering, so dass der Speedup mit 110 ziemlich nah am Optimum von 128 liegt.

3.7 Fazit

In diesem Kapitel wurden die grundlegenden Konzepte und Eigenschaften sowie die implementierten Erweiterungen der Münster Skelettbibliothek Muesli beschrieben. Die Bibliothek unterstützt die Erstellung paralleler Programme nach dem SPMD-Programmiermodell und ermöglicht datenparallele Berechnungen durch die Verwendung verteilter Datenstrukturen, indem die entsprechenden Daten in mehrere Partitionen zerlegt und auf die zur Verfügung stehenden Prozesse verteilt werden (siehe Abschn. 3.2.1). Derzeit sind verteilte Datenstrukturen für Felder, Matrizen sowie dünnbesetzte Matrizen implementiert (siehe Abschn. 3.3), zusätzliche Datenstrukturen können jedoch problemlos hinzugefügt werden. Sämtliche Datenstrukturen definieren einen Template-Parameter, der vom konkreten Typ der zu speichernden Objekte abstrahiert, und implementieren damit das Konzept des parametrischen Polymorphismus (siehe Abschn. 3.2.2). Zusätzlich stellt die Bibliothek einen automatisierten Serialisierungsmechanismus zur Verfügung, so dass innerhalb der verteilten Datenstrukturen neben elementaren Typen beliebige benutzerdefinierte Typen verwendet werden können (siehe Abschn. 3.2.5). Die parallele Manipulation der Datenstrukturen erfolgt mit Hilfe der Skelette *count*, *fold*, *map* und *zip* (siehe Abschn. 3.4). Letztere lassen sich als typische parallele Programmiermuster definieren und sind, konzeptionell betrachtet, Funktionen höherer Ordnung (siehe Abschn. 3.2.3), da jedes Skelett mindestens eine benutzerdefinierte Argumentfunktion erwartet, um aus dem abstrakten Muster eine konkrete Anwendung zu erzeugen. Die Argumentfunktion kann einerseits in Form eines gewöhnlichen Funktionszeigers, andererseits in Form eines speziellen Funktionsobjekts übergeben werden, welches mit Hilfe des implementierten Curry-Mechanismus

erzeugt werden kann und die partielle Applikation von Argumenten unterstützt (siehe Abschn. 3.2.4). Für den Anwender ist dies jedoch transparent, da sämtliche Skelette stets beide Varianten unterstützen.

Neben der Implementierung einer Datenstruktur für verteilte, dünnbesetzte Matrizen (siehe Kap. 4) stellt die Unterstützung von Mehrkernprozessoren eine weitere elementare Erweiterung der Skelettbibliothek dar (siehe Abschn. 3.5.1). In Kombination mit der OpenMP-Abstraktionsschicht (siehe Abschn. 3.5.2) skalieren unter Verwendung von Muesli erstellte Programme damit zusätzlich auf Parallelrechnern mit hybrider Speicherarchitektur (siehe Abschn. 2.2.2.3). Darüber hinaus wurden mit den Funktionen `msl:: broadcast`, `msl::allgather` und `msl::allreduce` drei kollektive Kommunikationsfunktionen implementiert, die, im Vergleich zu den entsprechenden MPI-Pendants (siehe Abschn. 2.3.3), das Senden und Empfangen einer beliebigen Anzahl serialisierbarer Objekte unterstützen (siehe Abschn. 3.5.3). Des Weiteren wurden mit der Organisation des Quelltextes in Header- und Quelldateien, den Klassen `Muesli` und `DistributedDataStructure` sowie den Funktionen `msl::read` und `msl::write` zusätzliche Erweiterungen vorgenommen (siehe Abschn. 3.5.4). Die durchgeführten Laufzeitmessungen haben gezeigt, dass die Skelettbibliothek die effiziente Parallelisierung unterschiedlicher Anwendungen ermöglicht und dem Anspruch, die Erstellung paralleler Programme zu vereinfachen, gerecht wird (siehe Abschn. 3.6).

Kapitel 4

Eine verteilte Datenstruktur für dünnbesetzte Matrizen

Inhalt

4.1 Einleitung

Neben der Eigenschaft zur automatisierten Unterstützung von Mehrkernprozessoren (siehe Abschn. 3.5.1) wurde die Skelettbibliothek um eine verteilte Datenstruktur für dünnbesetzte Matrizen ergänzt [99]. Der Begriff bezeichnet eine Matrix, die hauptsächlich mit Nullen besetzt ist [283, S. 619], so dass es ökonomisch sinnvoll ist, daraus einen Vorteil zu ziehen [300, S. 191]. Dieser Vorteil besteht i.d.R. darin, lediglich die von Null verschiedenen Elemente zu speichern, was zu einer deutlichen Reduktion des Speicherplatzverbrauchs sowie der Rechenzeit führt. Zur Anwendung kommen dünnbesetzte Matrizen vor allem in der numerischen Mathematik zur Darstellung von Gleichungssystemen oder in der Graphentheorie zur Speicherung von

Adjazenzmatrizen. Da die bestehende Datenstruktur für verteilte Matrizen aus der Eigenschaft, dass eine Matrix dünnbesetzt ist, keinen Vorteil zieht, wurde eine vollständige Neuentwicklung vorgenommen.

Die verteilte Datenstruktur für dünnbesetzte Matrizen ist, in Anlehnung an die Klassen `DistributedArray` und `DistributedMatrix` (siehe Abschn. 3.3.1 und 3.3.2), in der Klasse `DistributedSparseMatrix` definiert. In der Entwurfsphase der Entwicklung wurde eine Reihe von Designzielen festgelegt, deren Umsetzung in den folgenden Abschnitten ausführlich erläutert wird. Die wesentlichen Eigenschaften und Merkmale der verteilten Datenstruktur seien an dieser Stelle aber bereits vorweggenommen:

- Sämtliche Skelette der Datenstruktur verwenden OpenMP, um die Ausführung der Funktion auf Mehrkernprozessoren zu beschleunigen (siehe Abschn. 3.5.1). Hierbei ist wichtig, dass dieses Merkmal optionalen Charakter besitzt und durch die Verwendung der OpenMP-Abstraktionsschicht OAL beinahe vollständig ohne Laufzeitverluste deaktiviert werden kann (siehe Abschn. 3.5.2).

- Die Datenstruktur unterstützt beliebige benutzerdefinierte Kompressionsverfahren (siehe Abschn. 4.2.2). Bereits implementiert wurden die gebräuchlichen Verfahren *Compressed Row Storage* und *Block Sparse Row* (siehe Abschn. 4.3.2).

- Die Datenstruktur implementiert einen Mechanismus zur Lastverteilung, der ebenfalls benutzerdefiniert festgelegt werden kann (siehe Abschn. 4.2.1). Analog zu den Kompressionsverfahren wurden auch hier bereits einige Standardimplementierungen vorgenommen (siehe Abschn. 4.3.1).

- Die Größe der Datenstruktur, d.h. die Zeilen- und Spaltenanzahl, wird lediglich durch den Wertebereich von **`int`** eingeschränkt [213, S. 58–61]. Im Gegensatz zu den verteilten Datenstrukturen für Felder und Matrizen muss die Größe folglich nicht als Zweierpotenz darstellbar sein.

- Die Anzahl der verwendeten Prozesse np wird ebenfalls nicht eingeschränkt, muss jedoch mindestens 1 betragen. Im Gegensatz zu den

verteilten Datenstrukturen für Felder und Matrizen muss np folglich nicht als Zweierpotenz darstellbar sein.

- Die Datenstruktur kann, in Anlehnung an das Zwei-Schichten-Modell von P^{3}L (siehe Abschn. 2.6.14), innerhalb von taskparallelen Skeletten verwendet, d.h. geschachtelt werden. Diese Eigenschaft wird im Wesentlichen durch die Verwendung der kollektiven, serialisierten Kommunikationsfunktionen sichergestellt (siehe Abschn. 3.5.3).

Der Rest des Kapitels ist wie folgt aufgebaut: Zunächst werden in Abschnitt 4.2 die wesentlichen Konzepte beschrieben, die der Implementierung der Datenstruktur zu Grunde liegen. Anschließend befasst sich Abschnitt 4.3 mit den entsprechenden Implementierungsdetails. Hier werden die wichtigsten Klassen sowie deren Zusammenspiel erörtert. Das Hauptaugenmerk liegt jedoch auf den datenparallelen Skeletten, deren Implementierung im Detail beschrieben wird. Abschnitt 4.4 zeigt die Ergebnisse einiger Laufzeitmessungen, die die Effizienz der vorgenommenen Implementierung belegen, Abschnitt 4.5 schließt das Kapitel mit einem Fazit ab.

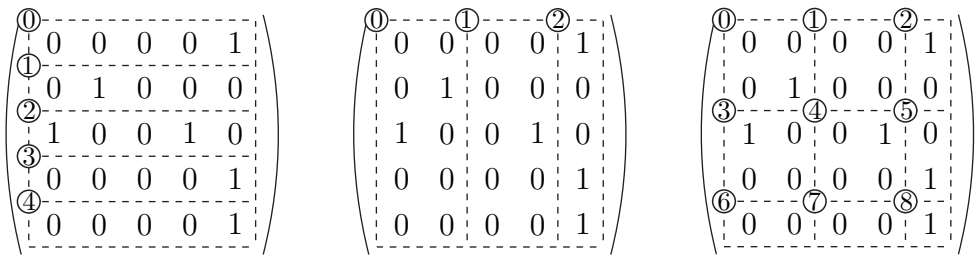
4.2 Konzepte

Die folgenden Abschnitte 4.2.1 bis 4.2.3 erörtern grundlegende Konzepte, die der Implementierung der verteilten Datenstruktur für dünnbesetzte Matrizen zu Grunde liegen. Während Abschnitt 4.2.1 einen Mechanismus zur Lastverteilung beschreibt, erörtert Abschnitt 4.2.2, wie die Elemente der dünnbesetzten Matrix komprimiert werden. Abschließend greift Abschnitt 4.2.3 kurz das Konzept des bereits in Abschnitt 3.2.2 beschriebenen parametrischen Polymorphismus auf und erläutert zusätzliche Anforderungen, die sich bei der Verwendung von benutzerdefinierten Typen ergeben.

4.2.1 Lastverteilung

Grundsätzlich stellt sich bei der Verwendung einer verteilten Datenstruktur auf einem Parallelrechner die Frage, wie deren Elemente unter den zur Verfügung stehenden Prozessen aufgeteilt werden. Die verteilte Datenstruktur für

dünnbesetzte Matrizen implementiert diesbezüglich, angelehnt an das sogenannte *Block Sparse Row* Kompressionsverfahren (siehe Abschn. 4.3.2.2), einen vergleichsweise simplen, aber flexiblen Ansatz. Dieser sieht vor, eine $n \times m$ Matrix zunächst in sogenannte *Submatrizen* der Größe $r \times c$ aufzuteilen, wobei i.d.R. $r \ll n$ und $c \ll m$ mit $n, m, r, c \in \mathbb{N}$ (siehe Abb. 4.1). Dieses Vorgehen ermöglicht es, die Matrix auf verschiedene Art und Weise aufzuteilen, so dass z.B. beim Rotieren von Zeilen bzw. Spalten kein Kommunikationsaufwand entsteht (siehe Abb. 4.1a und 4.1b). Zu beachten ist, dass weder $n \bmod r = 0$ noch $m \bmod c = 0$ gelten muss, vielmehr wird die Größe der Submatrizen am rechten bzw. unteren Rand der Matrix im Bedarfsfall entsprechend angepasst (siehe Abb. 4.1c).



(a) $r = 1$, $c = 5$ (b) $r = 5$, $c = 2$ (c) $r = c = 2$

Abbildung 4.1: Möglichkeiten zur Aufteilung einer dünnbesetzten 5×5 Matrix in Submatrizen. Die ID einer Submatrix ist jeweils in der linken oberen Ecke eingekreist.

Jede Submatrix erhält eine global eindeutige ID. Diese wird von einem speziellen `Distribution`-Objekt dazu benutzt, die Zuordnung zwischen Submatrizen und Prozessen vorzunehmen. M.a.W., ein `Distribution`-Objekt kapselt die Applikationslogik, die das Verteilungsschema der Submatrizen festlegt. Hierbei ist zu beachten, dass jede Submatrix genau *einem* Prozess zugeordnet wird, einem Prozess jedoch *mehrere* Submatrizen zugeordnet werden können. Der implementierte Mechanismus erlaubt beliebige benutzerdefinierte Verteilungsschemata und wird in Abschnitt 4.3.1 ausführlich erläutert.

4.2.2 Kompression

Die zentrale Fragestellung bei der Implementierung einer verteilten Datenstruktur für dünnbesetzte Matrizen ist die nach der Kompression. Analog zum Verteilungsmechanismus für die Submatrizen wurde hier ein flexibler, durch den Benutzer erweiterbarer Ansatz implementiert, mit dem beliebige Kompressionsverfahren unterstützt werden können. Diese Flexibilität ist wichtig, da jedes Kompressionsverfahren einen Kompromiss zwischen dem verbrauchten Speicherplatz und der Zugriffszeit eingeht. Da zwischen diesen beiden Größen eine antiproportionale Beziehung besteht, bleibt die Entscheidung, welche der beiden Eigenschaften wichtiger ist, geringe Zugriffszeit oder geringer Speicherplatzverbrauch, dem Benutzer überlassen. Konzeptionell findet die Kompression einer dünnbesetzten Matrix in zwei Stufen statt:

1. Wie bereits erwähnt, wird eine dünnbesetzte Matrix in mehrere Submatrizen aufgeteilt, die mit Hilfe eines `Distribution`-Objekts den verfügbaren Prozessen zugeordnet werden. Die Aufteilung der Matrix in Submatrizen ist nicht nur aus Gründen der Lastverteilung sinnvoll, sondern ermöglicht gleichzeitig auch eine erste Kompression. Diese besteht darin, leere Submatrizen, d.h. solche, die ausschließlich Nullen enthalten, nicht zu speichern. Auf diese Weise würden die Submatrizen 1, 6 und 7 aus Abbildung 4.1c verworfen werden.

2. In einem zweiten Schritt werden die nicht-leeren Submatrizen, d.h. solche, die mindestens ein von Null verschiedenes Element enthalten, in Abhängigkeit von dem durch die Submatrix implementierten Kompressionsverfahren komprimiert. Auf diese Weise würden die Submatrizen 0, 2, 3, 4, 5 und 8 aus Abbildung 4.1c komprimiert werden. Der implementierte Mechanismus erlaubt beliebige benutzerdefinierte Kompressionsverfahren und wird in Abschnitt 4.3.2 ausführlich erläutert.

4.2.3 Parametrischer Polymorphismus

Die Klasse `DistributedSparseMatrix` besitzt, analog zu den Klassen `DistributedArray` und `DistributedMatrix`, ebenfalls einen Template-Parameter `T`, der den Typ der zu speichernden Elemente festlegt. `T` kann mit sämtlichen integralen Datentypen sowie benutzerdefinierten Strukturen und Klassen instanziiert werden. Zu beachten ist, dass, falls Kommunikationsskelette, wie z.B. *fold*, verwendet werden sollen, die entsprechende Klasse den in Abschnitt 3.2.5 erläuterten Serialisierungsmechanismus unterstützen muss, d.h. die Klasse muss von `msl::Serializable` erben sowie die Funktionen `getSize`, `reduce` und `expand` implementieren. Darüber hinaus muss die Klasse die Operatoren `==`, `!=` und `<<` überschreiben, wobei letzterer lediglich für die Ausgabe benötigt wird.[19]

4.3 Implementierung

Nachdem in Abschnitt 4.2 die der Klasse `DistributedSparseMatrix` zugrunde liegenden Konzepte erläutert wurden, befassen sich die folgenden Abschnitte mit der entsprechenden Implementierung. Behandelt werden sowohl ausgewählte Details der in Abbildung 4.2 dargestellten Klassen und Schnittstellen als auch deren Beziehungen. Zunächst befassen sich die Abschnitte 4.3.1, 4.3.2 und 4.3.3 mit den Klassen `Distribution`, `Submatrix` und `RowProxy`. Anschließend wird in Abschnitt 4.3.4 mit der Klasse `DistributedSparseMatrix` das Kernstück der Implementierung der verteilten Datenstruktur vorgestellt. Abschließend erläutert Abschnitt 4.3.5 die von der verteilten Datenstruktur implementierten datenparallelen Skelette.

4.3.1 Distribution

Wie bereits erwähnt, kapselt ein `Distribution`-Objekt die Applikationslogik, die die Zuordnung zwischen Prozessen und Submatrizen vornimmt (siehe Abschn. 4.2.1). Die Schnittstelle ist in der Datei `Distribution.h`

[19] Die Klasse `Integer` erfüllt diese Anforderungen beispielhaft und kann Anhang A.3 entnommen werden.

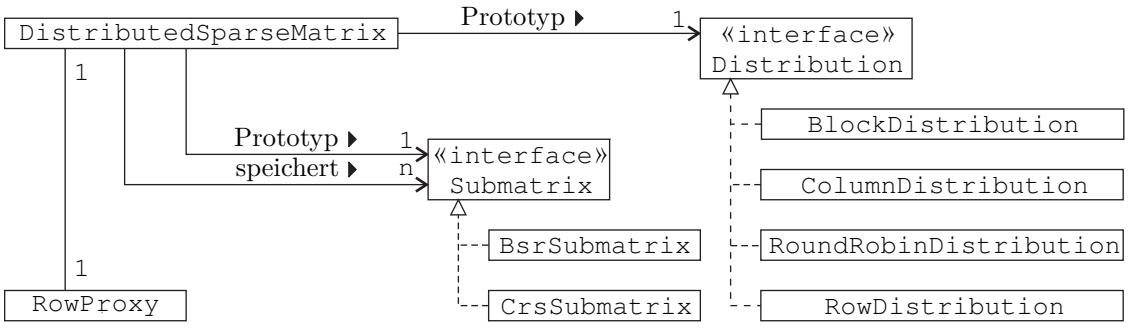


Abbildung 4.2: Beziehungen zwischen den bereitgestellten Klassen und Schnittstellen.

deklariert (siehe Lst. 4.1), die Klasse in der Datei `Distribution.cpp` definiert. Jedem `Distribution`-Objekt stehen für die Zuordnung von Submatrizen zu Prozessen die Variablen n, m, r, c, np und max zur Verfügung, wobei $max \in \mathbb{N}$ die maximale Anzahl zu verteilender Submatrizen ist (siehe Z. 5). Unterklassen können jedoch beliebige zusätzliche Attribute definieren und diese z.B. mit Hilfe eines Konstruktors initialisieren. Auf diese Weise werden Flexibilität und Erweiterbarkeit gewährleistet. Die Schnittstelle deklariert folgende rein virtuelle Funktionen:

`virtual int` `getIdProcess(int idSubmatrix)` **`const`** `= 0`. Die Funktion kapselt die Applikationslogik, d.h. die Zuordnung von Submatrizen zu Prozessen, und setzt dies mit Hilfe des Entwurfsmuster *Strategie* um [150, S. 315–323]. `getIdProcess` erwartet mit `idSubmatrix` die ID einer Submatrix als Argument und liefert die ID des Prozesses zurück, der für die Speicherung der Submatrix mit der übergebenen ID verantwortlich ist (siehe Z. 16). Die Funktion kann dabei auf oben erwähnte und/oder benutzerdefinierte Attribute zurückgreifen.

`virtual` `Distribution* clone()` **`const`** `= 0`. Die Funktion dient dem Kopieren eines `Distribution`-Objekts und ist erforderlich, da die Interaktion mit Objekten vom Typ `DistributedSparseMatrix` mit dem Entwurfsmuster *Prototyp* [150, S. 117–126] umgesetzt wurde. `clone` erstellt eine tiefe Kopie des `Distribution`-Objekts, auf dem die Funktion aufgerufen wurde, und liefert einen Zeiger auf diese Kopie zurück (siehe Z. 15). Die Funktion kann in einer Unterklasse i.d.R. durch den Aufruf des Compiler-generierten Kopierkonstruktors implementiert werden [1, S. 203–207, §12.8]. Falls eine Unter-

klasse jedoch Zeigerattribute definiert, muss der Benutzer einen eigenen Kopierkonstruktor definieren, der die referenzierten Objekte ebenfalls kopiert, da anderenfalls lediglich die Adressen kopiert werden. Die Definition eines (korrekten) Kopierkonstruktors kann jedoch nicht erzwungen werden. Eine weitere Schwachstelle der Umsetzung des Entwurfsmusters *Prototyp* in C++ ist die Tatsache, dass die `clone`-Funktion mit Hilfe des **`new`**-Operators zwar Speicher alloziert, diesen aber nicht wieder mit Hilfe des **`delete`**-Operators freigibt. Dies begünstigt das Auftreten von Speicherlecks, ist bei genauer Betrachtung aber nicht schwerwiegend, da `Distribution`-Objekte lediglich von `DistributedSparseMatrix`-Objekten geklont werden. Letztere nehmen dann im Destruktor den entsprechenden **`delete`**-Aufruf vor (siehe Abschn. 4.3.4.4).

Darüber hinaus implementiert die Klasse auf Basis der beiden oben genannten rein virtuellen Funktionen folgende Hilfsfunktionen:

`bool` `equals(`**`const`** `Distribution& d)` **`const`**. Die Funktion liefert **`true`** zurück, falls das übergebene `Distribution`-Objekt `d` die Zuordnung von Submatrizen zu Prozessen analog zu dem `Distribution`-Objekt durchführt, auf dem diese Funktion aufgerufen wurde und sämtliche Attribute mit denselben Werten belegt sind. Falls die Zuordnung bei mindestens einer Submatrix abweicht oder mindestens ein Parameter einen anderen Wert aufweist, gibt die Funktion **`false`** zurück.

`bool` `isStoredLocally(`**`int`** `idProcess,` **`int`** `idSubmatrix)` **`const`**. Die Funktion liefert **`true`** zurück, falls der Prozess mit der übergebenen ID für die Speicherung der Submatrix mit der übergebenen ID zuständig ist. Anderenfalls gibt die Funktion **`false`** zurück.

`virtual void` `initialize(`**`int`** `np,` **`int`** `n,` **`int`** `m,` **`int`** `r,` **`int`** `c,` **`int`** `max)`. Die Funktion überschreibt die Attribute des Objekts mit den entsprechenden übergebenen Werten. Da benutzerdefinierte Unterklassen eigene Attribute u.U. in Abhängigkeit von diesen Werten initialisieren möchten, ist die Funktion als **`virtual`** deklariert.

Um dem Benutzer die Erstellung eines parallelen Programms unter Verwendung der Klasse `DistributedSparseMatrix` zu erleichtern, bietet die Bibliothek bereits vier Implementierungen der `Distribution`-Schnittstelle

Listing 4.1: Schnittstelle der Klasse `Distribution`.

```
1  class Distribution {
2
3  protected:
4
5    int n;  int m;  int r;  int c;  int np;  int max;
6
7  public:
8
9    bool equals(const Distribution& d) const;
10   bool isStoredLocally(int idProcess,
11                        int idSubmatrix) const;
12   virtual void initialize(int np, int n, int m, int r,
13                           int c, int max);
14
15   virtual Distribution* clone() const = 0;
16   virtual int getIdProcess(int idSubmatrix) const = 0;
17
18 };
```

an. Diese werden in den folgenden Abschnitten 4.3.1.1 bis 4.3.1.4 erörtert. Darüber hinaus erläutert Abschnitt 4.3.4.1 das Zusammenspiel zwischen den Klassen `Distribution` und `DistributedSparseMatrix` und geht dabei insbesondere auf die Konfiguration des Verteilungsverfahrens ein.

4.3.1.1 BlockDistribution

Die Klasse `BlockDistribution` nimmt die Zuordnung von Submatrizen zu Prozessen in Blöcken von benachbarten Submatrizen vor, wobei die Nachbarschaft nicht räumlich, sondern über die ID einer Submatrix definiert ist. Die Schnittstelle ist in der Datei `BlockDistribution.h` deklariert, die Klasse in der Datei `BlockDistribution.cpp` definiert. Falls $max \bmod np = 0$, erhält jeder Prozess einen Block von max/np Submatrizen, wobei jeder Block stets Submatrizen mit zusammenhängenden IDs enthält. Falls $max \bmod np \neq 0$, erhalten die ersten Prozesse einen Block von $\lceil max/np \rceil$, die letzten Prozesse einen Block von $\lfloor max/np \rfloor$ Submatrizen (siehe Tab. 4.1).

4.3.1.2 ColumnDistribution

Die Klasse `ColumnDistribution` nimmt die Zuordnung von Submatrizen zu Prozessen spaltenweise alternierend vor. Die Schnittstelle ist in der Datei `ColumnDistribution.h` definiert, die Klasse in der Datei `ColumnDistribution.cpp` deklariert. Prozesse bekommen stets ganze Spalten von Submatrizen zugewiesen. Ein Prozess ist für die Speicherung einer Submatrix genau dann verantwortlich, wenn die Submatrix Bestandteil einer Spalte ist, die dem Prozess zugeordnet wurde. Ein Prozess ist für die Speicherung einer Spalte verantwortlich, wenn dessen ID dem Spaltenindex $j \bmod np$ entspricht (siehe Tab. 4.1).

4.3.1.3 RoundRobinDistribution

Die Klasse `RoundRobinDistribution` nimmt die Zuordnung von Submatrizen zu Prozessen zyklisch vor. Die Schnittstelle ist in der Datei `RoundRobinDistribution.h` deklariert, die Klasse in der Datei `RoundRobinDistribution.cpp` definiert. Ein Prozess ist für die Speicherung einer Submatrix genau dann verantwortlich, wenn dessen ID der ID der Submatrix $\bmod np$ entspricht (siehe Tab. 4.1).

4.3.1.4 RowDistribution

Die Klasse `RowDistribution` nimmt die Zuordnung von Submatrizen zu Prozessen zeilenweise alternierend vor. Die Schnittstelle ist in der Datei `RowDistribution.h` deklariert, die Klasse in der Datei `RowDistribution.cpp` definiert. Prozesse bekommen stets ganze Zeilen von Submatrizen zugewiesen. Ein Prozess ist für die Speicherung einer Submatrix genau dann verantwortlich, wenn die Submatrix Bestandteil einer Zeile ist, die dem Prozess zugeordnet wurde. Ein Prozess ist für die Speicherung einer Zeile verantwortlich, wenn dessen ID dem Zeilenindex $i \bmod np$ entspricht (siehe Tab. 4.1).

Tabelle 4.1: Verteilung der Submatrizen aus Abbildung 4.1c auf zwei Prozesse.

	Block-Distribution	Column-Distribution	RoundRobin-Distribution	Row-Distribution
p_0	0, 1, 2, 3, 4	0, 2, 3, 5, 6, 8	0, 2, 4, 6, 8	0, 1, 2, 6, 7, 8
p_1	5, 6, 7, 8	1, 4, 7	1, 3, 5, 7	3, 4, 5

4.3.2 Submatrix

Wie bereits erwähnt, kapselt ein `Submatrix`-Objekt die Applikationslogik, die für die Kompression der dünnbesetzten Matrix verantwortlich ist (siehe Abschn. 4.2.2). Die Schnittstelle des Klassen-Templates ist in der Datei `Submatrix.h` deklariert und kann auszugsweise Listing 4.2 entnommen werden. Die Klasse deklariert einen Template-Parameter `T`, der den Typ der zu speichernden Elemente festlegt (siehe Z. 1). Anzumerken bleibt, dass `T` standardmäßig mit **`double`** instanziiert wird [1, S. 237, §14.1.9]. Darüber hinaus werden folgende Attribute deklariert:

- `n` speichert die Zeilenanzahl der Submatrix (siehe Z. 9). Dieser Wert stimmt i.d.R. mit dem von r überein (siehe Abschn. 4.2.1). Eine Ausnahme bilden hier Submatrizen, die sich am rechten Rand der dünnbesetzten Matrix befinden und bei denen die Größe angepasst werden muss, da $n \bmod r \neq 0$.

- `m` speichert die Spaltenanzahl der Submatrix (siehe Z. 10). Dieser Wert stimmt i.d.R. mit dem von c überein (siehe Abschn. 4.2.1). Eine Ausnahme bilden hier Submatrizen, die sich am unteren Rand der dünnbesetzten Matrix befinden und bei denen die Größe angepasst werden muss, da $m \bmod c \neq 0$.

- `i0` speichert den globalen Zeilenindex, `j0` den globalen Spaltenindex des linken oberen Elements der Submatrix (siehe Z. 9 f.).

- `sid` speichert die global eindeutige ID der Submatrix (siehe Z. 9 und Abschn. 4.2.1). IDs sind ganzzahlige, fortlaufende Nummern und werden, mit 0 beginnend, zeilenweise vergeben, d.h. ID $\in [0; max - 1]$.

- `values` speichert die Werte der Submatrix (siehe Z. 10). Da deren Anzahl zur Laufzeit variieren kann, wird für die Speicherung kein simples Feld vom Typ `T*`, sondern eine Datenstruktur vom Typ `std::vector<T>` verwendet. Das Klassen-Template `std::vector<T>` ist Teil der STL [207] und implementiert einen Container zur Speicherung von Werten, dessen Größe dynamisch angepasst werden kann [1, S. 482–486, §23.2.4] [213, S. 121–126]. Der Template-Parameter `T` legt hierbei den Typ der zu speichernden Elemente fest.

- `zero` speichert das Nullelement für den Typ `T` und wird benötigt, damit die Klasse nicht nur mit primitiven, sondern auch mit benutzerdefinierten Typen verwendet werden kann (siehe Z. 10 und Abschn. 4.2.3). Das Attribut wird unter Verwendung der Funktion **`void`** `setZero(T)` im Konstruktor der Klasse `DistributedSparseMatrix` gesetzt (siehe Lst. 4.11, Abschn. 4.3.4.4).

Anzumerken bleibt, dass in benutzerdefinierten Unterklassen i.d.R. zusätzliche Attribute deklariert werden, um das entsprechende Kompressionsverfahren zu implementieren, wie z.B. in dem Klassen-Template `CrsSubmatrix<T>` (siehe Abschn. 4.3.2.1). Dies ist jedoch nicht obligatorisch. So kommt z.B. das Klassen-Template `BsrSubmatrix<T>` gänzlich ohne zusätzliche Attribute aus (siehe Abschn. 4.3.2.2).

Die Schnittstelle deklariert eine Reihe von rein virtuellen Funktionen, die in benutzerdefinierten Unterklassen das jeweilige Kompressionsverfahren implementieren und daher überschrieben werden müssen. Die wichtigsten sind `getElement` zum Auslesen (siehe Z. 16) und `setElement` zum Setzen von Elementen (siehe Z. 19). Hierbei ist wichtig, dass die übergebenen Indizes `i` und `j` als *lokale* Indizes interpretiert werden, d.h. nicht auf die dünnbesetzte Matrix, sondern lediglich auf die Submatrix bezogen werden. Die Umrechnung zwischen globalen und lokalen Indizes findet innerhalb der Klasse `DistributedSparseMatrix` statt. Dadurch wird eine Trennung der Applikationslogik vorgenommen, so dass Submatrizen keine Informationen über die dünnbesetzte Matrix, sondern lediglich über lokal gespeicherte Elemente vorhalten müssen. Anzumerken bleibt, dass beide Funktionen aus Gründen der Performanz die Gültigkeit der übergebenen Indizes nicht überprüfen, d.h. ob $0 \leqslant \texttt{i} < n$ und $0 \leqslant \texttt{j} < m$. Ist dies nicht der Fall, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen.

Listing 4.2: Die wichtigsten Attribute und Funktionen der Klasse `Submatrix`. Die mit Auslassungspunkten annotierten Funktionen werden bereits von der Schnittstelle implementiert.

```
1  template<typename T = double> class Submatrix {
2
3  private:
4
5    int getElementCount(bool includeZeros) const {...}
6
7  protected:
8
9    int n;  int i0;  int sid;
10   int m;  int j0;  std::vector<T> values;  T zero;
11
12   void init(int sid, int n, int m, int i0, int j0) {...}
13
14 public:
15
16   virtual T getElement(int i, int j) const = 0;
17   virtual T getElementLocal(int k) const {...}
18
19   virtual void setElement(T value, int i, int j) = 0;
20   virtual void setElementLocal(T value, int k) {...}
21
22   virtual int getColumnIndexLocal(int k) const = 0;
23   virtual int getRowIndexLocal(int k) const = 0;
24
25   virtual Submatrix* clone() const = 0;
26
27   virtual void initialize(int sid, int n, int m,
28     int i0, int j0) = 0;
29   virtual void initialize(int sid, int n, int m,
30     int i0, int j0, T value, int i, int j) = 0;
31   virtual void initialize(int sid, int n, int m,
32     int i0, int j0, const T* const * const mx) = 0;
33
34   int getElementCount() const {...}
35   int getElementCountLocal() const {...}
36
37   ...
38
39 };
```

Die Funktionen `getElement` und `setElement` lesen bzw. setzen Elemente unter Zuhilfenahme von lokalen Zeilen- bzw. Spaltenindexe. Für viele der von der Klasse `DistributedSparseMatrix` bereitgestellten Skelette ist ein solcher Zugriff auf die Elemente einer Submatrix jedoch ineffizient, da die beiden Funktionen, in Abhängigkeit vom implementierten Kompressionsschema, Mehraufwand erzeugen. Aus diesem Grund implementiert die Schnittstelle mit `getElementLocal` (siehe Z. 17) und `setElementLocal` (siehe Z. 20) zwei Funktionen, die einen direkten Zugriff auf die von der Submatrix gespeicherten Elemente ermöglichen. Beide Funktionen erwarten als Argument einen Index `k` und greifen mit dessen Hilfe unmittelbar auf ein Element im bereits erwähnten Vektor `values` zu. M.a.W. ermöglichen `getElementLocal` und `setElementLocal` das implementierte Kompressionsverfahren gewissermaßen zu umgehen und eignen sich somit insbesondere für den Zugriff auf die Elemente einer Submatrix, falls deren Zeilen- und Spaltenindexe nicht von Bedeutung sind, wie es z.B. bei *map* der Fall ist (siehe Abschn. 4.3.5.3). Anzumerken bleibt, dass beide Funktionen aus Gründen der Performanz die Gültigkeit des übergebenen Index nicht überprüfen, d.h. ob $0 \leqslant$ `k` $<$ `values.size()`. Ist dies nicht der Fall, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen.

Auch wenn der Zeilen- und Spaltenindex eines Elements von Bedeutung ist, verwenden die von der Klasse `DistributedSparseMatrix` implementierten Skelette i.d.R. die Funktionen `getElementLocal` und `setElementLocal` statt `getElement` und `setElement`. Dieses Vorgehen setzt jedoch voraus, dass aus dem Index `k` der lokale Zeilen- und Spaltenindex des entsprechenden Elements berechnet werden kann. Zu diesem Zweck deklariert die Schnittstelle die rein virtuellen Funktionen `getRowIndexLocal` und `getColumnIndexLocal` (siehe Z. 22 f.). Beide erwarten als Argument einen Index `k`,der sich auf ein Element im Vektor `values` bezieht, und berechnen aus diesem den lokalen Zeilen- bzw. Spaltenindex des entsprechenden Elements. Auch wenn nicht garantiert werden kann, dass die Kosten für diese Berechnung bei allen erdenklichen Kompressionsverfahren geringer sind als die Kosten für die Verwendung von `getElement` und `setElement`, so ist dies doch zumindest bei den gängigen Verfahren *Compressed Row Storage* und *Block Sparse Row* der Fall (siehe Abschn. 4.3.2), was den verwendeten Ansatz rechtfertigt. Anzumerken bleibt, dass mit den Funktionen `getRowIndexGlobal` und `getColumnIndexGlobal` zwei Funktio-

nen bereitgestellt werden, die aus dem Index `k` mit Hilfe der Funktionen `getRowIndexLocal` und `getColumnIndexLocal` sowie den Attributen `i0` und `j0` den globalen Zeilen- bzw. Spaltenindex des entsprechenden Elements berechnen.

Die Funktionen `getElementLocal` und `setElementLocal` können nur dann fehlerfrei verwendet werden, wenn die Anzahl der lokal gespeicherten Elemente abgefragt werden kann, so dass der Index `k` auf keine ungültige Position im Vektor `values` verweist. Um dies zu gewährleisten, definiert die Schnittstelle die private Funktion `getElementCount(` **`bool`** `includeZeros)` (siehe Z. 5). Diese zählt, in Abhängigkeit vom übergebenen Parameter `includeZeros`, die Elemente der Submatrix. Falls `includeZeros` = **`true`**, gibt die Funktion die Anzahl der im Vektor `values` gespeicherten Elemente zurück, d.h. inklusive möglicher Nullen. Falls `includeZeros` = **`false`**, gibt die Funktion die Anzahl der von Null verschiedenen Elemente der Submatrix zurück. Diese Unterscheidung ist für Kompressionsverfahren wichtig, die auch Null-Elemente speichern, wie z.B. *Block Sparse Row* (siehe Abschn. 4.3.2.2). Andererseits können Nullen auch bei anderen Kompressionsverfahren, wie z.B. *Compressed Row Storage* (siehe Abschn. 4.3.2.1), durch die Verwendung der Funktion `setElementLocal` entstehen, da in diesem Fall das Kompressionsverfahren umgangen und direkt auf den Vektor `values` zugegriffen wird. Da die Funktion nach außen hin nicht sichtbar ist, definiert die Schnittstelle die beiden Funktionen `getElementCount` und `getElementCountLocal`. Diese rufen lediglich die oben beschriebene private Funktion `getElementCount` mit einem entsprechenden Wert für den Parameter `includeZeros` auf:

`int` `getElementCount()` **`const`**. Die Funktion gibt durch einen Aufruf von `getElementCount(`**`false`**`)` die Anzahl der von Null verschiedenen Elemente im Vektor `values` zurück (siehe Z. 34).

`int` `getElementCountLocal()` **`const`**. Die Funktion gibt durch einen Aufruf von `getElementCount(`**`true`**`)` die Anzahl aller im Vektor `values` gespeicherten Elemente zurück, d.h. inklusive möglicher Nullen (siehe Z. 35).

Die Grundidee der `Submatrix`-Schnittstelle besteht darin, in der Klasse `DistributedSparseMatrix` vom konkreten Kompressionsverfahren zu ab-

strahieren und lediglich mit Objekten vom Typ `Submatrix` zu arbeiten. Das Kompressionsverfahrens wird hierbei durch den Benutzer mit Hilfe des Entwurfsmusters *Prototyp* festgelegt [150, S. 117–126]. Dieses sieht vor, dass von einem `Submatrix`-Objekt, dem sogenannten *Prototypen*, durch den wiederholten Aufruf einer `clone`-Funktion tiefe Kopien erstellt werden (siehe Z. 25). Die Verwendung von Konstruktoren ist in diesem Zusammenhang nicht möglich, da der Typ der zu erzeugenden Submatrizen nicht bekannt ist. Zur Initialisierung der geklonten Submatrizen deklariert die Schnittstelle drei rein virtuelle `init`-Funktionen (siehe Z. 27–32). Alle erwarten als Argumente mindestens die Parameter `sid`, `n`, `m`, `i0` und `j0` und sollten durch einen Aufruf der geschützten `init`-Funktion (siehe Z. 12) die entsprechenden Attribute der Submatrix mit den übergebenen Werten initialisieren:

`virtual void` `initialize(`**`int`** `sid, `**`int`** `n, `**`int`** `m, `**`int`** `i0, `**`int`** `j0) = 0`. Die Funktion initialisiert eine leere Submatrix (siehe Z. 27 f.).

`virtual void` `initialize(`**`int`** `sid, `**`int`** `n, `**`int`** `m, `**`int`** `i0, `**`int`** `j0, T value, `**`int`** `i, `**`int`** `j) = 0`. Die Funktion initialisiert ebenfalls eine leere Submatrix, setzt jedoch das Element mit dem Zeilenindex `i` und dem Spaltenindex `j` auf den übergebenen Wert `value` (siehe Z. 29 f.). Anzumerken bleibt, dass `i` und `j` als lokale Indizes interpretiert werden.

`virtual void` `initialize(`**`int`** `sid, `**`int`** `n, `**`int`** `m, `**`int`** `i0, `**`int`** `j0, `**`const`** `T* `**`const`** ` * `**`const`** `matrix) = 0`. Die Funktion initialisiert eine Submatrix mit Hilfe des übergebenen zweidimensionalen Feldes `mx`, indem ein Ausschnitt der Größe `n` × `m` aus dem Feld kopiert wird (siehe Z. 31 f.). Der Ursprung dieses Ausschnitts befindet sich an der Position `mx[i0][j0]`. Falls der Ausschnitt die Grenzen von `mx` überschreitet, wird die Ausführung des Programms u.U. mit einem Pufferüberlauf abgebrochen. Falls der Ausschnitt nur einen Teil von `mx` umfasst, werden die überschüssigen Elemente bei der Initialisierung der Submatrix ignoriert. Anzumerken bleibt, dass weder der Zeiger auf noch die Elemente von `mx` verändert werden, was durch die dreifache Verwendung des Schlüsselworts **`const`** sichergestellt wird.

Um dem Benutzer die Erstellung eines parallelen Programms unter Verwendung der Klasse `DistributedSparseMatrix` zu erleichtern, bietet die

Bibliothek mit den Klassen `CrsSubmatrix` und `BsrSubmatrix` bereits zwei Implementierungen der `Submatrix`-Schnittstelle an. Diese werden in den folgenden Abschnitten 4.3.2.1 und 4.3.2.2 erörtert. Darüber hinaus erläutert Abschnitt 4.3.4.2 das Zusammenspiel zwischen den Klassen `Submatrix` und `DistributedSparseMatrix` und geht dabei insbesondere auf die Konfiguration des Kompressionsverfahrens ein.

4.3.2.1 CrsSubmatrix

Die Klasse `CrsSubmatrix` implementiert das sogenannte *Compressed Row Storage* Kompressionsverfahren [136, 137]. Dieses speichert lediglich von Null verschiedene Elemente und benutzt dazu die folgenden drei Felder:

- Das Feld a dient der zeilenweisen Speicherung der von Null verschiedenen Elemente (siehe Abb. 4.3). a hat die Länge nnz, wobei nnz die Anzahl von Null verschiedener Elemente ist.
- Das Feld ia wird als Zeilenseparator genutzt und speichert an der Position $ia[i]$ den Index des ersten Elements der i-ten Zeile aus Feld a. Daraus ergibt sich, dass die Differenz von zwei benachbarten Werten, d.h. $ia[i+1] - ia[i]$ mit $0 \leqslant i \leqslant n$, gleich der Anzahl von Elementen in Zeile i ist. Da dieser Zusammenhang auch für die letzte Zeile gelten muss, d.h. für den Fall $i = n$, hat ia die Länge $n+1$. Das letzte Element von ia speichert den Index eines imaginären Elements, das nach dem letzten Element von a gespeichert werden würde, und enthält deshalb nnz.
- Das Feld ja speichert zu jedem im Feld a gespeicherten Element dessen Spaltennummer und hat deshalb ebenfalls die Länge nnz.

$$A = \left\langle \begin{array}{cc:cc} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 5 \end{array} \right\rangle \quad \begin{array}{rl} a = & [\,1\ 2\ 3\ 4\ 5\,] \\ ia = & [\,0\ 1\ 2\ 4\ 5\,] \\ ja = & [\,1\ 1\ 3\ 4\ 4\,] \end{array}$$

Abbildung 4.3: Komprimierung der dünnbesetzten Matrix A mit dem CRS-Verfahren.

Insgesamt werden zur Speicherung einer mit dem CRS-Verfahren komprimierten Matrix folglich $nnz+n+1+nnz = 2\cdot nnz+n+1$ Elemente benötigt. Verglichen mit der Speicherung sämtlicher Elemente einer $n\times m$ Matrix lohnt sich der Einsatz des Kompressionsverfahrens also, wenn folgender Zusammenhang gilt:

$$\begin{array}{lll} & 2\cdot nnz + n + 1 & < \quad n\cdot m \\ \Leftrightarrow & 2\cdot nnz & < \quad n\cdot m - n - 1 \\ \Leftrightarrow & nnz & < \quad \frac{1}{2}\cdot\left(n\cdot(m-1)-1\right) \end{array}$$

4.3.2.2 BsrSubmatrix

Die Klasse `BsrSubmatrix` implementiert einen Teil des sogenannten *Block Sparse Row* Kompressionsverfahrens [136, 137]. Dieses teilt eine dünnbesetzte Matrix A in Blöcke der Größe $r \times c$ auf, wobei $r < n$ und $c < m$.[20] Anschließend werden diejenigen Blöcke gespeichert, die mindestens eine Zahl ungleich Null enthalten, d.h. sämtliche Blöcke, die ausschließlich Nullen enthalten, werden verworfen. Zu beachten ist, dass Blöcke stets vollständig gespeichert werden, d.h. inklusive Nullen. Die Speicherung findet mit Hilfe von drei Feldern statt:

- Das Feld a dient der zeilenweisen Speicherung derjenigen Blöcke, die mindestens eine Zahl ungleich Null enthalten (siehe Abb. 4.4). Die Elemente eines Blocks werden ebenfalls zeilenweise gespeichert. a hat die Länge $nnzb \cdot r \cdot c$, wobei $nnzb$ die Anzahl von Blöcken ist, die mindestens eine von Null verschiedene Zahl speichern.
- Das Feld ia wird als Zeilenseparator genutzt, wobei die Zeileneinteilung hier blockweise vorgenommen wird, und speichert den Startindex des ersten Blocks in Zeile i. Daraus ergibt sich, dass die Differenz von zwei benachbarten Werten, d.h. $ia[i+1] - ia[i]$ mit $0 \leqslant i \leqslant \frac{n}{r}$, gleich der Anzahl von Blöcken in Zeile i ist. Da dieser Zusammenhang auch für die letzte Zeile gelten muss, d.h. für $i = \frac{n}{r}$, hat ia die Länge $\frac{n}{r} + 1$.

[20] Für die folgenden Erläuterungen wird der Einfachheit halber angenommen, dass $n \bmod r = m \bmod c = 0$. Die Klasse `BsrSubmatrix` unterliegt dieser Einschränkung jedoch nicht.

Das letzte Element von ia speichert den Startindex eines imaginären Blocks, der nach dem letzten Block von a gespeichert werden würde, und enthält deshalb $nnzb$.

- Das Feld ja speichert zu jedem im Feld a gespeicherten Block dessen Spaltennummer und hat deshalb die Länge $nnzb$.

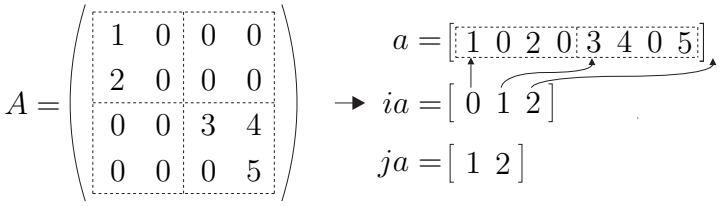


Abbildung 4.4: Komprimierung der dünnbesetzten Matrix A mit dem BSR-Verfahren.

Insgesamt werden zur Speicherung einer mit dem BSR-Verfahren komprimierten Matrix folglich $nnzb \cdot r \cdot c + \frac{n}{r} + 1 + nnzb$ Elemente benötigt. Anzumerken bleibt, dass, falls $r = c = 1$, der Speicherplatzverbrauch mit dem des CRS-Verfahrens (siehe Abschn. 4.3.2.1) identisch ist, da ein Block in diesem Fall lediglich ein einzelnes Element speichert und deshalb $nnzb$ durch nnz substituiert werden kann:

$$\begin{aligned} & nnzb \cdot r \cdot c + \tfrac{n}{r} + 1 + nnzb \\ = \; & nnz \cdot 1 \cdot 1 + \tfrac{n}{1} + 1 + nnz \\ = \; & 2 \cdot nnz + n + 1 \qquad \square \end{aligned}$$

4.3.3 RowProxy

Die Klasse `RowProxy` stellt Funktionalität zur Verfügung, um einzelne Elemente eines `DistributedSparseMatrix`-Objekts benutzerfreundlich mit dem Indexoperator auszulesen und ist in der Datei `RowProxy.h` definiert. Die Klasse ist notwendig, da der Indexoperator zwar überladen werden kann,

dieser aber genau *einen* Parameter erwartet [1, S. 230, §13.5.5]. Die Einschränkung hat zur Folge, dass für den Zugriff auf ein Element eines zweidimensionalen Feldes mx die Syntax $mx[i][j]$ statt der mathematischen Schreibweise $mx[i, j]$ verwendet werden muss, $i, j \in \mathbb{N}$. Da mehrdimensionale Felder nicht nativ, sondern lediglich als Felder vom Typ Zeiger unterstützt werden, wird der Ausdruck $mx[i][j]$ konzeptionell in zwei Schritten ausgewertet [1, S. 133 ff., §8.3.4]. Zunächst wertet $mx[i]$ zu einem temporären Zeiger t aus, da mx, wie bereits erwähnt, ein Feld von Zeigern ist. t enthält folglich die Adresse des ersten Elements der i-ten Zeile. Anschließend kann mit $t[j]$ auf das j-te Element der i-ten Zeile zugegriffen werden. Für eine dünnbesetzte Matrix A ergibt sich daher das Problem, dass der Ausdruck $A[i][j]$ zunächst zu einem Ausdruck auswertbar sein muss, der den Indexoperator unterstützt, d.h. zu einem sogenannten *lvalue* [1, S. 66, §5.2.1]. Eine Möglichkeit, dieser Anforderung nachzukommen, besteht darin, dass jeder Prozess sämtliche Werte der i-ten Zeile in einem temporären Feld

Listing 4.3: Definition der Klasse `RowProxy`.

```
1  template<typename T> class DistributedSparseMatrix;
2
3  template<typename T> class RowProxy {
4
5  private:
6
7    int i;    const DistributedSparseMatrix<T>& A;
8
9  public:
10
11   RowProxy(const DistributedSparseMatrix<T>& B): A(B) {
12   }
13
14   void setRowIndex(int i) {
15     this->i = i;
16   }
17
18   T operator[](int j) const {
19     return A.getElement(i, j);
20   }
21
22 };
```

sammelt und anschließend einen Zeiger auf dieses Feld zurückgibt. Diese Vorgehensweise erzeugt jedoch immensen und unnötigen Mehraufwand, da jeder Prozess sämtliche Werte der Zeile sammelt und anschließend lediglich ein einziger davon benötigt wird.

Zur Vermeidung des oben beschriebenen Mehraufwands deklariert die Klasse `RowProxy` eine vergleichsweise simple Schnittstelle (siehe Lst. 4.3). Die Grundidee hierbei besteht darin, den Ausdruck $A[i]$ nicht zu einer vollständigen Zeile einer dünnbesetzten Matrix auszuwerten, sondern zu einem Stellvertreter-Objekt vom Typ `RowProxy`. Dieses muss lediglich den Index der Zeile i speichern und den Indexoperator überladen. Der fehlende Spaltenindex j wird anschließend dem überladenen Indexoperator als Argument übergeben, so dass das `RowProxy`-Objekt auf das entsprechende Element der dünnbesetzten Matrix zugreifen kann. Zu diesem Zweck wird zunächst eine Vorwärtsdeklaration [1, S. 150 f., §9.1.2] vorgenommen, da sich die Klassen `RowProxy` und `DistributedSparseMatrix` gegenseitig referenzieren (siehe Z. 1). Daraus wiederum ergibt sich, dass die Klasse `RowProxy` ebenfalls einen Template-Parameter deklarieren muss, da anderenfalls die Klasse `DistributedSparseMatrix` nicht verwendet werden kann (siehe Z. 3). Die Klasse definiert folgende Funktionen:

`RowProxy(const DistributedSparseMatrix<T>& B)`. Konstruktor, der mit `B` eine Referenz auf das `DistributedSparseMatrix`-Objekt erwartet, dessen Wert ausgelesen werden soll (siehe Z. 11 f.). Die übergebene Referenz dient zur Initialisierung des privaten Attributs `A` (siehe Z. 7). Anzumerken bleibt, dass die erwähnte Initialisierung im Konstruktor mit Hilfe der Initialisierungsliste vorgenommen werden muss, da `A` vom Typ Referenz ist [1, S. 197, §12.6.2.4].

`void setRowIndex(int i)`. Die Funktion erwartet als einziges Argument den Zeilenindex `i` des Elements, welches im Folgenden ausgelesen werden soll (siehe Z. 14 ff.). Der übergebene Index wird im privaten Attribut `i` gespeichert (siehe Z. 7). Wie diese Funktion in der Klasse `DistributedSparseMatrix` zur Verwendung kommt, wird in Abschnitt 4.3.3 erläutert.

`T operator[](int j) const`. Die Funktion überschreibt den Indexoperator und erwartet mit `j` den Spaltenindex des auszulesenden Ele-

ments (siehe Z. 18 ff.). Da der Zeilenindex bereits im Vorhinein über die Funktion `setRowIndex` festgelegt wurde, kann mit Hilfe der Referenz `A` die `getElement`-Funktion des `DistributedSparseMatrix`-Objekts aufgerufen und der entsprechende Wert zurückgegeben werden. Anzumerken bleibt, dass der Indexoperator, entgegen der üblichen Vorgehensweise, keine Referenz, sondern einen Wert zurückgibt, da auch die Funktion `getElement` lediglich einen Wert zurückgibt. Diese Einschränkung ist der Tatsache geschuldet, dass die meisten Kompressionsverfahren, wie z.B. Compressed Row Storage (siehe Abschn. 4.3.2.1), keine Nullen speichern und es in diesem Fall nicht ohne weiteres möglich ist, eine Referenz auf einen nicht existenten Wert zurückzugeben. Infolgedessen kann die Klasse `RowProxy` ausschließlich dazu verwendet werden, auf einzelne Werte mit Hilfe des Indexoperators lesend zuzugreifen, ein schreibender Zugriff ist nicht möglich.

4.3.4 DistributedSparseMatrix

Das Kernstück der Implementierung einer verteilten Datenstruktur für dünnbesetzte Matrizen bildet das Klassentemplate `DistributedSparseMatrix` (siehe Lst. 4.4). Analog zu den Klassen `DistributedArray` und `DistributedMatrix` deklariert dieses einen Template-Parameter `T` und erbt von der Oberklasse `DistributedDataStructure` (siehe Z. 1 ff.). `T` legt den Typ der zu speichernden Elemente fest (siehe Abschn. 4.2.3) und wird standardmäßig mit **`double`** instanziiert, falls der Benutzer keine entsprechende Typisierung vornimmt [1, S. 237, §14.1.9]. Darüber hinaus werden folgende Attribute deklariert, die im Konstruktor initialisiert werden (siehe Abschn. 4.3.4.4):

- `np` speichert die Anzahl der Prozesse, die an der Speicherung der verteilten Datenstruktur beteiligt sind (siehe Z. 7).
- `m` speichert die Spaltenanzahl der dünnbesetzten Matrix (siehe Z. 8). Anzumerken bleibt, dass die Zeilenanzahl `n` von der Klasse `DistributedDataStructure` geerbt wird (siehe Abschn. 3.5.4.1).
- `r` speichert die Zeilenanzahl einer Submatrix (siehe Z. 9). Submatrizen am unteren Rand der dünnbesetzten Matrix weisen u.U. einen kleineren Wert auf, falls $n \bmod r \neq 0$ (siehe Abschn. 4.2.1).

- `c` speichert die Spaltenanzahl einer Submatrix (siehe Z. 10). Submatrizen am rechten Rand der dünnbesetzten Matrix weisen u.U. einen kleineren Wert auf, falls $m \bmod c \neq 0$ (siehe Abschn. 4.2.1).

- `submatrices` speichert die lokal verfügbaren Submatrizen eines Prozesses (siehe Z. 7). Da deren Anzahl zur Laufzeit variieren kann, wird für die Speicherung kein simples Feld vom Typ `Submatrix<T>*`, sondern eine Datenstruktur vom Typ `std::map<int, Submatrix<T>*>` verwendet. Das Klassentemplate `map<K, V>` ist Teil der STL [207] und implementiert einen assoziativen Container zur Speicherung von Schlüssel-Wert-Paaren, dessen Werte aufsteigend nach den Schlüsseln sortiert werden [1, S. 490–493, §23.3.1] [213, S. 436–449]. Die Template-Parameter `K` und `V` legen hierbei den Typ der Schlüssel bzw. der Werte fest. Anzumerken bleibt, dass als Schlüssel die ID der jeweiligen Submatrix verwendet wird. Die Verwendung der Datenstruktur erleichtert zwar die Verwaltung der Submatrizen, nichtsdestotrotz werden zusätzliche Hilfsfunktionen benötigt. Diese werden in Abschnitt 4.3.2 erörtert.

- `distribution` speichert die Adresse des `Distribution`-Objekts, das die Zuordnung von Submatrizen zu Prozessen vornimmt (siehe Z. 8). Wie das Verteilungsschema benutzerdefiniert festgelegt werden kann, wird in Abschnitt 4.3.1 erörtert.

- `submatrix` speichert die Adresse des `Submatrix`-Objekts, das als Prototyp für die Erzeugung neuer Submatrizen benutzt wird (siehe Z. 9).

- `rowProxy` speichert die Adresse des `RowProxy`-Objekts, das den Zugriff auf Elemente der dünnbesetzten Matrix mittels des Indexoperators ermöglicht (siehe Z. 10). Wie die Klasse `DistributedSparseMatrix` das Attribut verwendet, wird in Abschnitt 4.3.3 erörtert.

- `ids` speichert die IDs der MPI-Prozesse, die an der Speicherung der verteilten Datenstruktur beteiligt sind (siehe Z. 9). Diese werden benötigt, falls die verteilte Datenstruktur innerhalb eines taskparallelen Skeletts verschachtelt ist und kollektive, serialisierte Kommunikationsfunktionen verwendet werden (siehe Abschn. 3.5.3).

- `zero` speichert das Nullelement für den Typ `T`, d.h. dasjenige Element, welches von der dünnbesetzten Matrix komprimiert wird, und ist erforderlich, damit die Klasse nicht nur mit primitiven, sondern auch mit benutzerdefinierten Typen verwendet werden kann (siehe Z. 10 und Abschn. 4.2.3). Faktisch lassen sich damit durch eine geeignete Wahl auch von Null verschiedene Elemente komprimieren.

Anzumerken bleibt, dass die Attribute `distribution`, `submatrix` und `rowProxy` nicht vom Typ Objekt oder Referenz sein können, da diese sonst bei der Verwendung der Klasse `DistributedSparseMatrix` vor dem Aufruf des Konstruktors initialisiert werden müssten. Dies ist jedoch nicht möglich, da sowohl `Distribution` als auch `Submatrix<T>` abstrakte Klassen sind und `RowProxy` eine Referenz auf ein `DistributedSparseMatrix`-Objekt benötigt. Die Verwendung von Objekten bzw. Referenzen ist zu bevorzugen, da der hierfür allozierte Speicher vom Laufzeitsystem automatisch freigegeben wird, nachdem der Sichtbarkeitsbereich einer Variablen verlassen wurde [213, S. 310 ff.]. Im Gegensatz dazu muss dynamisch allozierter Speicher hingegen explizit wieder freigegeben werden [213, S. 217–222]. Für die genannten Attribute erfolgt die Allokation des Speichers in der `init`-Funktion der Klasse `DistributedSparseMatrix`, die Deallokation im entsprechenden Destruktor (siehe Abschn. 4.3.4.4).

Listing 4.4: Deklaration und Attribute des Klassentemplates `DistributedSparseMatrix`.

```
1  template<typename T = double>
2  class DistributedSparseMatrix:
3  public DistributedDataStructure {
4
5  private:
6
7    int np;  std::map<int, Submatrix<T>*> submatrices;
8    int m;   Distribution* distribution;
9    int r;   Submatrix<T>* submatrix;        int* ids;
10   int c;   RowProxy<T>*  rowProxy;         T   zero;
11
12 ...
13
14 };
```

Die folgenden Abschnitte 4.3.4.1 bis 4.3.4.4 befassen sich mit weiteren Implementierungsdetails und gehen dabei insbesondere auf die Verwendung der Attribute `distribution`, `submatrix` bzw. `submatrices` und `rowProxy` ein. Zunächst erläutert Abschnitt 4.3.4.1, wie das Verteilungsschema der Submatrizen benutzerdefiniert festgelegt werden kann. Anschließend erörtert Abschnitt 4.3.4.2, wie das Kompressionsverfahren der Submatrizen benutzerdefiniert festgelegt werden kann. Darüber hinaus werden hier die wichtigsten Hilfsfunktionen beschrieben, die zur Verwaltung der Submatrizen benötigt werden. Die Verwendung des überladenen Indexoperators und das Zusammenspiel mit der Klasse `RowProxy` wird in Abschnitt 4.3.4.3 gezeigt. Abschließend widmet sich Abschnitt 4.3.4.4 den zur Verfügung gestellten Konstruktoren sowie dem Destruktor.

4.3.4.1 Verteilung

Wie bereits erwähnt, wurde die Typisierung des Verteilungsschemas der Submatrizen mit Hilfe des Entwurfsmusters *Prototyp* implementiert (siehe Abschn. 4.3.1). Das Entwurfsmuster sieht vor, dass der Benutzer zunächst ein konkretes `Distribution`-Objekt `d` erzeugt und ggf. initialisiert (siehe Abb. 4.5). Anschließend wird ein neues `DistributedSparseMatrix`-Objekt erzeugt und `d` dem entsprechenden Konstruktor als Argument übergeben (siehe Abschn. 4.3.4.4). Innerhalb des Konstruktors wird durch einen Aufruf von `d.clone` eine Kopie von `d` erzeugt. Der Zeiger auf diese Kopie wird in dem lokalen Attribut `distribution` gespeichert (siehe Lst. 4.4, Z. 8). Abschließend werden durch einen Aufruf von `distribution->initialize` die Parameter `n`, `m`, `r`, `c`, `np` und `max` an das Objekt übergeben, welches daraufhin weitere Initialisierungen vornehmen kann.

Alternativ hätte der Mechanismus zur Typisierung des Verteilungsschemas auch mit Hilfe eines Template-Parameters implementiert werden können. Dies wäre jedoch aus mehreren Gründen unvorteilhaft gewesen, die im Folgenden erläutert werden. Hierfür sei angenommen, dass die Klasse `DistributedSparseMatrix` über einen zusätzlichen Template-Parameter `D` verfügt, der den Typ des Verteilungsschemas festlegt (siehe Lst. 4.4, Z. 1). In diesem Fall existieren für den Typ des Attributs `distribution` zwei Möglichkeiten (siehe Z. 8):

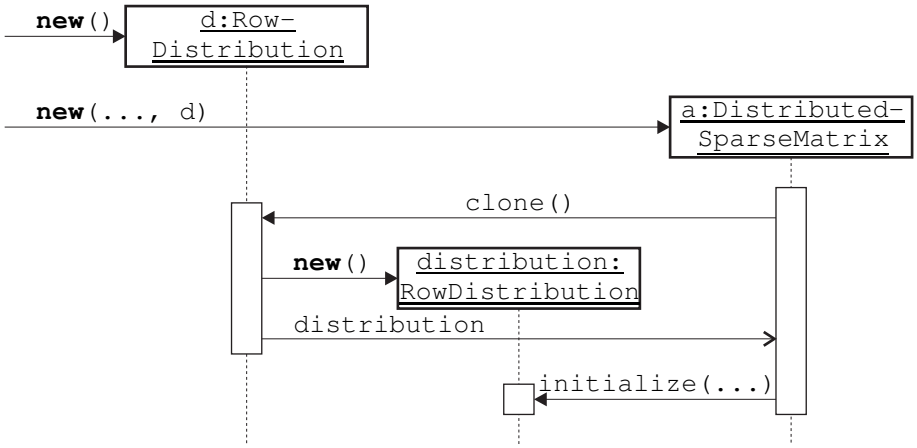


Abbildung 4.5: Sequenzdiagramm zur Konfiguration des Verteilungsschemas einer dünnbesetzten Matrix.

1. Der Typ wird in `D*` geändert, so dass ein neues `Distribution`-Objekt durch den Aufruf von `distribution = new D(...)` erzeugt werden kann. Diese Alternative vermeidet Typkonvertierungen, dies jedoch auf Kosten der Typsicherheit.

2. Der Typ wird nicht geändert, so dass ein neues `Distribution`-Objekt durch den Aufruf von `distribution = (Distribution*)new D(...)` erzeugt werden kann. Diese Alternative bietet zwar Typsicherheit, erfordert allerdings eine explizite Typkonvertierung.

Beide Alternativen weisen, neben den oben genannten Schwächen, einen weiteren erheblichen Nachteil auf, der durch die Auslassungspunkte beim Aufruf des entsprechenden Konstruktors angedeutet wird: Die Initialisierung von benutzerdefinierten Attributen ist schwer möglich, da zur Erzeugung eines neuen `Distribution`-Objekts lediglich ein Konstruktor mit vorher festgelegter Argumentliste aufgerufen werden kann. Diese Argumentliste muss für alle `Distribution`-Objekte identisch sein und kann, zur Gewährleistung der Abwärtskompatibilität, nachträglich nicht einfach geändert werden. Die Verwendung eines zusätzlichen Template-Parameters schränkt in diesem Zusammenhang folglich Erweiterbarkeit und Flexibilität ein, so dass von einer Implementierung auf Basis von Template-Parametern verzichtet wurde.

4.3.4.2 Submatrizen

Wie bereits erwähnt, wurde die Typisierung des Kompressionsverfahrens mit Hilfe des Entwurfsmusters *Prototyp* implementiert (siehe Abschn. 4.3.2). Das Entwurfsmuster sieht vor, dass der Benutzer zunächst ein konkretes `Submatrix`-Objekt `s` erzeugt und ggf. initialisiert (siehe Abb. 4.6). Anschließend wird ein neues `DistributedSparseMatrix`-Objekt erzeugt und `s` dem entsprechenden Konstruktor als Argument übergeben (siehe Abschn. 4.3.4.4). Innerhalb des Konstruktors wird durch einen Aufruf von `s.clone` eine Kopie von `s` erzeugt. Der Zeiger auf diese Kopie wird in dem lokalen Attribut `submatrix` gespeichert (siehe Lst. 4.4, Z. 9). Im Bedarfsfall, wie z.B. bei der Verwendung des Skeletts `zipInPlace`, werden weitere Kopien des Prototyps erzeugt und durch den Aufruf einer entsprechenden `initialize`-Methode initialisiert (siehe Lst. 4.2, Z. 27–32).

Submatrizen werden, wie bereits erwähnt, in einer Datenstruktur vom Typ `std::map< int, Submatrix<T>*>` gespeichert (siehe Abschn. 4.3.4). Dieses Vorgehen erleichtert zwar die Verwaltung der Submatrizen, erfordert

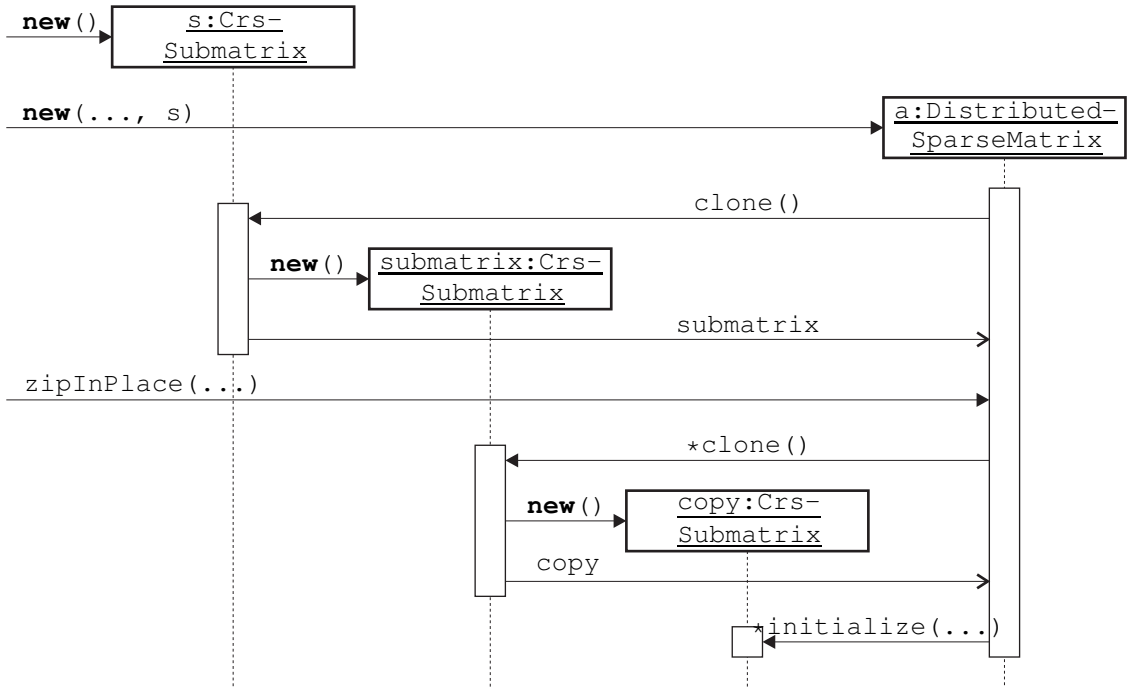


Abbildung 4.6: Sequenzdiagramm zur Konfiguration des Kompressionsverfahrens einer dünnbesetzten Matrix.

dennoch zusätzliche Hilfsfunktionen, um die Datenstruktur effektiv zu manipulieren. Da sämtliche Skelette auf diese Funktionalität angewiesen sind, werden im Folgenden die wichtigsten Hilfsfunktionen erörtert:

Listing 4.5: Quelltext der Funktion `addSubmatrix`.

```
1 void addSubmatrix(Submatrix<T>* s) {
2   submatrices.insert(std::make_pair(s->getId(), s));
3 }
```

`void` `addSubmatrix(Submatrix<T>* s)`. Die Funktion fügt den übergebenen Zeiger auf eine Submatrix der Abbildung hinzu, die die Zeiger auf die lokal verfügbaren Submatrizen speichert (siehe Lst. 4.5). Dazu wird die Funktion `insert` der Klasse `std::map<K, V>` aufgerufen, die als Argument ein Paar vom Typ `std::pair<K, V>` erwartet, d.h. die entsprechenden Typen der Abbildung und des Paares müssen identisch sein. Zur Erzeugung des Paares wird die Funktion `std::make_pair` aufgerufen, die als Argumente die beiden Elemente des Paares erwartet und dieses Paar anschließend zurückgibt [213, S. 443]. Anzumerken bleibt, dass mit der Funktion `getId` der Klasse `Submatrix` die ID der Submatrix bestimmt werden kann.

Listing 4.6: Quelltext der Funktion `getSubmatrix`.

```
1 Submatrix<T>* getSubmatrix(int idSubmatrix) const {
2   typename std::map<int, Submatrix<T>*>::const_iterator
3     iter = submatrices.find(idSubmatrix);
4
5   return iter == submatrices.end() ? NULL : iter->second;
6 }
```

`Submatrix<T>* getSubmatrix(`**`int`** `idSubmatrix)` **`const`** . Die Funktion gibt einen Zeiger auf die Submatrix mit der übergebenen ID zurück, falls diese lokal gespeichert ist (siehe Lst. 4.6). Anderenfalls wird `NULL` zurückgegeben. Dazu wird die Funktion `find` der Klasse `std::map<K, V>` aufgerufen, die als Argument den Schlüssel des gesuchten Elements erwartet. Falls ein Wert mit dem übergebenen Schlüssel existiert, gibt die Funktion einen Iterator auf das gesuchte Schlüssel-Wert-Paar zurück. Falls kein Wert mit dem übergebenen

Schlüssel existiert, wird ein Iterator zurückgegeben, der auf ein imaginäres Element nach dem letzten Element der Abbildung zeigt. Dieser Iterator kann mit der Funktion `end` der Klasse `std::map<K, V>` abgefragt werden. Da die Funktion keinen Zeiger auf das Schlüssel-Wert-Paar, sondern auf die gespeicherte Submatrix zurückgeben soll, wird mit dem Attribut `second` der Struktur `std::pair<T1, T2>` auf das zweite Element des Paares, d.h. den Zeiger auf die Submatrix, zugegriffen [1, S. 358 f., §20.2.2]. Anzumerken bleibt, dass der Deklaration des Iterators `iter` das Schlüsselwort **`typename`** vorangestellt ist, da der Compiler anderenfalls nicht feststellen kann, ob es sich bei `iter` um einen Typ oder einen Wert handelt [213, S. 740].

Listing 4.7: Quelltext der Funktion `getSubmatrixCount`.

```
1 int getSubmatrixCount() const {
2   return (int)submatrices.size();
3 }
```

`int` `getSubmatrixCount()` **`const`**. Die Funktion gibt die Anzahl der von einem Prozess lokal gespeicherten Submatrizen zurück (siehe Lst. 4.7). Dazu wird die Funktion `size` der Klasse `std::map<K, V>` aufgerufen, die die Anzahl der in der Abbildung gespeicherten Schlüssel-Wert-Paare zurückgibt. Zu beachten ist, dass dieser Rückgabewert vom Typ `size_t` [213, S. 136], d.h. vorzeichenlos, ist und statisch in einen Wert vom Typ **`int`**, d.h. vorzeichenbehaftet, konvertiert wird. Dies ist notwendig, da OpenMP 2.5 nur Schleifen parallelisieren kann, bei denen die Indexvariable einen ganzzahligen, vorzeichenbehafteten Typ aufweist [237, S. 34] und innerhalb der Skelette fast ausschließlich diejenigen Schleifen parallelisiert werden, die über die Anzahl der von einem Prozess lokal gespeicherten Submatrizen iterieren (siehe Abschn. 4.3.5). Anzumerken bleibt, dass OpenMP 3.0 diese Beschränkung aufhebt und folglich auch Schleifen parallelisieren kann, deren Indexvariable vom Typ `size_t` ist [238, S. 39].

`Submatrix<T>**` `getSubmatrices()` **`const`**. Die Funktion gibt ein Feld zurück, das Zeiger auf sämtliche lokal gespeicherten Submatrizen enthält (siehe Lst. 4.8). Dieses Vorgehen ist notwendig, da innerhalb der

Listing 4.8: Quelltext der Funktion `getSubmatrices`.

```
1  Submatrix<T>** getSubmatrices() const {
2    int j = 0;
3    typename std::map<int, Submatrix<T>*>::const_iterator
4      iter;
5    Submatrix<T>** tmp = new
6      Submatrix<T>*[getSubmatrixCount()];
7
8    for(iter  = submatrices.begin();
9        iter != submatrices.end(); iter++) {
10     tmp[j++] = iter->second;
11   }
12
13   return tmp;
14 }
```

Skelette fast ausschließlich diejenigen Schleifen parallelisiert werden, die über die Anzahl der von einem Prozess lokal gespeicherten Submatrizen iterieren (siehe Abschn. 4.3.5). Submatrizen werden jedoch in einer Datenstruktur vom Typ `std::map<int, Submatrix<T>*>` gespeichert, so dass ein indizierter Elementzugriff nicht möglich ist.[21] Stattdessen stellt die STL sogenannte *Iteratoren* zur Verfügung, um die Elemente einer Container-Klasse zu traversieren [213, S. 378–382]. OpenMP 2.5 kann jedoch keine Schleifen parallelisieren, die die Iterator-Schnittstelle einer STL Container-Klasse benutzen [179, S. 34f] [237, S. 34]. Um diese Einschränkung zu umgehen, werden sämtliche Zeiger auf lokal gespeicherte Submatrizen in ein temporäres Feld kopiert, auf dessen Elemente anschließend mit Hilfe des Indexoperators zugegriffen werden kann [179, S. 84–87]. Der Mehraufwand für den Aufruf der Funktion kann Tabelle 4.2 entnommen werden und fällt selbst bei $ns = 2^{23} = 8.388.608$ Submatrizen mit einer Sekunde kaum ins Gewicht.[22] Die typische Verwendung der Methode

[21] Abbildungen unterstützen zwar den Elementzugriff mit Hilfe des Indexoperators, hierbei wird der übergebene Index jedoch nicht als Position, sondern als Schlüssel interpretiert [213, S. 439 f.].

[22] Zum Vergleich: Bei dem in Abschnitt 4.4 vorgestellten Bellman-Ford-Algorithmus zur Berechnung der kürzesten Wege speichert die verwendete dünnbesetzte Matrix bei $np = nt = 1$ lediglich $400 < 2^9$ Submatrizen. Zwar wird die Funktion

Tabelle 4.2: Laufzeiten der Funktion `getSubmatrices` in Sekunden.

ns	$t[s]$	ns	$t[s]$	ns	$t[s]$	ns	$t[s]$
2^8	0,000008	2^{12}	0,000142	2^{16}	0,008098	2^{20}	0,116914
2^9	0,000013	2^{13}	0,000377	2^{17}	0,014383	2^{21}	0,249458
2^{10}	0,000024	2^{14}	0,002137	2^{18}	0,031919	2^{22}	0,485766
2^{11}	0,000050	2^{15}	0,004100	2^{19}	0,057265	2^{23}	1,019840

`getSubmatrices` zeigt Listing 4.9. Ein Nachteil dieser Vorgehensweise ist die Tatsache, dass die Aufrufe zur Allokation bzw. zur Deallokation des für das temporäre Feld benötigten Speichers nicht in derselben Funktion stattfinden, d.h. potenziell die Entstehung von Speicherlecks begünstigen. Da es sich bei der Funktion `getSubmatrices` jedoch um eine private Funktion handelt, kann dieser Nachteil in Kauf genommen werden. Anzumerken bleibt, dass OpenMP 3.0 die Parallelisierung von Schleifen unterstützt, die die Iterator-Schnittstelle einer STL Container-Klasse benutzen, falls der entsprechende Iterator einen wahlfreien Zugriff ermöglicht [238, S. 39]. `std::map<K, V>` erfüllt diese Bedingung jedoch nicht, so dass der gewählte Ansatz auch bei der Verwendung von OpenMP 3.0 beibehalten werden muss.

Listing 4.9: Typische Verwendung der Methode `getSubmatrices`.

```
1 Submatrix<T>** smxs = getSubmatrices();
2 ...
3 delete [] smxs;
```

4.3.4.3 Indexoperator

Wie bereits erwähnt, stellt die Klasse `RowProxy` die grundlegende Funktionalität zur Verfügung, um einzelne Elemente einer dünnbesetzten Matrix benutzerfreundlich mit Hilfe des Indexoperators auszulesen (siehe Abschn. 4.3.3). Zu diesem Zweck deklariert die Klasse `DistributedSparseMa-`

`getSubmatrices` ca. 4.000 Mal aufgerufen und verursacht somit einen Mehraufwand von ca. $4.000 \cdot 0,000013\,\text{s} = 0,052\,\text{s}$, in Relation zur gesamten Rechenzeit von ca. 230 s kann dieser aber vernachlässigt werden.

trix mit dem Attribut rowProxy einen Zeiger auf ein Objekt vom Typ RowProxy (siehe Lst. 4.4, Z. 10) und initialisiert diesen in der init-Funktion (siehe Lst. 4.11, Z. 9). Weiterhin stellt die Klasse einen überladenen Indexoperator zur Verfügung, der die von der Klasse RowProxy bereitgestellte Funktionalität benutzt. Wie bereits erwähnt, erwartet der Indexoperator genau ein Argument, so dass der Zugriff auf ein Matrixelement A[i][j] konzeptionell in zwei Schritten stattfindet. Der erste Schritt, d.h. die Auswertung des Ausdrucks a[i], findet innerhalb des von der Klasse DistributedSparseMatrix überladenen Indexoperators statt (siehe Lst. 4.10). Die Funktion erwartet als Argument den Zeilenindex i des auszulesenden Elements und speichert diesen mit Hilfe der setRowIndex-Funktion zunächst im rowProxy-Objekt (siehe Z. 2). Anschließend wird eine Referenz auf das rowProxy-Objekt zurückgegeben (siehe Z. 3). Der zweite Schritt, d.h. die Auswertung des Ausdrucks (*rowProxy)[j], findet innerhalb des von der Klasse RowProxy überladenen Indexoperators statt und wurde bereits in Abschnitt 4.3.3 erläutert (siehe Lst. 4.3, Z. 18 ff.). Anzumerken bleibt, dass der Indexoperator, im Gegensatz zur üblichen Vorgehensweise, nicht mit dem Modifikator **const** deklariert werden konnte, da die Funktion setRowIndex offensichtlich nicht konstant ist (siehe Lst. 4.3, Z. 14 ff.).

Listing 4.10: Überladener Indexoperator der Klasse DistributedSparseMatrix.

```
1 const RowProxy<T>& operator[](int i) {
2   rowProxy->setRowIndex(i);
3   return *rowProxy;
4 }
```

4.3.4.4 Konstruktoren

Die Klasse DistributedSparseMatrix stellt eine Reihe von Konstruktoren zur Verfügung, deren Semantik im Folgenden erläutert wird:

```
DistributedSparseMatrix(int n, int m, int r, int c, T zero, const
    Distribution& d = RoundRobinDistribution(), const Subma-
```

`trix<T>& s = CrsSubmatrix<T>())`. Der Konstruktor erzeugt eine leere `n` × `m` Matrix, die in Submatrizen der Größe `r` × `c` unterteilt wird. Als Verteilungsschema wird standardmäßig die Klasse `RoundRobinDistribution`, als Kompressionsschema die Klasse `CrsSubmatrix` verwendet. Anzumerken bleibt, dass die Validität der Argumente aus Gründen der Performanz nicht überprüft wird, d.h. es wird nicht getestet, ob $0 < r \leqslant n$ und $0 < c \leqslant m$.

`DistributedSparseMatrix(int n, int m, int r, int c, T zero, const T* const * const mx, const Distribution& d = RoundRobinDistribution(), const Submatrix<T>& s = CrsSubmatrix<T>())`. Der Konstruktor erzeugt eine `n` × `m` Matrix, die in Submatrizen der Größe `r` × `c` unterteilt wird, und initialisiert die Werte der Matrix mit Hilfe des zweidimensionalen Feldes `mx`. Als Verteilungsschema wird standardmäßig die Klasse `RoundRobinDistribution`, als Kompressionsschema die Klasse `CrsSubmatrix` verwendet. Anzumerken bleibt, dass, analog zu obigem Konstruktor, die Validität der Argumente nicht überprüft wird. Falls `mx` kleiner als die zu erzeugende Matrix ist, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen. Falls `mx` größer als die zu erzeugende Matrix ist, werden die überschüssigen Elemente bei der Initialisierung der Matrix ignoriert.

`DistributedSparseMatrix(const DistributedSparseMatrix<T>& A)`. Der Konstruktor erwartet als einziges Argument eine konstante Referenz auf ein Objekt vom Typ `DistributedSparseMatrix` und ist daher per Definition ein Kopierkonstruktor [1, S. 203, §12.8.2]. Die Verwendung des Compiler-generierten Kopierkonstruktors ist hier nicht möglich, da die Klasse Zeigerattribute besitzt (siehe Lst. 4.4, Z. 7–10). In diesem Fall kopiert der Compiler-generierte Kopierkonstruktor lediglich die Adressen der Objekte, nicht aber die Objekte selbst, erzeugt also nur eine flache Kopie[23] [1, S. 203–207, §12.8.8][24]. Da dieses

[23] Sei o ein Originalobjekt, welches ein Zeigerattribut a besitzt, d.h. a speichert die Adresse eines im Speicher befindlichen Objekts x. Im Gegensatz zu einer tiefen Kopie zeichnet sich eine flache Kopie c von o dadurch aus, dass $c.a$ denselben Wert hat wie $o.a$, d.h. $o.a$ und $c.a$ verweisen auf dasselbe Objekt x, da von diesem keine Kopie erzeugt wird. Bei einer tiefen Kopie wird von x ebenfalls eine tiefe Kopie x' erzeugt, so dass $c.a$ die Adresse von x' speichert, d.h. $o.a$ und $c.a$ verweisen auf unterschiedliche Objekte x und x'.

[24] Skalare Typen umfassen auch Zeiger auf skalare Typen [1, S. 52, §3.9.10]

Verhalten nicht erwünscht ist, stellt die Klasse einen eigenen Kopierkonstruktor zur Verfügung, der eine tiefe Kopie erzeugt.

Anzumerken bleibt, dass jeder Konstruktor die private Funktion `init` aufruft (siehe Lst. 4.11). Hier werden die privaten Attribute der Klasse initialisiert, das `RowProxy`-Objekt erzeugt sowie die Prototypen der Verteilungs- und Kompressionsverfahren geklont und initialisiert. Der entsprechende Destruktor der Klasse gibt die allozierten Ressourcen wieder frei (siehe Lst. 4.12).

Listing 4.11: Quelltext der Funktion `init`.

```
1  void init(const Distribution& d, const Submatrix<T>& s) {
2    pid = Muesli::MSL_myId - Muesli::MSL_myEntrance;
3    np  = Muesli::MSL_numOfLocalProcs;
4    ids = new int[np];
5
6    for(int i = 0; i < np; i++)
7      ids[i] = i + Muesli::MSL_myEntrance;
8
9    rowProxy     = new RowProxy<T>(*this);
10   submatrix    = s.clone();
11   submatrix->setZero(zero);
12   distribution = d.clone();
13   distribution->initialize(np, n, m, r, c,
14     getMaxSubmatrixCount());
15 }
```

4.3.5 Datenparallele Skelette

Nachdem in Abschnitt 3.4 die grundlegende Semantik der Skelette *count*, *fold*, *map* und *zip* erläutert wurde, wird in den folgenden Abschnitten 4.3.5.1 bis 4.3.5.5 zum einen mit *combine* ein neues Skelett vorgestellt, zum anderen für jedes der erwähnten Skelette eine Implementierungsvariante ausführlich beschrieben. Innerhalb der Skelette wird häufig auf die bereits in Abschnitt 4.3.2 erwähnten Funktionen der Schnittstelle `Submatrix` zurückgegriffen, außerdem werden die von der Klasse `DistributedSparseMatrix` implementierten Hilfsfunktionen benutzt (siehe Abschn. 4.3.4). Des Weiteren findet die in Abschnitt 3.5.2 erörterte OpenMP-Abstraktionsschicht

Listing 4.12: Destruktor der Klasse `DistributedSparseMatrix`.

```
1 ~DistributedSparseMatrix() {
2   deleteSubmatrices();
3   delete rowProxy;
4   delete submatrix;
5   delete distribution;
6   delete [] ids;
7 }
```

OAL Verwendung. Tabelle 4.3 listet die am häufigsten verwendeten Variablen auf und erläutert deren Semantik. Anzumerken bleibt, dass bei einigen Variablen zwischen der Matrix A, d.h. der Matrix, auf der das Skelett ausgeführt wird, und der übergebenen Argumentmatrix B unterschieden wird.

Tabelle 4.3: Semantik der Variablen, die bei der Implementierung der datenparallelen Skelette für die Klasse `DistributedSparseMatrix` häufig verwendet werden.

Variable	Semantik
`ci`, `ri`	Spalten- bzw. Zeilenindex des aktuellen Elements
`tid`	Thread-Nummer bezüglich des Thread-Teams
`lcl`, `glb`	lokales Zwischen- bzw. globales Endergebnis
`ne`	Anzahl der von der aktuellen Submatrix gespeicherten Elemente
`ns`	Anzahl der lokal vorhandenen Submatrizen
`nt`	Größe des Thread-Teams
`smxA`, `smxB`	Zeiger auf eine Submatrix von Matrix A bzw. Matrix B
`smxs`	Feld mit Adressen von lokal vorhandenen Submatrizen
`valA`, `valB`	Element von Matrix A bzw. Matrix B

Wie bereits erwähnt, verwenden sämtliche datenparallelen Skelette der Klasse `DistributedSparseMatrix` OpenMP, um die Ausführung auf Mehrkernprozessoren bzw. hybriden Speicherarchitekturen zu beschleunigen. Hierbei ist zu beachten, dass jedes Skelett stets in einer äußeren Schleife über

die Anzahl der lokal gespeicherten Submatrizen ns sowie in einer inneren Schleife über sämtliche Elemente der aktuellen Submatrix ne iteriert. Unter Berücksichtigung der bereits in Abschnitt 2.4.5 formulierten Regel, bei verschachtelten **for**-Schleifen stets die äußere Schleife zu parallelisieren, kann ein angemessener Speedup folglich nur dann erzielt werden, wenn jeder Rechenkern mindestens eine Submatrix verarbeitet. Die dünnbesetzte Matrix muss folglich derart in Submatrizen zerlegt werden, dass jeder Rechenknoten mindestens nt Submatrizen speichert, d.h. $ns \geqslant nt$. Falls $ns < nt$, wird die Beschleunigung suboptimal ausfallen, was jedoch i.d.R. durch eine geeignete Wahl der Parameter r und c verhindert werden kann, d.h. $r \ll n$ und $c \ll m$.

4.3.5.1 count

Nachdem die Semantik des *count*-Skeletts bereits in Abschnitt 3.4.1 beschrieben wurde, wird im Folgenden die entsprechende Implementierung erläutert (siehe Lst. 4.13). Die Grundidee des Skeletts besteht darin, dass jeder Prozess die übergebene Funktion `f` auf sämtliche Elemente aller lokal gespeicherten Submatrizen anwendet, sein lokales Ergebnis mit allen anderen Prozessen austauscht und gleichzeitig zum globalen Ergebnis reduziert und dieses dann zurückgibt. Zu diesem Zweck wird in einer äußeren **for**-Schleife zunächst über sämtliche lokal vorhandenen Submatrizen iteriert (siehe Z. 12). Hierbei ist wichtig, dass die Abarbeitung der Schleife mit der **parallel for**-Direktive beschleunigt wird (siehe Z. 10 f. und Abschn. 2.4.2.2). Die Direktive verwendet die Klauseln **reduction** und **private**, damit jeder Thread des Thread-Teams eine eigene Kopie der Variablen `lcl`, `ne` und `smxA` erhält und darüber hinaus die Thread-privaten Werte der Variable `lcl` nach der Abarbeitung des parallelen Bereichs aufsummiert werden (siehe Abschn. 2.4.3.1 und 2.4.3.2). In jeder Iteration der äußeren Schleife wird zunächst die Adresse der aktuellen Submatrix gespeichert, danach wird die Anzahl der von dieser Submatrix lokal gespeicherten Elemente bestimmt (siehe Z. 13 f.). Anschließend wird in einer weiteren **for**-Schleife über sämtliche von der aktuellen Submatrix gespeicherten Elemente iteriert (siehe Z. 16). In jeder Iteration dieser inneren Schleife wird die Funktion `f` auf das aktuelle Element angewendet (siehe Z. 17). Gibt die Funktion `f` **true** zurück, so wird der Thread-private Zähler `lcl` um eins inkrementiert (siehe Z. 18).

Listing 4.13: Quelltext des `count`-Skeletts.

```
1  template<typename F>
2  int getElementCount(Fct1<T, bool, F> f) const {
3    int glb = 0;
4    int lcl = 0;
5    int ne  = 0;
6    int ns  = getSubmatrixCount();
7    Submatrix<T>*  smxA = NULL;
8    Submatrix<T>** smxs = getSubmatrices();
9
10   #pragma omp parallel for reduction(+:lcl) \
11                            private(ne, smxA)
12   for(int i = 0; i < ns; i++) {
13     smxA = smxs[i];
14     ne   = smxA->getElementCountLocal();
15
16     for(int j = 0; j < ne; j++) {
17       if(f(smxA->getElementLocal(j))) {
18         lcl++;
19   } } }
20
21   msl::allreduce<int>(&lcl, &glb, add);
22
23   delete [] smxs;
24
25   return res;
26 }
27
28 template<typename T> T add(T a, T b) {
29   return a + b;
30 }
```

Nach der Abarbeitung des parallelen Bereichs tauschen sämtliche Prozesse ihre lokalen Ergebnisse aus und reduzieren diese mit Hilfe der Funktion `add` (siehe Z. 21, Z.28 ff. und Abschn. 3.5.3.3). Abschließend wird der Speicher für das Feld `smxs` freigegeben und das globale Ergebnis zurückgegeben (siehe Z. 23 ff.). Anzumerken bleibt, dass mit `getElementCountIndex` eine Variante des Skeletts zur Verfügung steht, welche der benutzerdefinierten Argumentfunktion neben dem aktuellen Wert zusätzlich dessen Zeilen- und Spaltenindex übergibt.

4.3.5.2 combine

Das *combine*-Skelett ist eine verallgemeinerte Variante der Matrix-Vektor-Multiplikation, bei dem die Operationen zum Verknüpfen der Elemente durch zwei benutzerdefinierte Argumentfunktionen festgelegt werden können, d.h. sowohl von der Multiplikations- als auch der Additionsoperation wird abstrahiert (siehe Lst. 4.14). Das Skelett erwartet mit `x` und `b` zwei Vektoren sowie mit `f` und `g` zwei Argumentfunktionen als Parameter und ist, da es den Zustand der dünnbesetzten Matrix nicht ändert, als **`const`** deklariert (siehe Z. 2 f.). Während `x` den mit der dünnbesetzten Matrix zu kombinierenden Vektor repräsentiert, wird im Vektor `b` das Ergebnis dieser Berechnung gespeichert. Aus diesem Grund müssen `x` und `b` mindestens die Größe `m` bzw. `n` aufweisen. Falls einer der Vektoren kleiner ist, wird die Ausführung des Programms in Folge eines Pufferüberlaufs u.U. abgebrochen. Ist einer der Vektoren größer, werden überschüssige Elemente ignoriert. Die Argumentfunktionen `f` und `g` sind für die Kombination der Elemente zuständig. Während `f` Elemente der Matrix mit Elementen des Vektors kombiniert, verknüpft `g` ausschließlich Elemente der Matrix, d.h. bei einer gewöhnlichen Matrix-Vektor-Multiplikation würde `f` die Multiplikation und `g` die Addition durchführen. Sowohl `f` als auch `g` müssen jeweils zwei Parameter vom Typ `T` entgegennehmen und jeweils einen Wert desselben Typs zurückgeben. Analog zum Skelett *fold* müssen beide sowohl assoziativ als auch kommutativ sein (siehe Abschn. 4.3.5.4).

Die Grundidee des Skeletts besteht darin, dass jeder Prozess über sämtliche lokal vorhandenen Elemente iteriert, diese mit Hilfe von `f` mit dem entsprechenden Element aus `x` kombiniert und das Ergebnis mit Hilfe von `g` mit dem bisherigen Ergebnis aus `b` verknüpft. Problematisch an diesem Ansatz ist jedoch der Umstand, dass eine Parallelisierung mit OpenMP nicht ohne weiteres möglich ist, da in diesem Fall mehrere Threads gleichzeitig sowohl lesend als auch schreibend auf den Vektor `b` zugreifen und es somit zu einer Wettlaufsituation kommen kann [98, S. 32 f.]. Grundsätzlich kann dieses Problem durch die Verwendung der **`critical`**-Direktive vermieden werden (siehe Abschn. 2.4.2.3). Deren Gebrauch ist jedoch vergleichsweise ineffizient, da die Direktive einen hohen Verwaltungsaufwand erzeugt (siehe Tab. 4.4). Um Wettlaufsituationen dennoch zu vermeiden, wurde stattdessen folgender Ansatz implementiert: Jedem Thread wird zunächst ein eige-

Listing 4.14: Quelltext des `combine`-Skeletts (1/2).

```
1  template<typename F>
2  void combine(const T* const x, T* const b,
3  Fct2<T, T, T, F> f, Fct2<T, T, T, F> g) const {
4    int ci  = 0;    int ns = getSubmatrixCount();
5    int ri  = 0;    int nt = oal::getMaxThreads();
6    int ne  = 0;    Submatrix<T>*  smxA = NULL;
7    int tid = 0;    Submatrix<T>** smxs = getSubmatrices();
8    T  valA = zero; T** lcl = new T*[nt];
9
10   #pragma omp parallel private(tid)
11   {
12     tid = oal::getThreadNum();
13
14     #pragma omp for
15     for(int i = 0; i < nt; i++) {
16       lcl[i] = new T[n];
17       memcpy(lcl[i], b, sizeof(T) * n);
18     }
19
20     #pragma omp for private(ci, ne, ri, smxA, val)
21     for(int i = 0; i < ns; i++) {
22       smxA = smxs[i];
23       ne   = smxA->getElementCountLocal();
24
25       for(int j = 0; j < ne; j++) {
26         ci   = smxA->getColumnIndexGlobal(j);
27         ri   = smxA->getRowIndexGlobal(j);
28         valA = f(smxA->getElementLocal(j), x[ci]);
29         lcl[tid][ri] = g(lcl[tid][ri], valA);
30     } }
31
32     ...  // siehe Lst. 4.15
```

ner Speicherbereich zur Verfügung gestellt, in den anschließend der Vektor `b` repliziert wird. Auf diese Weise besitzt jeder Thread eine eigene Kopie des Vektors und kann diesen, unabhängig von anderen Threads, manipulieren. Dieser Ansatz verbraucht zwar mehr Speicherplatz, vermeidet dafür aber die Verwendung der teuren **`critical`**-Direktive und wird im Folgenden als sogenannter *Thread-privater Speicherbereich* bezeichnet (siehe Abschn. 2.4.5).

Tabelle 4.4: Laufzeitvergleich verschiedener Implementierungen des *combine*-Skeletts. Der resultierende Speedup ist in Klammern angegeben. Zu erkennen ist, dass die aktuelle Implementierung (combine) gut skaliert, während eine Implementierung, die die **`critical`**-Direktive benutzt (critical), mit zunehmender Anzahl an Threads langsamer wird. Das Skelett wurde auf einer dünnbesetzten Matrix der Größe 10.000×10.000 ausgeführt, die Argumentfunktionen wurden so gewählt, dass eine gewöhnliche Matrix-Vektor-Multiplikation durchgeführt wird.

$np \setminus nt$		1	2	4	8
1	combine	15,76 (1,0)	7,90 (1,9)	3,95 (3,9)	1,98 (7,9)
	critical	19,69 (1,0)	22,63 (0,8)	43,25 (0,4)	80,04 (0,2)
2	combine	7,96 (1,9)	3,96 (3,9)	2,00 (7,8)	1,08 (14,5)
	critical	9,86 (2,0)	13,85 (1,4)	21,60 (0,9)	39,97 (0,4)
4	combine	3,98 (3,9)	2,65 (5,9)	1,01 (15,6)	0,71 (22,2)
	critical	4,92 (4,0)	5,66 (3,4)	10,76 (1,8)	20,08 (0,9)

Die Thread-privaten Speicherbereiche für den replizierten Vektor `x` werden durch das zweidimensionale Feld `lcl` bereitgestellt (siehe Z. 8). Die Größe des Feldes wird mit Hilfe der OpenMP Abstraktionsschicht bestimmt (siehe Z. 5 und Abschn. 3.5.2). Zur Unterscheidung der Thread-privaten Speicherbereiche wird die ID eines Threads verwendet. Um diese nicht bei jeder Schleifeniteration erneut zu bestimmen, wird zunächst mit der **`parallel`**-Direktive der parallel auszuführende Bereich markiert (siehe Z. 10 und Abschn. 2.4.2.1). Die Direktive verwendet die Klausel **`private`**, damit jeder Thread des Thread-Teams eine eigene Kopie der Variablen `tid` erhält (siehe Abschn. 2.4.3.1). Anschließend kann die ID eines Threads einmalig bestimmt und der Vektor `x` repliziert werden, wobei letzteres durch die Verwendung der **`for`**-Direktive beschleunigt wird (siehe Z. 12–18 und Abschn. 2.4.2.2). Der Kern des Skeletts wird von zwei geschachtelten **`for`**-Schleifen gebildet, wobei die äußere ebenfalls durch die Verwendung der **`for`**-Direktive beschleunigt wird (siehe Z. 20–30). In jeder Iteration der äußeren Schleife wird die Adresse der aktuellen Submatrix, danach die Anzahl der von dieser lokal gespeicherten Elemente bestimmt. In der inneren Schleife werden zunächst der Zeilen- und Spaltenindex des aktuellen Elements berechnet. Anschlie-

ßend wird der Wert des aktuellen Elements ausgelesen und mit Hilfe der Argumentfunktion `f` mit dem entsprechenden Element aus dem Vektor `x` verknüpft. Abschließend wird dieser Wert mit Hilfe der Argumentfunktion `g` mit dem bisherigen Ergebnis kombiniert. Hierbei kommen die Thread-privaten Speicherbereiche zur Geltung: Jeder Thread greift mit Hilfe seiner ID stets auf einen separaten Speicherbereich zu und aktualisiert denjenigen Wert, der dem Zeilenindex des aktuellen Elements entspricht.

Listing 4.15: Quelltext des `combine`-Skeletts (2/2).

```
33      ...  // siehe Lst. 4.14
34
35      #pragma omp for
36      for(int i = 0; i < n; i++) {
37        for(int j = 0; j < nt; j++) {
38          b[i] = g(b[i], lcl[j][i]);
39      } }
40
41    // alternative Implementierung:
42    // #pragma omp for
43    // for(int j = 0; j < nt; j++) {
44    //   for(int i = 0; i < n; i++) {
45    //     #pragma omp critical
46    //     b[i] = g(b[i], lcl[j][i]);
47    // } }
48
49      #pragma omp for
50      for(int i = 0; i < nt; i++) {
51        delete [] lcl[i];
52     } }
53
54    delete [] lcl;
55    delete [] smxs;
56
57    msl::allreduce(b, b, g, n);
58  }
```

Die Einrichtung von Thread-privaten Speicherbereichen ermöglicht auf der einen Seite eine effiziente Parallelisierung, erfordert auf der anderen Seite

zusätzlichen Rechenaufwand, da die Teilergebnisse der einzelnen Threads anschließend miteinander kombiniert werden müssen (siehe Lst. 4.15). Dieser zusätzliche Rechenschritt findet innerhalb von zwei geschachtelten **for**-Schleifen statt, die über sämtliche Elemente aller replizierten Vektoren iterieren, die Zwischenergebnisse mit Hilfe der Argumentfunktion `g` kombinieren und den Ergebnisvektor `b` entsprechend aktualisieren (siehe Z. 35–39). Bemerkenswert an diesem Teil der Implementierung ist weniger die Tatsache, dass die äußere Schleife durch die Verwendung der **for**-Direktive parallelisiert wird, sondern vielmehr der Umstand, dass hierbei die Reihenfolge der Schleifen eine wesentliche Rolle spielt. Wären die Schleifen vertauscht (siehe Z. 42–47), müsste der Zugriff auf den Ergebnisvektor durch die Verwendung der **critical**-Direktive synchronisiert werden, da es in diesem Fall durch den zeitgleichen Zugriff von mehreren Threads auf `b[i]` zu einer Wettlaufsituation kommen kann (siehe Z. 45). Bei der aktuellen Implementierung ist dies hingegen kein Problem, da hier diejenige Schleife parallelisiert wurde, die über die einzelnen Elemente iteriert, so dass jeder Thread exklusiv auf einen Bereich von `b` zugreifen kann (siehe Abschn. 2.4.5).

Nachdem die Teilergebnisse jedes Threads kombiniert und im Vektor `b` gespeichert wurden, wird der für die Felder `lcl` und `smxs` allozierte Speicher freigegeben (siehe Z. 49–55). Abschließend wird der Ergebnisvektor `b` durch einen Aufruf der Funktion `allreduce` unter allen Prozessen ausgetauscht und gleichzeitig kombiniert (siehe Z. 57 und Abschn. 3.5.3.3). Dieser Schritt ist notwendig, da jeder Prozess bisher nur lokal verfügbare Elemente kombiniert hat, zur Berechnung des globalen Ergebnisses müssen diese Elemente jedoch erneut verknüpft werden. Anzumerken bleibt, dass mit `multiply` eine Funktion zur Verfügung steht, die eine gewöhnliche Matrix-Vektor-Multiplikation durchführt. Die Funktion ruft dazu lediglich das oben beschriebene *combine*-Skelett mit zwei speziellen Argumentfunktionen zur Multiplikation bzw. zur Addition von Elementen auf.

4.3.5.3 map

Nachdem die Semantik des *map*-Skeletts bereits in Abschnitt 3.4.3 beschrieben wurde, wird im Folgenden die entsprechende Implementierung erläutert (siehe Lst. 4.16). Die Grundidee besteht darin, zunächst eine Kopie der

dünnbesetzten Matrix zu erzeugen, auf dem das Skelett aufgerufen wurde. Anschließend kann jeder Prozess die Argumentfunktion sukzessive auf jedes Element dieser Kopie anwenden und letztere dann zurückgeben. Zu diesem Zweck wird mit Hilfe des Kopierkonstruktors (siehe Abschn. 4.3.4.4) eine tiefe Kopie des `DistributedSparseMatrix`-Objekts erzeugt, auf dem das Skelett aufgerufen wurde (siehe Z. 5 f.). Anschließend wird ein Feld mit den Adressen sämtlicher Submatrizen dieser Kopie erstellt (siehe Z. 8). In einer äußeren **`for`**-Schleife wird über die Anzahl der lokal verfügbaren Submatrizen iteriert, die offensichtlich mit der Anzahl der von der Kopie gespeicherten Submatrizen übereinstimmt. Hierbei ist wichtig, dass die Abarbeitung der Schleife mit der **`parallel for`**-Direktive beschleunigt wird (siehe Z. 10 und Abschn. 2.4.2.2). Die Direktive verwendet die Klausel **`private`**, damit jeder Thread des Thread-Teams eine eigene Kopie der Variablen `ne` und `smxA` erhält (siehe Abschn. 2.4.3.1). In jeder Iteration der äußeren Schlei-

Listing 4.16: Quelltext des `map`-Skeletts.

```
1  template<typename F>
2  DistributedSparseMatrix<T>* map(Fct1<T, T, F> f) const {
3    int ne = 0;
4    int ns = getSubmatrixCount();
5    DistributedSparseMatrix<T>* B = new
6      DistributedSparseMatrix<T>(*this);
7    Submatrix<T>*  smxA = NULL;
8    Submatrix<T>** smxs = B->getSubmatrices();
9
10   #pragma omp parallel for private(ne, smxA)
11   for(int i = 0; i < ns; i++) {
12     smxA = smxs[i];
13     ne   = smxA->getElementCountLocal();
14
15     for(int j = 0; j < ne; j++) {
16       smxA->setElementLocal(
17         f(smxA->getElementLocal(j)), j);
18   } }
19
20   delete [] smxs;
21
22   return B;
23 }
```

fe wird zunächst die Adresse der aktuellen Submatrix gespeichert, danach wird die Anzahl der von dieser Submatrix lokal gespeicherten Elemente bestimmt (siehe Z. 12 f.). Anschließend wird in einer weiteren **for**-Schleife über sämtliche von der aktuellen Submatrix gespeicherten Elemente iteriert (siehe Z. 15). In jeder Iteration dieser inneren Schleife finden, konzeptionell betrachtet, drei Schritte statt: Zunächst wird das aktuelle Element ausgelesen, anschließend die Argumentfunktion `f` auf dieses angewendet und zuletzt das Ergebnis dieser Anwendung an dieselbe Position zurückgeschrieben (siehe Z. 16 f.). Nach der Abarbeitung des parallelen Bereichs wird der Speicher für das Feld `smxs` freigegeben und die Adresse der neu erzeugten dünnbesetzten Matrix `B` zurückgegeben (siehe Z. 20 ff.).

Anzumerken bleibt, dass der Zustand des `DistributedSparseMatrix`-Objekts, auf dem das Skelett aufgerufen wurde, durch die Funktion nicht verändert wird, weswegen diese mit dem Modifikator **const** deklariert ist (siehe Z. 2). Weiterhin wurden mit `mapIndex`, `mapInPlace` und `mapIndexInPlace` zusätzliche Varianten des *map*-Skeletts implementiert.

4.3.5.4 fold

Nachdem die Semantik des *fold*-Skeletts bereits in Abschnitt 3.4.2 beschrieben wurde, wird im Folgenden die Implementierung am Beispiel des *foldIndex*-Skeletts erläutert (siehe Lst. 4.17). Die Grundidee besteht darin, dass jeder Prozess zunächst sämtliche lokal vorhandenen Elemente mit Hilfe der Argumentfunktion `f` zu einem Zwischenergebnis faltet. Anschließend tauschen alle Prozesse dieses Zwischenergebnis aus und reduzieren dieses gleichzeitig zum globalen Endergebnis der Operation. Problematisch an diesem Ansatz ist jedoch der Umstand, dass, analog zum `combine`-Skelett (siehe Abschn. 4.3.5.2), eine Parallelisierung mit OpenMP nicht ohne weiteres möglich ist, da in diesem Fall mehrere Threads gleichzeitig sowohl lesend als auch schreibend auf das lokale Zwischenergebnis zugreifen und es somit zu einer Wettlaufsituation kommen kann [98, S. 32 f.]. Um letztere zu vermeiden, werden auch hier getrennte Speicherbereiche für die einzelnen Threads des Thread-Teams erzeugt, so dass Zwischenergebnisse unabhängig von anderen Threads berechnet werden können.

Die Thread-privaten Speicherbereiche werden durch das eindimensionale Feld `lcl` bereitgestellt, das für jeden Thread ein lokales Zwischenergebnis speichert, wobei die Größe des Feldes mit Hilfe der OpenMP Abstrakti-

Listing 4.17: Quelltext des `foldIndex`-Skeletts.

```
1  template<typename F>
2  T foldIndex(Fct4<T, T, int, int, T, F> f) const {
3    int ne = 0;                          int tid = 0;
4    int ns = getSubmatrixCount();        T  glb = zero;
5    int nt = oal::getMaxThreads();       T  tmp = zero;
6    Submatrix<T>*  smxA = NULL;          T* lcl = new T[nt];
7    Submatrix<T>** smxs = getSubmatrices();
8
9    #pragma omp parallel private(tid)
10   {
11     tid = oal::getThreadNum();
12
13     #pragma omp for
14     for(int i = 0; i < nt; i++)
15       lcl[i] = zero;
16
17     #pragma omp for private(ne, smxA)
18     for(int i = 0; i < ns; i++) {
19       smxA = smxs[i];
20       ne   = smxA->getElementCountLocal();
21
22       for(int j = 0; j < ne; j++)
23         lcl[tid] = f(lcl[tid], smxA->getElementLocal(j),
24                      smxA->getRowIndexGlobal(j),
25                      smxA->getColumnIndexGlobal(j));
26   } }
27
28   for(int i = 0; i < nt; i++)
29     tmp = f(tmp, lcl[i], -1, -1);
30
31   msl::allreduceIndex(&tmp, &glb, f);
32
33   delete [] smxs;
34   delete [] lcl;
35
36   return glb;
37 }
```

onsschicht bestimmt wird (siehe Z. 5 und Abschn. 3.5.2). Zur Unterscheidung der Speicherbereiche wird die ID eines Threads verwendet. Um diese nicht bei jeder Schleifeniteration erneut zu bestimmen, wird zunächst mit der **`parallel`**-Direktive der parallel auszuführende Bereich markiert (siehe Z. 9 und Abschn. 2.4.2.1). Die Direktive verwendet die **`private`**-Klausel, damit jeder Thread eine eigene Kopie der Variablen `tid` erhält (siehe Abschn. 2.4.3.1). Anschließend kann die ID eines Threads einmalig mit Hilfe der OpenMP Abstraktionsschicht OAL bestimmt werden (siehe Z. 11). Bevor die Faltung der lokal verfügbaren Elemente durchgeführt werden kann, wird jedes Element von `lcl` mit dem Nullelement initialisiert (siehe Z. 13 ff. und Abschn. 4.3.4). Der Kern des Skeletts wird von zwei geschachtelten **`for`**-Schleifen gebildet, wobei die äußere durch die Verwendung der **`for`**-Direktive beschleunigt wird (siehe Z. 17–26 und Abschn. 2.4.2.2). In jeder Iteration der äußeren Schleife wird zunächst die Adresse der aktuellen Submatrix, anschließend die Anzahl der von dieser lokal gespeicherten Elemente bestimmt. Jede Iteration der inneren Schleife findet, konzeptionell betrachtet, in drei Schritten statt: Zunächst wird das aktuelle Element bestimmt sowie dessen globaler Zeilen- und Spaltenindex berechnet. Anschließend wird das neue Zwischenergebnis berechnet, indem des aktuelle Element mit dem bisherigen Zwischenergebnis gefaltet wird. Abschließend wird das bisherige Zwischenergebnis mit dem neuen Zwischenergebnis überschrieben. Hierbei kommen die Thread-privaten Speicherbereiche zur Geltung: Jeder Thread greift mit Hilfe seiner ID ausschließlich auf das von ihm berechnete Zwischenergebnis zu.

Nach der Abarbeitung des parallelen Bereichs muss jeder Prozess die lokalen Ergebnisse jedes Threads erneut falten, um diese zu einem singulären Wert zu verdichten (siehe Z. 28 f.). Hierbei ist zu beachten, dass die lokalen Ergebnisse offensichtlich weder über einen Zeilen-, noch über einen Spaltenindex verfügen, die Argumentfunktion `f` diese jedoch erwartet. Statt eine zweite Argumentfunktion `g` zu verwenden, werden der Zeilen- und der Spaltenindex jeweils auf -1 gesetzt. Anschließend tauschen sämtliche Prozesse ihre lokalen Ergebnisse aus und reduzieren diese gleichzeitig mit Hilfe der Argumentfunktion `f` zum globalen Endergebnis (siehe Z. 31 und Abschn. 3.5.3.3). Abschließend wird der für die Felder `smxs` und `lcl` allozierte Speicher wieder freigegeben und das Ergebnis der Faltung zurückgegeben (siehe Z. 33–36).

4.3.5.5 zip

Nachdem die Semantik des `zip`-Skeletts bereits in Abschnitt 3.4.5 beschrieben wurde, wird im Folgenden die Implementierung am Beispiel des *zipInPlace*-Skeletts erläutert (siehe Lst. 4.18 und 4.19). Verglichen mit den Skeletten *combine*, *count*, *fold* und *map* erfordert *zip* deutlich mehr Aufwand, da beim Zusammenfügen zweier dünnbesetzter Matrizen A und B der Fall eintreten kann, dass Matrix A Elemente speichert, die Matrix B nicht speichert und umgekehrt. Nichtsdestotrotz müssen sämtliche von Null verschiedenen Elemente beider Matrizen berücksichtigt werden. Um dies zu gewährleisten, ist es zunächst erforderlich, dass beide Matrizen identische Verteilungsverfahren verwenden, d.h. korrespondierende Submatrizen von Matrix A und Matrix B müssen von demselben Prozess gespeichert werden (siehe Z. 11 und Abschn. 4.3.1). Anschließend wird mit der **`parallel`**-Direktive der parallel auszuführende Bereich markiert und gleichzeitig die Variable `tid` als **`private`** deklariert (siehe Z. 7, 12 und Abschn. 2.4.3.1). Letztere wird, analog zu den Skeletten `combine` und `fold`, mit Hilfe der OpenMP Abstraktionsschicht OAL bestimmt und ermöglicht die Benutzung von Thread-privaten Speicherbereichen (siehe Z. 14 und Abschn. 3.5.2). Weshalb diese hier benötigt werden, wird im weiteren Verlauf erörtert. Anschließend wird in einer äußeren **`for`**-Schleife über die Anzahl der Submatrizen iteriert, in die die dünnbesetzte Matrix durch die Parameter r und c aufgeteilt wird (siehe Z. 4 und 18). Hierbei ist wichtig, dass die Abarbeitung der Schleife durch die Verwendung der **`for`**-Direktive beschleunigt wird (siehe Z. 16 f. und Abschn. 2.4.2.2). Die Direktive verwendet die **`private`**-Klausel, damit jeder Thread eine eigene Kopie der Variablen `ci`, `idx`, `ne`, `ri`, `smxA`, `smxB`, `valA` und `valB` erhält (siehe Abschn. 2.4.3.1). In jeder Iteration der äußeren Schleife wird mit Hilfe des `distribution`-Objekts zunächst geprüft, ob die aktuelle Submatrix dem Prozess zugeordnet wurde, auf dem das Programm ausgeführt wird (siehe Z. 19). Ist dies nicht der Fall, kann mit der nächsten Submatrix fortgefahren werden. Anderenfalls werden mit Hilfe der Funktion `getSubmatrix` (siehe Abschn. 4.3.4.2, Lst. 4.6) die Adressen der Submatrizen mit der aktuellen ID bestimmt (siehe Z. 20 f.). Falls keine der beiden Submatrizen lokal gespeichert wird, d.h. falls sowohl `smxA == NULL` als auch `smxB == NULL` gilt, kann mit der nächsten Submatrix fortgefahren werden (siehe Z. 23 und Lst. 4.19, Z. 44). Falls Matrix A die aktuelle Sub-

Listing 4.18: Quelltext des `zipInPlace`-Skeletts (1/2).

```
1  template<typename F> void zipInPlace(const
2  DistributedSparseMatrix<T>& B, Fct2<T, T, T, F> f) {
3    int ci  = 0;  int nt  = oal::getMaxThreads();
4    int ne  = 0;  int max = getMaxSubmatrixCount();
5    int ns  = 0;  T  valA = zero;  Submatrix<T>* smxA;
6    int ri  = 0;  T  valB = zero;  Submatrix<T>* smxB;
7    int tid = 0;  std::set<MatrixIndex> idx;
8    std::vector<Submatrix<T>*>* tmp = new
9       std::vector<Submatrix<T>*>[nt];
10
11   if(distribution->equals(*B.distribution)) {
12     #pragma omp parallel private(tid)
13     {
14       tid = oal::getThreadNum();
15
16       #pragma omp for private(ci, idx, ne, ri, smxA, \
17                               smxB, valA, valB)
18       for(int idSmx = 0; idSmx < max; idSmx++) {
19         if(distribution->isStoredLocally(pid, idSmx)) {
20           smxA =   getSubmatrix(idSmx);
21           smxB = B.getSubmatrix(idSmx);
22
23           if(smxA != NULL) {
24             if(smxB == NULL)
25               valB = zero;
26
27             ne = smxA->getElementCountLocal();
28
29             for(int k = 0; k < ne; k++) {
30               valA = smxA->getElementLocal(k);
31               ri   = smxA->getRowIndexLocal(k);
32               ci   = smxA->getColumnIndexLocal(k);
33
34               if(smxB != NULL)
35                 valB = smxB->getElement(ri, ci);
36
37               smxA->setElementLocal(f(valA, valB), k);
38               idx.insert(MatrixIndex(ri, ci));
39           } }
40
41           ...  // siehe Lst. 4.19
```

matrix jedoch speichert, müssen sämtliche Werte dieser Submatrix mit den entsprechenden Werten der Submatrix von Matrix B kombiniert werden. Zu diesem Zweck wird zunächst bestimmt, ob die entsprechende Submatrix von Matrix B gespeichert wird und, falls dies nicht der Fall ist, der zu kombinierende Wert von Matrix B für den weiteren Verlauf der Iteration auf `zero` gesetzt (siehe Z. 24 f.). In einer inneren **`for`**-Schleife wird anschließend über die lokal gespeicherten Elemente von `smxA` iteriert (siehe Z. 27 und 29). In jeder Iteration dieser inneren Schleife wird der Wert, der Zeilen- sowie der Spaltenindex des aktuellen Elements von Matrix A bestimmt (siehe Z. 30 ff.). Falls die entsprechende Submatrix von Matrix B gespeichert wird, wird dessen Wert ausgelesen (siehe Z. 34 f.), anderenfalls gilt weiterhin `valB = zero` (siehe Z. 24 f.). Anschließend können beide Elemente mit Hilfe der Argumentfunktion `f` kombiniert und das Ergebnis in `smxA` gespeichert werden (siehe Z. 37). Abschließend wird mit `idx` eine Datenstruktur vom Typ `std::set<MatrixIndex>` verwendet, um im weiteren Verlauf der Funktion das wiederholte Kombinieren von Elementen zu verhindern (siehe Z. 7 und 38). Das Klassen-Template `std::set<T>` ist Teil der STL [207] und implementiert einen assoziativen Container zur Speicherung von Schlüsseln vom Typ `T` [1, S. 496–499, §23.3.3] [213, S. 450–453]. In diesem Fall werden Objekte vom Typ `MatrixIndex` (siehe Anh. A.4) verwendet, die den lokalen Zeilen- und Spaltenindex des jeweils aktuell kombinierten Elements von Matrix A kapseln. Durch dieses Vorgehen speichert `idx` für jede Iteration der äußeren Schleife, d.h. für jede Submatrix, die lokalen Zeilen- und Spaltenindexe derjenigen Elemente von Matrix A, die bereits kombiniert wurden.

Bis hierhin wurden sämtliche von Null verschiedenen Elemente von `smxA` mit den entsprechenden Elementen von `smxB` kombiniert. Um jedoch alle von Null verschiedenen Elemente beider Matrizen zu berücksichtigen, müssen darüber hinaus diejenigen Elemente von `smxB` mit den entsprechenden Elementen von `smxA` kombiniert werden, die bis jetzt noch nicht kombiniert wurden. Dieser Fall tritt ein, wenn `smxB` Elemente speichert, die `smxA` nicht speichert. Zu diesem Zweck muss, falls Matrix B die aktuelle Submatrix speichert, d.h. falls `smxB != NULL` gilt, über sämtliche Elemente von `smxB` iteriert werden (siehe Lst. 4.19, Z. 44). Dazu wird zunächst die Anzahl der von `smxB` lokal gespeicherten Elemente bestimmt, um anschließend über diese in einer inneren **`for`**-Schleife zu iterieren (siehe Z. 45 ff.). In jeder

Listing 4.19: Quelltext des `zipInPlace`-Skeletts (2/2).

```
42            ...  // siehe Lst. 4.18
43
44            if(smxB != NULL) {
45              ne = smxB->getElementCountLocal();
46
47              for(int k = 0; k < ne; k++) {
48                ri = smxB->getRowIndexLocal(k);
49                ci = smxB->getColumnIndexLocal(k);
50
51                if(idx.count(MatrixIndex(ri, ci)) == 0) {
52                  valA = f(zero, smxB->getElementLocal(k));
53
54                  if(smxA == NULL) {
55                    smxA = submatrix->clone();
56                    smxA->initialize(idSmx,
57                      getRowCount(idSmx),
58                      getColumnCount(idSmx),
59                      getRowIndexStart(idSmx),
60                      getColumnIndexStart(idSmx),
61                      valA, ri, ci);
62                    tmp[tid].push_back(smxA);
63                  }
64                  else {
65                    smxA->setElement(valA, ri, ci);
66            } } } }
67
68            idx.clear();
69      } } }
70
71      for(int i = 0; i < nt; i++) {
72        ns = tmp[i].size();
73
74        for(int j = 0; j < ns; j++) {
75          addSubmatrix(tmp[i][j]);
76      } }
77
78      delete [] tmp;
79  } }
```

Iteration dieser inneren Schleife wird der lokale Zeilen- sowie der lokale Spaltenindex des aktuellen Elements von Matrix B berechnet (siehe Z. 48 f.). Um das wiederholte Kombinieren von Elementen zu verhindern, wird zunächst überprüft, ob bereits ein Element mit entsprechenden Indexen kombiniert wurde. Hierfür wird ein temporäres Objekt vom Typ `MatrixIndex` erzeugt, das den Zeilen- und Spaltenindex des aktuellen Elements von Matrix B kapselt, und mit Hilfe der Funktion `count` der Klasse `std::set<T>` bestimmt, ob die Menge `idx` bereits ein Element mit identischen Indizes enthält (siehe Z. 51). Ist dies der Fall, wurde das aktuelle Element von Matrix B bereits kombiniert, so dass mit dem nächsten Element fortgefahren werden kann. Anderenfalls wird das aktuelle Element bestimmt und mit `zero` kombiniert (siehe Z. 52). Die pauschale Kombination mit `zero` ist korrekt, da bereits sämtliche von Null verschiedenen Elemente von `smxA` berücksichtigt wurden, so dass der Wert des entsprechenden Elements aus Matrix A nur `zero` sein kann. Abschließend muss der aktuell kombinierte Wert in Matrix A gespeichert werden. Falls die entsprechende Submatrix `smxA` bereits existiert, d.h. falls `smxA != NULL` gilt, kann der Wert mit Hilfe der Funktion `setElement` gespeichert werden (siehe Z. 64 f.). Anderenfalls muss die entsprechende Submatrix erst erzeugt werden, indem, wie bereits erwähnt, eine Kopie des Submatrix-Prototypen angefertigt wird (siehe Z. 54–60 und Abschn. 4.3.2). Nach der Initialisierung von `smxA` kommen die Thread-privaten Speicherbereiche zur Geltung: Statt `smxA` direkt der Matrix A hinzuzufügen, wird die Submatrix mit Hilfe der Variablen `tid` zunächst einem Thread-privaten Vektor angefügt (siehe Z. 62 und Lst. 4.18, Z. 8 f.). Die Verwendung von Vektoren ist an dieser Stelle notwendig, da die Anzahl der neu zu erzeugenden bzw. zu klonenden Submatrizen im Vorhinein nicht bekannt ist. Die Benutzung von Thread-privaten Speicherbereichen ist hier von Vorteil, weil dadurch die Verwendung der vergleichsweise langsamen **`critical`**-Direktive vermieden wird (siehe Abschn. 4.3.5.2). Letztere müsste jedoch verwendet werden, wenn `smxA` mit Hilfe der Funktion `addSubmatrix` direkt der Matrix A hinzugefügt werden würde, da die Funktion den Zustand der Matrix A verändert (siehe Abschn. 4.3.4.2). Diese Zustandsänderung erfolgt jedoch nicht atomar, so dass das gleichzeitige Ausführen der Funktion dazu führen kann, dass die Matrix A in einen inkonsistenten Zustand versetzt wird, was offensichtlich verhindert werden müsste (siehe Abschn. 2.4.5). Anzumerken bleibt, dass der Aufruf der Funktionen `setElement` und `initialize` nicht

synchronisiert werden muss, obwohl die Funktionen ebenfalls nicht mit dem Modifikator **const** deklariert sind (siehe Abschn. 4.3.2). Der Unterschied zur Funktion `addSubmatrix` besteht jedoch darin, dass diese den Zustand der *Matrix* manipuliert, während `setElement` den Zustand der aktuellen *Submatrix* verändert. Auf eine Submatrix greifen jedoch niemals mehrere Threads gleichzeitig zu, was durch die Verwendung der **parallel for**-Direktive sichergestellt ist (siehe Lst. 4.18, Z. 16 f.). Nach der Abarbeitung des parallelen Bereichs werden die von den Threads erzeugten bzw. geklonten Submatrizen der Matrix A hinzugefügt (siehe Z. 71–76). Anzumerken bleibt, dass die Verwendung der Funktion `addSubmatrix` nicht synchronisiert werden muss, da dieser Bereich nicht mehr parallel, sondern sequenziell ausgeführt wird. Abschließend wird der für das temporäre Feld `tmp` allozierte Speicher wieder freigegeben.

4.4 Ergebnisse

Im Folgenden werden die Ergebnisse einiger auf dem ZIVHPC (siehe Abschn. 2.5) durchgeführten Laufzeitmessungen erläutert. Das implementierte Testprogramm löst mit Hilfe des Bellman-Ford-Algorithmus [43, 148] ein sogenanntes *Single Source Shortest Path Problem*, berechnet also, ausgehend von einer Ecke, die kürzesten Wege zu allen anderen Ecken in einem gewichteten Digraph[25] [286]. Der Digraph wird durch eine Adjazenzmatrix repräsentiert, die mit Hilfe der Klasse `DistributedSparseMatrix` gespeichert wird (siehe Lst. 4.20). Im Gegensatz zu Dijkstras Algorithmus [129] können die Kantengewichte hierbei jedoch negativ sein, so dass bei der Suche u.U. Kreise mit einer negativen Länge gefunden werden [201, S. 181–186]. Infolgedessen kann die Entfernung zwischen zwei Knoten durch mehrmaliges Traversieren dieser Kreise beliebig reduziert werden, so dass in diesem Fall offensichtlich keine Lösung gefunden werden kann. Die vorgenommene Implementierung ist jedoch in der Lage, einen solchen Fall zu erkennen und den Kreis negativer Länge zu identifizieren, so dass dieser vom Benutzer manuell korrigiert werden kann. Anzumerken bleibt, dass die Suche nach kürzesten Wegen vorzeitig abgebrochen wird, falls in einer Iteration keine kürzeren Wege gefunden wurden.

[25] Ein Digraph ist ein gerichteter Graph (engl. directed graph).

Listing 4.20: Vereinfachte Implementierung des Bellman-Ford-Algorithmus zur Berechnung kürzester Wege in einem gewichteten Digraph.

```
1  void ssp(DistributedSparseMatrix<>& adjacencyMatrix,
2           int source) {
3    int* distances = new int[m];
4    DistributedSparseMatrix<> distanceMarker(n, m, r, c);
5
6    for(int i = 1; i < m; i++) {
7      distanceMarker.zipIndexInPlace(adjacencyMatrix,
8                                     &shorterDistance);
9      distanceMarker.foldColumns(&shortestDistance,
10                                distances);
11     ... // Test auf vorzeitigen Abbruch
12   }
13
14   ... // Kreise negativer Länge suchen
15 }
```

Die durchgeführten Benchmarks verwenden einen zufällig erzeugten Digraphen mit 4.000 Knoten und 25.000 Kanten, d.h. der Füllgrad der entsprechenden dünnbesetzten Matrix beträgt ca. 0,156 %. Die gemessenen Laufzeiten können Tabelle 4.5 entnommen werden. Zu erkennen ist, dass das Programm zwar mit steigender Anzahl an Prozessen deutlich besser skaliert als mit steigender Anzahl an Threads, der unter Verwendung von $np = 16$ Prozessen und $nt = 8$ Threads erzielte Speedup jedoch vergleichsweise gering ist. Hierfür gibt es im Wesentlichen zwei Gründe:

- Die vorgenommene Implementierung des Bellman-Ford-Algorithmus erfordert einen hohen Kommunikationsaufwand, der durch das *foldColumns*-Skelett verursacht wird. Dieser Aufwand nimmt mit steigender Anzahl an Prozessen zu. Nichtsdestotrotz kann die Dauer für die Berechnung der kürzesten Wege reduziert werden, da letztere auf mehrere Prozesse verteilt wird, so dass jeder Prozess weniger Elemente bearbeiten muss. Der erzielte Speedup wird jedoch durch den Kommunikationsaufwand beschränkt.
- Das schlechte Skalierungsverhalten bei einer Erhöhung der verwendeten Threads ist auf den hohen sequenziellen Anteil der Implementierung zurückzuführen, der jedoch aus Gründen der Übersichtlichkeit

nicht in Listing 4.20 enthalten ist. Während die unter Verwendung datenparalleler Skelette implementierten Teile des Algorithmus von einer höheren Anzahl an Threads profitieren, ist dies bei den sequenziellen Abschnitten erwartungsgemäß nicht der Fall. Diese sequenziellen Abschnitte stellen gewissermaßen Fixkosten dar, die durch eine Erhöhung der Threads nicht reduziert werden können und somit den erzielbaren Speedup beschränken.

Tabelle 4.5: Laufzeiten einer Suche nach kürzesten Wegen unter Verwendung des Bellman-Ford-Algorithmus in Sekunden. Der Speedup ist jeweils in Klammern angegeben.

$nt \setminus np$	1	2	4	8	16
1	230,6 (1,0)	116,5 (1,9)	62,6 (3,6)	48,8 (4,7)	27,5 (8,3)
2	117,4 (1,9)	59,9 (3,8)	32,4 (7,1)	23,6 (9,7)	17,9 (12,8)
4	84,8 (2,7)	43,6 (5,2)	21,4 (10,7)	13,3 (17,2)	10,3 (22,3)
8	61,4 (3,7)	30,4 (7,5)	14,6 (15,7)	9,6 (23,9)	7,3 (31,4)

4.5 Fazit

In diesem Kapitel wurde mit der verteilten Datenstruktur für dünnbesetzte Matrizen eine wesentliche Erweiterung der Skelettbibliothek beschrieben. Die Datenstruktur verfügt über ein flexibles Designkonzept und zerlegt, in Anlehnung an das bereits erläuterte Parallelisierungskonzept von Muesli (siehe Abschn. 3.2.1), die dünnbesetzte Matrix in mehrere Partitionen, sogenannte Submatrizen, die durch einen integrierten Mechanismus zur Lastverteilung den verfügbaren Prozessen zugeordnet werden (siehe Abschn. 4.2.1). Das von der Datenstruktur implementierte Kompressionsverfahren ist zweistufig konzipiert und gewährleistet, dass leere Submatrizen nicht gespeichert und solche, die mindestens ein Element speichern, komprimiert werden (siehe Abschn. 4.2.2). Anzumerken bleibt, dass durch die Verwendung eines Template-Parameters ebenfalls das Konzept des parametrischen Polymorphismus implementiert und so vom Typ der zu speichernden Elemente abstrahiert wird (siehe Abschn. 4.2.3).

Neben den grundlegenden Konzepten wurden in diesem Kapitel auch die entsprechenden Implementierungsdetails ausführlich erläutert. Während die Zuordnung zwischen Submatrizen und Prozessen mit Hilfe eines `Distribution`-Objekts erfolgt (siehe Abschn. 4.3.1), wird die Kompression von einem `Submatrix`-Objekt durchgeführt (siehe Abschn. 4.3.2). Durch die Verwendung dieser abstrakten Oberklassen ist gewährleistet, dass die Mechanismen zur Lastverteilung und Kompression benutzerdefiniert erweitert werden können, um so die Anforderungen einer speziellen Anwendung stärker zu berücksichtigen. Das Kernstück der Implementierung bildet das Klassen-Template `DistributedSparseMatrix`, dessen Eigenschaften und wichtigste Interna ebenfalls detailliert beschrieben wurden (siehe Abschn. 4.3.4). Darauf aufbauend wurden die von der Klasse zur Verfügung gestellten datenparallelen Skelette ausführlich erläutert (siehe Abschn. 4.3.5). Die durchgeführten Laufzeitmessungen haben gezeigt, dass die Datenstruktur die Parallelisierung einer realen Anwendungen ermöglicht und dem Anspruch, die Erstellung paralleler Programme zu vereinfachen, gerecht wird (siehe Abschn. 4.4).

Kapitel 5

Medizinische Bildrekonstruktion mit LM OSEM

Inhalt

5.1 Einleitung

Die Positronen-Emissions-Tomographie ist ein nicht-invasives, bildgebendes Verfahren der Nuklearmedizin, das hauptsächlich bei der Visualisierung von Stoffwechselprozessen in lebenden Organismen Anwendung findet [208, 263]. Dem Versuchsobjekt wird dabei eine schwach radioaktive Substanz verabreicht, die sich auf Grund ihrer chemischen Eigenschaften an den zu beobachtenden Stellen im Körper anreichert. Anschließend wird das Versuchsobjekt in einen PET-Scanner platziert, der den Zerfall der Substanz registriert und

aufzeichnet. Aus den so erzeugten Rohdaten wird abschließend ein dreidimensionales Bild rekonstruiert, das die Verteilung der Substanz im Körper des Versuchsobjekts visualisiert.

Für die eigentliche Bildrekonstruktion können verschiedene Algorithmen benutzt werden, die sich vor allem durch die Qualität des rekonstruierten Bildes und die Konvergenzrate unterscheiden. Ein gängiges Verfahren ist *Expectation Maximization–Maximum Likelihood* [271]. Der Algorithmus erzeugt zwar vergleichsweise hochwertige Bilder, dies jedoch auf Kosten einer geringen Konvergenzrate. Letzteres behebt *Ordered Subset Expectation Maximization* [182], indem die Rohdaten in mehrere disjunkte Teilmengen unterteilt werden. Der Algorithmus erzielt einen Beschleunigungsfaktor, der proportional zur Anzahl der Teilmengen ist. Der im Folgenden verwendete Algorithmus *List-Mode Ordered Subset Expectation Maximization* [258] kombiniert die Vorteile der oben genannten Verfahren, erzeugt also hochwertige Bilder mit einer schnellen Konvergenzrate. Nichtsdestotrotz dauert die Rekonstruktion eines Bildes auf einem Einkernprozessor mehrere Stunden [266], da die Rohdaten i.d.R. mehrere Millionen Zerfallsereignisse enthalten.

Im Folgenden wird beschrieben, wie der LM OSEM-Algorithmus unter Verwendung der Münster Skelettbibliothek parallelisiert werden kann [101]. Entwickelt wurden sowohl eine task- als auch zwei datenparallele Lösungen. Die Effizienz dieser Implementierungen wird mit mehreren Varianten verglichen, die Bibliotheken zur Parallelisierung eines Programms, wie z.B. MPI, OpenMP oder TBB, direkt benutzen. Diese Varianten wurden in der Arbeitsgruppe PVS an der Universität Münster entwickelt [101, 192, 263, 264, 265, 266, 267, 268]. Anzumerken bleibt, dass der LM OSEM-Algorithmus bereits erfolgreich und kosteneffektiv unter Verwendung von Grafikkarten parallelisiert wurde [267, 268]. Darüber hinaus existiert eine Implementierung, die in einer Grid-Umgebung skaliert [265]. Die beiden letztgenannten Versionen spielen in der folgenden Betrachtung jedoch keine Rolle.

Der Rest des Kapitels ist wie folgt aufgebaut: Zunächst werden in Abschnitt 5.2 die wesentlichen Grundlagen der medizinischen Bildrekonstruktion beschrieben. Darauf aufbauend werden in Abschnitt 5.3 mehrere Implementierungen vorgestellt, die den LM OSEM-Algorithmus einerseits mit Hilfe der Münster Skelettbibliothek, andererseits unter direkter Verwendung

von MPI, OpenMP und TBB parallelisieren. Abschnitt 5.4 zeigt die Ergebnisse der durchgeführten Laufzeitmessungen, Abschnitt 5.5 schließt das Kapitel mit einem Fazit ab.

5.2 Grundlagen

Die folgenden Abschnitte 5.2.1 bis 5.2.3 behandeln die wesentlichen Grundlagen der PET-basierten, medizinischen Bildrekonstruktion unter Verwendung des LM OSEM-Algorithmus. Abschnitt 5.2.1 erläutert zunächst, wie die für die Rekonstruktion benötigten Rohdaten erfasst werden. Darauf aufbauend erörtert Abschnitt 5.2.2 den für die Rekonstruktion verwendeten LM OSEM-Algorithmus. Abschließend werden in Abschnitt 5.2.3 Ansätze zur effizienten Parallelisierung der Bildrekonstruktion beschrieben.

5.2.1 Erfassung der Rohdaten

Die Erfassung der Rohdaten erfolgt in zwei Phasen und erfordert den kombinierten Einsatz eines Radiopharmakons sowie eines PET-Scanners [263]:

- Dem Versuchsobjekt wird ein sogenanntes *Radiopharmakon* verabreicht, wahlweise per Injektion oder Inhalation. Das Radiopharmakon, der sogenannte *Tracer*, ist eine mit einem Radionuklid markierte Substanz und reichert sich in einer sogenannten *Uptake-Phase* in Abhängigkeit von den pharmakologischen bzw. radiochemischen Eigenschaften des Nuklids in den zu beobachtenden Organen im Körper an. Der Tracer zerfällt auf Grund seiner radioaktiven Eigenschaften im Zeitverlauf und setzt dabei Positronen frei, die mit im Körper befindlichen Elektronen reagieren. Bei dieser Wechselwirkung, der sogenannten *Annihilation*, werden beide Teilchen unter Aussendung von zwei Photonen vernichtet. Das besondere Merkmal dieser *Vernichtungsstrahlung* ist der Umstand, dass die beiden Photonen in entgegengesetzter Richtung, d.h. unter einem Winkel von 180°, ausgesendet

werden.[26] Diese Eigenschaft ist die Grundlage des Funktionsprinzips eines PET-Scanners.

- Im Anschluss an die Uptake-Phase wird das Versuchsobjekt in den Sichtbereich (engl. field of view, FOV) des PET-Scanners gelegt. Der Scanner besteht aus mehreren Detektor-Ringen, die das Eintreffen der durch die Annihilation erzeugten koinzidenten Photonenpaare registrieren (siehe Abb. 5.1). Das Registrieren eines Photonenpaars wird als *Zerfallsereignis*, oder kurz als *Ereignis* bezeichnet, wobei hieran offensichtlich stets zwei gegenüberliegende Detektoren beteiligt sind. Diese bilden, imaginär durch eine Linie verbunden, eine sogenannte *Koinzidenzlinie* (engl. line of response, LOR). Die Koinzidenzlinie repräsentiert, bildlich gesprochen, den Weg des Photonenpaars nach der Annihilation. Die genaue Lokalisierung der Annihilation auf der Koinzidenzlinie erfolgt durch die Messung der zeitlichen Differenz zwischen dem Eintreffen der Photonen an den Detektoren. Auf diese Weise kann durch die zeitliche und räumliche Verteilung der Zerfallsereignisse auf die Verteilung des Radiopharmakons im Körperinneren geschlossen werden.

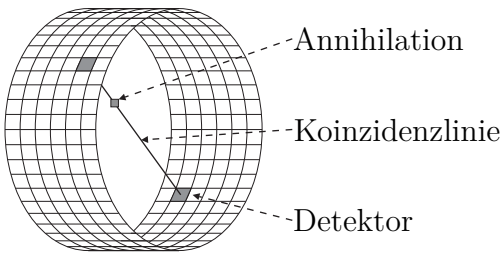


Abbildung 5.1: Detektion einer Annihilation in einem PET-Scanner. Quelle: [265].

[26] Auf Grund der Bewegung des Positrons vor der Annihilation beträgt der Winkel i.d.R. nicht exakt 180°, dieser sogenannte *Kollinearitätsfehler* wird aber von modernen PET-Scannern kompensiert.

5.2.2 Der LM OSEM-Algorithmus

Der LM OSEM-Algorithmus ist ein iteratives Verfahren zur schnellen und genauen Rekonstruktion von dreidimensionalen, PET-basierten Bildern [258] und beruht auf der bereits erwähnten *Expectation Maximization–Maximum Likelihood*-Methode [271]. Sei S der Rohdatensatz, der $x \in \mathbb{N}$ Ereignisse enthält. Der Algorithmus zerlegt S in $u \in \mathbb{N}$ disjunkte, gleichgroße Teilmengen, so dass jede Teilmenge $v = \frac{x}{u}$ Ereignisse enthält. In jeder Iteration $i \in \mathbb{N}, i \leqslant u$ wird aus sämtlichen Ereignissen der aktuellen Teilmenge $s_i \in S$ ein Teilbild rekonstruiert. Dieses wird mit dem bis dahin rekonstruierten Bild zusammengefügt, so dass nach jeder Iteration ein präziseres Bild entsteht:

$$h_{i+1} = q \cdot h_i \cdot d_i \quad \text{mit} \quad q = \frac{1}{P^T \cdot I} \quad \text{und} \quad d_i = \sum_{j=1}^{v} (P_j)^T \frac{1}{P_j \cdot h_i} \tag{5.1}$$

Hierbei repräsentieren $h, d \in \mathbb{R}^w$ das zu rekonstruierende Bild bzw. das Teilbild in Vektorform, $w \in \mathbb{N}$ entspricht der Anzahl Voxel von f bzw. d und damit der Auflösung des PET-Scanners. Zentraler Bestandteil der Rekonstruktion ist die sogenannte *Systemmatrix* $P \in \mathbb{R}^{v \times w}$, die auch in ihrer transponierten Form P^T verwendet wird. Jede Zeile $P_j, j \in \mathbb{N}, j \leqslant v$ speichert eine sogenannte *Projektion*, wobei letztere mit Hilfe des Siddon-Algorithmus [272] aus dem entsprechenden Ereignis e_j der aktuellen Teilmenge s_i berechnet wird. Ein Element einer Projektion $P_{j,k}, k \in \mathbb{N}, k \leqslant w$ gibt an, auf welcher Länge die zur Projektion bzw. zum Ereignis gehörende Koinzidenzlinie Voxel k des zu rekonstruierenden Bildes schneidet (siehe Abb. 5.2). Anzumerken bleibt, dass q als sogenannter *Normalisierungsvektor* bezeichnet wird und $I = h_0 = (1, 1, \ldots, 1)$.

5.2.3 Dekompositionsstrategien

Die zentrale Fragestellung bei der Beschleunigung der Bildrekonstruktion ist die nach der Dekomposition, d.h. der Zerlegung des Problems. Ausgehend von Formel 5.1 ergeben sich im Wesentlichen zwei Strategien [264]:

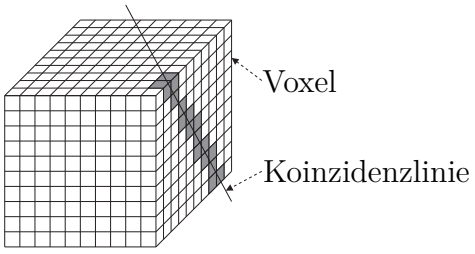


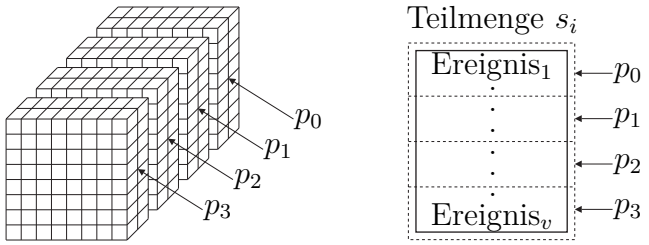
Abbildung 5.2: Die Koinzidenzlinie durchschneidet die Voxel des zu rekonstruierenden Bildes. Quelle: [265].

- Bei der sogenannten *Image Space Decomposition* (ISD, siehe Abb. 5.3a) findet eine Dekomposition des Bildraums statt, so dass jeder Prozess für die Rekonstruktion eines Teils des Bildes zuständig ist. Hierbei verarbeiten zwar alle Prozesse parallel dieselben Ereignisse, jeder Prozess berechnet aber für jedes Ereignis nur denjenigen Teil der Projektion, der durch den dem entsprechenden Prozess zugeordneten Bildraum verläuft. Die vollständige Rekonstruktion eines Teilbildes erfordert daher den Austausch sämtlicher lokal berechneter Normalisierungsvektoren in einem allgather-Kommunikationsschritt (siehe Abschn. 2.3.3.3)

- Bei der sogenannten *Projection Space Decomposition* (PSD, siehe Abb. 5.3b) findet eine Dekomposition des Ereignis- bzw. Projektionsraums statt, d.h. die Ereignisse der aktuellen Teilmenge s_i werden in np disjunkte Untermengen aufgeteilt, so dass jeder Prozess für die Verarbeitung genau einer Untermenge zuständig ist. Hierbei arbeiten sämtliche Prozesse stets auf demselben Bild, berechnen aber unterschiedliche Teilbilder. Die vollständige Rekonstruktion eines Teilbildes erfordert daher den Austausch sämtlicher lokal berechneter Teilbilder d_i^* in einem allgather-Kommunikationsschritt (siehe Abschn. 2.3.3.3).

Anzumerken bleibt, dass die Wahl der Dekompositionsstrategie einen beträchtlichen Einfluss auf das Verhältnis zwischen Rechen- und Kommunikationsaufwand hat. Während die beteiligten Prozesse bei der Dekomposition des Bildraums (ISD) mit dem Normalisierungsvektor q_i zwar nur geringe Datenmengen austauschen, dies aber bei jedem Ereignis notwendig ist, sind bei der Dekomposition des Projektionsraums (PSD) mit dem Teilbild d_i zwar größere Datenmengen auszutauschen, dies aber nur für jede Teilmenge

s_i. Wie eine effiziente Überlappung des Rechen- und Kommunikationsaufwands für den LM OSEM-Algorithmus erreicht werden kann, wird in [178] beschrieben, wobei die dort gewonnenen Erkenntnisse in die im Folgenden erörterte sequenzielle Implementierung eingeflossen sind.



(a) ISD (b) PSD

Abbildung 5.3: Verschiedene Dekompositionsstrategien, die bei der Parallelisierung des LM OSEM-Algorithmus verwendet werden.

5.3 Implementierungen

Die folgenden Abschnitte 5.3.1 bis 5.3.6 erörtern sowohl eine rein sequenzielle als auch fünf parallele Implementierungen des LM OSEM-Algorithmus. Zunächst wird in Abschnitt 5.3.1 eine von Philipp Kegel vorgenommene sequenzielle Variante präsentiert [101, 192]. Darauf aufbauend werden in den folgenden Abschnitten drei unter Verwendung von Muesli implementierte Varianten erläutert: Neben einer taskparallelen (siehe Abschn. 5.3.2) werden auch zwei datenparallele Implementierungen beschrieben, wobei die erste eine Dekomposition des Bildraums (siehe Abschn. 5.3.3), die zweite eine Dekomposition des Projektionsraums vornimmt (siehe Abschn. 5.3.4). Anschließend werden zu Vergleichszwecken zwei weitere Implementierungen von Philipp Kegel vorgestellt [101, 192]: Während die eine Variante eine Kombination aus MPI und OpenMP verwendet (siehe Abschn. 5.3.5), benutzt die andere MPI und TBB (siehe Abschn. 5.3.6).

5.3.1 Sequenziell

Die wesentlichen Teile der von Philipp Kegel vorgenommenen sequenziellen Implementierung des LM OSEM-Algorithmus können Listing 5.1 entnommen werden. In einer äußeren Schleife wird über die u Teilmengen iteriert, in die der Rohdatensatz aufgeteilt wurde (siehe Z. 4). Anschließend wird das Teilbild d initialisiert und die zur Teilmenge s_i gehörenden Ereignisse bestimmt (siehe Z. 5 f.). In einer weiteren Schleife wird über die v zur Teilmenge s_i gehörenden Ereignisse iteriert und für jedes Ereignis die entsprechende Projektion berechnet (siehe Z. 8 f.) Weiterhin wird für jedes Ereignis der Normalisierungsvektor q, der in dieser Variante der Implementierung ein singulärer Wert ist, zunächst initialisiert und anschließend berechnet (siehe Z. 10–13). Abschließend kann das Teilbild berechnet werden, indem die aktuelle Projektion mit dem Normalisierungswert gewichtet wird

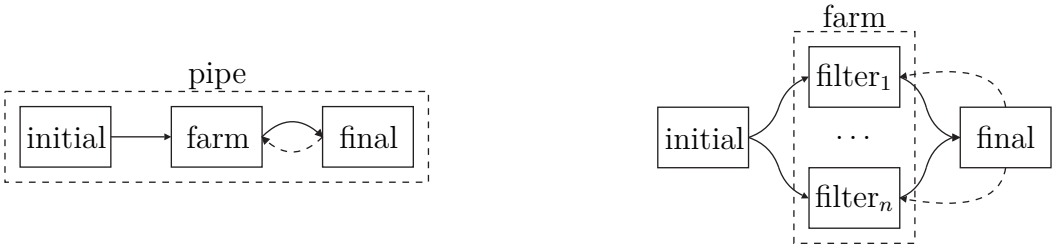
Listing 5.1: Vereinfachte, sequenzielle Implementierung des LM OSEM-Algorithmus.

```
1  double* d = new double[w];     double q = 0;
2  double* h = new double[w];     std::fill_n(&h[0], w, 1);
3
4  for(int i = 0; i < u; i++) {
5    std::fill_n(&d[0], w, 0);
6    Event* s_i = getEvents(i);
7
8    for(int j = 0; j < v; j++) {
9      Projection* P_j = getProjection(s_i[j]);
10     q = 0;
11
12     for(int k = 0; k < w; k++)
13       q += h[k] * P_j[k];
14
15     for(int k = 0; k < w; k++)
16       d[k] += q * P_j[k];
17   }
18
19   for(int k = 0; k < w; k++)
20     h[k] = h[k] * d[k];
21 }
```

(siehe Z. 15 f.) Nach der Abarbeitung einer Teilmenge s_i kann letztlich mit Hilfe des Teilbildes d_i ein neues, präziseres Bild h_{i+1} berechnet werden (siehe Z. 19 f.).

5.3.2 Taskparallel

Die Implementierung des LM OSEM-Algorithmus unter Verwendung taskparalleler Skelette basiert auf einer Dekomposition des Projektionsraums (siehe Abschn. 5.2.3). Die Bildrekonstruktion findet mit Hilfe der Skelette `Initial`, `Final`, `Filter`, `Pipe` und `Farm` statt [103, S. 42 ff. und 52 ff.], aus denen die in Abbildung 5.4 dargestellte Skelett-Topologie konstruiert wird. Äußerlich betrachtet besteht das Konstrukt zunächst aus einer dreistufigen `Pipe`, die die Skelette `Initial`, `Farm` und `Final` miteinander verbindet (siehe Abb. 5.4a). Bei genauerer Betrachtungsweise wird jedoch deutlich, dass das `Farm`-Skelett intern wiederum aus $n \in \mathbb{N}$ `Filter`-Skeletten besteht (siehe Abb. 5.4b). Innerhalb dieser Topologie übernehmen die Skelette folgende Aufgaben:



(a) Pipeline, die aus drei Stufen besteht (b) Farm, die aus $n \in \mathbb{N}$ Filtern besteht

Abbildung 5.4: (a) Abstrakte und (b) feingranulare Sicht auf die Skelett-Topologie der taskparallelen Implementierung des LM OSEM-Algorithmus.

- Das `Initial`-Skelett liest die Zerfallsereignisse des zu verarbeitenden Rohdatensatzes ein und versendet diese an das `Farm`-Skelett. Da es sich bei letzterem um eine sogenannte *dezentrale* Farm handelt (s.u.), werden die Ereignisse de facto direkt an die einzelnen `Filter`-Skelette geschickt. Die Ereignisse werden jedoch nicht einzeln, sondern in einem Paket, das mehrere Ereignisse enthält, versendet, um den Kommunikationsaufwand zu reduzieren. Anzumerken bleibt, dass das

`Initial`-Skelett durch das Versenden spezieller Pakete das Ende der aktuellen Teilmenge s_i bzw. des Rohdatensatzes S signalisiert.

- Wie bereits erwähnt, handelt es sich bei dem `Farm`-Skelett um eine sogenannte *dezentrale* Farm [251]. Diese zeichnet sich dadurch aus, dass es keinen expliziten Master-Prozess (Farmer) gibt, der für die Koordination der einzelnen Slave-Prozesse (Arbeiter), in diesem Fall der `Filter`-Prozesse, zuständig ist. Stattdessen übernimmt das `Farm`-Skelett die Koordination der Arbeiter, so dass der Farmer, insbesondere bei vielen zu koordinierenden Arbeitern, nicht zum Engpass bzw. Flaschenhals werden kann. Die einzige Aufgabe der Farm besteht folglich darin, eine entsprechende Anzahl an Arbeitern zu erzeugen und diese zu verwalten.

- Die `Filter`-Skelette empfangen die vom `Initial`-Skelett versendeten Pakete und berechnen für jedes darin enthaltene Ereignis die entsprechende Projektion P_j sowie den Normalisierungsvektor q, um daraus ein lokales Teilbild d_i^* zu erzeugen. Dafür ist es notwendig, dass jeder `Filter`-Prozess das zuletzt rekonstruierte Bild h_i vollständig im Speicher vorliegen hat. Erhält ein `Filter`-Prozess ein spezielles Paket, das das Ende der aktuellen Teilmenge s_i signalisiert, sendet der Prozess das lokal berechnete Teilbild an das `Final`-Skelett. Falls das Ende der Teilmenge gleichzeitig auch das Ende des Rohdatensatzes markiert, ist die Arbeit des `Filter`-Prozesses beendet. Anderenfalls wartet der Prozess auf den Erhalt des aktualisierten Bildes h_{i+1} und fährt mit der Verarbeitung der vom `Initial`-Prozess versendeten Pakete fort. Anzumerken bleibt, dass die Verwendung der `Filter`-Skelette hier zwingend erforderlich ist, da diese, im Gegensatz zu `Atomic`-Skeletten [103, S. 47], nicht für jede Eingabe eine Ausgabe produzieren müssen.

- Das `Final`-Skelett empfängt die von den `Filter`-Prozessen versendeten lokalen Teilbilder und berechnet daraus sowohl das globale Teilbild d_i als auch das aktualisierte Bild h_{i+1}. Wurden sämtliche Ereignisse des Rohdatensatzes verarbeitet, schreibt der `Final`-Prozess das rekonstruierte Bild in eine Datei. Anderenfalls sendet der Prozess das Bild per broadcast-Operation (siehe Abschn. 2.3.3.1) an alle `Filter`-Prozesse, die das aktualisierte Bild für die Verarbeitung der nächsten Teilmenge s_{i+1} benötigen.

Die Verwendung taskparalleler Skelette erlaubt eine vergleichsweise abstrakte Formulierung der Problemlösung (siehe Lst. 5.2). Die Rekonstruktion eines Bildes wird hierbei in drei logische Teilaufgaben zerlegt, die von den Funktionen `getEventPacket`, `processEventPacket` und `updateImage` implementiert werden:

Listing 5.2: Konstruktion der in Abbildung 5.4 dargestellten Skelett-Topologie.

```
1 Initial<EventPacket>        initial(&getEventPacket);
2 Filter<EventPacket, Image> filter(&processEventPacket,1);
3 Farm<EventPacket, Image>    farm(filter, np - 2);
4 Final<Image>                final(&updateImage);
5 Pipe                        pipe(initial, farm, final);
6
7 pipe.start();
```

`EventPacket* getEventPacket(Empty e)`. Die Funktion wird zur Parametrisierung des `Initial`-Skeletts verwendet (siehe Z. 1) und liest die in dem Rohdatensatz enthaltenen Ereignisse aus. Diese werden mit Hilfe der serialisierbaren Klasse `EventPacket` (siehe Anh. A.1) gebündelt und reihum unter Verwendung der Funktion `MSL_put`[27] [103, S. 47 f.] an die `Filter`-Skelette versendet. Nach der Abarbeitung sämtlicher Ereignisse des Rohdatensatzes gibt die Funktion `NULL` zurück, so dass die Prozesse ordnungsgemäß beendet werden können. Darüber hinaus versendet `getEventPacket` leere Pakete mit speziellen Flags, um den `Filter`-Skeletten das Ende einer Teilmenge bzw. des kompletten Rohdatensatzes zu signalisieren.

`void` `processEventPacket(Empty e)`. Die Funktion wird zur Parametrisierung der `Filter`-Skelette verwendet (siehe Z. 2) und empfängt mit Hilfe der Funktion `MSL_get` [103, S. 47 f.] die vom `Initial`-Skelett versendeten Pakete. Aus jedem darin enthaltenen Ereignis berechnet `processEventPacket` die entsprechende Projektion P_j sowie den Normalisierungswert q und aktualisiert das lokale Teilbild d_i^*. Signalisiert das Flag des empfangenen Pakets darüber hinaus das Ende der

[27] `MSL_put` implementiert eine Sendeoperation, deren Empfänger in Abhängigkeit von der konstruierten Skelett-Topologie bestimmt wird. Dieser kann die Nachricht durch einen Aufruf der Funktion `MSL_get` empfangen.

aktuellen Teilmenge bzw. des Rohdatensatzes, versendet die Funktion das bisher berechnete lokale Teilbild d_i^* inklusive des Flags mit Hilfe der serialisierbaren Klasse `Image` (siehe Anh. A.2) und der Funktion `MSL_put` an das `Final`-Skelett. Falls das Flag das Ende der aktuellen Teilmenge signalisiert hat, wartet die Funktion per `MPI_Bcast` (siehe Abschn. 2.3.3.1) auf den Empfang des neuen, aktualisierten Bildes h_{i+1}. Anschließend setzt `processEventPacket` den Empfang und die Abarbeitung weiterer Pakete fort. Falls das Flag das Ende des Rohdatensatzes signalisiert hat, ist die Rekonstruktion des Bildes abgeschlossen.

void `updateImage(`**const** `Image*` **const** `i)`. Die Funktion wird zur Parametrisierung des `Final`-Skeletts verwendet (siehe Z. 4) und empfängt mit Hilfe der Funktion `MSL_get` die von den `Filter`-Skeletten berechneten lokalen Teilbilder d_i^*. Aus diesen erstellt `updateImage` zunächst das Teilbild d_i, um anschließend das neue, aktualisierte Bild h_{i+1} zu generieren. Falls das in den lokalen Teilbildern enthaltene Flag das Ende des Rohdatensatzes signalisiert, schreibt die Funktion das rekonstruierte Bild in eine Datei. Anderenfalls verschickt `updateImage` das neue Bild per `MPI_Bcast` (siehe Abschn. 2.3.3.1) an sämtliche `Filter`-Skelette und wartet auf den Empfang neuer lokaler Teilbilder. Anzumerken bleibt, dass das Versenden der aktualisierten Bilder manuell per `MPI_Bcast` erfolgen muss, da der Datenstrom innerhalb des `Pipe`-Skeletts lediglich unidirektional konzipiert ist. Auf der einen Seite wird dadurch das Prinzip der Kapselung der Kommunikation durch die Skelettbibliothek verletzt, auf der anderen Seite gezeigt, dass Muesli auch Ad-hoc-Parallelität unterstützt.

Bei der Konstruktion der in Abbildung 5.4 dargestellten Skelett-Topologie ist zu beachten, dass lediglich das `Initial`-, das `Final`- und die `Filter`-Skelette von jeweils genau einem Prozess ausgeführt werden. Die gruppierenden Skelette `Pipe` und `Farm` hingegen benötigen keine weiteren Prozesse. Infolgedessen können innerhalb der Farm von den insgesamt np zur Verfügung stehenden Prozessen lediglich $np - 2$ Prozesse ein `Filter`-Skelett ausführen (siehe Z. 3). Im Anschluss an das `Initial`-, `Farm`- und `Final`-Skelett kann das `Pipe`-Skelett erzeugt und die Rekonstruktion des Bildes durch einen Aufruf der Funktion `start` begonnen werden (siehe Z. 5 ff.).

5.3.3 Datenparallel (ISD)

Die erste Variante der Implementierung des LM OSEM-Algorithmus unter Verwendung datenparalleler Skelette bzw. verteilter Datenstrukturen basiert auf einer Dekomposition des Bildraums (siehe Abschn. 5.2.3). Die Bildrekonstruktion findet mit Hilfe der Klasse `DistributedArray` (siehe Abschn. 3.3.1) sowie der Skelette *foldIndex* (siehe Abschn. 3.4.2), *mapIndexInPlace* (siehe Abschn. 3.4.3) und *zipInPlace* (siehe Abschn. 3.4.5) statt. Während die verteilte Datenstruktur zur Speicherung des zu rekonstruierenden Bildes h sowie des Teilbildes d verwendet wird, manipulieren die Skelette diese mit Hilfe entsprechender Argumentfunktionen. Dadurch wird die Berechnung des Bildes in drei Funktionen aufgeteilt:

double `computeQ(Projection* P_j, double q, double val, int k)`. Die Funktion wird als Argumentfunktion für das *fold*-Skelett verwendet und dient zur Berechnung des Normalisierungswerts q in Abhängigkeit von der aktuellen Projektion P_j. Die Parameter q und val repräsentieren den bisher akkumulierten Normalisierungswert sowie den Wert von Voxel k des zu rekonstruierenden Bildes h.

double `updateD(Projection* P_j, double q, double val, int k)`. Die Funktion wird als Argumentfunktion für das *map*-Skelett verwendet und dient zur Berechnung des Teilbildes d in Abhängigkeit von der aktuellen Projektion P_j. Die Parameter q und val repräsentieren den für die Projektion P_j berechneten Normalisierungswert sowie den bisherigen Wert von Voxel k des Teilbildes d.

double `updateH(double valH, double valD)`. Die Funktion wird als Argumentfunktion für das *zip*-Skelett verwendet und dient zur Berechnung des aktualisierten Bildes h_{i+1}. Die Parameter $valH$ und $valD$ repräsentieren den aktuellen Wert des Bildes h bzw. des Teilbildes d.

Durch die Verwendung datenparalleler Skelette bzw. verteilter Datenstrukturen in Kombination mit einer Dekomposition des Bildraums kann der wesentliche Teil der in Listing 5.1 skizzierten sequenziellen Implementierung auf wenige Zeilen verkürzt werden (siehe Lst. 5.3). In einer äußeren Schleife wird nach wie vor über die u Teilmengen iteriert, in die der Rohdatensatz aufgeteilt wurde, um anschließend die zur Teilmenge s_i gehörenden

Ereignisse zu bestimmen (siehe Z. 3 f.). Auch die innere Schleife, in der über sämtliche Ereignisse der aktuellen Teilmenge s_i iteriert und zu jedem Ereignis die entsprechende Projektion P_j berechnet wird, bleibt erhalten (siehe Z. 7). Im Vergleich zur sequenziellen Implementierung werden jedoch das zu rekonstruierende Bild h sowie das Teilbild d nicht in einem Feld vom Typ `double*`, sondern in einem w-elementigen, verteilten Feld vom Typ `DistributedArray<double>` gespeichert (siehe Z. 1 und 5). Somit kann die Berechnung des Normalisierungswerts q durch das Falten der Elemente von h erfolgen, wobei aus der Argumentfunktion `computeQ` eine partielle Applikation (siehe Abschn. 3.2.4) erzeugt wird, deren erster Parameter fest an die aktuelle Projektion P_j gebunden wird (siehe Z. 9). Die Rekonstruktion des Teilbildes geschieht durch die Anwendung der Argumentfunktion `updateD` auf sämtliche Elemente von d, wobei hier ebenfalls eine partielle Applikation erzeugt wird, deren erster bzw. zweiter Parameter fest an die aktuelle Projektion P_j bzw. den berechneten Normalisierungswert q gebunden wird (siehe Z. 10). Nach der Abarbeitung einer Teilmenge s_i kann das aktualisierte, präzisere Bild h_{i+1} berechnet werden, indem entsprechende Elemente des alten Bildes h und des Teilbildes d unter Verwendung der Funktion `updateH` miteinander kombiniert werden (siehe Z. 13).

Listing 5.3: Vereinfachte Implementierung des LM OSEM-Algorithmus unter Verwendung datenparalleler Skelette und einer auf der Zerlegung des Bildraums basierenden Dekompositionsstrategie.

```
1  DistributedArray<double> h(w, 1);    double q = 0;
2
3  for(i = 0; i < u; i++) {
4    Event* s_i = getEvents(i);
5    DistributedArray<double> d(w, 0);
6
7    for(j = 0; j < v; j++) {
8      Projection* P_j = getProjection(s_i[j]);
9      q = h.foldIndex(curry(&computeQ)(P_j));
10     d.mapIndexInPlace(curry(&updateD)(P_j)(q));
11   }
12
13   h.zipInPlace(d, &updateH);
14 }
```

5.3.4 Datenparallel (PSD)

Die zweite Variante der Implementierung des LM OSEM-Algorithmus unter Verwendung datenparalleler Skelette bzw. verteilter Datenstrukturen basiert auf einer Dekomposition des Projektionsraums (siehe Abschn. 5.2.3). Die Bildrekonstruktion findet mit Hilfe der Klasse `DistributedArray` (siehe Abschn. 3.3.1) sowie der Skelette *map* bzw. *mapInPlace* (siehe Abschn. 3.4.3), *foldPartitionsInPlace* (siehe Abschn. 3.4.2) und *zipInPlace* (siehe Abschn. 3.4.5) statt. Während die verteilte Datenstruktur zur Speicherung des zu rekonstruierenden Bildes h sowie des Teilbildes d verwendet wird, manipulieren die Skelette diese mit Hilfe entsprechender Argumentfunktionen. Dadurch wird die Berechnung des Bildes in vier Funktionen aufgeteilt:

`Projection computeP_j(Event& e)`. Die Funktion wird als Argumentfunktion für das *map*-Skelett verwendet (siehe Z. 6 f.) und dient zur Berechnung einer Projektion P_j in Abhängigkeit von dem gegebenen Zerfallsereignis e.

`Projection updateD(DistributedArray<`**`double`**`>& d, Projection& p)`. Die Funktion wird als Argumentfunktion für das zweite *map*-Skelett verwendet (siehe Z. 8) und berechnet mit Hilfe der gegebenen Projektion P_j den entsprechenden Normalisierungswert q, um anschließend das lokale Teilbild d zu aktualisieren.

`double``* combineD(`**`double`**` valA, `**`double`**` valB)`. Die Funktion wird als Argumentfunktion für das *fold*-Skelett verwendet (siehe Z. 9) und kombiniert die lokalen Teilbilder d_i^*. Die Parameter $valA$ und $valB$ repräsentieren den bisher kumulierten Wert des Teilbildes bzw. den eines noch zu kumulierenden lokalen Updates.

`double`` updateH(`**`double`**` valH, `**`double`**` valD)`. Die Funktion wird als Argumentfunktion für das *zip*-Skelett verwendet (siehe Z. 10) und dient zur Berechnung des aktualisierten Bildes h_{i+1}. Die Parameter $valH$ und $valD$ repräsentieren den aktuellen Wert des Bildes h bzw. des Teilbildes d.

Durch die Verwendung datenparalleler Skelette bzw. verteilter Datenstrukturen in Kombination mit einer Dekomposition des Projektionsraums kann

Listing 5.4: Vereinfachte Implementierung des LM OSEM-Algorithmus unter Verwendung datenparalleler Skelette und einer auf der Zerlegung des Projektionsraums basierenden Dekompositionsstrategie.

```
1  DistributedArray<double> h(w * np, 1);
2
3  for(i = 0; i < u; i++) {
4    DistributedArray<Event> s_i = getEvents(i);
5    DistributedArray<double> d(w * np, 0);
6    DistributedArray<Projection> P_j =
7      s_i.map(&computeP_j);
8    P_j.mapInPlace(curry(&updateD)(d));
9    d.foldPartitionsInPlace(&combineD);
10   h.zipInPlace(d, &updateH);
11 }
```

der wesentliche Teil der in Listing 5.1 skizzierten sequenziellen Implementierung auf wenige Zeilen verkürzt werden (siehe Lst. 5.4). Infolge der gewählten Dekompositionsstrategie rekonstruiert jeder der np Prozesse, basierend auf dem aktuellen Bild h_i, ein lokales Teilbild d_i^*, d.h. jeder Prozess muss sowohl über ein vollständiges Bild h_i als auch ein vollständiges Teilbild d_i verfügen. Zur Speicherung des Bildes bzw. des Teilbildes wird, analog zur datenparallelen Implementierung unter Verwendung einer auf der Zerlegung des Bildraums basierenden Dekompositionsstrategie, die Klasse `DistributedArray` (siehe Abschn. 3.3.1) verwendet. Die Größe des verteilten Feldes wird hierbei jedoch nicht auf w, sondern auf $w \cdot np$ gesetzt, so dass jeder Prozess ein vollständiges Bild bzw. Teilbild speichern kann (siehe Z. 1 und 5). In einer äußeren Schleife wird nach wie vor über die u Teilmengen iteriert, in die der Rohdatensatz aufgeteilt wurde, um anschließend die zur Teilmenge s_i gehörenden Ereignisse zu bestimmen (siehe Z. 3 f.). Letztere werden, entsprechend der Dekompositionsstrategie, in dem v-elementigen, verteilten Feld s_i gespeichert und damit unter den np Prozessen verteilt. Auf diese Weise kann die Berechnung der Projektionen durch das Anwenden der Funktion `computeP_j` auf sämtliche Elemente von s_i, die Berechnung der lokalen Teilbilder d_i^* durch das Anwenden der Funktion `updateD` auf sämtliche Elemente von P_j erfolgen (siehe Z. 6 ff.). Die Rekonstruktion des Teilbildes d_i geschieht durch das Falten der np lokalen Partitionen des verteilten Feldes d, wobei hier die bereits erwähnte Argumentfunktion

`combineD` verwendet wird (siehe Z. 9). Abschließend kann das aktualisierte, präzisere Bild h_{i+1} berechnet werden, indem entsprechende Elemente des alten Bildes h und des Teilbildes d unter Verwendung der Funktion `updateH` miteinander kombiniert werden (siehe Z. 10).

5.3.5 MPI und OpenMP

Die Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und OpenMP basiert auf einer Dekomposition des Projektionsraums (siehe Abschn. 5.2.3) und wurde von Philipp Kegel und Maraike Schellmann entwickelt [101, 192, 264]. Die wesentlichen Teile der Implementierung entsprechen denen der sequenziellen Implementierung (siehe Abschn. 5.3.1), verwenden zusätzlich aber zwei OpenMP-Direktiven sowie eine MPI-Kommunikationsfunktion (siehe List. 5.5). In einer äußeren Schleife wird nach wie vor über die u Teilmengen iteriert, in die der Rohdatensatz aufgeteilt wurde, um anschließend die zur Teilmenge s_i gehörenden Ereignisse zu bestimmen (siehe Z. 1). Letzteres geschieht mit Hilfe der Funktion `getOffset`, die in Abhängigkeit von der Prozess-ID `pid` einen Offset berechnet, so dass jeder Prozess, entsprechend der Dekompositionsstrategie, nur eine Untermenge der Teilmenge s_i bearbeiten muss. Auch die innere Schleife, in der über sämtliche Ereignisse der aktuellen Teilmenge s_i iteriert und zu jedem Ereignis die entsprechende Projektion P_j berechnet wird, bleibt erhalten (siehe Z. 5 f.). Hierbei ist wichtig, dass die Abarbeitung dieser inneren Schleife im Vergleich zur sequenziellen Implementierung durch die Verwendung der **`parallel for`**-Direktive beschleunigt wird (siehe Z. 4 und Abschn. 2.4.2.2). Auf diese Weise erfolgt nicht nur eine Zerlegung der Teilmenge s_i in mehrere disjunkte Untermengen, letztere werden darüber hinaus auf Parallelrechnern mit gemeinsamem Speicher (siehe Abschn. 2.2.2.1) durch mehrere OpenMP-Threads parallel bearbeitet. Zu diesem Zweck ist es notwendig, den Normalisierungswert q innerhalb einer **`private`**-Klausel zu deklarieren, damit jeder Thread eine eigene Kopie der Variablen erhält (siehe Abschn. 2.4.3.1). Die Berechnung der Projektion P_j (siehe Z. 6) sowie des Normalisierungswerts q (siehe Z. 9 f.) geschieht analog zur sequenziellen Variante, lediglich die Aktualisierung des Teilbildes d erfordert ein gesondertes Vorgehen. Da die Berechnung eines lokalen Teilbildes d_i^* auf Grund der **`parallel for`**-Direktive von mehreren Threads gleichzeitig durchgeführt

Listing 5.5: Vereinfachte Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und OpenMP sowie einer auf der Zerlegung des Projektionsraums basierenden Dekompositionsstrategie.

```
1  for(int i = 0; i < u; i++) {
2    Event* s_i = getEvents(i, getOffset(pid));
3
4    #pragma omp parallel for private(q)
5    for(int j = 0; j < v; j++) {
6      Projection P_j = getProjection(s_i[j]);
7      q = 0;
8
9      for(int k = 0; k < w; k++)
10       q += h[k] * P_j[k];
11
12     #pragma omp critical
13     for(int k = 0; k < w; k++)
14       d[k] += q * P_j[k];
15   }
16
17   MPI_Allreduce(d, d, w, MPI_DOUBLE, MPI_SUM,
18                 MPI_COMM_WORLD);
19
20   #pragma omp parallel for
21   for(j = 0; j < w; j++)
22     h[k] = h[k] * d[k];
23 }
```

wird, muss der Zugriff auf das Teilbild mit Hilfe der **`critical`**-Direktive (siehe Abschn. 2.4.2.3) synchronisiert werden, um Wettlaufsituationen zu vermeiden (siehe Z. 12 ff.). Nach der Abarbeitung der Teilmenge s_i werden die lokalen Teilbilder d_i^* mit Hilfe der Funktion `MPI_Allreduce` (siehe Abschn. 2.3.3.5) zum globalen Teilbild d_i zusammengesetzt (siehe Z. 17 f.). Anschließend berechnet jeder Prozess mit Hilfe des Teilbildes d_i das neue, präzisere Bild h_{i+1}, wobei auch diese Schleife durch die Verwendung der **`parallel for`**-Direktive beschleunigt wird (siehe Z. 20). Zwar erfolgt diese Berechnung für jeden Prozess redundant, erfordert dafür aber keinen zusätzlichen Kommunikationsaufwand.

5.3.6 MPI und TBB

Die Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und TBB basiert auf einer Dekomposition des Projektionsraums (siehe Abschn. 5.2.3) und wurde ebenfalls von Philipp Kegel vorgenommen [101]. TBB ist eine von Intel entwickelte C++ Template-Bibliothek, die die Programmierung von Mehrkernprozessoren ermöglicht [259]. Die Bibliothek übernimmt, ähnlich wie OpenMP (siehe Abschn. 2.4), die Erzeugung und Verwaltung von Threads und verwendet als zentralen Abstraktionsmechanismus sogenannte *Aufgaben* (engl. tasks). Diese werden mit Hilfe eines Scheduling-Mechanismus dynamisch den verfügbaren Rechenkernen zugeordnet, wobei das Laufzeitsystem eine automatisierte Lastverteilung vornimmt. Anzumerken bleibt, dass TBB, im Gegensatz zu OpenMP, Threads einmalig zu Beginn des Programms erzeugt und anschließend in einem Threadpool verwaltet.

Listing 5.6: Vereinfachte Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und TBB sowie einer auf der Zerlegung des Projektionsraums basierenden Dekompositionsstrategie.

```
1  for(int i = 0; i < u; i++) {
2    Event* s_i = getEvents(i, getOffset(pid));
3
4    parallel_for(blocked_range<size_t>(0, v, GRAIN_SIZE),
5      ProcessEvents(h, d, w, s_i));
6
7    MPI_Allreduce(d, d, w, MPI_DOUBLE, MPI_SUM,
8                  MPI_COMM_WORLD);
9
10   parallel_for(blocked_range<size_t>(0, w, GRAIN_SIZE),
11     UpdateImage(h, d));
12 }
```

Durch die Verwendung von TBB kann der wesentliche Teil der in Listing 5.1 skizzierten sequenziellen Implementierung auf wenige Zeilen verkürzt werden (siehe Lst. 5.6). Der Quelltext entspricht dem der Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und OpenMP (siehe Lst. 5.5), wobei jede OpenMP **`parallel for`**-Direktive inklusive der assoziierten Schleife durch die von TBB zur Verfügung gestellte

Template-Funktion `parallel_for` ersetzt wurde (siehe Z. 4 und 10). Die Template-Funktion verteilt, ähnlich der OpenMP **`for`**-Direktive (siehe Abschn. 2.4.2.2), die Iterationen einer **`for`**-Schleife auf mehrere Threads, erfordert jedoch die vollständige Auslagerung der zu parallelisierenden Schleife in eine separate Klasse. Auf diese Weise findet die Rekonstruktion eines Teilbildes bzw. eines Bildes mit Hilfe der Klassen `ProcessEvents` bzw. `UpdateImage` statt (siehe Z. 5 und 11). Während `ProcessEvents` die Schleife kapselt, die über die zu verarbeitenden Ereignisse iteriert (siehe Lst. 5.5, Z.4–15), kapselt `UpdateImage` die Schleife, die die Berechnung des neuen, präziseren Bildes durchführt (siehe Lst. 5.5, Z. 20 ff.). Anzumerken bleibt, dass die OpenMP **`critical`**-Direktive (siehe Lst. 5.5, Z. 12 und Abschn. 2.4.2.3) innerhalb der Klasse `ProcessEvents` durch einen von TBB zur Verfügung gestellten Mutex ersetzt wurde.

5.4 Ergebnisse

Im Folgenden werden die Ergebnisse einiger Laufzeitmessungen der in Abschnitt 5.3 beschriebenen Implementierungen erläutert (siehe Tab. 5.1 und Abb. 5.5). Sämtliche Messungen wurden auf dem ZIVHPC (siehe Abschn. 2.5) vorgenommen, wobei jeder MPI-Prozess auf einem separaten Rechenknoten ausgeführt wurde und, mit Ausnahme der taskparallelen und der rein auf MPI basierenden Implementierung, alle acht Rechenkerne des Knotens benutzt. Der verwendete Rohdatensatz enthält ca. $6 \cdot 10^7$ Ereignisse, von denen jedoch, um die Laufzeit zu reduzieren, nur etwa 10^6 verarbeitet werden, und zeigt rekonstruiert das Bild einer Maus (siehe Abb. 5.6). Anzumerken bleibt, dass der zur Erzeugung des Rohdatensatzes verwendete PET-Scanner im Jahr 2002 an der Klinik und Poliklinik für Nuklearmedizin des Universitätsklinikums Münster installiert wurde und eine Auflösung von $150 \times 150 \times 280$ Voxel erreicht [269].

- Die auf einer Dekomposition des Bildraums basierende, datenparallele Implementierung des LM OSEM-Algorithmus (siehe Abschn. 5.3.3) weist erwartungsgemäß das schlechteste Laufzeitverhalten auf. Die lange Laufzeit wird hauptsächlich durch das *foldIndex*-Skelett verursacht, da dieses für *jedes* Ereignis einmal aufgerufen wird. Zwar kann

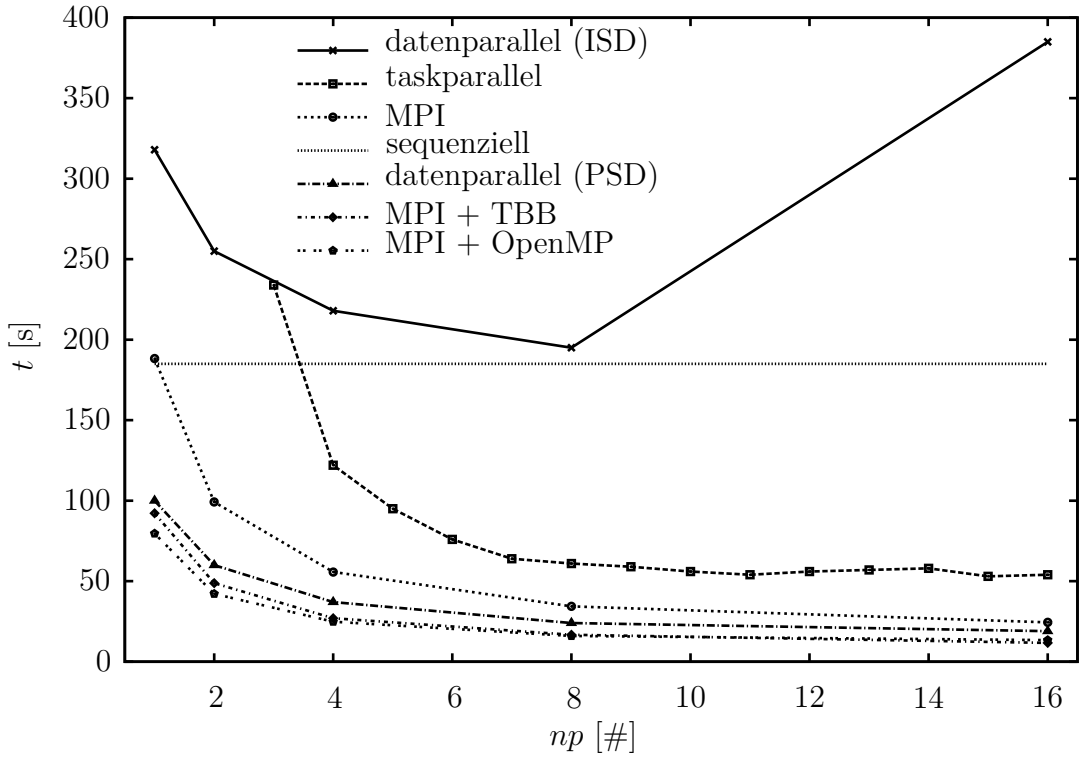


Abbildung 5.5: Laufzeiten der verschiedenen LM OSEM-Implementierungen.

der hohe Kommunikationsaufwand bis $np = 8$ durch die Verteilung der Bildrekonstruktion auf mehrere Knoten ausgeglichen werden, bei $np = 16$ ist dies jedoch offensichtlich nicht mehr der Fall. Der geringe Speedup ist jedoch nicht auf die Skelettbibliothek, sondern auf die gewählte Dekompositionsstrategie zurückzuführen, da eine Zerlegung des Bildraums zur Rekonstruktion des Normalisierungswerts zwangsläufig Kommunikation erfordert (siehe Abschn. 5.2.3). Insofern sind der Parallelisierung einer auf der Dekomposition des Bildraums basierenden Implementierung Grenzen gesetzt.

- Bei der zur taskparallelen Implementierung des LM OSEM-Algorithmus (siehe Abschn. 5.3.2) korrespondierenden Kurve fällt zunächst auf, dass diese, im Gegensatz zu allen anderen Kurzen, erst bei $np = 3$ beginnt. Dieser Umstand hängt mit der Tatsache zusammen, dass das `Initial-`, das `Final-` sowie die `Filter`-Skelette von jeweils genau einem Prozess ausgeführt werden, so dass die erzeugte Skelett-Topologie erst ab $np = 3$ funktionsfähig ist. Die taskparallele Variante weist deut-

Tabelle 5.1: Laufzeiten der verschiedenen LM OSEM-Implementierungen in Sekunden. Der Speedup ist jeweils in Klammern angegeben. Bei der taskparallelen Implementierung gibt np die Anzahl der `Filter`-Prozesse an.

Implementierung \ np	1	2	4	8	16
datenparallel (ISD)	318 (1,0)	255 (1,2)	216 (1,4)	195 (1,6)	385 (0,8)
taskparallel	234 (1,0)	122 (1,9)	76 (3,0)	56 (4,1)	54 (4,3)
MPI	188 (1,0)	99 (1,9)	56 (3,3)	34 (5,5)	24 (7,8)
sequenziell	185 (1,0)	185 (1,0)	185 (1,0)	185 (1,0)	185 (1,0)
datenparallel (PSD)	100 (1,0)	60 (1,6)	37 (2,6)	24 (4,0)	19 (5,0)
MPI + TBB	92 (1,0)	49 (1,8)	27 (3,4)	17 (5,4)	12 (7,6)
MPI + OpenMP	80 (1,0)	42 (1,9)	25 (3,2)	16 (5,0)	13 (6,1)

lich kürzere Laufzeiten sowie ein besseres Skalierungsverhalten auf als die datenparallele, auf einer Dekomposition des Bildraums basierende Implementierung. Die Laufzeiten liegen jedoch deutlich über denen der rein MPI-basierten Version, der entsprechende Mehraufwand muss offensichtlich als Preis für das höhere Abstraktionsniveau in Kauf genommen werden. Darüber hinaus fällt auf, dass die Implementierung ab $np = 8$ praktisch nicht mehr skaliert. Der Grund hierfür ist im iterativen Charakter des LM OSEM-Algorithmus zu suchen. Letzterer erfordert das manuelle Einfügen einer `MPI_Bcast`-Operation, mit deren Hilfe das `Final`-Skelett das neue, präzisere Bild h_{i+1} an sämtliche `Filter`-Prozesse sendet (siehe Abb. 5.4). Durch dieses Vorgehen treten jedoch zwangsweise Effizienzverluste auf, da jeder `Filter`-Prozess nach der Abarbeitung der empfangenen Ereignisse auf den Erhalt von h_{i+1} warten muss. Anzumerken bleibt, dass die Größe der vom `Initial`-Skelett versendeten Ereignispakete ab einer gewissen Paketgröße keinen Einfluss auf die Laufzeit hat, da die Abarbeitung des Pakets mehr Zeit als das Senden bzw. Empfangen desselbigen in Anspruch nimmt. Die für die Tests verwendete Paketgröße wurde auf 100 Ereignisse gesetzt.

- Die allein auf MPI basierende Implementierung des LM OSEM-Algorithmus entspricht der Variante unter direkter Verwendung von MPI und OpenMP (siehe Abschn. 5.3.5), beim Kompilieren wurde jedoch

die entsprechende OpenMP-Option deaktiviert (siehe Abschn. 2.4). Die korrespondierende Kurve kann somit als Referenz für den Vergleich mit Implementierungen herangezogen werden, die mit OpenMP oder TBB weitere Technologien zur Parallelisierung verwenden, um so den Grad der zusätzlichen Beschleunigung durch den Einsatz mehrerer Threads zu quantifizieren. Erwartungsgemäß ist die Laufzeit im Vergleich zu den zusätzlich auf OpenMP oder TBB basierenden Implementierungen länger, die rein MPI-basierte Variante weist im Gegenzug jedoch das beste Skalierungsverhalten auf (siehe Tab. 5.1).

- Die auf einer Dekomposition des Projektionsraums basierende, datenparallele Implementierung des LM OSEM-Algorithmus (siehe Abschn. 5.3.4) weist ein ähnliches Skalierungsverhalten auf wie die allein auf MPI basierende Variante. Die Laufzeit ist zwar durch die Verwendung von acht Rechenkernen pro Rechenknoten erwartungsgemäß deutlich kürzer, erreicht jedoch nicht ganz das Niveau der direkt auf MPI und TBB bzw. auf MPI und OpenMP basierenden Varianten. Der zusätzliche Mehraufwand wird durch die Verwendung der Skelettbibliothek verursacht und muss, analog zur taskparallelen Variante, als Preis für das höhere Abstraktionsniveau in Kauf genommen werden.

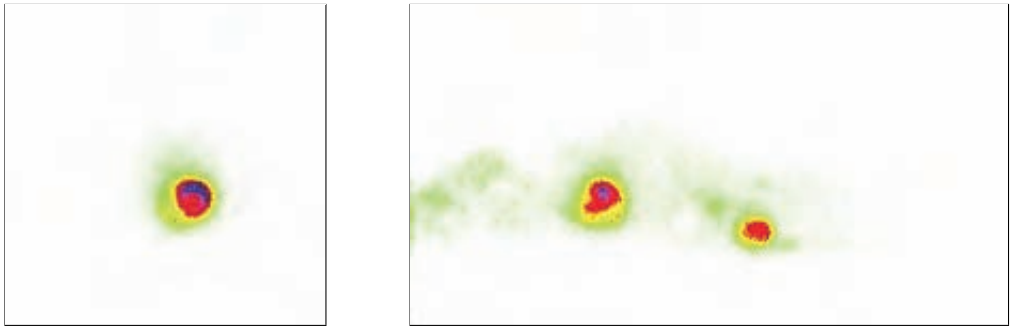
- Die Laufzeit der Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und TBB (siehe Abschn. 5.3.6) ist, verglichen mit der allein auf MPI basierenden Variante, wie erwartet deutlich kürzer. Obwohl bei TBB, im Vergleich zu OpenMP, der Mehraufwand für das wiederholte Erzeugen von Threads wegfällt, erreicht die Laufzeit bis $np = 8$ jedoch nicht ganz das Niveau der direkt auf MPI und OpenMP basierenden Implementierung (siehe Tab. 5.1). Der Mehraufwand wird durch den von TBB verwendeten dynamischen Scheduler erzeugt, da letzterer die zu bearbeitenden Aufgaben zur Laufzeit den verfügbaren Rechenkernen zuteilt und dabei zusätzlich eine automatisierte Lastverteilung vornimmt. Die Kosten für diese zusätzliche Funktionalität amortisieren sich offensichtlich erst ab $np = 8$, so dass die Laufzeiten nahezu identisch sind.

- Die Implementierung des LM OSEM-Algorithmus unter direkter Verwendung von MPI und OpenMP (siehe Abschn. 5.3.5) weist von allen Implementierungen die kürzeste Laufzeit auf. Verglichen mit der

TBB-basierten Variante ergeben sich jedoch nur bis $np = 4$ signifikante Unterschiede, da OpenMP die Iterationen der entsprechenden **`parallel for`**-Direktive nicht dynamisch, sondern statisch verteilt.[28] Ab $np = 8$ ergeben sich praktisch keine Unterschiede in der Laufzeit (siehe Tab. 5.1).

5.5 Fazit

In diesem Kapitel wurden mit dem LM OSEM-Algorithmus ein populäres Verfahren zur dreidimensionalen Rekonstruktion von PET-basierten Bildern sowie mehrere parallele Implementierungen desselbigen vorgestellt. Die für die Rekonstruktion benötigten Rohdaten werden hierbei mit Hilfe eines PET-Scanners erfasst, der die Zerfallsereignisse eines Radiopharmakons im Körper des Versuchsobjekts registriert (siehe Abschn. 5.2.1). Anschließend kann die eigentliche Bildrekonstruktion unter Verwendung des LM OSEM-Algorithmus erfolgen, die jedoch auf Grund der hohen Anzahl registrierter



(a) Transversalschnitt (b) Sagittalschnitt

Abbildung 5.6: Ein mit dem LM OSEM-Algorithmus rekonstruiertes Bild einer Maus. Zu erkennen ist die Verteilung des Radiopharmakons, wobei grüne Regionen eine geringe, gelbe eine mittlere und rote eine hohe Konzentration im Gewebe anzeigen.

[28] Die entsprechende **`parallel for`**-Direktive (siehe Lst. 5.5, Z. 4) verwendet die **`schedule`**-Klausel (siehe Abschn. 2.4.3.3) zwar nicht explizit. Falls letztere jedoch nicht spezifiziert wird, bestimmt die OpenMP-Kontrollvariable `def-sched-var`, wie der Iterationsraum aufgeteilt wird. Der verwendete Intel C++-Compiler 10.1 initialisiert die Variable mit **`static`** [185].

Ereignisse i.d.R. mehrere Stunden in Anspruch nimmt (siehe Abschn. 5.2.2). Um die Laufzeit zu reduzieren, bietet sich daher eine parallele Implementierung des Algorithmus an, wobei hier als Dekompositionsstrategie entweder eine Zerlegung des Bildraums oder eine Zerlegung des Projektionsraums in Frage kommt (siehe Abschn. 5.2.3). Der Schwerpunkt des Kapitels lag, neben der Vorstellung einer sequenziellen Variante, in der Beschreibung mehrerer paralleler Implementierungen des LM OSEM-Algorithmus, die einerseits mit Hilfe daten- bzw. taskparalleler Skelette, andererseits unter direkter Verwendung von MPI, OpenMP und TBB umgesetzt wurden (siehe Abschn. 5.3). Die durchgeführten Laufzeitmessungen haben gezeigt, dass eine effiziente Parallelisierung des LM OSEM-Algorithmus möglich ist, auch wenn die Reduzierung der Laufzeit durch die Verwendung zusätzlicher Rechenkerne nicht so hoch ausfällt wie durch die Verwendung zusätzlicher Prozesse (siehe Abschn. 5.4). Darüber hinaus konnte gezeigt werden, dass die Erstellung eines parallelen Programms durch die Verwendung einer Skelettbibliothek, verglichen mit dem direkten Gebrauch von MPI, OpenMP und TBB, deutlich einfacher ist. Zwar erreichen die Laufzeiten der daten- bzw. taskparallelen Varianten nicht ganz deren Niveau, der höhere Abstraktionsgrad vereinfacht die Erstellung der parallelen LM OSEM-Implementierung jedoch massiv, da die Verwendung von MPI und OpenMP durch die Bibliothek gekapselt wird und für den Benutzer somit transparent ist.

Kapitel 6

Paralleles Training für neuronale ART 2-Netze

Inhalt

6.1 Einleitung

Die Erforschung künstlicher neuronaler Netze [212, 302] hat ihren Ursprung in der Biologie und ist durch den Umstand motiviert, dass natürliche neuronale Netze selbst komplexe Aufgaben im Bruchteil einer Sekunde lösen, wohingegen moderne Rechner für dasselbe Problem um Größenordnungen mehr Zeit benötigen. So ist z.B. das menschliche Gehirn in der Lage, bekannte Personen oder Gegenstände in ca. 0,1 s zu erkennen, was bei einer angenommenen Reaktionszeit eines natürlichen Neurons von 1 ms somit lediglich 100 sequenzielle Verarbeitungsschritte erfordert [302, S. 26].

Eine vergleichbare Leistung können selbst aktuelle Rechner im Allgemeinen nicht erbringen. Das zentrale Merkmal natürlicher neuronaler Netze ist deren Lernfähigkeit gepaart mit einer enormen Robustheit gegenüber Störungen, so dass z.B. Personen auch nach mehreren Jahren wiedererkannt werden, obwohl sich deren Äußeres verändert hat. Auch die Klassifikation von Gegenständen oder die Erkennung einer Handschrift sind Aufgaben, für die ein menschliches Gehirn nach wie vor besser geeignet ist als ein Rechner. Künstliche neuronale Netze stellen somit den Versuch dar, die Art der Informationsverarbeitung bzw. die Eigenschaften und Fähigkeiten natürlicher neuronaler Netze zu imitieren.

In Analogie zur Beschaffenheit natürlicher neuronaler Netze bestehen künstliche neuronale Netze im Wesentlichen aus einer Menge an (künstlichen) Neuronen und einer Menge an Verbindungen zwischen diesen Neuronen. Deren genaue Anordnung hat maßgeblichen Einfluss auf die Eigenschaften des neuronalen Netzes, seit der ersten Veröffentlichung auf diesem Gebiet [230] wurden eine Vielzahl verschiedener Architekturen vorgeschlagen, wie z.B. Self-Organizing Maps [199], Hopfield-Netze [180] oder die adaptive Resonanztheorie (ART) [88]. Allen künstlichen neuronalen Netzen ist jedoch gemein, dass diese zunächst in einer Lernphase trainiert werden und sich anschließend insbesondere zur Klassifikation bzw. Mustererkennung eignen. Problematisch an diesem Umstand ist die Tatsache, dass die Lernphase eines künstlichen neuronalen Netzes für gewöhnlich viel Zeit in Anspruch nimmt, da eine beträchtliche Zahl von Trainingsdaten verarbeitet werden muss, um dem Netz das zur Lösung für dessen Aufgabe benötigte Wissen zu vermitteln. Zwar existieren diverse parallele Implementierungen für Backpropagation- und ART-Netze [183, 236, 241, 261], diese Verfahren basieren jedoch auf dem Paradigma der Knoten-Parallelität (engl. node parallelism) und nicht der Trainingsmuster-Parallelität (engl. training pattern parallelism). Die folgenden Abschnitte beschreiben, wie das Training eines ART 2-Netzes unter Verwendung der Münster Skelettbibliothek parallelisiert werden kann [100]. Der entwickelte Algorithmus implementiert ein Verfahren zur Aufteilung der Trainingsdaten und erhält gleichzeitig die grundsätzliche Vorgehensweise der adaptiven Resonanztheorie, d.h. das Konzept von ART 2 wird nicht modifiziert.

Der Rest des Kapitels ist wie folgt aufgebaut: Zunächst werden in Abschnitt 6.2 die wesentlichen Grundlagen künstlicher neuronaler Netze sowie der adaptiven Resonanztheorie beschrieben. Anschließend wird in Abschnitt 6.3 der Algorithmus erläutert, der das parallele Training eines ART 2-Netzes ermöglicht. Darauf aufbauend werden in Abschnitt 6.4 die mit Hilfe der Münster Skelettbibliothek vorgenommenen Implementierungen vorgestellt. Abschnitt 6.5 zeigt die Ergebnisse der durchgeführten Laufzeitmessungen, Abschnitt 6.6 schließt das Kapitel mit einem Fazit ab.

6.2 Grundlagen

Die folgenden Abschnitte 6.2.2 bis 6.2.3 behandeln die wesentlichen Grundlagen, die für das Verständnis der adaptiven Resonanztheorie im Allgemeinen und von ART 2-Netzen im Speziellen unabdingbar sind. Zu diesem Zweck führt Abschnitt 6.2.1 kurz in die Theorie künstlicher neuronaler Netze ein. Abschnitt 6.2.2 beschreibt die zentralen Merkmale und Eigenschaften der adaptiven Resonanztheorie, so dass darauf aufbauend ART 2-Netze als spezielle Ausprägung in Abschnitt 6.2.3 erläutert werden.

6.2.1 Künstliche neuronale Netze

Künstliche neuronale Netze haben ihren Ursprung in der Biologie und versuchen, das Verhalten natürlicher neuronaler Netze durch die Modellierung von Nervenzellen (Neuronen) und deren Verbindungen nachzuempfinden [212, 302]. Ein künstliches neuronales Netz, im Folgenden der Einfachheit halber kurz als neuronales Netz bezeichnet, ist formal definiert als Paar (N, V), wobei N einer Menge an Neuronen und V einer Menge an Verbindungen zwischen diesen Neuronen entspricht [212, S. 51]. Hierbei ist zu beachten, dass die Verbindungen zum einen gerichtet sind, zum anderen über ein spezifisches Gewicht verfügen, so dass ein Element $v \in V$ i.d.R. als Tripel (n_a, n_e, g) modelliert wird, wobei n_a dem Anfangsneuron, n_e dem Endneuron und g dem Gewicht der gerichteten Verbindung zwischen n_a und n_e entspricht. Den Gewichten kommt in diesem Zusammenhang eine besondere Bedeutung zu, da diese zur Speicherung des Erlernten dienen und folglich

sozusagen das Gedächtnis des neuronalen Netzes darstellen. Die Neuronen des Netzes werden für gewöhnlich gruppiert und in Schichten zusammengefasst. Anzumerken bleibt, dass die Anzahl der Neuronen, Verbindungen und Schichten nicht begrenzt ist und stark vom Verwendungszweck des Netzes abhängt. Eine vereinfachte Darstellung eines neuronalen Netzes kann Abbildung 6.1 entnommen werden.

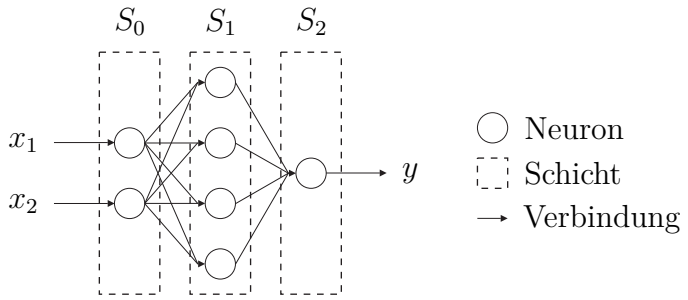


Abbildung 6.1: Vereinfachte Darstellung eines neuronalen Netzes. Der Übersichtlichkeit halber wurde auf die Darstellung der Gewichte verzichtet.

6.2.1.1 Arbeitsweise

Die Arbeitsweise neuronaler Netze hängt zwar stark von der Vernetzungsstruktur sowie dem verwendeten Lernverfahren ab (siehe Abschnitte 6.2.1.2 und 6.2.1.3), grundsätzlich basieren jedoch alle neuronalen Netze auf demselben Prinzip. In einer *Lernphase* werden dem Netz sukzessive n-elementige Eingabevektoren präsentiert, indem jedes Attribut x_i des Vektors mit genau einem Neuron der Eingabeschicht S_0 verbunden wird, $n \in \mathbb{N}, i \in \mathbb{N}, i \leqslant n$. Das neuronale Netz verarbeitet den Eingabevektor und erzeugt infolgedessen einen m-elementigen Ausgabevektor y, $m \in \mathbb{N}$. Abschließend adaptiert das neuronale Netz, in Abhängigkeit vom eingesetzten Lernverfahren, mit Hilfe von y die Gewichte der Verbindungen. Im Anschluss an die Lernphase folgt die eigentliche *Anwendungsphase*, in der dem neuronalen Netz bisher unbekannte Eingabevektoren präsentiert werden. Diese werden auf Basis der adaptierten Gewichte, d.h. des bisher Gelernten, ebenfalls in Ausgabevektoren transformiert, die, je nach Verwendungszweck des neuronalen Netzes,

z.B. die Klassifikation bisher unbekannter Daten ermöglicht. Anzumerken bleibt, dass in der Anwendungsphase i.d.R. keine Adaption der Gewichte stattfindet.

6.2.1.2 Vernetzungsstruktur

Bei der Konstruktion eines neuronalen Netzes spielt die Vernetzungsstruktur eine entscheidende Rolle, da diese, in Kombination mit dem verwendeten Lernverfahren (siehe Abschnitt 6.2.1.3), maßgeblichen Einfluss auf die Effektivität des Netzes hat. Grundsätzlich werden folgende Topologien unterschieden [302, S. 76–79]:

Vernetzungsstrukturen ohne Rückkopplung (engl. feedforward). Bei dieser Kategorie neuronaler Netze existiert kein Pfad, bei dem ein Neuron mehrmals traversiert wird [302, S. 78]. Als Unterarten lassen sich ebenenweise verbundene und allgemeine Feedforward-Netze unterscheiden. Bei ebenenweise verbundenen neuronalen Netzen existieren lediglich Verbindungen von einer Schicht S_i zur nächsten S_{i+1}. Allgemeine Feedforward-Netze erlauben darüber hinaus auch Verbindungen von einer Schicht S_i zu einer anderen Schicht $S_{i+k}, k \in \mathbb{N}$, ermöglichen also das Überspringen einer oder mehrerer Schichten.

Vernetzungsstrukturen mit Rückkopplung (engl. feedback). Bei dieser Kategorie neuronaler Netze existiert u.U. ein Pfad, bei dem ein Neuron mehrmals traversiert wird [302, S. 78]. Als Unterarten lassen sich Netze mit direkter Rückkopplung, Netze mit indirekter Rückkopplung, Netze mit Rückkopplung innerhalb einer Schicht und vollständig verbundene Netze unterscheiden. Bei Netzen mit direkter Rückkopplung kann die Ausgabe eines Neurons an den Eingang desselben Neurons gekoppelt sein. Netze mit indirekter Rückkopplung erlauben Verbindungen von einer Schicht S_i zu einer anderen Schicht $S_{i-k}, k \in \mathbb{N}$. Bei Netzen mit Rückkopplung innerhalb einer Schicht kann die Ausgabe eines Neurons einer Schicht auch an den Eingang eines Neurons derselben Schicht gekoppelt werden. Vollständig verbundene Netze weisen Verbindungen zwischen sämtlichen Neuronen auf.

6.2.1.3 Lernverfahren

Neben der Vernetzungsstruktur hat auch das verwendete Lernverfahren einen entscheidenden Einfluss auf die Effektivität eines neuronalen Netzes. Grundsätzlich werden folgende Arten unterschieden [302, S. 93–96]:

Überwachtes Lernen (engl. supervised learning). Die Anpassung der Gewichte findet in Abhängigkeit vom Feedback einer externen Komponente statt, dem sogenannten *Lehrer*. Dieser berechnet einen Fehlervektor, der die Abweichung zwischen dem Ausgabevektor und dem gewünschten Resultat quantifiziert. Der Fehlervektor wird ins Netz zurückgeleitet, woraufhin eine Anpassung der Gewichte mit dem Ziel erfolgt, den Fehlervektor zu verringern. Anzumerken bleibt, dass ein überwachtes Lernverfahren entsprechende Trainingsdaten inklusive gewünschter Ergebnisse voraussetzt.

Bestärkendes Lernen (engl. reinforced learning). Die Anpassung der Gewichte findet ebenfalls in Abhängigkeit vom Feedback eines Lehrers statt, hierbei wird jedoch lediglich berechnet, ob die Ausgabe richtig oder falsch ist. Diese Information wird, analog zum überwachten Lernen, in das Netz zurückgeleitet, woraufhin eine Anpassung der Gewichte erfolgt. Anzumerken bleibt, dass auch hier entsprechende Trainingsdaten inklusive gewünschter Ergebnisse vorausgesetzt werden.

Unüberwachtes Lernen (engl. unsupervised learning). Die Anpassung der Gewichte findet autonom statt, d.h. ohne das Feedback eines Lehrers. Die Trainingsdaten müssen lediglich aus einer beliebigen Anzahl an Eingabevektoren bestehen, die das Netz selbstständig zu klassifizieren versucht.

Anzumerken bleibt, dass neuronale Netze, die ein überwachtes Lernverfahren verwenden, i.d.R. schneller trainiert werden können, da zur Anpassung der Gewichte durch den Lehrer zusätzliche Informationen zur Verfügung gestellt werden [302, S. 95]. Häufig ist die Verfügbarkeit der gewünschten Ergebnisse jedoch nicht gegeben, so dass auf ein unüberwachtes Lernverfahren zurückgegriffen werden muss.

6.2.2 Adaptive Resonanztheorie

Der Begriff *adaptive Resonanztheorie* bezeichnet ein Designkonzept, das einer ganzen Familie neuronaler Netze zu Grunde liegt. Deren Entwicklung verfolgte das Ziel, eine Lösung für das sogenannte *Stabilitäts-Plastizitäts-Dilemma*[29] zu finden, das bisherige neuronale Netze aufwiesen [212, S. 134 f.]. ART-Netze eignen sich speziell zur Klassifikation bisher unbekannter Eingabedaten und implementieren, mit Ausnahme von ARTMAP (siehe unten), ein unüberwachtes Lernverfahren. Hierbei ist wichtig, dass die Anzahl der zu erlernenden Klassen autonom festgelegt wird, eine vom Benutzer definierte Obergrenze $m \in \mathbb{N}$ aber nicht überschreiten kann. Die ART-Familie umfasst folgende Modelle:

ART [88] ist die ursprüngliche Form und kann ausschließlich binäre Eingabedaten verarbeiten [212, S. 135–157].

ART 2 [89] erweitert ART um die Verarbeitung analoger, d.h. reellwertiger Eingabedaten [212, S. 157–164]. Mit ART 2-A [93] existiert darüber hinaus eine Variante von ART 2, die eine deutlich schnellere Konvergenzrate aufweist [212, S. 164 ff.].

ART 3 [90] ist eine Weiterentwicklung von ART 2, bei der die chemischen Vorgänge innerhalb einer Synapse mit Hilfe von Differentialgleichungen noch naturgetreuer modelliert werden [212, S. 166–169].

Fuzzy ART [94] ist eine Weiterentwicklung von ART um die Nutzung von Fuzzy-Logik, wodurch die Zuordnung eines Eingabedatums zu mehreren Klassen ermöglicht wird [212, S. 173–177].

ARTMAP [92] kombiniert zwei beliebige ART-Netze zu einem Gesamtnetz, so dass dieses überwacht trainiert werden kann [212, S. 170–173]. Mit der Fuzzy ARTMAP [91], der Distributed ARTMAP [96], der Instance-Counting ARTMAP [95] und der Default ARTMAP [87] existieren darüber hinaus Varianten der ARTMAP.

[29] Stabilität bezeichnet die Fähigkeit zur Beibehaltung des Gelernten, Plastizität die Fähigkeit zur Anpassung eines neuronalen Netzes. Das Stabilitäts-Plastizitäts-Dilemma bezeichnet den Umstand, dass ein neuronales Netze zwar in der Lage ist, Neues zu erlernen, hierbei aber dazu tendiert, in Folge einer Überanpassung bereits Gelerntes zu vergessen.

Grundsätzlich bestehen alle ART-Netze aus einer Vergleichsschicht F_1, einer Erkennungsschicht F_2 sowie einer Reset-Komponente R. F_1 und F_2 sind bidirektional voll vernetzt, wobei jede Verbindung mit einem spezifischen Gewicht bewertet ist. Diese Gewichte repräsentieren den Langzeitspeicher des ART-Netzes und werden in der Lernphase entsprechend den präsentierten Eingabedaten angepasst. Neben diversen Parametern spielt der sogenannte *Toleranzparameter* $\rho \in \mathbb{R}^+, \rho \leq 1$ eine entscheidende Rolle bei der Bildung neuer Klassen, da dieser das Abstraktionsniveau des ART-Netzes in der Lernphase festlegt: Während ein hoher Wert die Bildung vieler feingranularer Klassen bewirkt, verursacht ein niedriger Wert die Bildung weniger grobgranularer Klassen. Die Arbeitsweise während der Lernphase lässt sich prinzipiell wie folgt beschreiben:

Bottom-Up-Phase Zunächst wird der n-elementige Eingabevektor X an die Vergleichsschicht F_1 angelegt, $n \in \mathbb{N}$. F_1 verarbeitet den Eingabevektor und leitet das Ergebnis an die Erkennungsschicht F_2 weiter.

Auswahlphase Die Neuronen der Erkennungsschicht tragen untereinander einen Wettbewerb nach dem Alles-oder-nichts-Prinzip (engl. winner-take-all) aus: Nur dasjenige Neuron, dessen entsprechende Klasse die größte Ähnlichkeit zu dem Eingabedatum aufweist, wird aktiviert, alle anderen Neuronen werden unterdrückt. Das Ergebnis, die sogenannte *Resonanz*, wird an die Vergleichsschicht zurückgeleitet.

Top-Down-Phase Zunächst wird mit Hilfe des Toleranzparameters ρ die Ähnlichkeit zwischen der Gewinnerklasse und dem Eingabedatum berechnet. Anschließend ergeben sich zwei Möglichkeiten:

- Falls die Ähnlichkeit groß genug ist, konnte das Eingabedatum erfolgreich klassifiziert werden. Abschließend findet eine geringfügige Anpassung derjenigen Gewichte statt, die die Gewinnerklasse repräsentieren. Dieses Vorgehen ist der Grund für die Namensgebung der adaptiven Resonanztheorie: Das ART-Netz erzeugt in Folge eines Eingabedatums eine Resonanz und adaptiert daraufhin seinen internen Zustand.
- Falls die Ähnlichkeit nicht groß genug ist, blockiert die Reset-Komponente R das aktuelle Gewinnerneuron, so dass ein anderes Neuron den Wettbewerb der Auswahlphase gewinnt. Dieses

Vorgehen führt u.U. dazu, dass sämtliche Neuronen der Erkennungsschicht blockiert werden, da keine bisher erlernte Klasse eine ausreichend große Ähnlichkeit mit dem aktuellen Eingabedatum aufweist. In diesem Fall wird, falls die maximale Zahl zu erlernender Klassen m noch nicht erreicht ist, eine neue Klasse erzeugt, wobei der aktuelle Eingabevektor als Prototyp verwendet wird. Ist die Kapazität des ART-Netzes hingegen bereits erschöpft, gilt das Eingabedatum als nicht klassifizierbar und wird verworfen.

Mit der (nicht) erfolgten Klassifikation ist die Verarbeitung des Eingabevektors beendet, so dass mit der Präsentation des nächsten Vektors fortgefahren werden kann. Das oben beschriebene Vorgehen wiederholt sich so lange, bis sämtliche Eingabevektoren präsentiert wurden. Anschließend kann das Netz in die Anwendungsphase übergehen.

6.2.3 ART 2-Netze

Wie bereits erwähnt, stellt ART 2 [302, S. 259–268] eine Erweiterung der ART-Netze um die Verarbeitung kontinuierlicher, d.h. reellwertiger, n-elementiger Eingabevektoren dar, $n \in \mathbb{N}$. ART 2-Netze lernen unüberwacht, auch die Anzahl der zu erlernenden Klassen wird von dem Netz autonom festgelegt, darf aber eine vom Benutzer definierte Obergrenze $m \in \mathbb{N}$ nicht überschreiten. Das Netz besteht im Wesentlichen aus einer Eingabe-, einer Vergleichs- sowie einer Erkennungsschicht, wobei die Vernetzungsstruktur mehrere Rückkopplungen zwischen den sowie innerhalb der Schichten aufweist (siehe Abb. 6.2). Hierbei ist wichtig, dass die Eingabe- sowie die Vergleichsschicht jeweils n-elementige, die Erkennungsschicht hingegen m-elementige Vektoren verarbeitet. Die Umrechnung zwischen den beiden Dimensionen findet mit Hilfe der $n \times m$ Matrix BU (engl. bottom-up) sowie der $m \times n$ Matrix TD (engl. top-down) statt. Da TD im Wesentlichen der Transposition von BU entspricht, wird im Folgenden der Einfachheit halber von einer allgemeinen Gewichtsmatrix W gesprochen. W repräsentiert sozusagen das Gedächtnis des ART 2-Netzes, da hier die sogenannten *Prototypen* gespeichert werden. Ein Prototyp ist ein typischer Vertreter einer vom ART 2-Netz geformten Klasse und wird während der Lernphase gebildet. Die

Berechnung bzw. das Erlernen dieser Prototypen stellt das eigentliche Ziel der Lernphase dar, da das Netz in der sich anschließenden Anwendungsphase mit Hilfe dieser Prototypen bisher unbekannte Eingabedaten klassifizieren kann.

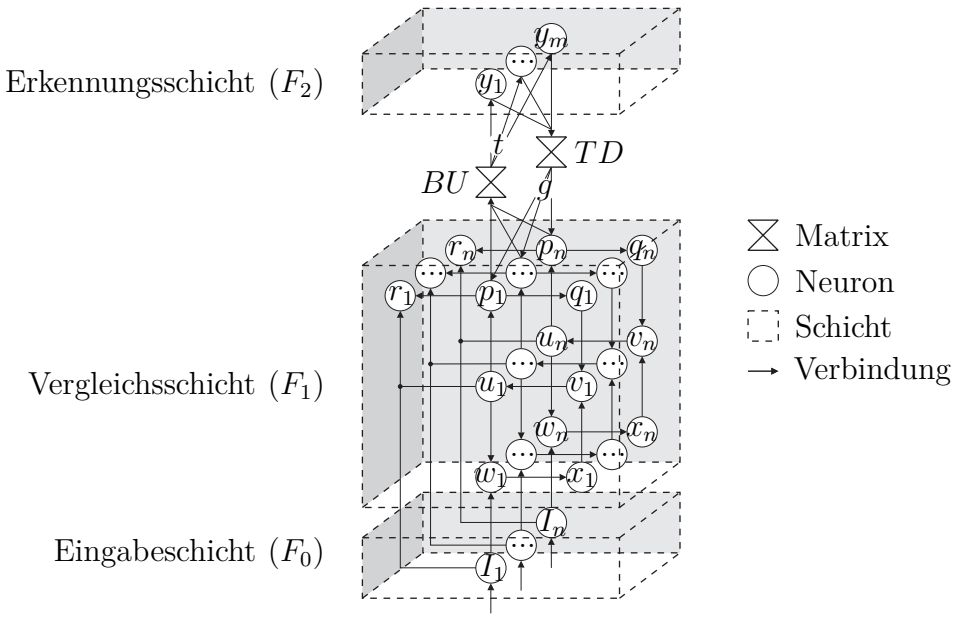


Abbildung 6.2: Aufbau eines ART 2-Netzes. Quelle: [302, S. 260].

Wie bereits erwähnt, muss ein ART 2-Netz in einer Lernphase zunächst trainiert werden, um in der Anwendungsphase bisher unbekannte Eingabedaten klassifizieren zu können. Die Arbeitsweise eines ART 2-Netzes während des Trainings orientiert sich dabei an dem in Abschnitt 6.2.2 beschriebenen dreistufigen Vorgehen, weist aber im Detail einige Unterschiede auf, die im Folgenden erörtert werden:

Bottom-Up-Phase Zunächst wird der n-elementige Eingabevektor X an die Eingabeschicht F_0 angelegt und i.d.R. unverändert[30] an die Vergleichsschicht F_1 weitergeleitet. In der Vergleichsschicht wird der Eingabevektor zunächst durch das Subsystem zur Aufmerksamkeitssteuerung verarbeitet, das aus den Neuronengruppen p, q, u, v, w und x be-

[30] Eine Modifikation kann erfolgen, um z.B. bestimmte Attribute zu verstärken bzw. abzuschwächen.

steht. Das Subsystem berechnet für jedes Attribut des Eingabevektors im Grunde die Lösung einer Differentialgleichung. Dies erfolgt jedoch nicht exakt, sondern mit Hilfe eines iterativen Näherungsverfahrens, das durch folgende Gleichungen beschrieben wird [302, S. 262 f.]:

$$f(x) = \begin{cases} 1 & \text{falls } 0 \leqslant x < \Theta \\ 0 & \text{sonst} \end{cases} \tag{6.1}$$

$$p_i = u_i + g_i \tag{6.2}$$

$$q_i = \frac{p_i}{e + \|P\|} \tag{6.3}$$

$$u_i = \frac{v_i}{e + \|V\|} \tag{6.4}$$

$$v_i = f(x_i) + b \cdot f(q_i) \tag{6.5}$$

$$w_i = I_i + a \cdot u_i \tag{6.6}$$

$$x_i = \frac{w_i}{e + \|W\|} \tag{6.7}$$

Der Definitionsbereich sowie die Semantik der verwendeten Parameter kann Tabelle 6.1 entnommen werden, $\|X\|$ bezeichnet die euklidische Norm des Vektors X. Infolge der beiden Feedback-Schleifen benötigt das Subsystem einige Zyklen, bis die Aktivierungszustände der Neuronen keine Änderungen mehr aufweisen. Sobald das Subsystem stabil ist, wird der Vektor t berechnet und an die Erkennungsschicht F_2 weitergeleitet:

$$t_i = \sum_{j=0}^{m-1} p_i \cdot BU_{i,j} \tag{6.8}$$

Auswahlphase Zwischen den Neuronen der Erkennungsschicht findet der bereits erwähnte Wettbewerb nach dem Alles-oder-nichts-Prinzip statt. Zu diesem Zweck wird zunächst das Element t_J mit der höchsten Aktivierung bestimmt. Anschließend darf lediglich Neuron y_J den Ausgabewert d produzieren, die Aktivierung aller anderen Neuronen wird unterdrückt [302, S. 263 f.]:

$$y_j = \begin{cases} d & \text{falls } t_j = \max(t_j, t_k) \ \forall k \in \mathbb{N}_0, k < m \\ 0 & \text{sonst} \end{cases} \tag{6.9}$$

Abschließend wird der Vektor g berechnet und an die Vergleichsschicht F_1 zurückgeleitet [302, S. 264]:

$$g_i = y_i \cdot TD_{J,i} \tag{6.10}$$

Top-Down-Phase Zunächst muss das oben erwähnte Gleichungssystem erneut gelöst werden, da der Vektor g den zuvor berechneten Gleichgewichtszustand beeinträchtigt. Sobald das Subsystem zur Aufmerksamkeitssteuerung abermals einen stabilen Zustand erreicht hat, wird das Subsystem zur Orientierungssteuerung aktiv. Letzteres besteht ausschließlich aus der Neuronengruppe r und berechnet, wie stark der Eingabevektor von der durch das Gewinnerneuron repräsentierten Klasse abweicht [302, S. 265]:

$$r_i = \frac{I_i + c \cdot p_i}{e + \|U\| + \|c \cdot P\|} \tag{6.11}$$

Das aktuelle Gewinnerneuron wird blockiert, falls die Ähnlichkeit zwischen dem Eingabevektor und der durch das Gewinnerneuron repräsentierten Klasse nicht hoch genug ist, d.h. falls folgende Bedingung wahr ist:

$$\frac{\rho}{e + \|R\|} > 1 \tag{6.12}$$

Anderenfalls konnte der Eingabevektor erfolgreich klassifiziert werden, so dass eine geringfügige Adaption der Gewichte stattfindet [302, S. 264 f.]:

$$BU_{i,J} = TD_{J,i} = \frac{u_i}{1-d} \tag{6.13}$$

Anzumerken bleibt, dass die sogenannte *Aktivierungsfunktion* f zur Kontrastverstärkung bzw. Rauschminderung verwendet wird und nicht zwangsweise auf die präsentierte Form beschränkt ist [302, S. 263]. Die Matrizen werden wie folgt initialisiert [302, S. 266]:

$$BU_{i,j} = \frac{1}{\sqrt{n} \cdot (1-d)} \tag{6.14}$$

$$TD_{i,j} = 0 \tag{6.15}$$

Tabelle 6.1: Auflistung aller von einem ART 2-Netz verwendeten Parameter.

$\mathbb{D}$	Semantik
$a \in \mathbb{N}^+, a \gg 1$	Feedback-Stärke von u zu w
$b \in \mathbb{N}^+, b \gg 1$	Feedback-Stärke von q zu v
$c \in \mathbb{R}^+, c < 1$	Skalierungsfaktor von P
$d \in \mathbb{R}^+, d < 1$	Ausgabe des Gewinnerneurons
$e \in \mathbb{R}^+, e \ll 1$	verhindert Division durch 0
$i \in \mathbb{N}_0, i < n$	Index für den Eingabevektor
$j \in \mathbb{N}_0, j < m$	Index für die Klasse
$J \in \mathbb{N}_0, J < m$	Index des Gewinnerneurons
$m \in \mathbb{N}^+$	maximale Klassenzahl
$n \in \mathbb{N}^+$	Dimension des Eingabevektors
$\Theta \in \mathbb{R}^+, \Theta < 1$	Parameter zur Rauschminderung

6.3 Paralleler Algorithmus

Bei der Entwicklung des Algorithmus, der das parallele Training von ART 2-Netzen ermöglicht, wurde eine Reihe von Designzielen festgelegt, so dass die Eigenschaften des Algorithmus wie folgt beschrieben werden können:

- Der Algorithmus ist in der Lage, die Trainingsphase eines ART 2-Netzes unter Ausnutzung von Trainingsmuster-Parallelität [302, S. 433] zu beschleunigen. Die Beschleunigung ist jedoch insofern beschränkt, als dass maximal $np = m$ Prozesse verwendet werden können.
- Der Algorithmus arbeitet im sogenannten *Offline-Modus*, d.h. Anpassungen an den Gewichtsmatrizen werden erst dann vorgenommen, wenn alle Trainingsmuster verarbeitet wurden [302, S. 107]. Dieses Vorgehen bietet den Vorteil, dass die Berechnung der Gewichtsmatrizen unabhängig von der Präsentationsreihenfolge der Trainingsmuster ist.
- Der Algorithmus identifiziert mit Hilfe eines benutzerdefinierten Parameters $k \in \mathbb{N}$ sogenannte *Ausreißer*, d.h. solche Trainingsmuster, deren Werte stark von denen anderer Trainingsmuster abweichen. Ausreißer werden auf einer schwarzen Liste vermerkt und im weiteren Verlauf der Trainingsphase ignoriert.
- Der Algorithmus erzeugt, falls keine Re-Klassifizierung[31] von Trainingsmustern stattfindet, dieselben Ergebnisse wie ein gewöhnlicher ART 2-Algorithmus, d.h. die Gewichtsmatrizen sind vollkommen identisch. Falls Re-Klassifizierungen vorkommen, was für gewöhnlich relativ selten der Fall ist, sind die Auswirkungen auf die Gewichtsmatrizen in Folge des ART 2-Designs marginal.

Der entwickelte Algorithmus basiert auf der Eigenschaft von ART 2, bei einer erfolgreichen Klassifizierung in der Lernphase nur die entsprechenden Gewichte der Gewinnerklasse zu adaptieren und arbeitet in zwei Phasen.

[31] Re-Klassifizierung bezeichnet den Umstand, dass ein Trainingsmuster in einem Durchlauf des Algorithmus in Klasse i, in einem späteren Durchlauf jedoch in Klasse j, $j \neq i$ eingeordnet wird.

Zunächst werden in einer ersten Phase die Trainingsmuster in bis zu m disjunkte Teilmengen aufgeteilt, wobei sämtliche Trainingsmuster einer Teilmenge derselben Klasse angehören. Anschließend werden in einer zweiten Phase mit Hilfe dieser Teilmengen sowie unter Verwendung des klassischen ART 2-Algorithmus bis zu m ART 2-Netze parallel trainiert. Im Folgenden wird die Arbeitsweise der beiden Phasen detaillierter beschrieben:

- In der ersten Phase des Algorithmus findet mit Hilfe eines modifizierten ART 2-Algorithmus eine Trennung der Trainingsmuster in bis zu m disjunkte Teilmengen statt, wobei jede Teilmenge ausschließlich diejenigen Trainingsmuster enthält, die in dieselbe Klasse eingeordnet wurden. Zu diesem Zweck wird in einer speziellen Datenstruktur für jedes Trainingsmuster festgehalten, in welche Klasse dieses einsortiert wurde. Mit dem Abschluss der Phase ist folglich die Anzahl der zu erlernenden Klassen sowie die Zuordnung von Trainingsmustern zu Klassen fix, d.h. eine Re-Klassifizierung ist nicht mehr möglich. Die Trennung der Trainingsmuster erfordert bis zu m Durchläufe durch den Trainingsdatensatz, wobei in jedem dieser Durchläufe eine neue Klasse erlernt wird. Letzteres erfordert wiederum mehrere Durchläufe durch den Trainingsdatensatz, die im Folgenden als *Zyklen* bezeichnet werden. Zu Beginn eines Durchlaufs wird der Prototyp der neu zu erlernenden Klasse mit dem sogenannten *Kandidaten* initialisiert. Unter einem Kandidaten wird im Folgenden ein Trainingsmuster verstanden, welches unter allen Trainingsmustern die maximale euklidische Distanz zum Kandidaten im vorherigen Durchlauf aufweist.[32] Dieses Vorgehen ermöglicht die schnelle Identifizierung von Ausreißern, was im Folgenden näher erläutert wird. Nach der Initialisierung des aktuellen Prototypen durchläuft der Algorithmus, wie bereits erwähnt, mehrere Zyklen. In jedem Zyklus wird versucht, sämtliche Trainingsmuster mit Hilfe des klassischen ART 2-Algorithmus in die bereits erlernten Klassen einzuordnen. Im Fall einer erfolgreichen Klassifizierung wird die vom ART 2-Algorithmus vorgenommene Adaption der Gewichtsmatrix W jedoch nicht sofort durchgeführt, sondern, entsprechend dem bereits erwähnten Offline-Modus, in einer speziellen $n \times m$ Update-Matrix U aufsummiert:[33]

[32] Im ersten Durchlauf ist der Kandidat das erste Trainingsmuster des Datensatzes.
[33] Sämtliche Werte von U werden zu Beginn jedes Zyklus mit 0 initialisiert.

$$U_{i,J}^{t} = U_{i,J}^{t-1} + \Delta W_{i,J} \quad \text{mit} \quad \Delta W_{i,J} = \frac{u_i}{1-d} - W_{i,J} \tag{6.16}$$

Sei U^* die Update-Matrix nach der Abarbeitung eines Zyklus. Gemäß dem verwendeten Offline-Modus wird die Gewichtsmatrix W erst nach der Abarbeitung eines Zyklus mit den in U^* aufsummierten Änderungen aktualisiert:

$$W^t = W^{t-1} + U^* \tag{6.17}$$

Falls ein Trainingsmuster in keine der bisher erlernten Klassen eingeordnet werden kann, wird mit dem nächsten Trainingsmuster fortgefahren. Ob und in welche Klasse ein Trainingsmuster eingeordnet wurde, wird mit Hilfe einer speziellen Datenstruktur erfasst, die im weiteren Verlauf eine wichtige Rolle spielt.

Nach der Abarbeitung eines Zyklus bzw. der Aktualisierung der Gewichtsmatrix W wird überprüft, ob die Klassifizierung der Trainingsmuster stabil ist, d.h. ob alle Trainingsmuster in dieselbe Klasse eingeordnet wurden wie im vorherigen Zyklus. Dieses Vorgehen ist notwendig, um die Wahrscheinlichkeit einer späteren Re-Klassifizierung der Trainingsmuster zu minimieren. Falls die Klassifizierung instabil ist, wird ein weiterer Zyklus durchgeführt. Anderenfalls findet die bereits erwähnte Identifizierung von Ausreißern statt. Hierbei wird überprüft, ob der im abgeschlossenen Durchlauf erlernten Klasse mindestens k Trainingsmuster zugeordnet wurden. Ist dies der Fall, ist die Bildung der aktuellen Klasse abgeschlossen, so dass mit dem Erlernen einer weiteren Klasse fortgefahren werden kann. Anderenfalls handelt es sich bei den Trainingsmustern per Definition um Ausreißer. Die entsprechenden Trainingsmuster werden auf einer Streichliste vermerkt und somit bei der Durchführung von weiteren Zyklen nicht mehr berücksichtigt, können also nicht als Kandidat bzw. als Prototyp für neu zu erlernende Klassen identifiziert werden. Dementsprechend wird auch die in diesem Durchlauf erlernte Klasse gelöscht, so dass mit dem Erlernen einer weiteren Klasse fortgefahren werden kann. Die erste Phase

des Algorithmus ist beendet, wenn alle Trainingsdaten entweder klassifiziert oder auf der Streichliste vermerkt wurden oder die maximale Zahl zu erlernender Klassen m erreicht wurde.

- In der zweiten Phase des Algorithmus findet mit Hilfe der in der ersten Phase erzeugten Teilmengen sowie unter Verwendung des klassischen ART 2-Algorithmus (siehe Abschn. 6.2.3) das parallele Training von bis zu m ART 2-Netzen statt. Hierbei ist wichtig, dass jedem beteiligten Prozess genau eine Teilmenge des Trainingsdatensatzes zugewiesen wird, d.h. jeder Prozess ist für die Berechnung des Prototypen einer anderen Klasse verantwortlich. Nach der Abarbeitung eines Durchlaufs findet die Rekonstruktion der Gewichtsmatrix W statt, indem alle beteiligten Prozesse die lokal berechneten Prototypen in einem allgather-Kommunikationsschritt (siehe Abschn. 2.3.3.3) austauschen. Falls die aktuelle Gewichtsmatrix W^t keine signifikanten Änderungen gegenüber der im vorherigen Durchlauf berechneten Gewichtsmatrix W^{t-1} aufweist, kann das Training beendet werden. Anderenfalls wird ein weiterer Durchlauf gestartet.

Anzumerken bleibt, dass es in Folge der Aufteilung der Trainingsmuster zu einer ungleichen Lastverteilung kommen kann, falls in eine Klasse wesentlich mehr Trainingsmuster eingeordnet werden als in andere Klassen. In diesem Fall bietet sich eine Reduzierung der Anzahl der Trainingsmuster der entsprechenden Klasse(n) an, die jedoch vom Benutzer manuell vorgenommen werden muss.

6.4 Implementierungen

Der in Abschnitt 6.3 beschriebene Algorithmus lässt sich prinzipiell sowohl mit Hilfe von task- als auch mit Hilfe von datenparallelen Skeletten implementieren. Im Folgenden werden beide Alternativen kurz erörtert, um anschließend in Abschnitt 6.5 die Laufzeiten der beiden Ansätze zu vergleichen. Beiden Implementierungen ist gemein, dass jeweils nur die zweite Phase des modifizierten ART 2-Algorithmus parallelisiert wird, da deren Abarbeitung die meiste Zeit in Anspruch nimmt. Somit erfolgt die Abarbeitung der ersten

Phase zwar durch jeden Prozess redundant, dieses Vorgehen vermeidet jedoch einen zusätzlichen Kommunikationsschritt, der erforderlich wäre, falls lediglich ein Prozess die Partitionierung des Datensatzes vornehmen und das Ergebnis an alle beteiligten Prozesse senden würde.

- Die datenparallele Implementierung verwendet zur Speicherung der Gewichtsmatrix W die Klasse `DistributedMatrix`. Die Partitionierung der verteilten Matrix erfolgt spaltenweise, so dass jeder Prozess genau eine Spalte der Gewichtsmatrix, d.h. einen Prototypen, lokal vorhält. Nach der Aufteilung der Trainingsmuster iteriert jeder Prozess über den lokalen Datensatz und nimmt, entsprechend dem ART 2-Algorithmus, Anpassungen an der Gewichtsmatrix vor. Nach der Abarbeitung eines Durchlaufs rekonstruieren die beteiligten Prozesse mit Hilfe eines allgather-Kommunikationsschritts (siehe Abschn. 2.3.3.3) die globale Gewichtsmatrix und überprüfen, ob die Prototypen stabil sind. Ist dies der Fall, kann das Training beendet werden und die rekonstruierte Gewichtsmatrix repräsentiert das Endergebnis. Anderenfalls durchlaufen sämtliche Prozesse den jeweiligen lokalen Datensatz erneut, um die Prototypen weiter anzupassen.
- Die taskparallele Implementierung trainiert das ART 2-Netz unter Verwendung der Klasse `Farm` [248, 250, 251]. Letztere besteht aus mehreren Arbeitern, die die eigentlichen Berechnungen durchführen. Nach der Aufteilung der Trainingsmuster iteriert jeder Arbeiter über den lokalen Datensatz und nimmt, entsprechend dem ART 2-Algorithmus, Anpassungen an der Gewichtsmatrix vor. Hierbei ist wichtig, dass zwar jeder Arbeiter mit Hilfe der Klasse `Matrix` eine vollständige Gewichtsmatrix vorhält, entsprechend dem lokalen Datensatz jedoch lediglich der Prototyp *einer* Klasse adaptiert wird, d.h. die restlichen Prototypen der Matrix werden nicht modifiziert. Nach der Abarbeitung eines Durchlaufs rekonstruieren die Arbeiter mit Hilfe eines allgather-Kommunikationsschritts (siehe Abschn. 2.3.3.3) die globale Gewichtsmatrix und überprüfen, ob die Prototypen stabil sind.[34][35]

[34] Analog zu Abschnitt 5.3.2 ist hierfür erneut Ad-hoc-Parallelität notwendig, da die Klasse `Farm` keine Kommunikation zwischen den Arbeitern vorsieht.

[35] Hierfür implementiert die Klasse `Matrix` den von Muesli zur Verfügung gestellten Serialisierungsmechanismus (siehe Abschn. 3.2.5).

Ist dies der Fall, kann das Training beendet werden und die rekonstruierte Gewichtsmatrix repräsentiert das Endergebnis. Anderenfalls durchlaufen sämtliche Arbeiter den jeweiligen lokalen Datensatz erneut, um die Prototypen weiter anzupassen.

6.5 Ergebnisse

Im Folgenden werden die Ergebnisse einiger Laufzeitmessungen der in Abschnitt 6.4 beschriebenen Implementierungen erläutert (siehe Tab. 6.2). Alle Messungen wurden auf dem ZIVHPC (siehe Abschn. 2.5) vorgenommen, wobei jeder MPI-Prozess auf einem separaten Rechenknoten ausgeführt wurde. Der verwendete Datensatz enthält 40 Trainingsmuster, die die vier Ecken eines Tetraeders repräsentieren, wobei jede Ecke von zehn Trainingsmustern beschrieben wird. Infolgedessen verwenden die parallelen Implementierungen des Algorithmus $np = 4$ Prozesse. Anzumerken bleibt, dass die Trainingsmuster einem gewissen Rauschen unterliegen, d.h. mit einer moderaten Varianz um die tatsächlichen Koordinaten der jeweiligen Ecke verstreut sind. Die Aufgabe des ART 2-Netzes besteht darin, die vier Ecken des Tetraeders zu identifizieren, die Trainingsmuster entsprechend zu klassifizieren und die Prototypen der Ecken zu berechnen.

Tabelle 6.2: Vergleich der sequenziellen und der parallelen ART 2-Implementierungen.

Implementierung	Laufzeit [s]	Speedup
sequenziell	21,47	1,00
datenparallel	6,11	3,51
taskparallel	6,07	3,54

Zunächst muss bemerkt werden, dass alle drei Implementierungen die vier Ecken des Tetraeders korrekt erkennen und exakt dieselben Prototypen berechnen. Im Vergleich zur sequenziellen Variante erfolgt dies durch die paral-

lelen Implementierungen jedoch erheblich schneller, so dass die Laufzeit von ca. 21 s auf rund 6 s reduziert werden kann, was einem Speedup von etwa 3,5 entspricht. Der Unterschied zwischen den parallelen Implementierungen ist vernachlässigbar gering. Der optimale Speedup von 4 wird aufgrund des Mehraufwands in der ersten Phase des Algorithmus sowie des zusätzlichen Kommunikationsaufwands nicht erreicht. Nichtsdestotrotz zeigen die Ergebnisse, dass das Training eines ART 2-Netzes unter Verwendung des vorgestellten Algorithmus effizient parallelisiert sowie eine deutliche Reduktion der Laufzeit erzielt werden kann.

6.6 Fazit

In diesem Kapitel wurde eine Variante des ART 2-Algorithmus zur unüberwachten Klassifikation von Eingabedaten vorgestellt, der das parallele Training eines ART 2-Netzes ermöglicht. Zunächst wurde in die allgemeine Thematik künstlicher neuronaler Netze eingeführt, wobei die Aspekte Arbeitsweise, Vernetzungsstruktur und Lernverfahren als Unterscheidungsmerkmal neuronaler Netze herausgearbeitet wurden (siehe Abschn. 6.2.1). Als spezieller Vertreter neuronaler Netze wurden anschließend das Konzept und die Funktionsweise der adaptiven Resonanztheorie sowie die damit verbundene Familie von ART-Netzen ausführlich beschrieben (siehe Abschn. 6.2.2). Den Abschluss des Grundlagenteils bildete der Abschnitt über die Theorie von ART 2-Netzen zur Verarbeitung von reellwertigen Eingabedaten, wobei hier insbesondere der Aufbau, die Arbeitsweise sowie das zu lösende Gleichungssystem detailliert erläutert wurden (siehe Abschn. 6.2.3). Darauf aufbauend wurde ein Algorithmus entwickelt, der das parallele Training eines ART 2-Netzes ermöglicht, sowie dessen Eigenschaften und Merkmale erörtert (siehe Abschn. 6.3). Der parallele Algorithmus wurde mit Hilfe von Muesli unter Verwendung sowohl von daten- als auch taskparallelen Skeletten implementiert (siehe Abschn. 6.4). Die durchgeführten Laufzeitmessungen haben gezeigt, dass die parallelen Implementierungen das Training eines ART 2-Netzes im Vergleich zur sequenziellen Variante erheblich verkürzen, d.h. eine effiziente Parallelisierung ermöglichen (siehe Abschn. 6.5).

Kapitel 7

Schlussbetrachtungen

Inhalt

7.1 Zusammenfassung

Die vorliegende Arbeit hat sich mit dem datenparallelen Teil der Münster Skelettbibliothek Muesli beschäftigt und ist dabei insbesondere auf die zahlreichen implementierten Erweiterungen sowie praktische Anwendungen eingegangen. Zunächst wurde in Kapitel 1 auf die Problematiken der Erstellung paralleler Programme eingegangen, um damit die Verwendung algorithmischer Skelette zu motivieren (siehe Abschn. 1.1). Diese kapseln typische parallele Rechen- bzw. Kommunikationsmuster und vermeiden somit das Auftreten häufiger Fehler bei der parallelen Programmierung. Neben einer Klassifikation wurden insbesondere die Vorteile der skelettalen Programmierung herausgearbeitet. Im Anschluss an die Formulierung der zu erfüllenden Zielsetzungen (siehe Abschn. 1.2) wurde kurz auf den Aufbau der Arbeit eingegangen (siehe Abschn. 1.3).

Kapitel 2 hat die in Bezug auf Muesli relevanten Grundlagen der parallelen Programmierung thematisiert. Zu diesem Zweck wurde zunächst mit

der Flynn'schen Taxonomie ein Modell zur Klassifikation von Parallelrechnern vorgestellt (siehe Abschn. 2.2). Da die Betrachtung von Daten- und Instruktionsströmen zur Klassifikation aktueller Parallelrechner nicht mehr ausreichend ist, wurde das Modell mit der Speicherarchitektur um ein zusätzliches Kriterium erweitert. Dieses Vorgehen ist naheliegend, da die Speicherarchitektur erheblichen Einfluss auf die zur Programmierung eines Parallelrechners verwendeten Technologien hat. In den folgenden Abschnitten wurden mit dem Message Passing Interface MPI (siehe Abschn. 2.3) und der Open-Multi Processing API OpenMP (siehe Abschn. 2.4) zwei von Muesli verwendete Schnittstellen zur parallelen Programmierung vorgestellt, wobei hier nur die für Muesli relevanten Features erwähnt wurden. Im Anschluss an eine kurze Beschreibung des ZIVHPC, der Testumgebung für die Laufzeitmessungen (siehe Abschn. 2.5), erfolgte eine ausführliche Beschreibung des aktuellen Forschungsstands, um Muesli im Allgemeinen und den Beitrag dieser Arbeit im Speziellen besser einordnen zu können (siehe Abschn. 2.6).

Aufbauend auf den in Kapitel 2 vermittelten Grundlagen, insbesondere dem aktuellen Forschungsstand, hat sich Kapitel 3 mit der Münster Skelettbibliothek Muesli befasst. Neben einer Beschreibung des Parallelisierungskonzepts sowie der grundlegenden Eigenschaften (Funktionen höherer Ordnung, parametrischer Polymorphismus, partielle Applikationen, Currying, Serialisierung, siehe Abschn. 3.2) wurde kurz auf die momentan implementierten verteilten Datenstrukturen für Felder, Matrizen und dünnbesetzte Matrizen sowie die zur Verfügung gestellten Skelette *count*, *fold*, *map*, *permute*, *zip* und deren Varianten eingegangen (siehe Abschn. 3.3 und 3.4). Im Anschluss wurden die, neben der in Kapitel 4 beschriebenen verteilten Datenstruktur für dünnbesetzte Matrizen, wesentlichen an Muesli vorgenommenen Erweiterungen erörtert (siehe Abschn. 3.5). Zu diesen zählen die transparente und optionale Unterstützung hybrider Speicherarchitekturen bzw. Mehrkernprozessoren unter Verwendung der OpenMP Abstraktionsschicht OAL, die implementierten kollektiven, serialisierten Kommunikationsfunktionen `msl::allgather`, `msl::allreduce` und `msl::broadcast` sowie diverse kleinere Ergänzungen. Die Effizienz der vorgenommenen Erweiterungen wurde mit Hilfe einiger Benchmarks verdeutlicht (siehe Abschn. 3.6).

Neben den in Kapitel 3 beschriebenen Erweiterungen hat sich Kapitel 4 mit einer weiteren, essentiellen Ergänzung der Skelettbibliothek befasst, ei-

ner verteilten Datenstruktur für dünnbesetzte Matrizen. Die Datenstruktur implementiert, analog zu den verteilten Datenstrukturen für Felder und Matrizen, das Konzept des parametrischen Polymorphismus sowie benutzerdefinierbare Schemata zur Kompression bzw. Lastverteilung (siehe Abschn. 4.2). Neben einer ausführlichen Beschreibung der verwendeten Schnittstellen und Klassen wurde besonderes Augenmerk auf die implementierten Skelette gelegt, um die Unterstützung hybrider Speicherarchitekturen bzw. das Zusammenwirken von MPI und OpenMP zu verdeutlichen und damit die bisher einmaligen Skalierungseigenschaften von Muesli darzulegen (siehe Abschn. 4.3). Auch hier wurde die Effizienz der vorgenommenen Implementierung mit Hilfe einiger Benchmarks verdeutlicht (siehe Abschn. 4.4).

Kapitel 5 hat einerseits mit der Positronen-Emissions-Tomographie als bildgebendem Verfahren der Nuklearmedizin, andererseits mit dem LM OSEM-Algorithmus als Verfahren zur Rekonstruktion PET-basierter Bilder einen praktischen Anwendungsfall für die in Kapitel 3 vorgestellte Skelettbibliothek beschrieben. Neben den für das Verständnis erforderlichen Grundlagen, wie der Erfassung der Rohdaten, dem LM OSEM-Algorithmus sowie den möglichen Dekompositionsstrategien (siehe Abschn. 5.2), wurde besonderes Augenmerk auf die vorgenommenen Implementierungen sowie deren Vergleich gelegt (siehe Abschn. 5.3). Ausgehend von einer rein sequenziellen Variante wurde der LM OSEM-Algorithmus einerseits unter Verwendung daten- bzw. taskparalleler Skelette, andererseits unter direkter Verwendung von MPI und OpenMP bzw. MPI und TBB implementiert. Die verschiedenen Varianten wurden im Hinblick auf die erforderliche Laufzeit bzw. den erzielten Speedup mit Hilfe einiger Benchmarks sowie den erreichten Abstraktionsgrad miteinander verglichen (siehe Abschn. 5.4).

Kapitel 6 hat mit einem Verfahren zum parallelen Training neuronaler ART 2-Netze einen weiteren Anwendungsfall für die in Kapitel 3 vorgestellte Skelettbibliothek beschrieben. Aufbauend auf den Grundlagen künstlicher neuronaler Netze, der adaptiven Resonanztheorie im Allgemeinen sowie der Theorie von ART 2-Netzen im Speziellen (siehe Abschn. 6.2) wurde anschließend zunächst das Konzept des parallelen Algorithmus erläutert (siehe Abschn. 6.3). Danach wurde sowohl eine daten- als auch eine taskparallele Implementierung kurz vorgestellt (siehe Abschn. 6.4). Analog zu den vorherigen Kapiteln wurde auch hier die Effizienz der vorgenommenen Implementierung mit Hilfe einiger Benchmarks verdeutlicht (siehe Abschn. 6.5).

7.2 Ausblick

Die folgenden Abschnitte vermitteln einen Ausblick auf mögliche zukünftige Forschungsthemen in Bezug auf Muesli. Die vorgestellten Themen erheben keinen Anspruch auf Vollständigkeit, beruhen aber allesamt auf der Annahme, dass im Hochleistungsrechnen in Zukunft (noch) mehr Heterogenität zu erwarten ist. Die Annahme basiert einerseits auf dem Moore'schen Gesetz, nach dem sich die Transistorzahl je Prozessor etwa alle 18 Monate verdoppelt [232], anderseits auf dem seit dem Erscheinen der ersten Mehrkernprozessoren zu beobachtendem Umstand, nach dem die Leistung eines Prozessors nicht mehr durch eine Erhöhung der Taktfrequenz, sondern durch eine Erhöhung der Anzahl an Rechenkernen erzielt wird. Für das Hochleistungsrechnen bedeutet dies, dass die Rechenknoten von Parallelrechnern noch häufiger als dies momentan ohnehin bereits der Fall ist aus Mehrkernprozessoren bestehen werden. Die nächste Generation von Mehrkernprozessoren, sogenannte *Manycores*, zielen bereits darauf ab, mehrere Hundert Rechenkerne auf einem Prozessor zu integrieren und befinden sich momentan in der Erforschung [72, 164]. Für Muesli bedeutet dies, die Unterstützung hybrider Speicherarchitekturen einerseits durch die Integration neuer Technologien, wie z.B. C++0x oder OpenMP 3.0 (siehe Abschn. 7.2.1), noch effizienter zu gestalten, andererseits auf die komplette Bibliothek auszuweiten, d.h. sowohl auf die kollektiven, serialisierten Kommunikationsfunktionen (siehe Abschn. 7.2.2) als auch auf die taskparallelen Skelette (siehe Abschn. 7.2.3). Ein weiterer Forschungsschwerpunkt könnte sich mit dem Thema GPGPU befassen, d.h. mit der Erstellung einer Version von Muesli, die zusätzlich auch auf Grafikkarten skaliert (siehe Abschn. 7.2.4). Die Implementierung einer Java-basierten Version von Muesli erscheint ebenfalls erstrebenswert (siehe Abschn. 7.2.5).

7.2.1 C++0x und OpenMP 3.0

In naher Zukunft wird es einen mit C++0x bezeichneten Nachfolger des aktuellen C++03-Standards geben [2]. C++0x befindet sich momentan in der Abstimmungsphase. Mit einer Veröffentlichung wird 2011, spätestens 2012

gerechnet, in Teilen wird der zukünftige Standard jedoch bereits von aktuellen Compilern unterstützt. Vorrangig werden eine Reihe von Spracherweiterungen vorgenommen, aber auch der Umfang der Standard Template Library wird erweitert. Von besonderem Interesse ist hier die Integration eines Thread-Konzepts zum Ausdrücken von Nebenläufigkeit auf Systemen mit gemeinsamem Speicher [67]. Neben der Klasse `std::thread` werden auch Java-ähnliche Futures unterstützt, die Synchronisation findet mit Hilfe von Mutexen statt. Bezüglich Muesli wäre zu untersuchen, ob für die Unterstützung von Parallelrechnern mit hybridem Speicher auf die Verwendung von OpenMP 2.5 verzichtet werden kann und die implementierten Features stattdessen ausschließlich mit Hilfe von C++0x umgesetzt werden können. Hierbei sollte allerdings darauf geachtet werden, den aktuellen OpenMP 3.0-Standard zu berücksichtigen. Dieser erweitert, verglichen mit OpenMP 2.5, u.a. die **`for`**-Direktive, so dass die Indexvariable der entsprechenden Schleife vom Typ `size_t` sein darf (siehe Abschn. 4.3.4.2). Des Weiteren ermöglicht OpenMP 3.0 die Parallelisierung von Schleifen, die das Iterator-Interface einer STL Container-Klasse benutzen, um über die Elemente des entsprechenden Objekts zu traversieren. Dies setzt allerdings voraus, dass die entsprechende Klasse einen Iterator definiert, der einen wahlfreien Zugriff ermöglicht, z.B. `std::deque` oder `std::vector`.

7.2.2 Kollektive, serialisierte Kommunikationsfunktionen für Mehrkernprozessoren

Die in Abschnitt 3.5.3 beschriebenen kollektiven, serialisierten Kommunikationsfunktionen profitieren momentan nicht von mehreren Rechenkernen pro Rechenknoten, sondern verwenden zur Durchführung der Kommunikation lediglich einen MPI-Prozess pro Rechenknoten, d.h. etwaige zusätzliche Rechenkerne bleiben ungenutzt. Grundsätzlich ist die Verwendung zusätzlicher Rechenkerne pro Rechenknoten jedoch durchaus möglich, wie Listing 7.1 demonstriert. Hier werden die Rechenkerne eines Rechenknotens dazu verwendet, die ID des Wurzelprozesses, d.h. desjenigen Prozesses, für den `pid == 0` gilt (siehe Z. 7), schneller an die beteiligten Prozesse zu verteilen. Zu diesem Zweck erzeugt der Wurzelprozess zunächst mit Hilfe der

parallel-Direktive mehrere Threads und verwendet dabei die **private**-Klausel, so dass jeder Thread eine eigene Kopie der Variablen `dst` erhält (siehe Z. 8). Dieses Vorgehen ist wichtig, da `dst` die ID des empfangenden MPI-Prozesses speichert (siehe Z. 10). Anschließend kann jeder Thread als Sender einer Nachricht auftreten, wobei jeder Thread die ID des Wurzelprozesses als Quelle der Sendeoperation angibt (siehe Z. 11 und Abschn. 2.3.2). Das Empfangen einer Nachricht erfordert keine weiteren Vorkehrungen (siehe Z. 14 f.).

Listing 7.1: Durchführung einer broadcast-Operation unter Verwendung mehrerer Rechenkerne.

```
1  void main(int argc, char** argv) {
2    int dst, pid;    MPI_Status status;
3
4    MPI_Init(&argc, &argv);
5    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
6
7    if(pid == 0) {
8      #pragma omp parallel private(dst)
9      {
10       dst = oal::getThreadNum() + 1;
11       MPI_Send(&pid, 1, MPI_INT, dst, 0, MPI_COMM_WORLD);
12   } }
13   else {
14     MPI_Recv(&pid, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
15              &status);
16   }
17
18   MPI_Finalize();
19 }
```

Vor einer möglichen Umsetzung sollte zunächst untersucht werden, ob die verwendete Kommunikationsinfrastruktur überhaupt eine effiziente Parallelisierung erlaubt, da der vorgestellte Ansatz durch das Erzeugen und Zerstören von Threads einen gewissen Mehraufwand verursacht. Im schlimmsten Fall serialisiert die Kommunikationsinfrastruktur das parallele Versenden der Nachrichten, so dass der vorgestellte Ansatz durch den von OpenMP verursachten Mehraufwand sogar eine längere Laufzeit aufweisen würde.

7.2.3 Taskparallele Skelette für Mehrkernprozessoren

Wie Tabelle 2.7 auf Seite 74 entnommen werden kann, unterstützt Muesli gemeinsame/hybride Speicherarchitekturen bzw. Mehrkernprozessoren nur eingeschränkt. Diese Einschränkung resultiert aus der Tatsache, dass die OpenMP-Erweiterungen lediglich für den datenparallelen Teil der Skelettbibliothek implementiert wurden (siehe Abschn. 3.5.1). Grundsätzlich ist es jedoch erstrebenswert, auch den taskparallelen Teil der Skelettbibliothek entsprechend zu erweitern, um das in aktuellen Parallelrechnern vorhandene Parallelitätspotenzial vollständig und effizient auszunutzen. Vor einer möglichen Implementierung sollte zunächst untersucht werden, welche Technologien eine effektive Umsetzung ermöglichen. Eventuell ist das vom zukünftigen C++-Standard [2] implementierte Thread-Konzept ausreichend [67], u.U. müssen zusätzliche Technologien, wie z.B. OpenMP 3.0 [238] oder TBB [259], verwendet werden. Analog zum datenparallelen Teil der Skelettbibliothek sollte bei der Erweiterung des taskparallelen Teils darauf geachtet werden, dass die Unterstützung von hybriden Speicherarchitekturen einerseits optionalen Charakter hat, so dass sämtliche taskparallelen Skelette auch weiterhin auf Parallelrechnern mit verteiltem Speicher ausführbar sind, und andererseits, gemäß dem Grundgedanken einer Skelettbibliothek, für den Benutzer vollkommen transparent ist.

7.2.4 GPGPU und OpenCL

Neben der Entwicklung von Mehrkernprozessoren hat sich in der jüngsten Vergangenheit ein weiterer Trend im Hochleistungsrechnen etabliert: General-Purpose Computation on Graphics Processing Units (GPGPU), d.h. die Verwendung des Grafikprozessors für nicht grafikbezogene (allgemeine) Berechnungen [141, 193, 239, 246, 281]. Das Konzept gewinnt vermehrt an Popularität, da aktuelle GPUs, im Vergleich zu aktuellen CPUs, sowohl eine höhere absolute Rechenleistung als auch das bessere Preis-Leistungs-Verhältnis aufweisen [79]. Der eigentliche Grund für den hohen Stellenwert im HPC-Bereich liegt jedoch in der Hardware-Architektur moderner Grafikkarten begründet. Letztere besteht mittlerweile aus mehreren Tausend von

Recheneinheiten und bietet somit ein enormes Parallelitätspotenzial.[36] Dieses Potenzial auszuschöpfen, d.h. Muesli in einer Version zu implementieren, die zusätzlich auch auf Grafikkarten skaliert, erscheint daher als sinnvolles und erstrebenswertes Ziel.

Vor einer möglichen Implementierung sollte zunächst untersucht werden, welche Technologien überhaupt eine effiziente Umsetzung erlauben. Während ATI mit Stream [33] bzw. NVIDIA mit CUDA (Compute Unified Device Architecture) [193, 235, 281] zwei proprietäre und zueinander inkompatible Schnittstellen zur Programmierung der eigenen Grafikkarten anbieten, ermöglicht die Khronos Group mit der Open Computing Language OpenCL [193, 290, 294] die gleichzeitige Programmierung von CPUs, GPUs und DSPs (engl. digital signal processor). Mit OpenCL geschriebene Programme werden zur Laufzeit auf sämtliche OpenCL-fähigen Recheneinheiten verteilt und abstrahieren somit von der unterliegenden parallelen Hardware. Verglichen mit Stream und CUDA garantiert OpenCL folglich eine stärkere Plattformunabhängigkeit. Im Gegensatz zu CUDA, das mittlerweile über eine Sprachanbindung zu C++ verfügt, müssen OpenCL-Programme jedoch in einer speziellen Programmiersprache entwickelt werden, nämlich OpenCL C. Letztere verfügt jedoch (noch) nicht über die für Muesli notwendigen Spracheigenschaften, wie z.B. Templates, Klassen/Objekte und Funktionszeiger, so dass Features wie parametrischer Polymorphismus, partielle Applikationen und Currying nicht bzw. nur eingeschränkt umgesetzt werden können.

7.2.5 Neuimplementierung von Muesli mit Java

Wie Tabelle 2.7 auf Seite 74 entnommen werden kann, wurde eine Vielzahl aktueller Skelettbibliotheken unter Verwendung der Programmiersprache Java implementiert. Verglichen mit C++ bietet der Einsatz von Java eine

[36] Die momentan schnellste verfügbare Grafikkarte *ATI Radeon HD 5970* verfügt über 3.200 sogenannte *Stream Processing Units* und erreicht bei einfacher Genauigkeit eine Rechenleistung von 4,64 TeraFLOPS. Zum Vergleich: Der momentan schnellste in den TOP500 [293] gelistete Parallelrechner *Jaguar* besteht aus 224.162 AMD Opteron Six Core Prozessoren @ 2,6 GHz und erreicht eine theoretische Rechenleistung von 2,331 PetaFLOPS.

Reihe von Vorteilen, wie z.B. ein automatisches Speichermanagement [200, S. 107 f.], ein integriertes Thread-Konzept [200, S. 499–527] zur effizienten Unterstützung von Mehrkernprozessoren oder einen unkomplizierten Serialisierungsmechanismus [200, S. 957–980]. Andererseits lassen sich gewisse Features von Muesli, wie z.B. parametrischer Polymorphismus in Form von Template-Parametern (siehe Abschn. 3.2.2), Funktionen höherer Ordnung (siehe Abschn. 3.2.3) und partielle Applikationen (siehe Abschn. 3.2.4), in Folge der begrenzten Spracheigenschaften nicht ohne weiteres implementieren. Insbesondere der parametrische Polymorphismus lässt sich in Java unter Verwendung von Generics [200, S. 371–384] nur teilweise umsetzen und verursacht, wie bereits in Abschnitt 3.2.2 beschrieben, zur Laufzeit einen erheblichen Mehraufwand. Nichtsdestotrotz ist die vollständige Neuimplementierung einer Java-basierten Version von Muesli grundsätzlich erstrebenswert, in [142] wird mit JMuesli ein erster Prototyp für den datenparallelen Teil der Skelettbibliothek beschrieben. Der Prototyp verwendet Java Generics, um vom verwendeten Datentyp zu abstrahieren, definiert eine spezielle Schnittstelle zur Simulation von Funktionen höherer Ordnung, partiellen Applikationen und Currying und verwendet MPJ Express [41, 270] zur Durchführung der Kommunikation. Es werden sämtliche bereits von Muesli bereitgestellten verteilten Datenstrukturen implementiert (siehe Abschn. 3.3). Letzte wiederum implementieren, analog zur C++-Version von Muesli, die Skelette *fold*, *map* und *zip* inklusive diverser Varianten (siehe Abschn. 3.4). Als größte Unzulänglichkeit von JMuesli muss die im Vergleich zu Muesli schlechte Performanz bemängelt werden, die durch eine hohe Anzahl an Objekten und das damit verbundene Speichermanagement verursacht wird. In einer endgültigen Version muss dieser Nachteil ohne Frage beseitigt werden. In diesem Zusammenhang sind jedoch Ergebnisse ermutigend, die bei einer speziell auf den Datentyp `int` zugeschnittenen Variante von JMuesli keine Laufzeitunterschiede zwischen der Java- und der C++-basierten Version von Muesli ergeben haben. Als weitere Forschungstätigkeit könnte JMuesli auch auf den taskparallelen Teil der Skelettbibliothek erweitert werden.

7.3 Schlussfazit

Obwohl die Grundgedanken algorithmischer Skelette bereits 1989 veröffentlicht wurden [104], ist die Relevanz des Konzepts in Anbetracht der steigenden Heterogenität moderner Parallelrechner größer als je zuvor. Bestand die Aufgabe zu Beginn der skelettalen Programmierung zunächst darin, die Programmierung von Parallelrechnern mit verteiltem und später auch mit gemeinsamem Speicher zu vereinfachen, müssen mittlerweile zusätzlich hybride Speicherarchitekturen effizient unterstützt werden. Zukünftige Skelettbibliotheken müssen darüber hinaus Technologien berücksichtigen, die ein noch höheres Maß an paralleler Verarbeitung ermöglichen, wie z.B. GPGPU und Manycores.

Vor diesem Hintergrund wurde der datenparallele Teil der Münster Skelettbibliothek Muesli erweitert, um die aktuellen Anforderungen und Gegebenheiten im Hochleistungsrechnen eingehender zu berücksichtigen. Ausgehend von den in Abschnitt 1.2 formulierten Zielsetzungen wurden auf der einen Seite sämtliche bereits implementierten datenparallelen Skelette der verteilten Datenstrukturen für Felder und Matrizen überarbeitet, um effizient auf hybriden Speicherarchitekturen bzw. Mehrkernprozessoren zu skalieren. Auf der anderen Seite wurde Muesli mit der Klasse `DistributedSparseMatrix` um eine flexible und erweiterbare Datenstruktur zur Handhabung dünnbesetzter Matrizen ergänzt. Durch zahlreiche praktische Anwendungen, wie z.B. die Suche nach kürzesten Wegen mit dem Bellman-Ford-Algorithmus, die medizinische Bildrekonstruktion mit dem LM OSEM-Algorithmus oder das parallele Training von ART 2-Netzen, konnte darüber hinaus die Praktikabilität und Effizienz der skelettalen Programmierung demonstriert werden. Somit konnten sämtliche formulierten Zielsetzungen vollständig erfüllt werden.

Die vorliegende Arbeit hat gezeigt, dass die skelettale Programmierung im Allgemeinen und Muesli im Speziellen die Erstellung paralleler Programme durch die Implementierung typischer paralleler Rechen- und Kommunikationsmuster erheblich vereinfacht. Die Bereitstellung algorithmischer Skelette vermeidet häufige Probleme bei der parallelen Programmierung und ermöglicht, verglichen mit einer manuellen Implementierung unter direkter Verwendung von MPI, OpenMP oder TBB, durch die Kapselung der

Applikations- und Kommunikationslogik ein deutlich höheres Abstraktionsniveau. Infolgedessen kann bei der Erstellung eines parallelen Programms die Implementierung des Algorithmus in den Vordergrund rücken, während die Details der Parallelisierung durch die Skelette festgelegt werden. Zum jetzigen Zeitpunkt existiert mit Muesli, im Vergleich zu anderen Forschungsarbeiten (siehe Tab. 2.7, S. 74), eine der fortschrittlichsten und umfangreichsten Skelettbibliotheken überhaupt. Durch die vorgenommenen Überarbeitungen und Erweiterungen wurde gewährleistet, dass der datenparallele Teil der Skelettbibliothek optional auch auf modernen Parallelrechnern mit hybrider Speicherarchitektur effizient skaliert. In Anbetracht der in nicht allzu ferner Zukunft verfügbaren Manycores ist dies eine Eigenschaft von erheblicher praktischer Relevanz, die in konkurrierenden Skelettbibliotheken erst nachgerüstet werden muss.

Anhang A

Hilfsklassen

Inhalt

A.1 EventPacket

Die Klasse `EventPacket` wird bei der taskparallelen Implementierung des LM OSEM-Algorithmus verwendet und bündelt mehrere Ereignisse, so dass diese von dem `Initial`-Skelett nicht einzeln versendet werden müssen (siehe Abschn. 5.3.2). Insgesamt werden `noe` Ereignisse in dem zweidimensionalen Feld `events` gespeichert (siehe Z. 10), für jedes Ereignis werden `NOP = 7` Eigenschaften vom Typ **`double`** vorgehalten (siehe Z. 4). Die Reihenfolge, in der letztere in dem Feld `events` abgelegt werden, wird durch die Konstanten `X1`, `X2`, `Y1`, `Y2`, `Z1`, `Z2` und `WG` festgelegt (siehe Z. 1–4). Anzumerken bleibt, dass die Variable `flag` anzeigt, ob das Ende der aktuellen Teilmenge s_i bzw. des gesamten Rohdatensatzes erreicht wurde (siehe Z. 10). Gemäß dem von der Skelettbibliothek implementierten Serialisierungsmechanismus (siehe Abschn. 3.2.5) erbt `EventPacket` von der Klasse `msl::Serializable` und überschreibt die Funktionen `reduce`, `expand`

und `getSize` (siehe Z. 6 und Z. 67–90). Hierbei kommen die Funktionen `msl::read` und `msl::write` zum Einsatz, die die Verwendung von Puffern vom Typ **`void`**`*` vereinfachen (siehe Abschn. 3.5.4.4).

Listing A.1: Definition der Klasse `EventPacket`.

```
1  static const int X1  = 0;  static const int X2  = 1;
2  static const int Y1  = 2;  static const int Y2  = 3;
3  static const int Z1  = 4;  static const int Z2  = 5;
4  static const int WG  = 6;  static const int NOP = 7;
5
6  class EventPacket: public msl::Serializable {
7
8  private:
9
10   int flag;    int noe;    double** events;
11
12   void init() {
13     events = new double*[noe];
14
15     for(int i = 0; i < noe; i++)
16       events[i] = new double[NOP];
17   }
18
19   void free() {
20     for(int i = 0; i < noe; i++)
21       delete [] events[i];
22
23     delete [] events;
24   }
25
26 public:
27
28   EventPacket(): noe(0) {
29     init();
30   }
31
32   EventPacket(int size, int flag):
33   noe(size), flag(flag) {
34     init();
35   }
36
37   ~EventPacket() {
```

```
38      free();
39    }
40
41    int getFlag() const {
42      return flag;
43    }
44
45    int getNumberOfEvents() const {
46      return noe;
47    }
48
49    double* getEvent(int idx) {
50      return events[idx];
51    }
52
53    void setEvent(int idx, Event* e) {
54      events[idx][X1] = e->x1;    events[idx][X2] = e->x2;
55      events[idx][Y1] = e->y1;    events[idx][Y2] = e->y2;
56      events[idx][Z1] = e->z1;    events[idx][Z2] = e->z2;
57      events[idx][WG] = e->weight;
58    }
59
60    void setNumberOfEvents(int count) {
61      if(count != noe) {
62        free();
63        noe = count;
64        init();
65    } }
66
67    void reduce(void* buffer, int size)
68      for(int i = 0; i < noe; i++)
69        for(int j = 0; j < NOP; j++)
70          msl::write(buffer, events[i][j], i * NOP *
71                     msl::SOD + j * msl::SOD);
72
73      msl::write(buffer, flag, noe * NOP * msl::SOD);
74    }
75
76    void expand(void* buffer, int size) {
77      setNumberOfEvents((int)((size - msl::SOD) / NOP /
78                        msl::SOD));
79
80      for(int i = 0; i < noe; i++)
81        for(int j = 0; j < NOP; j++)
```

```
82          events[i][j] = msl::read<double>(buffer,
83            i * NOP * msl::SOD + j * msl::SOD);
84
85      flag = msl::read<int>(buffer, noe * NOP * msl::SOD);
86    }
87
88    int getSize() {
89      return noe * NOP * msl::SOD + msl::SOI;
90    }
91
92  };
```

A.2 Image

Die Klasse `Image` wird bei der taskparallelen Implementierung des LM OSEM-Algorithmus verwendet und kapselt ein Bild bzw. ein Teilbild, so dass diese zwischen den `Filter`-Skeletten und dem `Final`-Skelett ausgetauscht werden können (siehe Abschn. 5.3.2). Das zu speichernde Bild wird in dem Feld `image`, die Länge bzw. Anzahl der Elemente in der Variablen `length` vorgehalten (siehe Z. 5). Analog zur Klasse `EventPacket` (siehe Anh. A.1) gibt die Variable `flag` an, ob das Ende der aktuellen Teilmenge bzw. des gesamten Rohdatensatzes erreicht wurde (siehe Z. 5). Gemäß dem von der Skelettbibliothek implementierten Serialisierungsmechanismus (siehe Abschn. 3.2.5) erbt `Image` von der Klasse `msl::Serializable` und überschreibt die Funktionen `reduce`, `expand` und `getSize` (siehe Z. 1 und Z. 68–81). Hierbei kommen die Funktionen `msl::read` und `msl::write` zum Einsatz, die die Verwendung von Puffern vom Typ **`void`**`*` vereinfachen (siehe Abschn. 3.5.4.4).

Listing A.2: Definition der Klasse `Image`.

```
1  class Image: public msl::Serializable {
2
3  private:
4
5    int flag;    int length;    double* image;
6
```

```
7    void init() {
8      image = new double[size];
9    }
10
11   void free() {
12     delete [] image;
13   }
14
15   void resize(int length) {
16     free();
17     this->length = length;
18     init();
19   }
20
21 public:
22
23   Image(): length(0), flag(0) {
24     init();
25   }
26
27   Image(int flag): length(0), flag(flag) {
28     init();
29   }
30
31   Image(int length, int flag):
32   length(length), flag(flag) {
33     init();
34   }
35
36   Image(double* image, int length, int flag):
37   length(length), flag(flag) {
38     init();
39     memcpy(this->image, image, length * SOD);
40   }
41
42   Image(const Image& img):
43   length(img.length), flag(img.flag) {
44     init();
45     memcpy(this->image, img.image, length * SOD);
46   }
47
48   ~Image() {
49     free();
50   }
```

```
51
52    int getFlag() const {
53      return flag;
54    }
55
56    int getLength() const {
57      return length;
58    }
59
60    double* getImage() const {
61      return image;
62    }
63
64    double& operator[](int index) const {
65      return image[index];
66    }
67
68    void reduce(void* buffer, int size) {
69      memcpy((double*)buffer, image, length * SOD);
70      msl::write(buffer, flag, length * msl::SOD);
71    }
72
73    void expand(void* buffer, int size) {
74      resize((size - SOI) / SOD);
75      memcpy(image, (double*)buffer, length * SOD);
76      flag = msl::read<int>(buffer, length * msl::SOD);
77    }
78
79    int getSize() {
80      return length * SOD + SOI;
81    }
82
83  };
```

A.3 Integer

Die Klasse `Integer` dient als Wrapper-Klasse für einen Wert vom Typ **`int`** (siehe Z. 5) und implementiert beispielhaft sämtliche Anforderungen an eine benutzerdefinierte Klasse, die innerhalb der verteilten Datenstruktur für dünnbesetzte Matrizen verwendet werden kann (siehe Abschn. 4.2.3):

- `Integer` erbt von `msl::Serializable` (siehe Z. 1) und überschreibt die Funktionen `reduce`, `expand` und `getSize`, um den in Abschnitt 3.2.5 beschriebenen automatisierten Serialisierungsmechanismus zu verwenden (siehe Z. 17–27). Anzumerken bleibt, dass hierbei die Funktionen `msl::read` und `msl::write` zum Einsatz kommen (siehe Abschn. 3.5.4.4).
- Die Klasse definiert einen parameterlosen Konstruktor, der beim Deserialisieren der `Integer`-Objekte benötigt wird (siehe Z. 9).
- Die Operatoren ==, != und << werden überladen, so dass diese jeweils Argumente vom Typ **`const`** `Integer&` erwarten (siehe Z. 31–42).

Listing A.3: Definition der Klasse `Integer`.

```
1  class Integer: public msl::Serializable {
2
3  private:
4
5    int val;
6
7  public:
8
9    Integer(): val(0) { }
10
11   Integer(int value): val(value) { }
12
13   int getValue() const {
14     return val;
15   }
16
17   void reduce(void* buf, int size) {
18     msl::write(buf, val, 0);
19   }
20
21   void expand(void* buf, int size) {
22     val = msl::read<int>(buf, 0);
23   }
24
25   int getSize() {
26     return msl::SOI;
27   }
```

```
28
29 };
30
31 bool operator==(const Integer& i, const Integer& j) {
32   return i.getValue() == j.getValue();
33 }
34
35 bool operator!=(const Integer& i, const Integer& j) {
36   return i.getValue() != j.getValue();
37 }
38
39 std::ostream& operator<<(std::ostream& s,
40                          const Integer& i) {
41   return s << i.getValue();
42 }
```

A.4 MatrixIndex

Die Klasse `MatrixIndex` wird in den *zip*-Skeletten verwendet und verhindert dort das wiederholte Kombinieren von Elementen (siehe Abschn. 4.3.5.5). Zu diesem Zweck speichert ein `MatrixIndex`-Objekt den Zeilen- sowie den Spaltenindex eines Matrixelements (siehe Lst. A.4, Z 5). Die Indizes werden mit Hilfe des Konstruktors initialisiert und können durch entsprechende `get`-Funktionen ausgelesen werden. Von spezieller Bedeutung ist der überladene Operator < (siehe Z. 23–33). Dieser wird benötigt, damit `MatrixIndex`-Objekte als Schlüsseltyp `T` in Mengen vom Typ `std::set<T>` verwendet werden können [213, S. 436 f.]. Der Operator definiert eine sogenannte *strikt schwache Ordnung* und sortiert `MatrixIndex`-Objekte zeilenweise von links nach rechts. Anzumerken bleibt, dass der Operator == nicht überladen werden muss, da per Definition zwei Schlüssel identisch sind, falls beide nicht kleiner als der jeweils andere sind.

Listing A.4: Definition der Klasse `MatrixIndex`.

```
1 class MatrixIndex {
2
3 private:
```

```
4
5   int ri;    int ci;
6
7  public:
8
9   MatrixIndex(int rowIndex, int columnIndex) {
10    ri = rowIndex; ci = columnIndex;
11  }
12
13  int getColumnIndex() const {
14    return ci;
15  }
16
17  int getRowIndex() const {
18    return ri;
19  }
20
21 };
22
23 bool operator<(const MatrixIndex& i,
24                const MatrixIndex& j) {
25   if(i.getRowIndex() < j.getRowIndex())
26     return true;
27   else
28     if(i.getRowIndex() == j.getRowIndex() &&
29     i.getColumnIndex() < j.getColumnIndex())
30       return true;
31     else
32       return false;
33 }
```

Literaturverzeichnis

[1] 14882:1998, ISO/IEC: *Programming languages – C++*. ISO/IEC, 1998 (Zitiert auf den Seiten 80, 96, 102, 117, 121, 122, 130, 131, 132, 133, 139, 143 und 159)

[2] 9899:201x, ISO/IEC: *Programming languages – C*. ISO/IEC, 2010 (Zitiert auf den Seiten 216 und 219)

[3] Akon, Mohammad M. ; Goswami, Dhrubajyoti ; Li, Hon F.: A Model for Designing and Implementing Parallel Applications Using Extensible Architectural Skeletons. In: *Parallel Computing Technologies* Bd. 3606. Springer, 2005, S. 367–380 (Zitiert auf Seite 62)

[4] Akon, Mohammad M. ; Goswami, Dhrubajyoti ; Li, Hon F.: Super-PAS: A Parallel Architectural Skeleton Model Supporting Extensibility and Skeleton Composition. In: *Parallel and Distributed Processing and Applications* Bd. 3358. Springer, 2005, S. 985–996 (Zitiert auf Seite 62)

[5] Akon, Mohammad M. ; Singh, Ajit ; Goswami, Dhrubajyoti ; Li, Hon F.: Extensible Parallel Architectural Skeletons. In: *High Performance Computing - HiPC 2005* Bd. 3769. Springer, 2005, S. 290–301 (Zitiert auf Seite 62)

[6] Akon, Mohammad M. ; Singh, Ajit ; Shen, Xuemin ; Goswami, Dhrubajyoti ; Li, Hon F.: Developing High-Performance Parallel Applications Using EPAS. In: *Parallel and Distributed Processing and Applications* Bd. 3758. Springer, 2005, S. 431–441 (Zitiert auf Seite 62)

[7] Alba, Enrique ; Almeida, Francisco ; Blesa, Maria J. ; Cabeza, J. ; Cotta, Carlos ; Díaz, Manuel ; Dorta, Isabel ; Gabarró,

Joaquim ; León, Coromoto ; Luna, J. ; Moreno, Luz M. ; Pablos, C. ; Petit, Jordi ; Rojas, Angélica ; Xhafa, Fatos: MALLBA: A Library of Skeletons for Combinatorial Optimisation. In: *Euro-Par 2002 Parallel Processing* Bd. 2400. Springer, 2002, S. 927–932 (Zitiert auf Seite 59)

[8] Alba, Enrique ; Almeida, Francisco ; Blesa, Maria J. ; Cotta, Carlos ; Díaz, Manuel ; Dorta, Isabel ; Gabarró, Joaquim ; León, Coromoto ; Luque, Gabriel ; Petit, Jordi ; Rodríguez, Casiano ; Rojas, Angélica ; Xhafa, Fatos: Efficient Parallel LAN/WAN Algorithms for Optimization. The MALLBA Project. In: *Parallel Computing* 32(5–6) (2006), S. 415–440 (Zitiert auf Seite 59)

[9] Alba, Enrique ; Luque, Gabriel ; García-Nieto, José ; Ordóñez, Guillermo ; Leguizamón, Guillermo: MALLBA: A Software Library to Design Efficient Optimisation Algorithms. In: *International Journal of Innovative Computing and Applications* 1(1) (2007), S. 74–85 (Zitiert auf Seite 59)

[10] Aldinucci, Marco: *Dynamic shared data in structured parallel programming frameworks*, Università di Pisa, Diss., 2003 (Zitiert auf Seite 59)

[11] Aldinucci, Marco ; Bertolli, Carlo ; Campa, Sonia ; Coppola, Massimo ; Vanneschi, Marco ; Veraldi, Luca ; Zoccolo, Corrado: Self-configuring and self-optimizing grid components in the GCM model and their ASSIST implementation. In: *Proceedings of PHC-GECO/CompFrame*, 2006, S. 45–52 (Zitiert auf Seite 50)

[12] Aldinucci, Marco ; Bracciali, Andrea ; Lio', Pietro ; Sorathiya, Anil ; Torquati, Massimo: StochKit-FF: Efficient Systems Biology on Multicore Architectures / Università di Pisa. 2010 (TR-10-12). – Forschungsbericht (Zitiert auf Seite 55)

[13] Aldinucci, Marco ; Campa, Sonia ; Ciullo, Pierpaolo ; Coppola, Massimo ; Danelutto, Marco ; Pesciullesi, Paolo ; Ravazzolo, Roberto ; Torquati, Massimo ; Vanneschi, Marco ; Zoccolo, Corrado: A framework for experimenting with structured

parallel programming environment design. In: *Advances in Parallel Computing* Bd. 13. Elsevier, 2003, S. 617–624 (Zitiert auf Seite 50)

[14] Aldinucci, Marco ; Campa, Sonia ; Ciullo, Pierpaolo ; Coppola, Massimo ; Danelutto, Marco ; Pesciullesi, Paolo ; Ravazzolo, Roberto ; Torquati, Massimo ; Vanneschi, Marco ; Zoccolo, Corrado: ASSIST Demo: A High Level, High Performance, Portable, Structured Parallel Programming Environment at Work. In: *Euro-Par 2003 Parallel Processing* Bd. 2790. Springer, 2004, S. 1295–1300 (Zitiert auf Seite 50)

[15] Aldinucci, Marco ; Campa, Sonia ; Ciullo, Pierpaolo ; Coppola, Massimo ; Magini, Silvia ; Pesciullesi, Paolo ; Potiti, Laura ; Ravazzolo, Roberto ; Torquati, Massimo ; Vanneschi, Marco ; Zoccolo, Corrado: The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In: *Euro-Par 2003 Parallel Processing* Bd. 2790. Springer, 2004, S. 712–721 (Zitiert auf Seite 50)

[16] Aldinucci, Marco ; Campa, Sonia ; Coppola, Massimo ; Magini, Silvia ; Pesciullesi, Paolo ; Potiti, Laura ; Ravazzolo, Roberto ; Torquati, Massimo ; Zoccolo, Corrado: Targeting Heterogeneous Architectures in ASSIST: Experimental Results. In: *Euro-Par 2004 Parallel Processing* Bd. 3149. Springer, 2004, S. 638–643 (Zitiert auf Seite 50)

[17] Aldinucci, Marco ; Coppola, Massimo ; Danelutto, Marco ; Tonellotto, Nicola ; Vanneschi, Marco ; Zoccolo, Corrado: High level Grid programming with ASSIST. In: *Computational Methods in Science and Technology* 12(1) (2006), S. 21–32 (Zitiert auf Seite 50)

[18] Aldinucci, Marco ; Danelutto, Marco: Stream Parallel Skeleton Optimization. In: *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, 1999, S. 955–962 (Zitiert auf Seite 59)

[19] Aldinucci, Marco ; Danelutto, Marco: Skeleton-based parallel programming: Functional and parallel semantics in a single shot. In:

Computer Languages, Systems and Structures 33(3-4) (2007), S. 179–192 (Zitiert auf Seite 59)

[20] Aldinucci, Marco ; Danelutto, Marco: The cost of security in skeletal systems. In: *Proceedings fo the 15th International Conference on Parallel, Distributed and Network-Based Processing*, 2007, S. 213–220 (Zitiert auf Seite 60)

[21] Aldinucci, Marco ; Danelutto, Marco: Securing skeletal systems with limited performance penalty: The muskel experience. In: *Journal of Systems Architecture* 54(9) (2008), S. 868–876 (Zitiert auf Seite 60)

[22] Aldinucci, Marco ; Danelutto, Marco ; Dazzi, Patrizio: Muskel: A Skeleton Library Supporting Skeleton Set Expandability. In: *Scalable Computing: Practice and Experience* 8(4) (2007), S. 325–341 (Zitiert auf Seite 60)

[23] Aldinucci, Marco ; Danelutto, Marco ; Dünnweber, Jan: Optimization Techniques for Implementing Parallel Skeletons in Grid Environments. In: *Proceedings of the 4th International Workshop on Constructive Methods for Parallel Programming*, 2004, S. 35–47 (Zitiert auf Seite 59)

[24] Aldinucci, Marco ; Danelutto, Marco ; Kilpatrick, Peter ; Meneghin, Massimiliano ; Torquati, Massimo: Accelerating sequential programs using FastFlow and self-offloading / Università di Pisa. 2010 (TR-10-03). – Forschungsbericht (Zitiert auf Seite 55)

[25] Aldinucci, Marco ; Danelutto, Marco ; Meneghin, Massimiliano ; Torquati, Massimo ; Kilpatrick, Peter: Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In: *Parallel Computing: From Multicores and GPU's to Petascale* Bd. 19. IOS Press, 2010, S. 273–280 (Zitiert auf Seite 55)

[26] Aldinucci, Marco ; Danelutto, Marco ; Teti, Paolo: An advanced environment supporting structured parallel programming in Java. In: *Future Generation Computer Systems* 19(5) (2003), S. 611–626 (Zitiert auf Seite 59)

[27] Aldinucci, Marco ; Meneghin, Massimiliano ; Torquati, Massimo: Efficient Smith-Waterman on Multi-core with FastFlow. In: *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, S. 195–199 (Zitiert auf Seite 55)

[28] Aldinucci, Marco ; Torquati, Massimo ; Meneghin, Massimiliano: FastFlow: Efficient Parallel Streaming Applications on Multi-core / Università di Pisa. 2009 (TR-09-12). – Forschungsbericht (Zitiert auf Seite 55)

[29] Aldinucci, Marco ; Torquati, Salvatore Ruggieri M.: Porting Decision Tree Algorithms to Multicore using FastFlow / Università di Pisa. 2010 (TR-10-11). – Forschungsbericht (Zitiert auf Seite 55)

[30] Alexandrescu, Andrei: *Modernes C++ Design. Generische Programmierung und Entwurfsmuster angewendet.* mitp, 2003 (Zitiert auf Seite 84)

[31] Alonso, José M. ; Hernández, Vicente ; Moltó, Germán ; Proença, Alberto J. ; Sobral, João Luís F.: Grid Enabled JaSkel Skeletons with GMarte. In: *Proceedings of the 1st Iberian Grid Infrastructure Conference*, 2007, S. 301–312 (Zitiert auf Seite 57)

[32] Alt, Martin ; Dünnweber, Jan ; Müller, Jens ; Gorlatch, Sergei: HOCs: Higher-Order Components for Grids. In: *Component Models and Systems for Grid Applications.* Springer US, 2005, S. 157–166 (Zitiert auf Seite 56)

[33] AMD: *ATI Stream SDK v2.1 with OpenCL 1.0 Support.* `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`, 21. Juli 2010 (Zitiert auf Seite 220)

[34] Anvik, John ; Macdonald, Steve ; Szafron, Duane ; Schaeffer, Jonathan ; Bromling, Steven ; Tan, Kai: Generating Parallel Programs from the Wavefront Design Pattern. In: *Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Society Press, 2002, S. 1–8 (Zitiert auf Seite 51)

[35] Bacci, Bruno ; Cantalupo, B. ; Danelutto, Marco ; Orlando, Salvatore ; Pasetto, D. ; Pelagatti, Susanna ; Vanneschi, Marco: An Environment for Structured Parallel Programming. In: *Advances in High Performance Computing*. Kluwier Academic Publishers, 1997, S. 219–234 (Zitiert auf Seite 61)

[36] Bacci, Bruno ; Danelutto, Marco ; Orlando, Salvatore ; Pelagatti, Susanna ; Vanneschi, Marco: P^{3}L: a Structured High-level Programming Language, and its Structured Support. In: *Concurrency: Practice and Experience* Bd. 7, 1995, S. 225–255 (Zitiert auf Seite 61)

[37] Bacci, Bruno ; Danelutto, Marco ; Pelagatti, Susanna: Resource Optimization Via Structured Parallel Programming. In: *Programming Environments for Massively Parallel Distributed Systems*, 1993, S. 13–25 (Zitiert auf Seite 61)

[38] Bacci, Bruno ; Danelutto, Marco ; Pelagatti, Susanna ; Vanneschi, Marco: SkIE: A heterogeneous environment for HPC applications. In: *Parallel Computing* 25(13-14) (1999), S. 1827–1852 (Zitiert auf Seite 68)

[39] Baker, Mark ; Carpenter, Bryan: MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing. In: *Parallel and Distributed Processing* Bd. 1800. Springer, 2000, S. 552–559 (Zitiert auf Seite 71)

[40] Baker, Mark ; Carpenter, Bryan ; Fox, Geoffrey ; Ko, Sung H. ; Lim, Sang: mpiJava: An Object-Oriented Java Interface to MPI. In: *Advances in Parallel Computing* Bd. 1586. Springer, 1999, S. 748–762 (Zitiert auf Seite 71)

[41] Baker, Mark ; Carpenter, Bryan ; Shafi, A.: MPJ Express: Towards Thread Safe Java HPC. In: *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006, S. 1–10 (Zitiert auf den Seiten 71 und 221)

[42] Balzert, Helmut: *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodel-*

lierung. Spektrum Akademischer Verlag, 1998 (Lehrbücher der Informatik) (Zitiert auf Seite 1)

[43] Bellman, Richard: On a routing problem. In: *Quarterly of Applied Mathematics* 16 (1958), S. 87–90 (Zitiert auf Seite 162)

[44] Benoit, Anne ; Cole, Murray: Two Fundamental Concepts in Skeletal Parallel Programming. In: *Computational Science - ICCS 2005* Bd. 3515. Springer, 2005, S. 764–771 (Zitiert auf Seite 54)

[45] Benoit, Anne ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: Evaluating the Performance of Skeleton-Based High Level Parallel Programs. In: *Computational Science - ICCS 2004* Bd. 3038. Springer, 2004, S. 289–296 (Zitiert auf Seite 54)

[46] Benoit, Anne ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: Evaluating the performance of pipeline-structured parallel programs with skeletons and process algebra. In: *Scalable Computing: Practice and Experience* 6(4) (2005), S. 1–16 (Zitiert auf Seite 54)

[47] Benoit, Anne ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: Flexible Skeletal Programming with eSkel. In: *Euro-Par 2005 Parallel Processing* Bd. 3648. Springer, 2005, S. 761–770 (Zitiert auf Seite 54)

[48] Benoit, Anne ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: Scheduling Skeleton-Based Grid Applications Using PEPA and NWS. In: *The Computer Journal* 48(3) (2005), S. 369–378 (Zitiert auf Seite 54)

[49] Benoit, Anne ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: Using eSkel to Implement the Multiple Baseline Stereo Application. In: *Parallel Computing: Current & Future Issues of High-End Computing*. Central Institute for Applied Mathematics, 2005 (John von Neumann Institute for Computing Series), S. 673–680 (Zitiert auf Seite 54)

[50] Benoit, Anne ; Robert, Yves: Mapping Pipeline Skeletons onto Heterogeneous Platforms. In: *Computational Science - ICCS 2007* Bd. 4487. Springer, 2007, S. 591–598 (Zitiert auf Seite 54)

[51] Berthold, Jost: Towards a Generalised Runtime Environment for Parallel Haskells. In: *Computational Science - ICCS 2004* Bd. 3038. Springer, 2004, S. 297–305 (Zitiert auf Seite 53)

[52] Berthold, Jost: Dynamic Chunking in Eden. In: *Implementation of Functional Languages* Bd. 3145. Springer, 2005, S. 102–117 (Zitiert auf Seite 53)

[53] Berthold, Jost: *Explicit and Implicit Parallel Functional Programming: Concepts and Implementation*, Philipps-Universität Marburg, Diss., 2008 (Zitiert auf Seite 53)

[54] Berthold, Jost ; Dieterle, Mischa ; Lobachev, Oleg ; Loogen, Rita: Distributed Memory Programming on Many-Cores - A Case Study Using Eden Divide-&-Conquer Skeletons. In: *Proceedings of the 22th International Conference on Architecture of Computing Systems*, 2009, S. 47–56 (Zitiert auf Seite 53)

[55] Berthold, Jost ; Dieterle, Mischa ; Lobachev, Oleg ; Loogen, Rita: Parallel FFT with Eden Skeletons. In: *Parallel Computing Technologies* Bd. 5698. Springer, 2009, S. 73–83 (Zitiert auf Seite 53)

[56] Berthold, Jost ; Dieterle, Mischa ; Loogen, Rita: Implementing Parallel Google Map-Reduce in Eden. In: *Euro-Par 2009 Parallel Processing* Bd. 5704. Springer, 2009, S. 990–1002 (Zitiert auf Seite 53)

[57] Berthold, Jost ; Dieterle, Mischa ; Loogen, Rita ; Priebe, Steffen: Hierarchical Master-Worker Skeletons. In: *Practical Aspects of Declarative Languages* Bd. 4902. Springer, 2008, S. 248–264 (Zitiert auf Seite 53)

[58] Berthold, Jost ; Klusik, Ulrike ; Loogen, Rita ; Priebe, Steffen ; Weskamp, Nils: High-Level Process Control in Eden. In: *Euro-Par 2003 Parallel Processing* Bd. 2790. Springer, 2004, S. 732–741 (Zitiert auf Seite 53)

[59] Berthold, Jost ; Loogen, Rita: Parallel Coordination Made Explicit in a Functional Setting. In: *Implementation and Application*

of Functional Languages Bd. 4449. Springer, 2007, S. 73–90 (Zitiert auf Seite 53)

[60] Berthold, Jost ; Loogen, Rita: Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In: *Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, 2007, S. 121–128 (Zitiert auf Seite 53)

[61] Berthold, Jost ; Loogen, Rita: The Impact of Dynamic Channels on Functional Topology Skeletons. In: *Parallel Processing Letters* 18(1) (2008), S. 101–116 (Zitiert auf Seite 53)

[62] Berthold, Jost ; Weskamp, Nils: The Eden Porting Project - Porting the Eden Runtime-System from GHC 2.10 to GHC 5.00.2 / Philipps-Universität Marburg. 2002. – Forschungsbericht (Zitiert auf Seite 53)

[63] Bird, Richard S.: An Introduction to the Theory of Lists. In: *Proceedings of the NATO Advanced Study Institute on Logic Programming and Calculi of Discrete Design*, 1987, S. 5–42 (Zitiert auf Seite 67)

[64] Bischof, Holger: *Systematic Development of Parallel Programs Using Skeletons*, Westfälische Wilhelms-Universität Münster, Diss., 2005 (Zitiert auf Seite 52)

[65] Bischof, Holger ; Gorlatch, Sergei ; Leshchinskiy, Roman: DatTeL: A Data-Parallel C++ Template Library. In: *Parallel Processing Letters* 13(3) (2003), S. 461–472 (Zitiert auf Seite 52)

[66] Bischof, Holger ; Gorlatch, Sergei ; Leshchinskiy, Roman: Generic Parallel Programming Using C++ Templates and Skeletons. In: *Domain-Specific Program Generation* Bd. 3016. Springer, 2004, S. 107–126 (Zitiert auf Seite 52)

[67] Boehm, Hans-Jürgen ; Adve, Sarita V.: Foundations of the C++ Concurrency Memory Model. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, S. 68–78 (Zitiert auf den Seiten 217 und 219)

[68] Botorog, George H. ; Kuchen, Herbert: Efficient parallel programming with algorithmic skeletons. In: *Euro-Par'96 Parallel Processing* Bd. 1123. Springer, 1996, S. 718–731 (Zitiert auf Seite 69)

[69] Botorog, George H. ; Kuchen, Herbert: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In: *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, 1996, S. 243–252 (Zitiert auf Seite 69)

[70] Botorog, George H. ; Kuchen, Herbert: Using algorithmic skeletons with dynamic data structures. In: *Parallel Algorithms for Irregularly Structured Problems* Bd. 1117. Springer, 1996, S. 263–276 (Zitiert auf Seite 69)

[71] Botorog, George H. ; Kuchen, Herbert: Efficient High-Level Parallel Programming. In: *Theoretical Computer Science* 196(1-2) (1998), S. 71–107 (Zitiert auf Seite 69)

[72] Bowles, Richard ; King, David ; Held, Jim ; Douglas, Stuart ; Lacey, Marian: Addressing the Challenges of Tera-scale Computing / Intel Corporation. 2009 (13(4)). – Forschungsbericht (Zitiert auf Seite 216)

[73] Breitinger, S. ; Loogen, R. ; Ortega-Mallén, Y. ; Peña, R.: Eden: Language Definition and Operational Semantics / Philipps-Universität Marburg. 1996 (10). – Forschungsbericht (Zitiert auf Seite 53)

[74] Breitinger, Silvia: *Design and Implementation of the Parallel Functional Language Eden*, Philipps-Universität Marburg, Diss., 1998 (Zitiert auf Seite 53)

[75] Breitinger, Silvia ; Klusik, Ulrike ; Loogen, Rita: Implementation of Functional Languages. In: *An Implementation of Eden on Top of Concurrent Haskell* Bd. 1268. Springer, 1997, S. 141–161 (Zitiert auf Seite 53)

[76] Breitinger, Silvia ; Klusik, Ulrike ; Loogen, Rita: From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of

View. In: *Principles of Declarative Programming* Bd. 1490. Springer, 1998, S. 318–334 (Zitiert auf Seite 53)

[77] Breitinger, Silvia ; Klusik, Ulrike ; Loogen, Rita ; Ortega-Mallén, Yolanda ; Peña, Ricardo: DREAM: The DistRibuted Eden Abstract Machine. In: *Implementation of Functional Languages* Bd. 1467. Springer, 1998, S. 250–269 (Zitiert auf Seite 53)

[78] Breitinger, Silvia ; Loogen, Rita ; Ortega-Mallén, Yolanda ; Peña-Marí, Ricardo: Eden — The Paradise of Functional Concurrent Programming. In: *Euro-Par'96 Parallel Processing* Bd. 1123. Springer, 1996, S. 710–713 (Zitiert auf Seite 53)

[79] Brodtkorb, André ; Dyken, Erik C. ; Hagen, Trond R. ; Hjelmervik, Jon M. ; Storaasli, Olaf O.: State-of-the-art in heterogeneous computing. In: *Scientific Programming* 18(1) (2010), S. 1–33 (Zitiert auf Seite 219)

[80] Butenhof, David: *Programming with POSIX Threads*. Addison-Wesley, 1997 (Addison-Wesley Professional Computing) (Zitiert auf Seite 36)

[81] Caromel, Denis ; Delbé, Christian ; Constanzo, Alexandre di ; Leyton, Mario: ProActive: an Integrated platform for programming and running applications on Grids and P2P systems. In: *Computational Methods in Science and Technology* 12(1) (2006), S. 69–77 (Zitiert auf Seite 51)

[82] Caromel, Denis ; Henrio, Ludovic ; Leyton, Mario: Type Safe Algorithmic Skeletons. In: *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, IEEE Computer Society, 2008, S. 45–53 (Zitiert auf Seite 51)

[83] Caromel, Denis ; Leyton, Mario: Fine Tuning Algorithmic Skeletons. In: *Euro-Par 2007 Parallel Processing* Bd. 4641. Springer, 2007, S. 72–81 (Zitiert auf Seite 51)

[84] Caromel, Denis ; Leyton, Mario: A transparent non-invasive file data model for algorithmic skeletons. In: *Proceedings of the 22nd*

IEEE International Symposium on Parallel and Distributed Processing, IEEE Computer Society, 2008, S. 1–10 (Zitiert auf Seite 51)

[85] Caromel, Denis ; Leyton, Mario: ProActive Parallel Suite: From Active Objects-Skeletons-Components to Environment and Deployment. In: *Euro-Par 2008 Workshops - Parallel Processing* Bd. 5415. Springer, 2009, S. 423–437 (Zitiert auf Seite 51)

[86] Carpenter, Bryan ; Getov, Vladimir ; Judd, Glenn ; Skjellum, Tony ; Fox, Geoffrey: MPJ: MPI-like Message Passing for Java. In: *Concurrency: Practice and Experience* 12(11) (2000), S. 1019–1038 (Zitiert auf Seite 71)

[87] Carpenter, Gail A.: Default ARTMAP. In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2003, S. 1396–1401 (Zitiert auf Seite 199)

[88] Carpenter, Gail A. ; Grossberg, Stephen: A massively parallel architecture for a self-organizing neural recognition machine. In: *Computer Vision, Graphics, And Image Processing* (1987), Nr. 37, S. 54–115 (Zitiert auf den Seiten 194 und 199)

[89] Carpenter, Gail A. ; Grossberg, Stephen: ART 2: Self-organization of stable category recognition codes for analog input patterns. In: *Applied Optics* (1987), Nr. 26(23), S. 4919–4930 (Zitiert auf Seite 199)

[90] Carpenter, Gail A. ; Grossberg, Stephen: ART 3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures. In: *Neural Networks* (1990), Nr. 3, S. 129–152 (Zitiert auf Seite 199)

[91] Carpenter, Gail A. ; Grossberg, Stephen ; Markuzon, Natalya ; Reynolds, John H. ; Rosen, David B.: Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps. In: *IEEE Transactions on Neural Networks* (1992), Nr. 3(5), S. 698–713 (Zitiert auf Seite 199)

[92] Carpenter, Gail A. ; Grossberg, Stephen ; Reynolds, John H.: ARTMAP: Supervised real-time learning and classification

of nonstationary data by a self-organizing neural network. In: *Neural Networks* (1991), Nr. 4, S. 565–588 (Zitiert auf Seite 199)

[93] Carpenter, Gail A. ; Grossberg, Stephen ; Rosen, David B.: ART 2-A: An Adaptive Resonance Algorithm for Rapid Category Learning and Recognition. In: *Neural Networks* (1991), Nr. 4, S. 493–504 (Zitiert auf Seite 199)

[94] Carpenter, Gail A. ; Grossberg, Stephen ; Rosen, David B.: Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. In: *Neural Networks* (1991), Nr. 4, S. 759–771 (Zitiert auf Seite 199)

[95] Carpenter, Gail A. ; Markuzon, Natalya: ARTMAP-IC and medical diagnosis: Instance counting and inconsistent cases. In: *Neural Networks* (1998), Nr. 11, S. 323–336 (Zitiert auf Seite 199)

[96] Carpenter, Gail A. ; Milenova, Boriana L. ; Noeske, Benjamin W.: Distributed ARTMAP: a Neural network for fast distributed supervised learning. In: *Neural Networks* (1998), Nr. 11, S. 793–813 (Zitiert auf Seite 199)

[97] Ceccolini, A. ; Danelutto, Marco ; Orsini, G. ; Pelagatti, Susanna: A Tool for the Development of Structured Parallel Applications. In: *High-Performance Computing and Networking* Bd. 1067. Springer, 1996, S. 485–492 (Zitiert auf Seite 61)

[98] Chapman, Barbara ; Jost, Gabriele ; Pas, Ruud van d.: *Using OpenMP - Portable Shared Memory Parallel Programming.* MIT Press, 2008 (Scientific and Engineering Computation) (Zitiert auf den Seiten 34, 36, 38, 148 und 154)

[99] Ciechanowicz, Philipp: Algorithmic Skeletons for General Sparse Matrices on Multi-Core Processors. In: *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 2008, S. 188–197 (Zitiert auf den Seiten 3, 75, 87 und 111)

[100] Ciechanowicz, Philipp ; Dlugosz, Stephan ; Kuchen, Herbert ; Müller-Funk, Ulrich: Exploiting Training Example Parallelism

With a Batch Variant of the ART 2 Classification Algorithm. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2008, S. 195–201 (Zitiert auf den Seiten 3, 75 und 194)

[101] Ciechanowicz, Philipp ; Kegel, Philipp ; Schellmann, Maraike ; Gorlatch, Sergei ; Kuchen, Herbert: Parallelizing the LM OSEM Image Reconstruction on Multi-Core Clusters. In: *Parallel Computing: From Multicores and GPU's to Petascale* Bd. 19. IOS Press, 2010, S. 169–176 (Zitiert auf den Seiten 3, 75, 168, 173, 183 und 185)

[102] Ciechanowicz, Philipp ; Kuchen, Herbert: Enhancing Muesli's Data Parallel Skeletons for Multi-Core Computer Architectures. In: *Proceedings of The 12th IEEE International Conference on High Performance Computing and Communication (HPCC)*, IEEE Computer Society Press, 2010, S. X–X (Zitiert auf den Seiten 3, 75 und 76)

[103] Ciechanowicz, Philipp ; Poldner, Michael ; Kuchen, Herbert: The Münster Skeleton Library Muesli - A Comprehensive Overview / Westfälische Wilhelms-Universität Münster. 2009. – Forschungsbericht (Zitiert auf den Seiten 3, 75, 76, 84, 85, 86, 175, 176 und 177)

[104] Cole, Murray: *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989 (Zitiert auf den Seiten 2 und 222)

[105] Cole, Murray: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. In: *Parallel Computing* 30(3) (2004), S. 389–406 (Zitiert auf Seite 54)

[106] Comeau Computing: *Comeau C/C++ Compiler 4.3.3*. `http://www.comeaucomputing.com`, 22. Januar 2010 (Zitiert auf Seite 103)

[107] Coudarcher, Rémi ; Sérot, Jecelyn ; Dérutin, Jean-Pierre: Implementation of a Skeleton-Based Parallel Programming Environment Supporting Arbitrary Nesting. In: *High-Level Parallel Programming*

Models and Supportive Environments Bd. 2026. Springer, 2001, S. 71–84 (Zitiert auf Seite 70)

[108] Danelutto, Marco: Dynamic Run Time Support for Skeletons. In: *Proceedings of the International Conference on Parallel Computing*, 1999, S. 460–467 (Zitiert auf Seite 59)

[109] Danelutto, Marco: Task Farm Computations in Java. In: *High Performance Computing and Networking* Bd. 1823. Springer, 2000, S. 385–394 (Zitiert auf Seite 59)

[110] Danelutto, Marco: Efficient support for skeletons on workstation clusters. In: *Parallel Processing Letters* 11(1) (2001), S. 41–56 (Zitiert auf Seite 59)

[111] Danelutto, Marco: QoS in parallel programming through application managers. In: *Proceedings fo the 13th International Conference on Parallel, Distributed and Network-Based Processing*, 2005, S. 282–289 (Zitiert auf Seite 60)

[112] Danelutto, Marco ; Cosmo, Roberto D. ; Leroy, Xavier ; Pelagatti, Susanna: Parallel Functional Programming with Skeletons: the ocamlp3l experiment. In: *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998, S. 31–39 (Zitiert auf Seite 61)

[113] Danelutto, Marco ; Dazzi, Patrizio: Joint Structured/Unstructured Parallelism Exploitation in muskel. In: *Computational Science - ICCS 2006*. Springer, 2006, S. 937–944 (Zitiert auf Seite 60)

[114] Danelutto, Marco ; Meglio, Robert D. ; Orlando, Salvatore ; Pelagatti, Susanna ; Vanneschi, Marco: A Methodology for the Development and Support of Massively Parallel Programs. In: *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1994, S. 319–334 (Zitiert auf Seite 61)

[115] Danelutto, Marco ; Pasqualetti, Fabrizio ; Pelagatti, Susanna: Skeletons for Data Parallelism in P3L. In: *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing*, 1997, S. 619–628 (Zitiert auf Seite 61)

[116] Danelutto, Marco ; Stigliani, Massimiliano: SKElib: Parallel Programming with Skeletons in C. In: *Euro-Par 2000 Parallel Processing* Bd. 1900. Springer, 2000, S. 1175–1184 (Zitiert auf Seite 66)

[117] Danelutto, Marco ; Teti, Paolo: Lithium: A Structured Parallel Programming Environment in Java. In: *Computational Science - ICCS 2002* Bd. 2330. Springer, 2002, S. 844–853 (Zitiert auf Seite 59)

[118] Darlington, John: Structured parallel programming: a generic coordination model and a parallel Fortran. In: *19th CERN School of Computing* (1996), S. 195–201 (Zitiert auf Seite 64)

[119] Darlington, John: Structured parallel programming: parallel abstract data types. In: *19th CERN School of Computing* (1996), S. 203–210 (Zitiert auf Seite 64)

[120] Darlington, John ; Field, Anthony J. ; Harrison, Peter G. ; Kelly, Paul H. J. ; Sharp, David W. N. ; Wu, Qiang ; While, Lyndon: Parallel Programming Using Skeleton Functions. In: *PARLE '93 Parallel Architectures and Languages Europe* Bd. 694. Springer, 1993, S. 146–160 (Zitiert auf Seite 64)

[121] Darlington, John ; Guo, Yi ke ; To, Hing W. ; Yang, Jin: Functional Skeletons for Parallel Coordination. In: *EURO-PAR '95 Parallel Processing* Bd. 966. Springer, 1995, S. 55–66 (Zitiert auf Seite 64)

[122] Darlington, John ; Guo, Yi ke ; To, Hing W. ; Yang, Jin: Parallel Skeletons for Structured Composition. In: *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, S. 19–28 (Zitiert auf Seite 64)

[123] Díaz, Manuel ; Romero, Sergio ; Rubio, Bartholomé ; Soler, Enrique ; Troya, José M.: An Aspect Oriented Framework for Scientific Component Development. In: *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005, S. 290–296 (Zitiert auf Seite 63)

[124] Díaz, Manuel ; Romero, Sergio ; Rubio, Bartholomé ; Soler, Enrique ; Troya, José M.: Dynamic Reconfiguration of Scientific

Components Using Aspect Oriented Programming: A Case Study. In: *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE* Bd. 4276. Springer, 2006, S. 1351–1360 (Zitiert auf Seite 63)

[125] Díaz, Manuel ; Romero, Sergio ; Rubio, Bartholomé ; Soler, Enrique ; Troya, José M.: Using SBASCO to Solve Reaction-Diffusion Equations in Two-Dimensional Irregular Domains. In: *Computational Science - ICCS 2006* Bd. 3992. Springer, 2006, S. 912–919 (Zitiert auf Seite 63)

[126] Díaz, Manuel ; Romero, Sergio ; Rubio, Bartholomé ; Soler, Enrique ; Troya, José M.: Adding Aspect-Oriented Concepts to the High-Performance Component Model of SBASCO. In: *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, S. 21–27 (Zitiert auf Seite 63)

[127] Díaz, Manuel ; Rubio, Bartholomé ; Soler, Enrique ; Troya, José M.: SBASCO: Skeleton-based Scientific Components. In: *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2004, S. 318–324 (Zitiert auf Seite 63)

[128] Dieterle, Mischa ; Horstmeyer, Thomas ; Loogen, Rita: Skeleton Composition Using Remote Data. In: *Practical Aspects of Declarative Languages* Bd. 5937. Springer, 2010, S. 73–87 (Zitiert auf Seite 53)

[129] Dijkstra, Edsger: A Note on Two Problems in Connexion with Graphs. In: *Numerische Mathematik* 1 (1959), S. 269–271 (Zitiert auf Seite 162)

[130] Dünnweber, Jan: *Higher-Order Components for Web-Enabled Grid Applications*, Westfälische Wilhelms-Universität Münster, Diss., 2008 (Zitiert auf Seite 56)

[131] Dünnweber, Jan ; Gorlatch, Sergei: HOC-SA: A Grid Service Architecture for Higher-Order Components. In: *Proceedings of*

the 2004 IEEE International Conference on Services Computing, 2004, S. 288–294 (Zitiert auf Seite 56)

[132] Dünnweber, Jan ; Gorlatch, Sergei: Component-Based Grid Programming Using the HOC-Service Architecture. In: *Proceedings of the 4th International Conference on New Trends in Software Methodologies, Tools and Techniques*, 2005, S. 311–329 (Zitiert auf Seite 56)

[133] Dünnweber, Jan ; Gorlatch, Sergei: *Higher-Order Components for Grid Programming - Making Grids More Usable.* Springer, 2009 (Software Patterns) (Zitiert auf Seite 56)

[134] Dünnweber, Jan ; Gorlatch, Sergei ; Aldinucci, Marco ; Campa, Sonia ; Danelutto, Marco: Adaptable Parallel Components for Grid Programming. In: *Integrated Research in GRID Computing.* Springer US, 2007, S. 43–57 (Zitiert auf Seite 56)

[135] Dumitrescu, Catalin L. ; Dünnweber, Jan ; Lüdeking, Philipp ; Gorlatch, Sergei ; Raicu, Ioan ; Foster, Ian T.: Simplifying Grid Application Programming Using Web-Enabled Code Transfer Tools. In: *Towards Next Generation Grids - Proceedings of the CoreGRID Symposium 2007.* Springer, 2007, S. 225–235 (Zitiert auf Seite 56)

[136] Eisenstat, Stanley C. ; Gursky, M. C. ; Schultz, Martin H. ; Sherman, Andrew H.: Yale Sparse Matrix Package. II. The Nonsymmetric Codes / Department of Computer Science, Yale University. 1977. – Forschungsbericht (Zitiert auf den Seiten 127 und 128)

[137] Eisenstat, Stanley C. ; Gursky, M. C. ; Schultz, Martin H. ; Sherman, Andrew H.: Yale Sparse Matrix Package. I. The Symmetric Codes. In: *International Journal of Numerical Methods in Engineering* (1982), Nr. 18, S. 1145–1151 (Zitiert auf den Seiten 127 und 128)

[138] Emoto, Kento ; Hu, Zhenjiang ; Kakehi, Kazuhiko ; Takeichi, Masato: A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays. In: *International Journal of Parallel Programming* 35(6) (2007), S. 615–658 (Zitiert auf Seite 67)

[139] Emoto, Kento ; Matsuzaki, Kiminori ; Hu, Zhenjiang ; Takeichi, Masato: Surrounding Theorem: Developing Parallel Programs for Matrix-Convolutions. In: *Euro-Par 2006 Parallel Processing* Bd. 4128. Springer, 2006, S. 605–614 (Zitiert auf Seite 67)

[140] Emoto, Kento ; Matsuzaki, Kiminori ; Hu, Zhenjiang ; Takeichi, Masato: Domain-Specific Optimization Strategy for Skeleton Programs. In: *Euro-Par 2007 Parallel Processing* Bd. 4641. Springer, 2007, S. 705–714 (Zitiert auf Seite 67)

[141] Engel, Wolfgang: *GPU Pro*. A K Peters, 2010 (Zitiert auf Seite 219)

[142] Ernsting, Steffen: *Datenparallele Skelette in Java*, Westfälische Wilhelms-Universität Münster, Diplomarbeit, 2010 (Zitiert auf Seite 221)

[143] Falcou, Joël: *Un cluster pour la Vision Temps Réel Architecture, Outils et Applications*, Université Blaise Pascal - Clermont II, Diss., 2006 (Zitiert auf Seite 63)

[144] Falcou, Joël ; Sérot, Jocelyn: Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. In: *Parallel Computing: Architectures, Algorithms and Applications* Bd. 38, 2007 (NIC 3), S. 243–252 (Zitiert auf Seite 63)

[145] Falcou, Joël ; Sérot, Jocelyn ; Chateau, Thierry ; Lapresté, Jean-Thierry: QUAFF: efficient C++ design for parallel skeletons. In: *Parallel Computing* 32(7) (2006), S. 604–615 (Zitiert auf Seite 63)

[146] Ferreira, João F. ; Sobral, João Luís F. ; Proença, Alberto J.: JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing. In: *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, 2006, S. 301–304 (Zitiert auf Seite 57)

[147] Flynn, Michael: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* C-21 (1972), S. 948–960 (Zitiert auf Seite 12)

[148] Ford, Lester: Network flow theory / The RAND Corporation. 1956. – Forschungsbericht. – Paper P-923 (Zitiert auf Seite 162)

[149] Galán, Luis A. ; Pareja, Cristóbal ; Peña, Ricardo: Functional Skeletons Generate Process Topologies in Eden. In: *Programming Languages: Implementations, Logics, and Programs* Bd. 1140. Springer, 1996, S. 289–303 (Zitiert auf Seite 53)

[150] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2008 (Professional Computing) (Zitiert auf den Seiten 2, 117 und 126)

[151] Geist, Al ; Beguelin, Adam ; Dongarra, Jack ; Jiang, Weicheng ; Manchek, Robert ; Sunderam, Vaidy: *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994 (Scientific and Engineering Computation) (Zitiert auf Seite 53)

[152] Ginhac, Dominique ; Sérot, Jocelyn ; Dérutin, Jean-Pierre: Fast prototyping of image processing applications using functional skeletons on a MIMD-DM architecture. In: *IAPR Workshop on Machine Vision and Applications*, 1998, S. 468–471 (Zitiert auf Seite 70)

[153] González-Vélez, Horacio ; Cole, Murray: Towards Fully Adaptive Pipeline Parallelism for Heterogeneous Distributed Environments. In: *Parallel and Distributed Processing and Applications* Bd. 4330. Springer, 2006, S. 916–926 (Zitiert auf Seite 54)

[154] González-Vélez, Horacio ; Cole, Murray: Adaptive Structured Parallelism for Computational Grids. In: *Principles and Practice of Parallel Programming* (2007), S. 140–141 (Zitiert auf Seite 54)

[155] González-Vélez, Horacio ; Cole, Murray: An Adaptive Parallel Pipeline Pattern for Grids. In: *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society Press, 2008, S. 1–11 (Zitiert auf Seite 54)

[156] González-Vélez, Horacio ; Cole, Murray: Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. In: *Concurrency and Computation: Practice and Experience* (2010). – DOI: 10.1002/cpe.1549 (Zitiert auf Seite 54)

[157] Gorlatch, Sergei ; Dünnweber, Jan: From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In: *Future Generation Grids*. Springer US, 2006, S. 241–261 (Zitiert auf Seite 56)

[158] Goswami, Dhrubajyoti: *Parallel Architectural Skeletons: Re-Usable Building Blocks for Parallel Applications*, University of Waterloo, Diss., 2001 (Zitiert auf Seite 62)

[159] Goswami, Dhrubajyoti ; Singh, Ajit ; Preiss, Bruno R.: Architectural Skeletons: The Re-Usable Building-Blocks For Parallel Applications. In: *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applciations*, 1999, S. 1250–1256 (Zitiert auf Seite 62)

[160] Goswami, Dhrubajyoti ; Singh, Ajit ; Preiss, Bruno R.: Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons. In: *Computing in Object-Oriented Parallel Environments* Bd. 1732. Springer, 1999, S. 130–141 (Zitiert auf Seite 62)

[161] Goswami, Dhrubajyoti ; Singh, Ajit ; Preiss, Bruno R.: From Design Patterns to Parallel Architectural Skeletons. In: *Journal of Parallel and Distributed Computing* 62(4) (2002), S. 669–695 (Zitiert auf Seite 62)

[162] Gropp, William ; Lusk, Ewing ; Skjellum, Anthony: *Using MPI - Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1996 (Scientific and Engineering Computation) (Zitiert auf Seite 17)

[163] Hammond, Kevin ; Berthold, Jost ; Loogen, Rita: Automatic Skeletons in Template Haskell. In: *Parallel Processing Letters* 13(3) (2003), S. 413–424 (Zitiert auf Seite 53)

[164] Held, Jim ; Bautista, Jerry ; Koehl, Sean: From a Few Cores to Many: A Tera-scale Computing Research Overview / Intel Corporation. 2006. – Forschungsbericht. – Document number: 315297-001US (Zitiert auf Seite 216)

[165] Hernández, Félix ; Peña, Ricardo ; Rubio, Fernando: From GranSim to Paradise. In: *Trends in Functional Programming* Bd. 1. Intellect Books, 2000, S. 11–19 (Zitiert auf Seite 53)

[166] Herrmann, Christoph A.: *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*, Universität Passau, Diss., 2000 (Zitiert auf Seite 56)

[167] Herrmann, Christoph A. ; Lengauer, Christian: On the Space-Time Mapping of a Class of Divide-and-Conquer Recursions. In: *Parallel Processing Letters* 6(4) (1996), S. 525–537 (Zitiert auf Seite 56)

[168] Herrmann, Christoph A. ; Lengauer, Christian: Transformation of Divide & Conquer to Nested Parallel Loops. In: *Programming Languages: Implementations, Logics, and Programs* Bd. 1292. Springer, 1997, S. 95–109 (Zitiert auf Seite 56)

[169] Herrmann, Christoph A. ; Lengauer, Christian: Size Inference of Nested Lists in Functional Programs. In: *Proceedings of the 10th International Workshop on Implementation of Functional Languages*, 1998, S. 347–364 (Zitiert auf Seite 56)

[170] Herrmann, Christoph A. ; Lengauer, Christian: Static Parallelization of Functional Programs: Elimination of Higher-Order Functions & Optimized Inlining. In: *Euro-Par'99 Parallel Processing* Bd. 1685. Springer, 1999, S. 930–934 (Zitiert auf Seite 56)

[171] Herrmann, Christoph A. ; Lengauer, Christian: HDC: A higher-order language for divide-and-conquer. In: *Parallel Processing Letters* 10(2–3) (2000), S. 239–250 (Zitiert auf Seite 56)

[172] Herrmann, Christoph A. ; Lengauer, Christian: The HDC Compiler Project. In: *Proceedings of the 8th International Workshop on Compilers for Parallel Computers*, 2000, S. 239–254 (Zitiert auf Seite 56)

[173] Herrmann, Christoph A. ; Lengauer, Christian: A Transformational Approach which Combines Size Inference and Program Optimization. In: *Semantics, Applications, and Implementation of*

Program Generation Bd. 2196. Springer, 2001, S. 199–218 (Zitiert auf Seite 56)

[174] Herrmann, Christoph A. ; Lengauer, Christian: Transforming Rapid Prototypes to Efficient Parallel Programs. In: *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002, S. 65–94 (Zitiert auf Seite 56)

[175] Hidalgo-Herrero, Mercedes ; Ortega-Mallén, Yolanda: A Distributed Operational Semantics for a Parallel Functional Language. In: *Trends in Functional Programming* Bd. 2. Intellect Books, 2000, S. 89–102 (Zitiert auf Seite 53)

[176] Hidalgo-Herrero, Mercedes ; Ortega-Mallén, Yolanda: An Operational Semantics for the Parallel Language Eden. In: *Parallel Processing Letters* 12(2) (2002), S. 211–228 (Zitiert auf Seite 53)

[177] Hidalgo-Herrero, Mercedes ; Ortega-Mallén, Yolanda: Continuation Semantics for Parallel Haskell Dialects. In: *Implementation of Functional Languages* Bd. 2895. Springer, 2003, S. 303–321 (Zitiert auf Seite 53)

[178] Hoefler, Torsten ; Schellmann, Maraike ; Gorlatch, Sergei ; Lumsdaine, Andrew: Communication Optimization for Medical Image Reconstruction Algorithms. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface* Bd. 5205. Springer, 2008, S. 75–83 (Zitiert auf Seite 173)

[179] Hoffmann, Simon ; Lienhart, Rainer: *OpenMP*. Springer, 2008 (Informatik im Fokus) (Zitiert auf den Seiten 34, 35, 36, 40 und 140)

[180] Hopfield, John J.: Neural networks and physical systems with emergent collective computational abilities. In: *Proceedings of the National Academy of Sciences of the United States of America* 79(8) (1982), S. 2554–2558 (Zitiert auf Seite 194)

[181] Hudak, Paul ; Hughes, John ; Jones, Simon P. ; Wadler, Philip: A History of Haskell: Being Lazy with Class. In: *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming*

Languages, ACM, 2007, S. 12–1–12–55 (Zitiert auf den Seiten 53 und 56)

[182] Hudson, Malcolm ; Larkin, Richar: Accelerated Image Reconstruction using Ordered Subsets of Projection Data. In: *IEEE Transactions on Medical Imaging* 13(4) (1982), S. 601–609 (Zitiert auf Seite 168)

[183] Hwang, Jenq-Neng ; Kung, Sun-Yuan: Parallel Algorithms/Architectures for Neural Networks. In: *Journal of VLSI Signal Processing* 1(3) (1989), S. 221–251 (Zitiert auf Seite 194)

[184] IBM (Hrsg.): *Compiler Reference – IBM XL C/C++ for Linux, V10.1*. IBM, 2008 (Zitiert auf Seite 35)

[185] Intel Corporation (Hrsg.): *Intel C++ Compiler Documentation*. Intel Corporation. – Document number: 304967-021US (Zitiert auf den Seiten 48 und 190)

[186] Intel Corporation (Hrsg.): *Intel C++ Compiler User and Reference Guides*. Intel Corporation. – Document number: 304968-023US (Zitiert auf den Seiten 35 und 48)

[187] Iwasaki, Hideya ; Hu, Zhenjiang: A New Parallel Skeleton for General Accumulative Computations. In: *International Journal of Parallel Programming* 32(5) (2004), S. 389–414 (Zitiert auf Seite 67)

[188] Jones, Simon P. (Hrsg.): *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, 2003 (Zitiert auf den Seiten 53 und 56)

[189] Kakehi, Kazuhiko ; Matsuzaki, Kiminori ; Emoto, Kento: Efficient Parallel Tree Reductions on Distributed Memory Environments. In: *Computational Science - ICCS 2007* Bd. 4488. Springer, 2007, S. 601–608 (Zitiert auf Seite 67)

[190] Karasawa, Yuki ; Iwasaki, Hideya: Parallel Skeletons for Sparse Matrices in SkeTo Skeleton Library. In: *IPSJ Digital Courier* 4 (2008), S. 167–181 (Zitiert auf Seite 67)

[191] Karasawa, Yuki ; Iwasaki, Hideya: A Parallel Skeleton Library for Multi-core Clusters. In: *Proceedings of the 2009 International Conference on Parallel Processing*, 2009, S. 84–91 (Zitiert auf den Seiten 67 und 68)

[192] Kegel, Philipp ; Schellmann, Maraike ; Gorlatch, Sergei: Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores. In: *Euro-Par 2009 Parallel Processing* Bd. 5604. Springer, 2009, S. 654–665 (Zitiert auf den Seiten 168, 173 und 183)

[193] Kirk, David ; Hwu, Wen-mei W.: *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufman, 2010 (Zitiert auf den Seiten 219 und 220)

[194] Klusik, Ulrike: *An Efficient Implementation of the Parallel Functional Language Eden on Distributed-Memory System*, Philipps-Universität Marburg, Diss., 2003 (Zitiert auf Seite 53)

[195] Klusik, Ulrike ; Loogen, Rita ; Priebe, Steffen: Controlling Parallelism and Data Distribution in Eden. In: *Trends in Functional Programming* Bd. 2. Intellect Books, 2000, S. 53–64 (Zitiert auf Seite 53)

[196] Klusik, Ulrike ; Loogen, Rita ; Priebe, Steffen ; Rubio, Fernando: Implementation Skeletons in Eden: Low-Effort Parallel Programming. In: *Implementation of Functional Languages* Bd. 2011. Springer, 2001, S. 71–88 (Zitiert auf Seite 53)

[197] Klusik, Ulrike ; Ortega-Mallén, Yolanda ; Peña, Ricardo: Implementing Eden — or: Dreams Become Reality. In: *Implementation of Functional Languages* Bd. 1595. Springer, 1999, S. 649–664 (Zitiert auf Seite 53)

[198] Klusik, Ulrike ; Peña, Ricardo ; Segura, Clara: Bypassing of Channels in Eden. In: *Trends in Functional Programming* Bd. 1. Intellect Books, 2000, S. 2–10 (Zitiert auf Seite 53)

[199] Kohonen, Teuvo: Self-organized formation of topologically correct feature maps. In: *Biological Cybernetics* 43(1) (1982), S. 59–69 (Zitiert auf Seite 194)

[200] Krüger, Guido: *Handbuch der Java-Programmierung*. 4. Addison Wesley, 2006 (Zitiert auf den Seiten 20, 72, 80, 83 und 221)

[201] Krumke, Sven ; Noltemeier, Hartmut: *Graphentheoretische Konzepte und Algorithmen*. Teubner, 2005 (Leitfäden der Informatik) (Zitiert auf Seite 162)

[202] Kuchen, Herbert: *Datenparallele Programmierung von MIMD-Rechnern mit verteiltem Speicher*. Shaker, 1996 (Zitiert auf Seite 69)

[203] Kuchen, Herbert: A Skeleton Library. In: *Proceedings of Euro-Par*, Springer, 2002, S. 620–629 (Zitiert auf den Seiten 3, 4, 5, 6 und 75)

[204] Kuchen, Herbert: Optimizing Sequences of Skeleton Calls. In: *Domain-Specific Program Generation* Bd. 3016. Springer, 2004, S. 99–140 (Zitiert auf den Seiten 75 und 86)

[205] Kuchen, Herbert ; Cole, Murray: The Integration of Task and Data Parallel Skeletons. In: *Parallel Processing Letters* 12(2) (2002), S. 141–155 (Zitiert auf den Seiten 4 und 76)

[206] Kuchen, Herbert ; Striegnitz, Jörg: Higher-Order Functions and Partial Applications for a C++ Skeleton Library. In: *Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, ACM, 2002, S. 122–130 (Zitiert auf Seite 81)

[207] Kuhlis, Stefan ; Schrader, Martin: *Die C++-Standardbibliothek. Einführung und Nachschlagewerk*. 4. Springer, 2005 (Zitiert auf den Seiten 122, 133 und 159)

[208] Langer, Jens: *Event-Driven Motion Compensation in Positron Emission Tomography: Development of a Clinically Applicable Method*, Technische Universität Dresden, Diss., November 2008 (Zitiert auf Seite 167)

[209] Leyton, Mario: *Advanced Features for Algorithmic Skeleton Programming*, Universität Nizza Sophia-Antipolis, Diss., Oktober 2008 (Zitiert auf den Seiten 49 und 51)

[210] Leyton, Mario ; Piquer, José M.: A Skandium based Parallelization of DNSSEC. In: *Proceedings of the XIII Workshop on Distributed Systems and Parallelism as part of the XXVIII International Conference of the Chilean Computer Society*, 2009, S. 30–37 (Zitiert auf Seite 65)

[211] Leyton, Mario ; Piquer, José M.: Skandium: Multi-core Programming with Algorithmic Skeletons. In: *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2010, S. 289–296 (Zitiert auf Seite 65)

[212] Lippe, Wolfram-Manfred: *Soft-Computing*. Springer, 2006 (eXamen.press) (Zitiert auf den Seiten 193, 195 und 199)

[213] Lippman, Stanley ; Lajoie, Josée ; Moo, Barbara: *C++ Primer*. Addison-Wesley, 2006 (Zitiert auf den Seiten 19, 20, 40, 52, 80, 81, 102, 103, 112, 122, 133, 134, 138, 139, 140, 159 und 232)

[214] Loidl, Hans-Wolfgang ; Klusik, Ulrike ; Hammond, Kevin ; Loogen, Rita ; Trinder, Philip W.: GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster. In: *Trends in Functional Programming* Bd. 2. Intellect Books, 2000, S. 39–52 (Zitiert auf Seite 53)

[215] Loidl, Hans-Wolfgang ; Rubio, Fernando ; Scaife, Norman ; Hammond, Kevin ; Horiguchi, Susumu ; Klusik, Ulrike ; Loogen, Rita ; Michaelson, Greg J. ; Peña, Ricardo ; Priebe, Stefan ; J. Rebón Portillo Álvaro ; Trinder, Philip W.: Comparing Parallel Functional Languages: Programming and Performance. In: *Higher-Order and Symbolic Computation* 16(3) (2003), S. 203–251 (Zitiert auf Seite 53)

[216] Loogen, Rita: Programming Language Constructs. In: *Research Directions in Parallel Functional Programming*. Springer, 1999, S. 63–91 (Zitiert auf Seite 53)

[217] Loogen, Rita ; Ortega-Mallén, Yolanda ; Peña, Ricardo ; Priebe, Steffen ; Rubio, Fernando: Parallelism Abstractions in Eden. In: *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003, S. 95–128 (Zitiert auf Seite 53)

[218] Loogen, Rita ; Ortega-Mallén, Yolanda ; Peña-Marí, Ricardo: Parallel Functional Programming in Eden. In: *Journal of Functional Programming* 15(3) (2005), S. 431–475 (Zitiert auf Seite 53)

[219] MacDonald, Steve ; Anvik, John ; Bromling, Steven ; Schaeffer, Jonathan ; Szafron, Duane ; Tan, Kai: From patterns to frameworks to parallel programs. In: *Parallel Computing* 28(12) (2002), S. 1663–1683 (Zitiert auf Seite 51)

[220] MacDonald, Steve ; Szafron, Duane ; Schaeffer, Jonathan: Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks. In: *Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems*, 1999, S. 29–43 (Zitiert auf Seite 51)

[221] MacDonald, Steve ; Szafron, Duane ; Schaeffer, Jonathan ; Bromling, Steven: Generating Parallel Program Frameworks from Parallel Design Patterns. In: *Euro-Par 2000 Parallel Processing* Bd. 1900. Springer, 2000, S. 95–104 (Zitiert auf Seite 51)

[222] MacDonald, Steven: *From Patterns to Frameworks to Parallel Programs*, Universität Alberta, Diss., 2002 (Zitiert auf Seite 51)

[223] Matsuzaki, Kiminori: Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers. In: *Computational Science - ICCS 2007* Bd. 4488. Springer, 2007, S. 609–616 (Zitiert auf Seite 67)

[224] Matsuzaki, Kiminori ; Hu, Zhenjiang ; Kakehi, Kazuhiko ; Takeichi, Masato: Systematic Derivation of Tree Contraction Algorithms. In: *Parallel Processing Letters* 15(3) (2005), S. 321–336 (Zitiert auf Seite 67)

[225] Matsuzaki, Kiminori ; Hu, Zhenjiang ; Takeichi, Masato: Parallelization with Tree Skeletons. In: *Euro-Par 2003 Parallel Processing* Bd. 2790. Springer, 2004, S. 789–798 (Zitiert auf Seite 67)

[226] Matsuzaki, Kiminori ; Hu, Zhenjiang ; Takeichi, Masato: Parallel skeletons for manipulating general trees. In: *Parallel Computing* 32(7-8) (2006), S. 590–603 (Zitiert auf Seite 67)

[227] Matsuzaki, Kiminori ; Hu, Zhenjiang ; Takeichi, Masato: Towards Automatic Parallelization of Tree Reductions in Dynamic Programming. In: *Proceedings of the 8th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2006, S. 39–48 (Zitiert auf Seite 67)

[228] Matsuzaki, Kiminori ; Kakehi, Kazuhiko ; Iwasaki, Hideya ; Hu, Zhenjiang ; Akashi, Yoshiki: A Fusion-Embedded Skeleton Library. In: *Euro-Par 2004 Parallel Processing* Bd. 3149. Springer, 2004, S. 644–653 (Zitiert auf Seite 67)

[229] Mattson, Timothy ; Sanders, Beverly ; Massingill, Berna: *Patterns for Parallel Programming*. Addison-Wesley, 2005 (Software Patterns) (Zitiert auf Seite 2)

[230] McCulloch, Warren S. ; Pitts, Walter: A logical calculus of the ideas immanent in nervous activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133 (Zitiert auf Seite 194)

[231] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), 2009 (Zitiert auf den Seiten 2, 17, 18 und 48)

[232] Moore, Gordon E.: Cramming more components onto integrated circuits. In: *Electronics* 38(8) (1965), S. 4–7 (Zitiert auf Seite 216)

[233] Morel, Matthieu: *Components for Grid Computing*, Universität Nizza Sophia-Antipolis, Diss., November 2006 (Zitiert auf Seite 51)

[234] Neves, Diogo T. ; Sobral, João Luís F.: Improving the Separation of Parallel Code in Skeletal Systems. In: *Proceedings of the 8th International Symposium on Parallel and Distributed Computing*, 2009, S. 257–260 (Zitiert auf Seite 58)

[235] Nguyen, Hubert (Hrsg.): *GPU Gems 3 - Programming Techniques for High-Performance Graphics General-Purpose Computation*. Addison-Wesley, 2007 (Zitiert auf Seite 220)

[236] Nordström, Tomas ; Svensson, Bertil: Using and Designing Massively Parallel Computers for Artificial Neural Networks. In: *Journal of Parallel and Distributed Computing* 14(3) (1992), S. 260–285 (Zitiert auf Seite 194)

[237] OpenMP Architecture Review Board: *OpenMP Application Program Interface, Version 2.5*, 2005 (Zitiert auf den Seiten 2, 34, 38, 39, 40, 41, 42, 43, 44, 139 und 140)

[238] OpenMP Architecture Review Board: *OpenMP Application Program Interface, Version 3.0*, 2008 (Zitiert auf den Seiten 2, 34, 45, 139, 141 und 219)

[239] Owens, John D. ; Luebke, David ; Govindaraju, Naga ; Harris, Mark ; Krüger, Jens ; Lefohn, Aaron E. ; Purcell, Timothy J.: A Survey of General-Purpose Computation on Graphics Hardware. In: *Computer Graphics Forum* 26(1) (2007), S. 80–113 (Zitiert auf Seite 219)

[240] Pareja, Cristóbal ; Peña, Ricardo ; Rubio, Fernando ; Segura, Clara: Optimizing Eden by Transformation. In: *Trends in Functional Programming* Bd. 2. Intellect Books, 2000, S. 13–26 (Zitiert auf Seite 53)

[241] Patnaik, Lalit M. ; Ghare, Gaurav. D.: Implementation of ART 1 and ART 2 Artificial Neural Networks on Ring and Mesh Architectures. In: *International Journal of High Speed Computing* 9(1) (1997), S. 41–56 (Zitiert auf Seite 194)

[242] Peña, Ricardo ; Rubio, Fernando: Parallel functional programming at two levels of abstraction. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2001, S. 187–198 (Zitiert auf Seite 53)

[243] Peña, Ricardo ; Segura, Clara: Non-determinism Analysis in a Parallel-Functional Language. In: *Implementation of Functional Languages* Bd. 2011. Springer, 2001, S. 1–18 (Zitiert auf Seite 53)

[244] Pelagatti, Susanna: Compiling and Supporting Skeletons on MPP. In: *Proceedings of the 3rd Working Conference on Massively Parallel Programming Models*, IEEE Computer Society Press, 1997, S. 140–150 (Zitiert auf Seite 61)

[245] Pelagatti, Susanna ; Vanneschi, Marco: Task and Data Parallelism in P3L. In: *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003, S. 155–186 (Zitiert auf Seite 61)

[246] Pharr, Matt: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005 (Software Patterns) (Zitiert auf Seite 219)

[247] Pointon, Robert F. ; Priebe, Steffen ; Loidl, Hans-Wolfgang ; Loogen, Rita ; Trinder, Phil W.: Functional Vs Object-Oriented Distributed Languages. In: *Computer Aided Systems Theory – EUROCAST 2001* Bd. 2178. Intellect Books, 2001, S. 642–656 (Zitiert auf Seite 53)

[248] Poldner, Michael: *Task Parallel Algorithmic Skeletons*, Westfälische Wilhelms-Universität Münster, Diss., 2008 (Zitiert auf den Seiten 3, 7, 75, 76 und 210)

[249] Poldner, Michael ; Kuchen, Herbert: Algorithmic Skeletons for Branch and Bound. In: *Proceedings of the 1st International Conference on Software and Data Technology (ICSOFT)* Bd. 1, 2006, S. 291–300 (Zitiert auf den Seiten 3, 75 und 76)

[250] Poldner, Michael ; Kuchen, Herbert: Scalable Farms. In: *Proceedings of the International Conference on Parallel Computing (ParCo)* Bd. 33, 2006 (NIC Series), S. 795–802 (Zitiert auf den Seiten 3, 75, 76 und 210)

[251] Poldner, Michael ; Kuchen, Herbert: On Implementing the Farm Skeleton. In: *Parallel Processing Letters* 18(1) (2008), S. 117–131 (Zitiert auf den Seiten 3, 75, 76, 176 und 210)

[252] Poldner, Michael ; Kuchen, Herbert: Optimizing Skeletal Stream Processing for Divide and Conquer. In: *Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT)*, 2008, S. 181–189 (Zitiert auf den Seiten 3, 75 und 76)

[253] Poldner, Michael ; Kuchen, Herbert: Skeletons for Divide and Conquer Algorithms. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2008, S. 181–188 (Zitiert auf den Seiten 3, 75 und 76)

[254] Priebe, Steffen: Preprocessing Eden with Template Haskell. In: *Generative Programming and Component Engineering* Bd. 3676. Springer, 2005, S. 357–372 (Zitiert auf Seite 53)

[255] Quinn, Michael: *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003 (Zitiert auf den Seiten 12, 13, 14, 15, 16 und 17)

[256] Rabhi, Fethi (Hrsg.) ; Gorlatch, Sergei (Hrsg.): *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003 (Zitiert auf Seite 2)

[257] Rauber, Thomas ; Rünger, Gudula: *Parallele Programmierung*. 2. Springer, 2007 (eXamen.press) (Zitiert auf den Seiten 12 und 34)

[258] Reader, Andrew J. ; Erlandsson, Kjell ; Flower, Maggie A. ; Ott, Robert J.: Fast accurate iterative reconstruction for low-statistics positron volume imaging. In: *Physics in Medicine and Biology* 43(4) (1998), S. 823–834 (Zitiert auf den Seiten 168 und 171)

[259] Reinders, James: *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007 (Zitiert auf den Seiten 185 und 219)

[260] Rubio, Fernando: *Programación Funcional Paralela Eficiente en Eden*, Universität Complutense Madrid, Diss., November 2001 (Zitiert auf Seite 53)

[261] Saratchandran, Paramasivan ; Sundararajan, Narasimhan: Analysis of Training Set Parallelism for Backpropagation Neural Networks. In: *International Journal of Neural Systems* 6(1) (1995), S. 61–78 (Zitiert auf Seite 194)

[262] Sato, Shigeyuki ; Iwasaki, Hideya: A Skeletal Framework with Fusion Optimizer for GPGPU Programming. In: *Programming Languages and Systems* Bd. 5904. Springer, 2009, S. 79–94 (Zitiert auf den Seiten 67 und 68)

[263] Schellmann, Maraike: *Efficient PET Image Reconstruction on Modern Parallel and Distributed Systems*, Westfälische Wilhelms-Universität Münster, Diss., 2009 (Zitiert auf den Seiten 167, 168 und 169)

[264] Schellmann, Maraike ; Gorlatch, Sergei: Comparison of Two Decomposition Strategies for Parallelizing the 3D List-Mode OSEM Algorithm. In: *Proceedings of the 9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D), Workshop on High-Performance Image Reconstruction (HPIR)*, 2007, S. 37–40 (Zitiert auf den Seiten 168, 171 und 183)

[265] Schellmann, Maraike ; Gorlatch, Sergei ; Meiländer, Dominik ; Kösters, Thomas ; Schäfers, Klaus ; Wübbeling, Frank ; Burger, Martin: Parallel Medical Image Reconstruction: From Graphics Processors to Grids. In: *Parallel Computing Technologies* Bd. 5698. Springer, 2008, S. 457–473 (Zitiert auf den Seiten 168, 170 und 172)

[266] Schellmann, Maraike ; Koesters, Thomas ; Gorlatch, Sergei: Parallelization and Runtime Prediction of the Listmode OSEM Algorithm for 3D PET Reconstruction. In: *IEEE Nuclear Science Symposium Conference Record* Bd. 4, 2006, S. 2190–2195 (Zitiert auf Seite 168)

[267] Schellmann, Maraike ; Vörding, Jürgen ; Gorlatch, Sergei: Systematic Parallelization of Medical Image Reconstruction for Gra-

phics Hardware. In: *Euro-Par 2008 - Parallel Processing* Bd. 5168. Springer, 2008, S. 811–821 (Zitiert auf Seite 168)

[268] Schellmann, Maraike ; Vörding, Jürgen ; Gorlatch, Sergei ; Meiländer, Dominik: Cost-Effective Medical Image Reconstruction: From Clusters to Graphics Processing Units. In: *Proceedings of the 5th Conference on Computing Frontiers*, 2008, S. 283–292 (Zitiert auf Seite 168)

[269] Schäfers, Klaus P. ; Reader, Andrew J. ; Kriens, Michael ; Knoess, Christof ; Schober, Otmar ; Schäfers, Michael: Performance Evaluation of the 32-Module quadHIDAC Small-Animal PET Scanner. In: *The Journal of Nuclear Medicine* 46(6) (2005), S. 996–1004 (Zitiert auf Seite 186)

[270] Shafi, Aamir ; Carpenter, Bryan ; Baker, Mark: Nested Parallelism for Multi-Core HPC Systems Using Java. In: *Journal of Parallel and Distributed Computing* 69(6) (2009), S. 532–545 (Zitiert auf den Seiten 71 und 221)

[271] Shepp, Lawrence A. ; Vardi, Yehuda: Maximum Likelihood Reconstruction for Emission Tomography. In: *IEEE Transactions on Medical Imaging* 1(2) (1982), S. 113–122 (Zitiert auf den Seiten 168 und 171)

[272] Siddon, Robert L.: Fast calculation of the exact radiological path for a three-dimensional CT array. In: *Medical Physics* 12(2) (1985), S. 252–255 (Zitiert auf Seite 171)

[273] Snir, Marc ; Otto, Steve ; Huss-Lederman, Steven ; Walker, David ; Dongarra, Jack: *MPI: The Complete Reference*. MIT Press, 1996 (Zitiert auf den Seiten 2, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30, 31, 32 und 33)

[274] Sobral, João Luís F. ; Proença, Alberto J.: Enabling JaSkel Skeletons for Clusters and Computational Grids. In: *Proceedings of the IEEE International Conference on Cluster Computing*, 2007, S. 365–371 (Zitiert auf Seite 57)

[275] Spector, Alfred ; Gifford, David: The Space Shuttle Primary Computer System. In: *Communications of the ACM* 27(9) (1984), S. 872–900 (Zitiert auf Seite 14)

[276] Sérot, Jocely ; Ginhac, Dominique: Skeletons for parallel image processing: an overview of the SKIPPER project. In: *Parallel Computing* 28(12) (2002), S. 1685–1708 (Zitiert auf Seite 70)

[277] Sérot, Jocelyn: Tagged-Token Data-Flow for Skeletons. In: *Parallel Processing Letters* 11(4) (2001), S. 377–392 (Zitiert auf Seite 70)

[278] Sérot, Jocelyn ; Ginhac, Dominique ; Chapuis, Roland ; Dérutin, Jean-Pierre: Fast prototyping of parallel-vision applications using functional skeletons. In: *Machine Vision and Applications* 12(6) (2001), S. 271–290 (Zitiert auf Seite 70)

[279] Sérot, Jocelyn ; Ginhac, Dominique ; Dérutin, Jean-Pierre: SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In: *Parallel Computing Technologies* Bd. 1662. Springer, 1999, S. 767–776 (Zitiert auf Seite 70)

[280] Stallings, William: *Betriebssysteme: Prinzipien und Umsetzung*. 4. Pearson Studium, 2002 (Zitiert auf Seite 35)

[281] Stallings, William: *CUDA by Example - An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010 (Zitiert auf den Seiten 219 und 220)

[282] Stallmann, Richard: *Using the GNU Compiler Collection – For GCC version 4.4.2*, 2009 (Zitiert auf Seite 35)

[283] Stoer, Josef ; Bulirsch, Roland: *Introduction to Numerical Analysis*. 3. Springer, 2002 (Zitiert auf Seite 111)

[284] Striegnitz, Jörg: Making C++ Ready for Algorithmic Skeletons / Forschungszentrum Jülich. 2000. – Forschungsbericht. – FZJ-ZAM-IB-2000-08 (Zitiert auf Seite 82)

[285] Sun Microsystems (Hrsg.): *Sun Studio 12 Update 1: OpenMP API User's Guide*. Sun Microsystems, 2009 (Zitiert auf Seite 35)

[286] Sunke, Holger: *Datenparallele Skelette - Anwendungen für dünnbesetzte Matrizen*, Westfälische Wilhelms-Universität Münster, Diplomarbeit, 2009 (Zitiert auf Seite 162)

[287] Tan, Kai ; Szafron, Duane ; Schaeffer, Jonathan ; Anvik, John ; MacDonald, Steve: Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. In: *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, S. 203–215 (Zitiert auf Seite 51)

[288] Tanno, Haruto ; Iwasaki, Hideya: Parallel Skeletons for Variable-length Lists in SkeTo Skeleton Library. In: *Euro-Par 2009 Parallel Processing* Bd. 5704. Springer, 2009, S. 666–677 (Zitiert auf Seite 67)

[289] The Globus Alliance: *Globus*. `http://www.globus.org`, 7. Juli 2010 (Zitiert auf Seite 57)

[290] The Khronos Group: *OpenCL*. `http://www.khronos.org/opencl`, 4. Mai 2010 (Zitiert auf Seite 220)

[291] The Open MPI Project: *Open MPI: Open Source High Performance Computing*. `http://www.open-mpi.org`, 10. Februar 2010 (Zitiert auf Seite 48)

[292] Thompson, Simon: *Haskell: The Craft of Functional Programming*. 2. Addison-Wesley, 1999 (International Computer Science) (Zitiert auf den Seiten 53 und 56)

[293] TOP500.Org: *Home | TOP500 Supercomputing Sites*. `http://www.top500.org`, 10. Februar 2010 (Zitiert auf den Seiten 48 und 220)

[294] Tsuchiyama, Ryoji ; Nakamura, Takashi ; Iizuka, Takuro ; Asahara, Akihiro ; Miki, Satoshi: *The OpenCL Programming Book. Parallel Programming for MultiCore CPU and GPU*. Fixstars Corporation, 2010 (Zitiert auf Seite 220)

[295] Vandevoorde, David ; Josuttis, Nicolai: *C++ Templates. The Complete Guide*. Addison-Wesley, 2003 (Zitiert auf den Seiten 80, 102 und 103)

[296] Vanneschi, Marco: Heterogeneous HPC Environments. In: *Euro-Par'98 Parallel Processing* Bd. 1470. Springer, 1998, S. 21–34 (Zitiert auf Seite 68)

[297] Vanneschi, Marco: The programming model of ASSIST, an environment for parallel and distributed portable applications. In: *Parallel Computing* 28(12) (2001), S. 1709–1732 (Zitiert auf Seite 50)

[298] Wei, Zunce ; Li, Hon F. ; Goswami, Dhrubajyoti: Composable Skeletons for Parallel Programming. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* Bd. 3, 2004, S. 1256–1261 (Zitiert auf Seite 62)

[299] Wikipedia, the free encyclopedia: *Algorithmic skeleton*. `http://en.wikipedia.org/wiki/Algorithmic_skeleton`, 18. März 2010 (Zitiert auf Seite 49)

[300] Wilkinson, James H. ; Reinsch, Christian: *Handbook for Computation, Volume II, Linear Algebra*. Springer, 1971 (Zitiert auf Seite 111)

[301] Yaikhom, Gagarine ; Cole, Murray ; Gilmore, Stephen ; Hillston, Jane: A Structural Approach for Modelling Performance of Systems Using Skeletons. In: *Electronic Notes in Theoretical Computer Science* 190(3) (2007), S. 167–183 (Zitiert auf Seite 54)

[302] Zell, Andreas: *Simulation Neuronaler Netze*. Addison-Wesley, 1994 (Zitiert auf den Seiten 193, 195, 197, 198, 201, 202, 203, 204, 205 und 206)

[303] Zentrum für Informationsverarbeitung: *ZIVPHC*. `https://www.uni-muenster.de/ZIV/Technik/ZIVHPC/index.html`, 1. Dezember 2009 (Zitiert auf Seite 47)

[304] Zentrum für Informationsverarbeitung: *PALMA*. `https://www.uni-muenster.de/ZIVwiki/bin/view/Anleitungen/PALMA`, 21. Juli 2010 (Zitiert auf Seite 48)

Index

Symbole

A

Q

R

S

Y

Z

Datenparallele algorithmische Skelette

Philipp Ciechanowicz

Die Arbeit thematisiert den datenparallelen Bestandteil der Münster Skelettbibliothek Muesli und beschreibt neben einer Reihe implementierter Erweiterungen auch mit Muesli parallelisierte Anwendungen. Eine der wichtigsten Neuerungen ist die Unterstützung von Mehrkernprozessoren durch die Verwendung von OpenMP, infolgedessen mit Muesli entwickelte Programme auch auf Parallelrechnern mit hybrider Speicherarchitektur skalieren. Eine zusätzliche Erweiterung stellt die Neuentwicklung einer verteilten Datenstruktur für dünnbesetzte Matrizen dar. Letztere implementiert ein flexibles Designkonzept, was die Verwendung benutzerdefinierter Kompressions- sowie Lastverteilungsmechanismen ermöglicht. Darüber hinaus werden mit dem LM OSEM-Algorithmus und den ART 2-Netzen zwei Anwendungen vorgestellt, die mit Muesli parallelisiert wurden. Neben einer Beschreibung der Funktionsweise sowie der Eigenschaften und Konzepte von MPI und OpenMP wird darüber hinaus der aktuelle Forschungsstand skizziert.

ISBN 978-3-8405-0029-9 EUR 19,90
01990
9 783840 500299