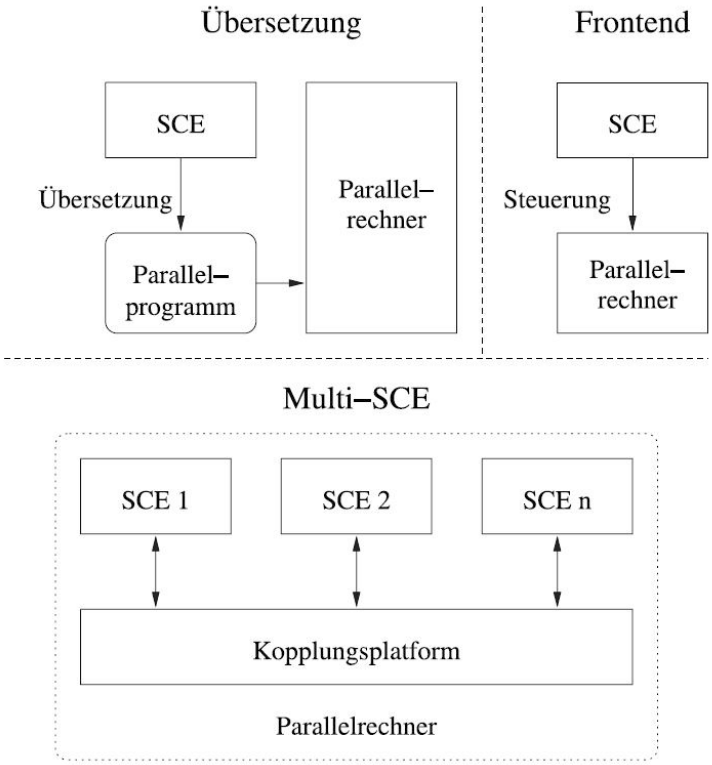
FORTSCHRITTSBERICHT SIMULATION FBS 12



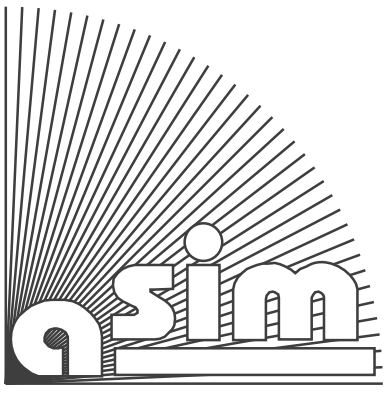
René Fink

Untersuchungen zur Parallelverarbeitung mit Wissenschaftlich-technischen Berechnungsumgebungen

ARGESIM

ISBN Ebook 978-3-903347-12-0
ISBN Print 978-3-901608-62-9

DOI: 10.11128/fbs.12


argesim


ARGESIM

Fortschrittsberichte Simulation

FBS Band 12

Herausgegeben von **ASIM**
Arbeitsgemeinschaft **Sim**ulation, Fachausschuss der GI im
Fachbereich ILW – Informatik in den Lebenswissenschaften

René Fink

Untersuchungen zur Parallelverarbeitung mit wissenschaftlich-technischen Berechnungsumgebungen

ARGESIM / ASIM – Verlag, Wien, 2008
ISBN Print 978-3-901608-62-9

Ebook Reprint 2020
ISBN Ebook 978-3-903347-12-0
DOI: 10.11128/fbs.12

Fortschrittsberichte Simulation

Herausgegeben von **ASIM**, Arbeitsgemeinschaft Simulation, Fachausschuß der GI im Fachbereich ILW – Informatik in den Lebenswissenschaften

Betreuer der Reihe:

Prof. Dr.-Ing. Th. Pawletta (ASIM)
Hochschule Wismar
Phillip-Müller-Str., 23952 Wismar, Germany
Tel: +49-3841-753-406, Fax: +49-3841-753-132
Email: **pawel@mb.hs-wismar.de**

Dr.-Ing. habil. D.P.F. Schwarz (ASIM)
Fraunhofer-Institut für Integrierte Schaltungen
Zeunerstr. 38, 01069 Dresden, Germany
Tel: +49-351- 4640 - 730, Fax: +49-351-4640-703
Email: **schwarz@eas.iis.fhg.de**

Prof. Dr. F. Breitenecker (ARGESIM / ASIM)
Technische Universität Wien
Wiedner Hauptstraße 8 - 10, 1040 Wien, Austria
Tel: +43-1-58801-10115, Fax: +43-1-58801-10199
Email: **Felix.Breitenecker@tuwien.ac.at**

FBS Band 12

Titel: Untersuchungen zur Parallelverarbeitung mit wissenschaftlich-technischen Berechnungsumgebungen

Autor: Dipl.-Ing. (FH) Dr.-Ing. René Fink
IAV GmbH
Philipp-Müller-Strasse 12
23966 Wismar
Germany
Email: **rene.fink@iav.de**

Begutachter des Bandes:

Prof. Dr.-Ing. B. Lampe, Prof. Dr.-Ing. S. Pawletta, Prof. Dr. F. Breitenecker

ARGESIM / ASIM – Verlag, Wien, 2008

ISBN Print 978-3-901608-62-9

Ebook Reprint 2020

ISBN Ebook 978-3-903347-12-0

DOI: 10.11128/fbs.12

Untersuchungen zur Parallelverarbeitung mit wissenschaftlich-technischen Berechnungsumgebungen

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

René Fink

eingereicht am
9. Juli 2007

verteidigt am
13. Dezember 2007

begutachtet von
Prof. Dr.-Ing. habil. Dr. h.c. Bernhard Lampe,
Institut für Automatisierungstechnik, Universität Rostock

Prof. Dr.-Ing. Sven Pawletta,
Bereich Elektrotechnik und Informatik, Hochschule Wismar

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Felix Breitenecker,
Institut für Analysis und Scientific Computing, Technische Universität Wien

Kurzfassung

Wissenschaftlich-technische Berechnungsumgebungen, wie Matlab, Scilab oder Octave, die nach Pawletta hier als SCEs bezeichnet werden, sind heute unverzichtbare Werkzeuge in vielen ingenieurtechnischen Anwendungsbereichen. Mit ihrer Hilfe können wissenschaftliche beziehungsweise technische Berechnungsprogramme interaktiv entwickelt, ausgeführt und ausgewertet werden, sodass ingenieurtechnische Entwurfsaufgaben im Gegensatz zum klassischen compilerbasierten Arbeiten hier deutlich schneller umgesetzt werden können. Durch die interpretative Arbeitsweise von SCEs ergibt sich grundsätzlich eine geringere Ausführungsgeschwindigkeit gegenüber kompilierten Programmen, wodurch die Effektivität interaktiver Entwurfsprozesse verringert wird. Es existieren daher verschiedene Ansätze zur Programmbeschleunigung in SCEs. Die SCE-basierte Parallelverarbeitung repräsentiert einen dieser Ansätze.

Trotz einer Vielzahl an spezifischen Softwaresystemen ist die SCE-basierte Parallelverarbeitung bisher kaum in ingenieurtechnischen Bereichen verbreitet. Die Gründe dafür sind einerseits eine fehlende Evaluierung der Softwaresysteme für ingenieurtechnische Anwendungsfelder und andererseits ein Mangel an Erkenntnissen bezüglich der Entwicklung paralleler SCE-Programme, die Aussagen über den zu erwartenden Aufwand und Nutzen zulassen.

Die vorliegende Arbeit hat das Ziel, die Anwendung der SCE-basierten Parallelverarbeitung in ingenieurtechnischen Bereichen stärker zu etablieren. Aufbauend auf der Arbeit von Pawletta wird dabei der aktuelle Entwicklungsstand wiedergegeben und eine neue Klassifikation für dieses Gebiet vorgeschlagen. Es wird gezeigt, dass aus der neuen Klassifikation insbesondere der Multi-SCE-Ansatz für ingenieurtechnische Bereiche interessant ist. Verfügbare Multi-SCE-Systeme werden anhand qualitativer und quantitativer Merkmale verglichen. Darüber hinaus erfolgt die Präsentation weiterentwickelter Multi-SCE-Systeme, die zum Teil explizit auf industrielle Anwendungsgebiete fokussieren. Eine anhand verschiedener ingenieurtechnischer Applikationen durchgeführte anwendungsbezogene Untersuchung zeigt beispielhaft Parallelisierungsaufwand und Laufzeitgewinn bei verschiedenen Anwendungstypen und Multi-SCE-Systemen.

Abstract

Scientific and technical computing environments, referred to as SCEs following Pawletta, are essential tools in today's engineering domains. They enable scientific and technical computations to be developed, executed and analysed within one environment. Therefore, design processes can be realized significantly faster within SCEs in opposite to classical compiler based programming. Caused by the interpretative way of working SCE program execution times are significantly higher than compiled program execution times, which decreases the efficiency of interactive design processes. Therefore, several approaches for SCE program execution acceleration exist. SCE based parallel processing represents one of these approaches.

Despite the large number of dedicated software systems, SCE based parallel processing is not widely established in engineering domains today. Reasons for that are, on one hand, the lack of software system evaluation, especially for engineering domains. On the other hand, there is little knowledge about parallel SCE program development which enables predictions of estimated speedup and parallelization effort.

The following work aims the further establishment of SCE-based parallel processing in engineering domains. As a follow-up of Pawletta's work, the current development state in this area is presented and a new taxonomy is proposed. It is shown that out of this new taxonomy, the Multi-SCE approach is particularly of interest for engineering domains. Available Multi-SCE systems are compared with respect to qualitative and quantitative characteristics. Furthermore, self-developed Multi-SCE systems, particularly focussed on industrial domains are presented. Based on six engineering applications, selected Multi-SCE systems are compared regarding speedup and parallelization effort.

Danksagung

Die vorliegende Arbeit entstand im Rahmen eines kooperativen Promotionsprojektes zwischen der Forschungsgruppe Computational Engineering und Automation der Hochschule Wismar und dem Institut für Automatisierungstechnik der Universität Rostock. Finanziell wurde das Projekt in Teilen durch das Bundesministerium für Bildung und Forschung sowie das Bildungsministerium Mecklenburg-Vorpommern[1] unterstützt. Die praktischen Anwendungsbeispiele wurden in Kooperation mit der Ingenieurgesellschaft Auto und Verkehr, IAV GmbH, sowie der Daimler Chrysler AG entwickelt.

Mein Dank gilt allen, die diese Arbeit ermöglicht und unterstützt haben. Insbesondere danke ich meinen Betreuern, Prof. Dr.-Ing. Sven Pawletta und Prof. Dr.-Ing. habil. Dr. h.c. Bernhard Lampe sowie den weiteren Professoren und Mitarbeitern der Forschungsgruppe Computational Engineering und Automation der Hochschule Wismar, ohne deren Initiative und Unterstützung dieses Projekt nicht hätte durchgeführt werden können. Darüber hinaus danke ich meiner Frau Bianca, die durch ihre Geduld und die von ihr ausgehende Motivation einen wesentlichen Beitrag zum Entstehen der vorliegenden Arbeit geleistet hat.

[1] HWP310 - FHW13

Inhaltsverzeichnis

1 Einleitung

Ingenieurtechnische Problemstellungen werden heute zu einem großen Teil mit Hilfe von Computern bewältigt. Als wichtige Hilfsmittel auf diesem Gebiet haben sich Softwaresysteme etabliert, mit denen wissenschaftliche beziehungsweise technische Berechnungsprogramme interaktiv entwickelt, ausgeführt und ausgewertet werden können. Für die weite Verbreitung derartiger Systeme gibt es mehrere Gründe. Zum Einen stellen sie dem Anwender spezielle Algorithmen für viele technische und wissenschaftliche Disziplinen bereit. Zum Anderen ermöglicht ihre interpretative Arbeitsweise einen interaktiven Zugriff auf diese Algorithmen und unterstützt somit das schnelle Entwickeln von Softwareprototypen.

Das heute am weitesten verbreitete System dieser Art ist Matlab, entwickelt durch The MathWorks Inc., dessen dominante Stellung zur Bildung des Oberbegriffs *Matlab-like Systems* führte. Seitens The MathWorks Inc. wird Matlab als *scientific and technical computing environment* bezeichnet ([27]), wovon sich die im Rahmen der vorliegenden Arbeit verwendete Abkürzung *SCE* sowie die deutsche Bezeichnung *wissenschaftlich-technische Berechnungsumgebung* als Oberbegriff für derartige Softwaresysteme ableitet.

Durch die interpretative Arbeitsweise von wissenschaftlich-technischen Berechnungsumgebungen ergibt sich für SCE-Programme grundsätzlich eine geringere Ausführungsgeschwindigkeit gegenüber kompilierten Programmen. Dieser Nachteil ist insbesondere für interaktive ingenieurtechnische Entwurfsprozesse kritisch, da hier eine kurze Antwortzeit von Applikationen notwendig ist. Es existieren daher verschiedene Ansätze zur Beschleunigung von SCE-Programmen. Einen dieser Ansätze stellt die SCE-basierte Parallelverarbeitung dar.

Die ersten dokumentierten Versuche zur SCE-basierten Parallelverarbeitung erfolgten durch The MathWorks Inc. am System Matlab. In einem 1995 erschienenen Artikel berichtet Moler ([40]) über in den 80er Jahren durchgeführte Versuche mit parallelen Matlab-Versionen. Diese frühen Versuche lieferten aufgrund der zu geringen Granularität der verwendeten parallelen Algorithmen negative Ergebnisse, sodass die Entwicklung seitens The MathWorks eingestellt wurde. Moler weist in seinem Artikel darauf hin, dass eine sinnvolle Anwendung Matlab-basierter Parallelverarbeitung für sehr grobgranulare Probleme, das heisst bei Parallelisierung auf höchster Programmebene, denkbar ist, aber wesentliche Veränderungen in der Matlab-Architektur erfordern würde.

In den 90er Jahren erfolgte im Rahmen verschiedener Forschungsprojekte die Entwicklung von Matlab-Toolboxen, die exakt auf dieses Segment fokussierten. Zu den frühesten Softwaresystemen dieser Art zählen die DP-Toolbox ([42]), MultiMATLAB ([44])sowie die PT Toolbox ([41]). In seiner 1998 erschienenen Dissertation fasst Pawletta die bis dahin erfolgte Entwicklung der SCE-basierten parallelen und verteilten Verarbeitung zusammen. Darüber hinaus wird erstmals der Multi-SCE-Ansatz vorgestellt, der die SCE-

basierte Parallelverarbeitung unter Nutzung mehrerer kooperierender SCE-Instanzen beschreibt. Als beispielhafte Multi-SCE-Implementierung wird dabei die *Distributed and Parallel Application Toolbox* (DP-Toolbox) präsentiert.

Zwischen 1998 und 2006 entstand eine Vielzahl weiterer Softwaresysteme zur SCE-basierten Parallelverarbeitung, die fast ausschließlich den Multi-SCE-Ansatz verfolgen. Einen vorläufigen Höhepunkt stellt die Entwicklung der *Distributed Computing Toolbox* (DC-Toolbox, [73]) seitens The MathWorks Inc. dar, die 2004, also circa zehn Jahre nach Molers kritischem Artikel veröffentlicht wurde.

Trotz dieser starken Entwicklungsdynamik ist die SCE-basierte Parallelverarbeitung in ingenieurtechnischen Bereichen bis heute kaum verbreitet. Die Ursachen dafür sind vielfältig. So besteht zum Einen ein Überangebot an Softwaresystemen, für die bisher kaum Vergleichskriterien existieren. Zum Anderen sind die betreffenden Softwaresysteme nur selten für ingenieurtechnische Anwendungen evaluiert. Drittens gibt es bisher nur wenig Erkenntnisse über die Entwicklung paralleler SCE-Anwendungen, so dass eine Abschätzung von Aufwand und Nutzen derzeit kaum möglich ist.

Die vorliegende Arbeit hat das Ziel, den Einsatz der SCE-basierten Parallelverarbeitung unter Verwendung des Multi-SCE-Ansatzes in ingenieurtechnischen Bereichen weiter zu etablieren. In verschiedenen Punkten wird dabei auf der Arbeit von Pawletta aufgebaut. So erfolgt eine Analyse des Entwicklungsstandes sowie der Entwurf einer neuen vereinheitlichenden Klassifikation zur SCE-basierten Parallelverarbeitung. Darüber hinaus werden bestehende Multi-SCE-Systeme miteinander verglichen und Ansätze für die Weiterentwicklung der DP-Toolbox identifiziert. Die weiterentwickelte DP-Toolbox sowie weitere experimentelle Systeme werden präsentiert und mit bestehenden Multi-SCE-Systemen verglichen. In einem anwendungsbezogenen Vergleich von Multi-SCE-Systemen wird die Praxistauglichkeit verschiedener Systeme untersucht. Es werden Erkenntnisse für die Anwendungsentwicklung gewonnen, wobei bestimmte Anwendungsmerkmale zur Abschätzung von Entwicklungsaufwand und Laufzeitgewinn betrachtet werden.

In Kapitel 2 werden zunächst die für die späteren Ausführungen notwendigen Grundlagen der Parallelverarbeitung eingeführt. Anschließend erfolgt in Kapitel 3 eine einführende Betrachtung zu wissenschaftlich-technischen Berechnungsumgebungen, in der auch die verschiedenen Möglichkeiten zur Beschleunigung von SCE-Programmen aufgezeigt werden.

Kapitel 4 befasst sich mit der SCE-basierten Parallelverarbeitung im Allgemeinen. Darin wird zunächst eine Klassifikation entwickelt, die alle bisher bekannten Klassifikationen zu dieser Thematik in sich vereint. Anschließend erfolgt die Einordnung prinzipieller Techniken und konkreter Softwaresysteme zur SCE-basierten Parallelverarbeitung in die neue Klassifikation. Eine abschließende Betrachtung verdeutlicht die Gründe für die nachfolgende Fokussierung auf die Klasse der Multi-SCE-Systeme.

In Kapitel 5 werden verfügbare Multi-SCE-Systeme hinsichtlich verschiedener Kriterien miteinander verglichen. Einerseits erfolgt der Vergleich anhand bestimmter qualitativer Eigenschaften, zum Anderen wird ein quantitativer Vergleich anhand von Messungen zur Kommunikationsleistung der Systeme vorgenommen.

Die Weiterentwicklung der DP-Toolbox ist Gegenstand von Kapitel 6. Darin wird

die weiterentwickelte DP-Toolbox als ein System für industrielle Anwendungen vorgestellt. Daneben erfolgt die Präsentation eines experimentellen Multi-SCE-Systems sowie weiterer prototypischer Implementierungen. Abschließend erfolgt ein Vergleich der entwickelten Systeme in Analogie zu Kapitel 5.

Kapitel 7 befasst sich mit der anwendungsorientierten Untersuchung von Multi-SCE-Systemen. Darin erfolgt die Präsentation von sechs Anwendungen, die jeweils mit drei ausgewählten Multi-SCE-Systemen parallelisiert wurden. Die Anwendungen umfassen sowohl Benchmarkprobleme als auch industrielle Anwendungen und stellen Optimierungs- beziehungsweise Simulationsprobleme dar. Der Vergleich der Systeme erfolgt einerseits hinsichtlich des erzielten Laufzeitgewinns, andererseits aber auch bezüglich des notwendigen Parallelisierungsaufwands. Darüber hinaus wird ein direkter Vergleich der Anwendungen vorgenommen, aus dem allgemeine Aussagen über den zu erwartenden Parallelisierungsaufwand und Laufzeitgewinn bestimmter Anwendungstypen abgeleitet werden.

Der Umfang der anwendungsorientierten Betrachtungen zeigt, dass bei der vorliegenden Arbeit die Nutzung der SCE-basierten Parallelverarbeitung in ingenieurtechnischen Anwendungen im Vordergrund steht. Die Parallelverarbeitung selbst ist dabei nicht der primäre Betrachtungsgegenstand, sondern wird in diesem Sinne als ein Mittel zur Programmbeschleunigung angesehen, das es effektiv anzuwenden gilt.

2 Grundlagen der Parallelverarbeitung

Die Parallelverarbeitung verfolgt das Ziel, durch Vervielfachung von Verarbeitungselementen rechentechnische Probleme in kürzerer Zeit zu lösen, als dies mit nur einem Verarbeitungselement möglich ist. Die Grundidee besteht darin, ein Problem in Teilprobleme zu zerlegen, die durch die Verarbeitungselemente zeitgleich und koordiniert, das heisst parallel, bearbeitet werden. Neben der Problemlösung in kürzerer Zeit ist die Lösung komplexerer Probleme in gleicher Zeit (gegenüber der sequentiellen Verarbeitung) eine weitere, oft genannte Zielstellung der Parallelverarbeitung.

Das Gebiet der Parallelverarbeitung umfasst ein breites Spektrum an Teilbereichen, die unter anderem die Hardwarearchitektur, parallele Algorithmen, die Erstellung paralleler Programme und die Laufzeit- und Leistungsanalyse paralleler Programme umfassen. Aus dieser Menge werden im Folgenden die für die vorliegende Arbeit relevanten Aspekte Hardwarearchitekturen, Programmiermodelle und Leistungsbewertung diskutiert.

2.1 Hardwarearchitekturen

2.1.1 Klassifikation

In der Literatur findet sich eine Vielzahl an Taxonomien (z.B. [3, 5, 7]), die Parallelverarbeitungshardware auf unterschiedlichen Abstraktionsniveaus klassifizieren. Eine weit verbreitete Klassifikation, die häufig als Einstiegspunkt in die Thematik genutzt wird, repräsentiert die Klassifikation nach Flynn ([1]). Flynns Taxonomie besitzt einen vergleichsweise hohen Abstraktionsgrad gegenüber realer Parallelverarbeitungshardware, sodass verschiedene Ansätze für ihre Verfeinerung existieren. Ein häufig verwendeter Verfeinerungsansatz für Flynns Taxonomie ist die Klassifikation nach Art der Speicherorganisation. Die Kombination beider Klassifikationen wird unter anderem zur Einordnung der heute leistungsstärksten Parallelverarbeitungsplattformen durch van der Steen und Dongarra ([18]) herangezogen.

2.1.1.1 Klassifikation nach Flynn

Die Flynnsche Klassifikation ist einer der frühesten Ansätze, parallele Rechnerarchitekturen zu unterscheiden. Sie ermöglicht eine relativ grobe Einteilung, wobei vor allem die Abgrenzung gegenüber der sequentiellen Verarbeitung deutlich wird.

Flynn unterscheidet Architekturen anhand der Behandlung von Daten- und Befehlsströmen, wobei die Ströme als unabhängig voneinander betrachtet werden. Die Verarbeitung der Ströme wird von Verarbeitungselementen (processing elements) übernommen.

Diesen wird jeweils ein Befehls- und Datenstrom zugeführt und ein Datenstrom entnommen. Nach Flynn ergeben sich daraus die folgenden vier Hardwareklassen (s. Abb. 2.1):

SISD: single instruction stream, single data stream

SIMD: single instruction stream, multiple data streams

MISD: multiple instruction streams, single data stream

MIMD: multiple instruction streams, multiple data streams

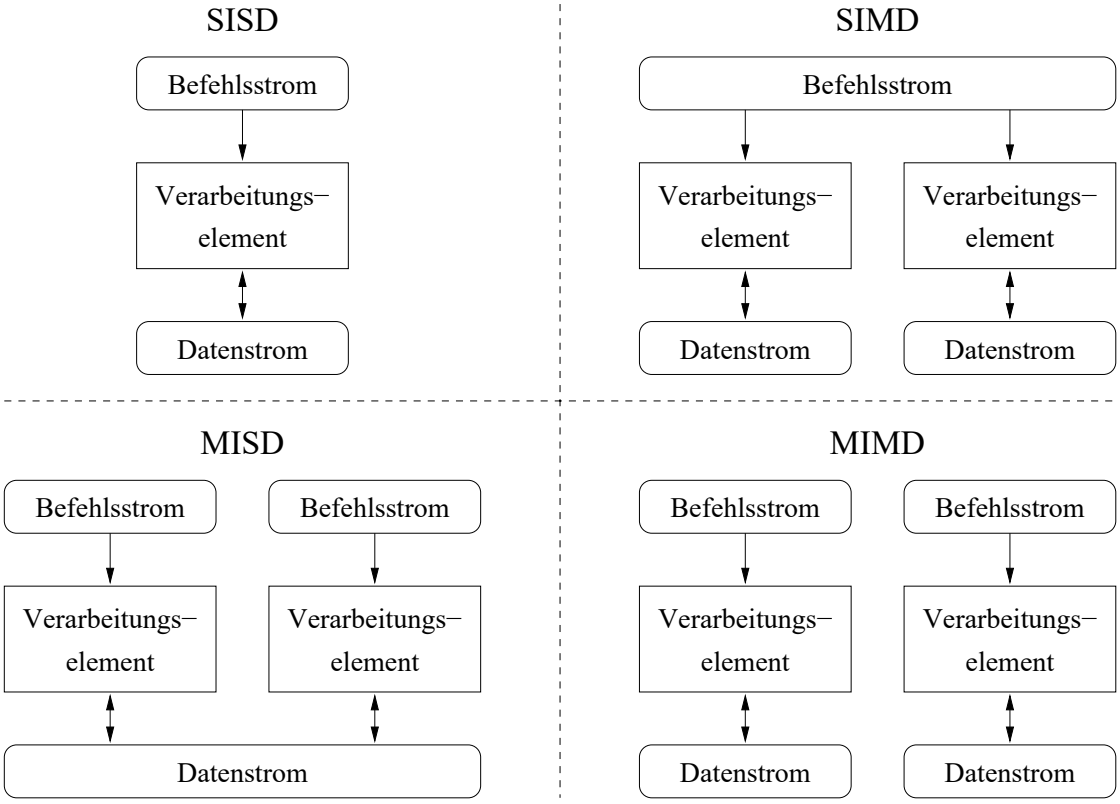


Abbildung 2.1: Struktur der Flynnschen Hardwareklassen

In *SISD*-Systemen wird ein Befehlsstrom mittels eines Verarbeitungselements auf einen Datenstrom angewandt. Dieses Verarbeitungsprinzip entspricht dem klassischen von-Neumann-Prinzip und enthält keinerlei Parallelität. Die Flynnsche Klassifikation enthält die SISD-Klasse als Abgrenzung sequentieller Hardwarestrukturen gegenüber parallelen Strukturen. Systeme der *SIMD*-Klasse wenden einen Befehlsstrom durch mehrere Verarbeitungselemente gleichzeitig auf mehrere Datenströme an. Da nur ein Befehlsstrom existiert, erfolgt die Abarbeitung durch die Verarbeitungselemente synchron. *MISD*-Systeme wenden mehrere Befehlsströme durch mehrere Verarbeitungselemente

gleichzeitig auf einen Datenstrom an. In der Literatur wird diese Klasse häufig als leer angenommen, da die Arbeitsweise von MISD-Systemen zu keinem eindeutigen Ergebnis führen würde. Flynn selbst sieht dagegen die MISD-Klasse als nicht leer an ([13]). *MIMD*-Systeme wenden mehrere Befehlsströme mittels mehrerer Verarbeitungselemente auf mehrere Datenströme an. Da mehrere Befehlsströme existieren, kann die Abarbeitung hier asynchron erfolgen.

Obwohl Flynns Taxonomie ursprünglich zur Klassifikation von Hardwarearchitekturen entworfen wurde, ist es aufgrund ihres hohen Abstraktionsgrades möglich, sie auch auf der Softwareebene anzuwenden, wie durch Pawletta ([47]) erfolgt.

2.1.1.2 Klassifikation nach Art der Speicherorganisation

Prinzipiell wird bei der Art der Speicherorganisation zwischen zwei Klassen unterschieden:

Systeme mit gemeinsamem Speicher: Mehrere Verarbeitungselemente arbeiten über einem gemeinsamen Speicherbereich.

Systeme mit verteiltem Speicher: Mehrere Verarbeitungselemente arbeiten über individuellen Speicherbereichen und sind durch ein Verbindungsnetzwerk gekoppelt.

Eine Mischform aus beiden Klassen stellt die Kopplung mehrerer Systeme mit gemeinsamem Speicher mittels eines Verbindungsnetzwerkes dar. Dabei entsteht ein hierarchisches System aus gemeinsamem Speicher auf unterer Ebene und verteiltem Speicher auf höherer Ebene. Derartige Mischformen werden im Folgenden als *Systeme mit hybridem Speicher* bezeichnet (s. Abb. 2.2).

Die Art der Speicherorganisation kann wie Flynns Taxonomie zur Klassifikation von sowohl Hard- als auch Softwarestrukturen herangezogen werden. Auf Ebene der Hardware spricht man daher auch von physikalischer Speicherorganisation, während auf Softwareebene von logischer Speicherorganisation gesprochen wird. Die logische Speicherorganisation stellt dabei die Sicht des Programmierers auf das System dar und hat direkte Auswirkungen auf das anzuwendende parallele Programmiermodell (s. Abschn. 2.2).

2.1.2 Reale Hardware

Nicht alle Klassen aus Flynns Taxonomie sind in der praktischen Parallelverarbeitung von Bedeutung. So besitzt die SISD-Klasse aufgrund der fehlenden Parallelität keine Relevanz, während das Verarbeitungsprinzip der MISD-Klasse bisher durch keine Hardwarearchitektur realisiert wurde. Die SIMD-Klasse war bis in die 90er Jahre in der Parallelverarbeitung von Bedeutung, verlor jedoch seit dem stetig an Relevanz. Heute sind SIMD-Architekturen vor allem in Spezialbereichen, zum Beispiel als Grafik- oder Signalprozessoren, oder in Form von Erweiterungen herkömmlicher Prozessoren, wie MMX[1] oder SSE[2], verbreitet. In der praktischen Parallelverarbeitung ist heute einzig die MIMD-Klasse von Bedeutung.

[1] Multi Media Extensions
[2] Internet Streaming SIMD Extensions

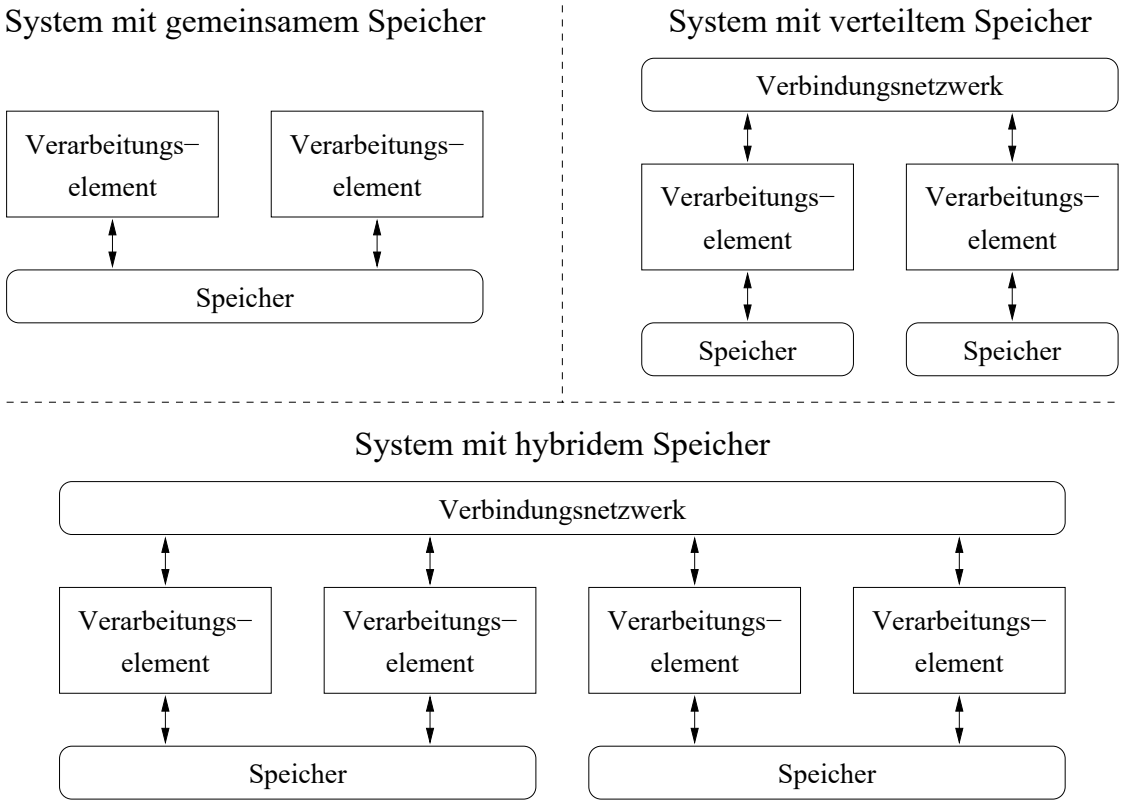


Abbildung 2.2: Arten der Speicherorganisation

Zur exakteren Klassifikation aktueller Hardwarearchitekturen wird die MIMD-Klasse häufig bezüglich der Art der Speicherorganisation weiter untergliedert (s. [9, 14, 17]), sodass zwischen MIMD-Systemen mit gemeinsamem, verteiltem oder hybridem Speicher unterschieden werden kann. Die Art der Speicherorganisation beeinflusst dabei stark die Skalierbarkeit des Systems und die Kommunikationsleistung zwischen den Verarbeitungselementen. MIMD-Systeme mit gemeinsamem Speicher besitzen durch die Speicherkopplung eine vergleichsweise hohe Kommunikationsleistung, während ihre Skalierbarkeit aufgrund des gemeinsamen Speicherzugriffs begrenzt ist. MIMD-Systeme mit verteiltem Speicher verfügen dagegen durch ihr Verbindungsnetzwerk über eine hohe Skalierbarkeit bei vergleichsweise geringer Kommunikationsleistung. Die Nachteile beider Hardwareklassen werden durch MIMD-Systeme mit hybridem Speicher kompensiert. Hier besteht eine hohe Kommunikationsleistung durch Speicherkopplung auf unterer Hierarchieebene, während die Kopplung mittels Verbindungsnetzwerk eine hohe Skalierbarkeit auf oberer Ebene gewährleistet.

In der praktischen Parallelverarbeitung werden statt den Klassifikationstermini häufig gängigere Begriffe zur Spezifikation bestimmter Architekturtypen verwendet. Für eine breite Übersicht über reale Parallelverarbeitungshardware folgt hier eine Analyse von Architekturtypen und entsprechenden Klassen für den Bereich kleiner Installationen von Parallelverarbeitungsplattformen (Low-End-Bereich) und den Bereich des Supercompu-

ting (High-End-Bereich).

2.1.2.1 Hardware im Low-End-Bereich

Zur Übersicht über Hardwarearchitekturen im Low-End-Bereich der Parallelverarbeitung wurden die zwischen 1994 und 2003 eingereichten Lösungsbeiträge zur SNE Comparison Parallel ([84]) analysiert. Die detaillierten Untersuchungsergebnisse finden sich in [91]. Für den Low-End-Bereich wurden drei relevante Architekturtypen identifiziert:

- Computercluster
- Symmetrische Multiprozessoren (SMP)
- Transputer

Computercluster bestehen aus unabhängigen Prozessoren mit eigenem physikalischen Speicher, gekoppelt durch ein herkömmliches Verbindungsnetzwerk, wie zum Beispiel Ethernet. Sie können als MIMD-Systeme mit verteiltem Speicher klassifiziert werden. *Symmetrische Multiprozessoren* bestehen aus mehreren homogenen Prozessoren mit einem gemeinsamen Speicher. Sie gehören somit zur Klasse der MIMD-Systeme mit gemeinsamem Speicher. *Transputer* bestehen aus mehreren Prozessoren mit zusätzlicher Hardware zur schnellen Prozessorkopplung. Sie gehören zur Klasse der MIMD-Systeme mit verteiltem Speicher. Transputer waren während der 90er Jahre im Low-End-Bereich weit verbreitet, sind heute aber nicht mehr von Bedeutung.

2.1.2.2 Hardware im High-End-Bereich

Für eine Übersicht über Hardwarearchitekturen im High-End-Bereich der Parallelverarbeitung wurde die Zusammenfassung der aktuellen Top500-Liste[3] ([26]) analysiert. Es wurden ebenfalls drei Architekturtypen identifiziert:

- Computercluster
- Constellations
- MPP[4]-Systeme

Computercluster im High-End-Bereich unterscheiden sich von denen im Low-End-Bereich durch die Verwendung schneller Verbindungsnetzwerke, wie zum Beispiel Myrinet. *Constellations* bestehen aus mehreren symmetrischen Multiprozessoren, ebenfalls gekoppelt über ein schnelles Verbindungsnetzwerk. Constellations gehören zur Klasse der MIMD-Rechner mit hybridem Speicher. *MPP-Systeme* besitzen eine sehr große Prozessorzahl (häufig mehr als eintausend Prozessoren) mit gemeinsamem oder verteiltem physikalischen Speicher, gekoppelt durch schnelle Verbindungsnetzwerke in mehreren Ebenen. Sie gehören zur Klasse der MIMD-Systeme mit verteiltem oder hybridem Speicher.

[3] In der Top500-Liste sind die nach dem LINPACK-Benchmark ([20]) 500 leistungsfähigsten Computer aufgeführt.

[4] Massively Parallel Processing

2.1.2.3 Zusammenfassung

In der praktischen Parallelverarbeitung ist heute nur die MIMD-Klasse nach Flynns Taxonomie relevant. Aktuelle Rechnerarchitekturen unterscheiden sich im Rahmen der aufgeführten Klassifikationen somit nur in der Art der Speicherorganisation. Im Low-End-Bereich dominieren MIMD-Systeme mit verteiltem und gemeinsamem Speicher, während im High-End-Bereich vorwiegend MIMD-Systeme mit verteiltem und hybridem Speicher zu finden sind. Systeme mit gemeinsamem Speicher sind aufgrund ihrer begrenzten Skalierbarkeit heute nicht mehr in der Top500-Liste enthalten. Aufgrund der aktuellen Entwicklung von Multikernprozessoren[5] für den Endverbraucherbereich ist anzunehmen, dass MIMD-Systeme mit hybridem Speicher sich in Zukunft auch im Low-End-Bereich durchsetzen werden, da auf diese Weise eine kostengünstige Installation von Constellations möglich wird.

2.2 Programmiermodelle

Ein paralleles Programmiermodell stellt die Schnittstelle zwischen dem Programmierer und der darunter liegenden Hardware dar. Es dient sowohl der vereinfachten Programmierung durch die Abstraktion von Hardwarestrukturen als auch der Etablierung von Programmierstandards. Parallele Programmiermodelle können auf der Ebene von Programmiersprachen, Compilern oder Bibliotheken unterstützt werden. Ein wesentliches Unterscheidungsmerkmal paralleler Programmiermodelle ist die implizite beziehungsweise explizite Darstellung von Aufgaben der parallelen Programmierung. Nach Skillicorn und Talia ([15]) umfasst die Programmierung paralleler Programme die folgenden Aufgabenbereiche:

Zerlegung: Die Zerlegung eines Gesamtproblems in Teilaufgaben, häufig durch Zerlegung einer Datenstruktur. Dazu zählt auch die Zerlegung eines ursprünglich sequentiellen Befehlsstroms in mehrere parallele Befehlsströme.

Prozessorzuordnung (Mapping): Die Zuordnung paralleler Befehlsströme (Prozesse) zu vorhandenen Prozessoren. Dazu zählt auch die Instanziierung von Prozessen.

Kommunikation: Der Datenaustausch zwischen den Prozessen als Mittel zur koordinierten Zusammenarbeit.

Synchronisation: Das Sicherstellen, dass sich Prozesse in einem bestimmten Zustand befinden als Mittel zur koordinierten Zusammenarbeit.

Werden in einem Programmiermodell alle Aufgaben automatisch erfüllt, spricht man von einem impliziten Modell. Setzt ein Programmiermodell dagegen die Erfüllung einzelner oder mehrerer Aufgabenbereiche durch den Programmierer voraus, so handelt es sich um ein explizites Modell. Im Folgenden werden verschiedene explizite und implizite Programmiermodelle vorgestellt.

[5] Multikernprozessor (Multi-Core-CPU): mehrere Prozessorkerne in einem Schaltkreis integriert

2.2.1 Implizite Programmiermodelle

Implizite Programmiermodelle besitzen den höchstmöglichen Abstraktionsgrad, da ein Programm keine Details der parallelen Ausführung enthält und der Programmierer wie bei der Entwicklung eines sequentiellen Programms verfahren kann. Allerdings können durch das Verbergen aller parallelverarbeitungstechnischen Details häufig nicht alle Möglichkeiten der Plattform ausgeschöpft werden.

2.2.1.1 Parallelisierende Compiler

Ein parallelisierender Compiler erzeugt aus einem sequentiellen Programm automatisch parallelen Code. Er muss dafür die Abhängigkeiten zwischen den durchzuführenden Berechnungen erkennen und diese effektiv in Teilaufgaben zerlegen können. Die Bewältigung dieser zum Teil sehr komplexen Aufgabe überfordert parallelisierende Compiler jedoch oft, sodass sie unbefriedigende Ergebnisse liefern.

Parallelisierende Compiler existierten zunächst nur für SIMD-Plattformen und wurden daher als *vektorisierende* Compiler bezeichnet. Heutige Compiler können Programme für SIMD-, MIMD-, und Mischplattformen erzeugen. Bedeutende Vertreter sind die Compilerfamilien von The Portland Group ([25]) oder Intel ([22, 21]).

2.2.1.2 Datenparallele Programmierung

Die datenparallele Programmierung basiert auf der Ausführung von gleichen, voneinander unabhängigen Operationen auf mehreren Elementen einer Datenstruktur. Im einfachsten Fall wird eine Vektoroperation auf alle Elemente eines Feldes angewandt, wobei die gegenseitige Unabhängigkeit der Operationen die gleichzeitige Abarbeitung erlaubt. Für die Entwicklung datenparalleler Programme ist eine spezielle Programmiersprache erforderlich, die die üblichen Operationen sequentieller Programmiersprachen zu Vektoroperationen erweitert.

Die datenparallele Programmierung ist stark an das Verarbeitungsprinzip von SIMD-Plattformen angelehnt, welches Vektoroperationen auf Hardwareebene ausführt. Prinzipiell lassen sich datenparallele Programme aber für SIMD- sowie MIMD-Plattformen übersetzen. Die datenparallele Programmierung wird häufig in Zusammenhang mit parallelisierenden Compilern benutzt, wobei die Aufgabe der Vektorisierung hier für den Compiler entfällt. Vertreter datenparalleler Programmiersprachen sind Fortran 90 oder Dataparallel C.

2.2.1.3 Programmierung mit parallelen Bibliotheken

Die Programmierung mit parallelen Bibliotheken wird in der Literatur oft nicht als paralleles Programmiermodell betrachtet, da die parallel ablaufenden Programmabschnitte nicht von Programmierer selbst entworfen werden. Im Rahmen der vorliegenden Arbeit ist es jedoch notwendig, auch diese Programmiertechnik in die Klasse der impliziten parallelen Programmiermodelle aufzunehmen. Darüber hinaus wird wie bei allen anderen

parallelen Programmiermodellen auch hier eine standardisierte Schnittstelle zwischen Programmierer und beliebiger Parallelverarbeitungsplattform bereitgestellt.

Bei der Programmierung mit parallelen Bibliotheken werden rechenintensive Programmabschnitte durch spezielle parallele Standardroutinen ausgeführt. Der Programmierer muss dabei lediglich die Aufrufsemantik einer Routine kennen, während die Aufgaben der parallelen Programmierung (Zerlegung, Mapping, Kommunikation, Synchronisation) innerhalb der Routine erledigt werden. Die Verwendung paralleler Bibliotheken kann aus einer beliebigen Programmiersprache heraus erfolgen, solange eine Aufrufschnittstelle für die Sprache existiert.

Parallele Bibliotheken stehen für SIMD- und MIMD-Plattformen zur Verfügung und sind in numerischen Anwendungen weit verbreitet. Wichtige Vertreter für MIMD-Systeme sind ScaLAPACK ([12]), sowie die NAG Parallel Library ([24]). Die Numerikbibliotheken ACML von AMD ([19]) sowie MKL von Intel ([23]) stellen durch Vektorfunktionen Schnittstellen zur Nutzung von SIMD-Erweiterungen, wie zum Beispiel SSE bereit. Darüber hinaus existiert eine Vielzahl anwendungsspezifischer paralleler Bibliotheken, die meist nur für eine spezielle Applikation entwickelt werden.

2.2.2 Explizite Programmiermodelle

Im Folgenden werden drei explizite Programmiermodelle, die für die vorliegende Arbeit von besonderer Bedeutung sind, vorgestellt: Message-Passing-Programmierung, Shared-Memory-Programmierung und RPC[6]-Programmierung. Die Modelle unterscheiden sich durch verschiedene Mengen explizit wahrzunehmender Programmieraufgaben. Signifikante Unterschiede bestehen in der expliziten beziehungsweise impliziten Ausführung von Kommunikations- und Synchronisationsaufgaben. Hinsichtlich Zerlegung und Prozessmapping existieren dagegen starke Ähnlichkeiten: die Zerlegung in Teilaufgaben muss in jedem Programmiermodell explizit vorgenommen werden, wohingegen die Prozessorzuordnung implizit *oder* explizit erfolgt und eine eher untergeordnete Rolle spielt.

Eine weitere Gemeinsamkeit der Modelle ist die Fokussierung auf MIMD-Systeme. Diese Hardwarearchitektur setzt aufgrund ihrer unterschiedlichen Verarbeitungsströme ein Modell aus mehreren, asynchron ablaufenden Prozessen voraus, die, um eine gemeinsame Aufgabe zu bewältigen, über Koordinationsmechanismen verfügen müssen. Tabelle 2.1 fasst die Gemeinsamkeiten und Unterschiede dieser drei Modelle zusammen.

Programmiermodell	Zerlegung	Mapping	Kommunikation	Synchronisation
Message Passing	explizit	impl. o. expl.	explizit	implizit
Shared Memory	explizit	impl. o. expl.	implizit	explizit
RPC	explizit	impl. o. expl.	implizit	implizit

Tabelle 2.1: Programmieraufgaben in expliziten parallelen Programmiermodellen

[6] Remote Procedure Call

2.2.2.1 Message-Passing-Programmierung

Das Message-Passing-Programmiermodell geht von einer Menge von Prozessen aus, die jeweils über ihrem lokalen Speicher arbeiten. Es eignet sich somit besonders für Plattformen mit logisch verteilter Speicherorganisation. Zur koordinierten Abarbeitung erfolgt zwischen den Prozessen eine explizite Kommunikation mittels Nachrichtenaustausch. Für die Übermittlung einer Nachricht muss auf der Seite des sendenden Prozesses eine Sendeoperation und auf der Seite des empfangenden Prozesses eine entsprechende Empfangsoperation ausgeführt werden. Die Synchronisation ergibt sich implizit aus der Kausalordnung des Nachrichtentransports: eine Empfangsoperation kann nur erfolgreich abgeschlossen werden, wenn zuvor die dazugehörige Sendeoperation ausgeführt wurde.

Wichtige Vertreter von Message-Passing-Systemen sind PVM[7], das in den 90er Jahren einen inoffiziellen Standard in der Parallelprogrammierung darstellte, und verschiedene Implementierungen des Message-Passing-Standards MPI[8], der PVM inzwischen abgelöst hat.

2.2.2.2 Shared-Memory-Programmierung

Die Shared-Memory-Programmierung basiert auf einem Modell von Prozessen, die neben ihrem lokalen Speicher auch Zugriff auf gemeinsame Speicherbereiche besitzen. Diese Eigenschaft macht die Shared-Memory-Programmierung zum bevorzugten Programmiermodell für Plattformen mit logisch gemeinsamer Speicherorganisation. Der gemeinsame Speicherbereich ist hierbei das Medium für eine implizite Prozesskommunikation. Um den Zugriff auf das Kommunikationsmedium zu regeln, sind explizite Synchronisationsoperationen erforderlich.

Ein einfacher und in der Parallelverarbeitung verbreiteter Synchronisationsmechanismus ist die *Barrier*-Synchronisation ([14], S.73 f.). Jeder Prozess, der während seiner Abarbeitung einen Barrier-Befehl erreicht, wartet dort solange, bis alle anderen Prozesse ebenfalls diese Barriere erreichen. Anschließend setzen alle Prozesse ihre Abarbeitung fort. Weitere Synchronisationsmechanismen, die weitestgehend der Betriebssystementwicklung entstammen, sind Semaphore, Mutex-Variablen oder Bedingungsvariablen ([4]).

Wichtige Vertreter von Shared-Memory-Programmiertechniken sind die Threadprogrammierung, die heute von vielen Betriebssystemen und Programmiersprachen unterstützt wird und der Standard OpenMP ([16]), der speziell für die Programmierung von Systemen mit logisch gemeinsamem Adressraum entwickelt wurde.

2.2.2.3 RPC-Programmierung

Die RPC-Programmierung stellt kein klassisches paralleles Programmiermodell dar, sondern stammt ursprünglich aus der verteilten Verarbeitung ([6]). Diese unterscheidet sich

[7] Parallel Virtual Machine [10]
[8] Message Passing Interface [11]

von der Parallelverarbeitung durch andere Zielstellungen wie zum Beispiel der Abbildung realer Systemstrukturen, der Erhöhung der Verfügbarkeit oder der Erhöhung der Dezentralisierung. Da die RPC-Programmierung im Rahmen der vorliegenden Arbeit von besonderer Relevanz ist, ihr in der üblichen Parallelverarbeitungsliteratur jedoch kaum Platz eingeräumt wird, soll an dieser Stelle ausführlicher auf dieses Modell eingegangen werden.

Das ursprüngliche RPC-Modell erweitert das Prinzip lokaler Funktionsaufrufe dahingehend, dass aufrufende Funktion und aufgerufene Funktion in unterschiedlichen Prozessen (oft auf unterschiedlichen Rechnern) residieren. Anders als in den vorhergehenden expliziten Programmiermodellen basiert das RPC-Programmiermodell auf einem Prozessmodell mit festem Rollenverhalten. Die aufrufende Funktion befindet sich im so genannten *Clientprozess*, während die aufgerufene Funktion Bestandteil des *Serverprozesses* ist.

Wie bei lokalen Funktionsaufrufen blockiert im klassischen RPC-Modell die aufrufende Funktion für die Dauer der Abarbeitung. In der verallgemeinerten Darstellung der Funktionsschnittstelle unterscheidet sich ein entfernter Prozeduraufruf nur durch einen den Serverprozess spezifizierenden Parameter vom lokalen Vorbild:

```
lokal:    rückgabewert=funktion(parameter1,parameter2,...)
entfernt: rückgabewert=funktion(server,parameter1,parameter2,...)
```

Die Analogie zu lokalen Funktionsaufrufen ist gleichzeitig der große Vorteil des RPC-Modells, da die Entwicklung von verteilten Anwendungen somit transparent und systemunabhängig wird.

Ein Schwerpunkt der verteilten Verarbeitung ist bis heute die Nutzung gemeinsamer Ressourcen. So stellt ein Serverprozess häufig einen Dienst bereit, der von mehreren Clientprozessen in Anspruch genommen wird. Um die RPC-Programmierung sinnvoll in der Parallelverarbeitung einzusetzen, muss dieses Schema invertiert werden: *ein* Clientprozess wird von *mehreren* Serverprozessen (gleichzeitig) bedient. Jeder Serverprozess bearbeitet somit eine Teilaufgabe des Gesamtproblems.

Die RPC-Programmierung lässt sich, übertragen auf die Parallelverarbeitung, den expliziten MIMD-Programmiermodellen zuordnen. Gegenüber der Message-Passing- und Shared-Memory-Programmierung erfolgt hier sowohl Kommunikation als auch Synchronisation implizit. Kommunikation findet lediglich bei der Übergabe von Aufrufparametern an einen Serverprozess sowie bei der Rückgabe von Ergebnisparametern an den Clientprozess statt. Somit besitzt das RPC-Modell als einziges betrachtetes Programmiermodell ein festes Interprozess-Kommunikationsschema: Kommunikation kann nur zwischen Client- und Serverprozess erfolgen (niemals zwischen zwei Serverprozessen) und beginnt immer auf der Clientseite. Die Synchronisation zwischen Prozessen erfolgt durch den Start und die Finalisierung der entfernten Prozeduren.

Die Forderung nach gleichzeitiger Abarbeitung entfernter Prozeduren macht eine Erweiterung des ursprünglichen RPC-Modells in asynchrones beziehungsweise vektorielles RPC notwendig. Asynchrone RPC-Mechanismen ([8]) basieren auf einem nichtblockierenden entfernten Prozeduraufruf, verbunden mit einer blockierenden Finalisie-

rungsoperation. Verallgemeinert lässt sich dieses Prinzip durch die folgende Funktionsschnittstelle darstellen:

```
Funktionsaufruf: handle=funktion(server,parameter1,parameter2,...)
Finalisierung:   rückgabewert=finalisiere(handle)
```

Ein nach diesem Modell arbeitendes Parallelprogramm muss zunächst alle Teilaufgaben in einer Programmschleife starten und anschließend in einer weiteren Schleife deren Ergebnisse auslesen. Als Abgrenzung gegenüber vektoriellem RPC wird dieses Prinzip im Folgenden als asynchrones *skalares* RPC bezeichnet.

Vektorielle RPC-Mechanismen ([47], S. 61 f.) beruhen dagegen auf der gleichzeitigen Ausführung einer Funktion in mehreren Serverprozessen mit jeweils unterschiedlichen Parametern. Der Funktionsaufruf eines vektoriellen RPC unterscheidet sich gegenüber dem herkömmlichen RPC-Aufruf durch die vektorielle Darstellung von Parametern und Rückgabewerten, ist jedoch im Gegensatz zum asynchronen RPC-Modell blockierend. Verallgemeinert kann vektorielles RPC durch die folgende Funktionsschnittstelle veranschaulicht werden:

```
rückgabewerte[]=funktion(server[],parameter1[],parameter2[],...)
```

Im Vergleich zum asynchronen RPC-Modell reduziert sich bei Verwendung von vektoriellem RPC der Programmieraufwand auf den *einmaligen*, zeitgleichen Aufruf aller entfernten Prozeduren. Gleichfalls als Abgrenzung gegenüber asynchronem RPC wird diese Technik im Folgenden als *synchron* vektorielles RPC bezeichnet.

Wegen ihres eingeschränkten Kommunikationsschemas (Kommunikation und Synchronisation nur zu Beginn und Ende der entfernten Funktionsabarbeitung) eignet sich die RPC-Programmierung nur für Probleme, bei denen innerhalb der Abarbeitung einer Teilaufgabe durch einen Prozess keine Kommunikation mit anderen Prozessen notwendig ist. Diese Art von Problemen wird in der Literatur als taskparallel ([17], S. 136 ff.) oder embarrassingly[9] parallel ([62]) bezeichnet. Die Verwendung dieses unwissenschaftlichen Terms ist allerdings nicht gerechtfertigt. Tatsächlich bergen derartige Probleme das größte Potential an Parallelität, da keine Koordination zwischen den Teilaufgaben erforderlich ist und somit keine synchronisationsbedingten Wartezeiten zwischen den Prozessen entstehen. Die RPC-Programmierung mit ihren parallelen Erweiterungen (asynchron/vektoriell) erweist sich für derartige Probleme als komfortables Programmiermodell.

2.3 Leistungsbewertung

Die Leistung von Parallelprogrammen kann anhand unterschiedlicher Kriterien bewertet werden. In erster Linie ist dabei das Laufzeitverhalten von Interesse, aber auch die Skalierbarkeit eines Programms oder die Auslastung der Prozessoren können ein Kriterium

[9] embarrassigly: peinlich, beschämend

zur Leistungsbewertung sein. Die Bewertung eines Programms kann dabei, je nach Verfahren, bereits in der Entwurfsphase erfolgen oder erst auf Basis von Laufzeitmessungen nach mehreren Programmläufen.

2.3.1 Granularität

Bei der qualitativen Abschätzung des Laufzeitverhaltens paralleler Programme spielt die Granularität eine entscheidende Rolle. Diese beschreibt das Verhältnis von Rechenaufwand und Kommunikationsaufwand im Parallelprogramm und kann bereits in der Entwurfsphase zur Laufzeitabschätzung benutzt werden.

Für eine Laufzeitabschätzung wird ein vorliegendes oder geplantes Programm als grob-, mittel- oder feingranular eingestuft und seine Granularität auf das Verhältnis von Rechen- und Kommunikationsleistung der ausführenden Parallelverarbeitungsplattform bezogen. Es wird somit geprüft, ob die Programmstruktur zur Plattform „passt“.

Im Allgemeinen eignen sich feingranulare Probleme eher für SIMD-Systeme und mittelgranulare Probleme für MIMD-Systeme mit gemeinsamem Speicher. Grobgranulare Probleme sind dagegen prädestiniert für MIMD-Systeme mit verteiltem Speicher und relativ langsamen Verbindungsnetzwerken.

2.3.2 Speedup und Effizienz

Die quantitative Bewertung eines Parallelprogramms erfolgt im einfachsten Fall anhand seiner gemessenen Laufzeit, verglichen mit der Laufzeit einer sequentiellen Programmvariante. Im günstigsten Fall läuft dabei ein paralleles Programm zum Beispiel auf zwei Prozessoren doppelt so schnell ab, wie sein sequentielles Pendant auf einem Prozessor.

Aus dieser pragmatischen Vorbetrachtung ergibt sich die Definition des Speedups. Dieser errechnet sich aus dem Verhältnis der Laufzeit der optimalen sequentiellen Implementierung eines Problems T^* zur Laufzeit einer parallelen Implementierung des Problems T_p, wobei p die Anzahl der Prozessoren spezifiziert:

$$S_p = \frac{T^*}{T_p}. \tag{2.1}$$

Durch den Overhead im parallelen Programm ist die Laufzeit des sequentiellen Programms (T^*) meist kleiner als die Laufzeit des parallelen Programms auf einem Prozessor (T_1). Um den Einfluss des Overheads bei der Laufzeitanalyse zu eliminieren, wird neben dem „tatsächlichen“ Speedup, also dem realen Laufzeitgewinn, der Kennwert des algorithmischen Speedups ($\overline{S}_p$) verwendet. Dabei wird statt der Laufzeit der sequentiellen Implementierung die Laufzeit der parallelen Implementierung auf einem Prozessor T_1 als Bezugsgröße verwendet.

Im Idealfall entspricht der Speedup der Anzahl der Prozessoren. Es sind jedoch auch Situationen denkbar, in denen $S_p > p$ ist, man spricht dann von *hyperlinearem* Speedup. Diese werden häufig durch Auslagerungseffekte verursacht, das heisst wenn eine

Datenstruktur im sequentiellen Programm zu groß ist um vollständig im Arbeitsspeicher gehalten zu werden, im parallelen Programm jedoch auf die Arbeitsspeicher aller beteiligten Rechner aufgeteilt werden kann ([14], S. 26).

Ein weiteres, auf dem Speedup beruhendes, qualitatives Bewertungsmaß stellt die Effizienz dar. Hierbei wird der Speedup S_p durch die Anzahl der Prozessoren p dividiert:

$$E_p = \frac{S_p}{p} = \frac{T^*}{p \cdot T_p}. \tag{2.2}$$

Die Effizient quantifiziert damit die durchschnittliche Auslastung aller beteiligten Prozessoren. Sie beträgt im Idealfall eins und im ungünstigsten Fall null, kann aber bei hyperlinearem Speedup auch Werte größer als eins annehmen.

Geht man von der Berechnungsformel des Speedups aus, so läßt sich dieser für genügend kleine parallele Abarbeitungszeiten bis ins Unendliche steigern. Diese Annahme wird durch das *Ahmdahlsche Gesetz* ([2]) entkräftet. Danach besteht ein sequentielles Programm zu einem bestimmten Anteil f $(0 \leq f \leq 1)$ aus Instruktionen, die sich nicht parallelisieren lassen und zu einem Anteil $1 - f$ aus solchen, die parallelisierbar sind. Die parallele Laufzeit T_p ergibt sich dann aus:

$$T_p = f \cdot T^* + (1 - f)\frac{T^*}{p} \ . \tag{2.3}$$

Für den Speedup S_p folgt:

$$S_p = \frac{T^*}{f \cdot T^* + (1 - f)\frac{T^*}{p}} = \frac{1}{f + \frac{1-f}{p}} \ . \tag{2.4}$$

Die Grenzwertbetrachtung für $p \to \infty$ ergibt nun den maximal erreichbaren Speedup:

$$\lim_{p \to \infty} S_p = \lim_{p \to \infty} \left(\frac{1}{f + \frac{1-f}{p}} \right) = \frac{1}{f} \ . \tag{2.5}$$

Das bedeutet, nur wenn der Anteil der nicht parallelisierbaren Instruktionen genügend klein ist, kann ein paralleler Laufzeitgewinn in sinnvollen Größenordnungen erreicht werden.

3 Wissenschaftlich-technische Berechnungsumgebungen (SCEs)

Als wissenschaftlich-technische Berechnungs- beziehungsweise Entwicklungsumgebung wird im Rahmen dieser Arbeit ein Softwaresystem verstanden, mit dem es möglich ist, wissenschaftliche beziehungsweise technische Anwendungen interpretativ zu entwickeln, auszuführen und auszuwerten.

Bisher hat sich in der Literatur kein einheitlicher Begriff für derartige Softwaresysteme durchgesetzt. Eine häufig verwendete umgangssprachliche Bezeichnung ist der Term *Matlab-like System*, da Matlab das am weitesten verbreitete Softwaresystem dieser Art ist. Da das Grundprinzip derartiger Softwareumgebungen unabhängig von konkreten Systemen ist, spiegelt dieser Term jedoch eine subjektive Sichtweise wider.

Werden die betrachteten Softwaresysteme ausschließlich mit numerischen Berechnungen assoziiert, so können sie als Gegenstück zur Klasse der Computeralgebrasysteme (CAS) angesehen werden. Durch F. Breitenecker wurde hierfür der Begriff *Computer Numerical System* (CNS) geprägt. Tatsächlich ist in der heutigen Softwareentwicklung eine Vermischung von interpretativen Systemen für numerische und symbolische Berechnungen zu beobachten, sodass eine Differenzierung anhand dieser Merkmale zunehmend schwieriger wird.

S. Pawletta leitet in [47] den Begriff *wissenschaftlich-technisches Berechnungs- und Visualisierungssystem* sowie die Abkürzung *SCE* aus der englischen Bezeichnung *scientific and technical computing environment for numeric computation and visualization*[1] ab. Da diese Systeme neben der Ausführung vor allem auch die effektive Erstellung von Programmen unterstützen, wird in [47] der Begriff *wissenschaftlich-technische Entwicklungsumgebung* als synonyme Bezeichnung verwendet. Das Hauptanliegen für die Einführung des SCE-Begriffes war die Schaffung einer geeigneten Klassenbezeichnung, die auch die Einordnung der inzwischen hybriden Systeme, in denen sowohl numerische als auch symbolische Methoden bereitgestellt werden, erlaubt.

In der vorliegenden Arbeit werden die in [47] eingeführten Bezeichnungen verwendet. Die Analyse realer SCEs sowie deren Verwendung zur Parallelverarbeitung beschränkt sich jedoch auf das numerische Rechnen.

3.1 Eigenschaften

Für die Durchführung wissenschaftlicher Berechnungen in kompilierbaren Sprachen wie Fortran oder C werden häufig zusätzliche Numerikbibliotheken verwendet. Um ein feh-

[1] Diese Bezeichnung wurde einer früheren Dokumentation des Systems Matlab ([27]) entnommen.

lerfreies Programm zu entwickeln muss dabei der Zyklus Programmieren-Übersetzen-Testen-Fehlersuche mehrere Male durchlaufen werden. Die Visualisierung der Ergebnisse erfordert darüber hinaus zusätzlichen Programmieraufwand oder den Export der Daten an ein weiteres Programm.

SCEs ermöglichen es dagegen, alle Entwicklungs- und Ausführungsschritte in *einer* Umgebung durchzuführen. Sie besitzen folgende Eigenschaften:

interpretative Arbeitsweise: Befehlszeilen werden unmittelbar nach ihrer Eingabe ausgeführt.

höhere Programmiersprache: Programme werden in einer so genannten *Very High Level Language* verfasst, die einen höheren Abstraktionsgrad als konventionelle Hochsprachen (Fortran oder C) besitzen.

integrierte Numerikbibliotheken: Algorithmen zur linearen Algebra, Signalverarbeitung, Simulation und Datenanalyse stehen zur Verfügung.

integrierte Visualisierungsfunktionen: Ergebnisse lassen sich innerhalb der Entwicklungsumgebung darstellen.

Die Programmiersprachen von SCEs weisen dabei folgende Eigenschaften auf:

dynamische Speicherverwaltung: Variablen können an jeder Stelle eines Programms angelegt und dimensioniert werden.

implizite Typisierung: Variablen können ohne explizite Spezifikation ihrer Datentypen angelegt werden.

Arrays als Basisdatentypen: Jedes Datum wird als multidimensionales Array abgelegt (Matrizen, Vektoren und Skalare sind somit Spezialfälle).

Datenparallelität: Auf Variablen können datenparallele Operationen angewandt werden.

3.2 Aufbau

Konventionelle SCEs arbeiten prinzipiell nach dem SISD-Modell: Ein Befehlsstrom wird mittels eines Verarbeitungselements auf einen Eingangsdatenstrom angewandt, anschließend werden die Ergebnisse einem Ausgangsdatenstrom zugeführt. Der in Abbildung 3.1 dargestellte prinzipielle Aufbau einer SCE spiegelt dieses Verarbeitungsprinzip wider.

Der Befehlsstrom und die Datenströme der SCE können über unterschiedliche Kanäle zu- oder abgeführt werden. So wird der Eingangsdatenstrom entweder über die Tastatur oder, bei größeren Datenmengen, über eine Datei eingelesen und dem Inputmodul zugeführt. Über die gleichen Kanäle nimmt ein Parser- und Interpretermodul den Befehlsstrom entgegen. Das Einlesen der Befehle über die Tastatur wird dabei als *interaktiver Modus*, das Einlesen über Dateien als *Stapelverarbeitungsmodus* bezeichnet. Analog dazu

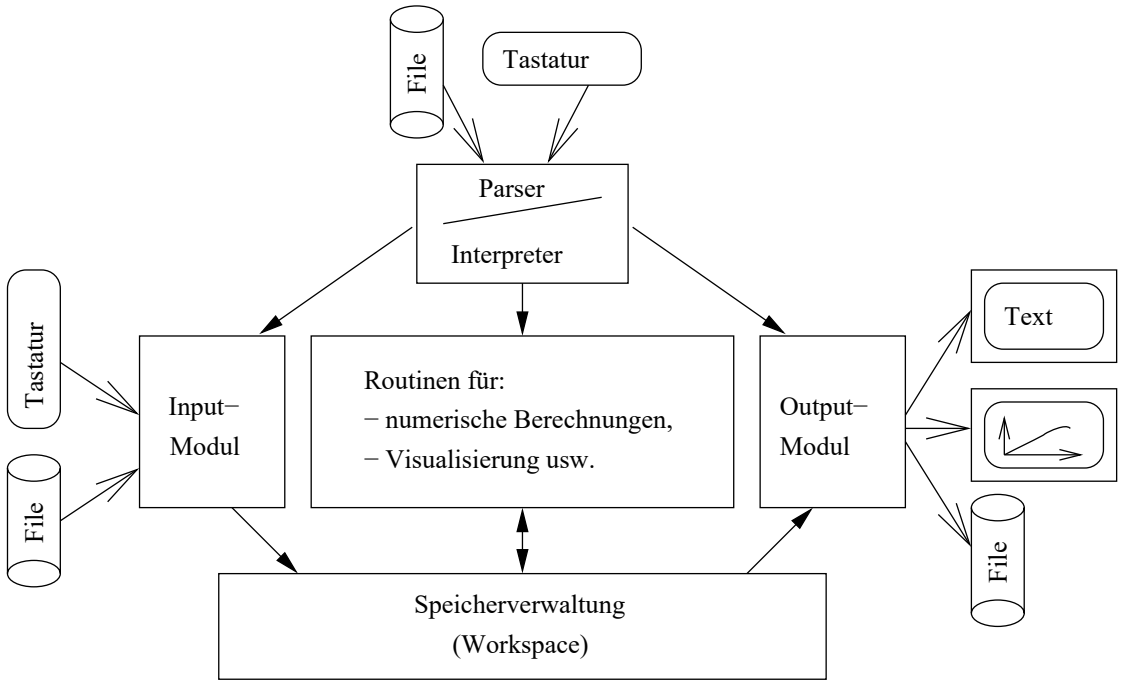


Abbildung 3.1: Prinzipieller Aufbau einer SCE nach Pawletta ([47])

kann der Ausgangsdatenstrom wahlweise als textuelle Bildschirmausgabe, in eine Datei oder als Grafik über das Ouputmodul ausgegeben werden.

Neben den Modulen zur Daten- und Befehlsstrombehandlung besitzen SCEs eine Speicherverwaltungskomponente, in der alle Variablen zusammen mit ihren Typ- und Dimensionsinformationen abgelegt sind. Die Speicherverwaltung der SCE sorgt darüber hinaus für die dynamische Allokation und Deallokation von Speicherbereichen, zum Beispiel beim Vergrößern oder Verkleinern von Arrays, und die Zuordnung von Variablennamen zu Speicheradressen. Die Variablen selbst residieren in unterschiedlichen Variablenräumen (*Workspaces*), welche ihre Sichtbarkeit (und somit den Zugriff) auf unterschiedlichen Programmebenen festlegen. Die folgenden, durch die SCE Matlab verwendeten, Variablenräume sind in vielen Systemen unter abweichenden Bezeichnungen wiederzufinden:

Base-Workspace: enthält Variablen der höchsten Programmebene

Function-Workspace: enthält lokale Funktionsvariablen

Global-Workspace: enthält globale Variablen

Das Kernstück einer SCE bildet ein Berechnungsmodul, welches den Zugriff auf integrierte Numerik- und Visualisierungsroutinen ermöglicht. Die Bibliotheken liegen je nach Komplexität und Rechenaufwand entweder in der SCE-eigenen Programmiersprache oder in kompiliertem Maschinencode vor.

Der Funktionsumfang einer SCE kann durch den Anwender erweitert werden. Auf diese Weise ist die Verwendung zusätzlicher numerischer Routinen möglich, aber auch die Bereitstellung zusätzlicher Schnittstellen in der SCE. Wie im Berechnungsmodul der SCE können auch hier die Funktionen in SCE-Code oder Maschinencode vorliegen. Zur Einbindung von Maschinencode muss häufig eine Anpassung zwischen nativer Schnittstelle und SCE-Schnittstelle über SCE-spezifische Kapselungsfunktionen (z.B. Matlab: mex, Octave: oct) vorgenommen werden. Die Kapselungsfunktionen können mittels dynamischer Bindung interaktiv über den Interpreter gerufen werden. In einigen SCEs kann Maschinencode auch direkt aufgerufen werden. Voraussetzung dafür ist allerdings die explizite Schnittstellendeklaration der Maschinencodefunktion in der SCE.

3.3 Vertreter

Ausgehend von den SCE-spezifischen Eigenschaften (s. Abschn. 3.1) lassen sich unter anderem die in Tabelle 3.1 aufgelisteten Softwaresysteme der Klasse der wissenschaftlich-technischen Berechnungsumgebungen zuordnen.

SCE	Hersteller	Lizenzmodell	veröffentlicht
IDL [36]	RSI Research Systems Inc.	kommerziell	1977
Gauss [34]	Aptech Systems Inc.	kommerziell	1983
Matlab [38]	The MathWorks Inc.	kommerziell	1984
Mathematica [35]	Wolfram Research Inc.	kommerziell	1988
O-Matrix [37]	Harmonic Software Inc.	kommerziell	1992
Octave [30]	J. W. Eaton	nichtkommerziell	1993
Rlab [32]	I. Searle	nichtkommerziell	1994
Scilab [31]	INRIA	nichtkommerziell	1994
Tela [29]	P. Janhunen	nichtkommerziell	1994
Euler [33]	R. Grothmann	nichtkommerziell	1996
Yorick [28]	D. H. Munro	nichtkommerziell	1996

Tabelle 3.1: Auswahl kommerzieller und freier SCEs

Das System Matlab nimmt dabei eine Sonderposition ein, da es die am weitesten verbreitete SCE ist und sich Syntax und Funktionsumfang vieler SCEs an dieses System anlehnen. Matlab wird darüber hinaus kontinuierlich um Funktionsbibliotheken (sog. *Toolboxen*) und Subsysteme für verschiedene wissenschaftliche und technische Disziplinen erweitert. Ein Subsystem, das stark zur Etablierung von Matlab in technischen Disziplinen beigetragen hat, ist zum Beispiel das Simulationssystem *Simulink*.

Das System IDL wurde als Visualisierungswerkzeug für physikalische Vorgänge entwickelt, besitzt heute aber alle Eigenschaften einer SCE. In ähnlicher Weise stellte das System Mathematica ursprünglich ein Computeralgebrasystem dar, verfügt heute aber

ebenfalls über alle SCE-spezifischen Eigenschaften. Die Syntax von IDL und Mathematica unterscheidet sich jedoch grundlegend von der anderer Systeme. Gauss und O-Matrix spielen im Vergleich zu Matlab und weit verbreiteten freien SCEs eine eher untergeordnete Rolle.

Eine der bedeutendsten freien SCEs stellt das System Octave dar. Seine Hauptmerkmale sind die Matlab-identische Syntax sowie die Matlab-identischen Grundfunktionen, die Octave besonders für Lehrzwecke interessant machen. Die zweite weit verbreitete freie SCE wird durch Scilab repräsentiert. Dieses System besitzt als einzige freie SCE ein Simulationssystem (*Scicos*), welches Simulink sehr ähnelt.

3.4 SCE-Programme

Die Abarbeitung von SCE-Programmen unterscheidet sich wesentlich von der Programmabarbeitung auf Betriebssystemebene. Auf Ebene des Betriebssystems liegt ein Programm als ausführbare Datei in Maschinencode vor. Beim Start des Programms wird der Maschinencode und die dazugehörigen Daten in einem Prozess gekapselt. Unterstellt man ein Multitasking-System, so können auf Grund dieser Kapselung mehrere Prozesse simultan durch das Betriebssystem verwaltet werden, ohne einander zu beeinflussen.

In einer SCE wird ein Programm durch ein SCE-Skript oder eine SCE-Funktion dargestellt, die in der Programmiersprache der SCE vorliegen. Sowohl Skripte als auch Funktionen können während ihrer Abarbeitung weitere SCE-Skripte und -Funktionen aufrufen, sodass ein Programm aus einem Hauptskript beziehungsweise einer Hauptfunktion und mehreren Unterskripten beziehungsweise -funktionen bestehen kann. Die Abarbeitung erfolgt dabei streng sequentiell, sodass zu jedem Zeitpunkt nur ein Programm von der SCE abgearbeitet werden kann.

Ein SCE-Skript stellt eine Stapelverarbeitungsdatei dar, in der mehrere Kommandos zusammengefasst werden, die sonst durch den Nutzer interaktiv über den Interpreter eingegeben werden. Die Daten eines Skriptprogramms werden stets im aktuellen Workspace abgelegt, das heisst beim interaktiven Aufruf im Base-Workspace und bei Aufruf durch eine Funktion im entsprechenden Function-Workspace. Beim interaktiven Aufruf eines SCE-Skriptes bleiben die Daten des Skriptprogramms nach seiner Abarbeitung durch den Nutzer inspizierbar und für nachfolgende Skripte zugreifbar. Durch die fehlende Prozesskapselung unterscheidet sich die Abarbeitung von SCE-Skriptprogrammen somit fundamental von der Programmabarbeitung auf Betriebssystemebene.

Eine SCE-Funktion ähnelt gegenüber SCE-Skripten stärker einem Prozess auf Betriebssystemebene. Die Daten einer SCE-Funktion sind im funktionseigenen Workspace abgelegt und stehen nach Funktionsabarbeitung nicht mehr zur Verfügung, sodass eine Beeinflussung nachfolgend abgearbeiteter SCE-Funktionen nicht direkt möglich ist. Dennoch existieren auch hier Möglichkeiten, die nachfolgende Abarbeitung von SCE-Funktionen zu manipulieren. So ist zum Beispiel der Zugriff und die Modifikation des Base- und Global-Workspace durch eine Funktion möglich.

Analog zu Funktionen beziehungsweise Routinen in kompilierbaren Sprachen besitzen SCE-Funktionen eine Menge an Eingabe- und Ausgabeparametern, die im Funktionskopf

deklariert werden. Im Gegensatz zu Fortran-Routinen oder C-Funktionen stehen dabei die Eingabeparameter stets rechtsseitig des Funktionsnamens, während Ausgabeparameter stets linksseitig des Namens deklariert werden, wie folgendes Schema verdeutlicht:

```
function [ output1 , output2 , ... ] = name ( input1 , input2 , ... )
```

Beim Aufruf einer SCE-Funktion werden die Eingabeparameter vollständig in den Workspace der Funktion kopiert (call by value), sodass ihre Modifikation während der Funktionsabarbeitung keine Auswirkung auf Variablen anderer Workspaces hat.

3.5 Programmentwicklung

Die Programmentwicklung in einer kompilierbaren Sprache erfolgt grundsätzlich iterativ, wobei der Zyklus Programmieren-Übersetzen-Testen-Fehlersuche mehrfach bis zum finalen Programm durchlaufen werden muss. Das Programm ist dabei in mehrere Module gegliedert, deren Entwicklung zwei Richtlinien folgen kann: Top-Down-Entwicklung beziehungsweise Bottom-Up-Entwicklung. In der Top-Down-Entwicklung wird das Programm ausgehend von der obersten Programmebene entwickelt, wobei eventuelle Unterprogramme schrittweise implementiert und verfeinert werden. In der Bottom-Up-Entwicklung wird das Programm dagegen ausgehend von den untersten Programmebenen entwickelt, die schrittweise in übergeordneten Funktionen zusammengefasst werden. Für den Test von Funktionen muss dabei stets ein expliziter Testrahmen in Form eines speziellen Hauptprogramms entwickelt werden, durch den die Nutzung der Funktion im realen Programm simuliert wird.

Die Programmentwicklung in einer SCE erfolgt ebenfalls iterativ, jedoch entfällt der Schritt des Übersetzens aus dem Entwicklungszyklus. Wie im vorangegangenen Abschnitt erwähnt, kann ein SCE-Programm ebenfalls modular aufgebaut sein. Hinsichtlich der Entwicklungsrichtlinie wird dabei insbesondere die Bottom-Up-Entwicklung unterstützt. Die Entwicklung einer Funktion erfolgt häufig in den folgenden Schritten:

1. Interaktive Erprobung von Befehlszeilen
2. Zusammenfassen erprobter Befehlszeilen in Skriptprogrammen
3. Kapselung von Skript-Programmen in Funktionen

Der Test einer implementierten Funktion entspricht dabei wiederum dem ersten Schritt der Funktionsentwicklung. Da auf diese Weise sehr schnell Funktionsprototypen entwickelt werden können, wird die beschriebene Methode oft auch als *Rapid Software Prototyping* bezeichnet. Der Zwischenschritt der Skriptprogrammierung hat den Vorteil, dass nach Programmabarbeitung oder beim Auftreten eines Fehlers alle Variablen des Skriptprogramms im Base-Workspace liegen und somit interaktiv inspiziert werden können, was die Fehlersuche deutlich erleichtert. In der Praxis wird der Schritt der Kapselung für das endgültige Hauptprogramm aus diesem Grund oft unterlassen, sodass nach einem Programmlauf alle Variablen des Hauptprogramms für eine nachfolgende Auswertung im Base-Workspace zur Verfügung stehen.

3.6 Beschleunigung von SCE-Programmen

Die Vorteile des SCE-Einsatzes für wissenschaftlich-technische Berechnungen wurden bereits angedeutet: Durch die interpretative Arbeitsweise können Programmprototypen effektiv entwickelt und getestet werden, dabei können alle Arbeiten innerhalb der SCE erfolgen. Durch die Verwendung eines Interpreters erfolgt die Abarbeitung von SCE-Code jedoch deutlich langsamer als bei kompiliertem Maschinencode. Es existieren daher verschiedene miteinander kombinierbare Ansätze, die Ausführungszeit von SCE-Programmen zu verkürzen:

- SCE-Code-Optimierung
- Übersetzung von SCE-Code in Maschinencode
- Reimplementierung von SCE-Code in einer kompilierbaren Sprache
- Parallele Abarbeitung von SCE-Code

Bei der Optimierung von SCE-Code werden zeitaufwendige Programmzeilen durch effizientere Programmstrukturen ersetzt. Beispiele dafür sind die Vermeidung dynamischer Vergrößerungen von Arrays durch explizite Speicherallokation oder das Ersetzen von Schleifen durch äquivalente datenparallele Operationen.

Bei der Übersetzung von SCE-Code in Maschinencode wird ein gesamtes SCE-Programm oder ein Teil eines Programms in kompilierbaren Zwischencode überführt und anschließend durch einen Compiler in Maschinencode übersetzt. Der Vorgang kann manuell durch den Anwender vorgenommen werden oder, durch einen so genannten Just-in-time-Compiler, automatisch durch die SCE erfolgen.

Die Reimplementierung von Programmabschnitten ist eine weit verbreitete Methode, SCE-Code zu beschleunigen. Viele SCEs stellen dem Programmierer Schnittstellen bereit, die den Aufruf von Fortran, C-, C++- oder Java-Code aus der SCE heraus ermöglichen. Da der Programmabschnitt bereits in einer höheren Programmiersprache entworfen und mit der SCE getestet wurde, erfordert die Reimplementierung deutlich weniger Zeit als der direkte Programmentwurf in der kompilierbaren Sprache.

Bei der parallelen Abarbeitung von SCE-Code werden zeitaufwändige Programmabschnitte oder vollständige SCE-Programme parallelisiert und auf einer Parallelverarbeitungsplattform ausgeführt. Die parallele Abarbeitung von SCE-Programmen ist Gegenstand der vorliegenden Arbeit und wird in den folgenden Kapiteln ausführlich betrachtet.

Als nützliches Werkzeug zur Programmbeschleunigung erweist sich ein so genannter *Profiler*. Mit diesem ist es möglich, die Abarbeitungszeit jeder Codezeile zu analysieren und somit Ansatzpunkte für eine eventuelle Programmbeschleunigung zu finden. Ein Profiler gehört bisher nur bei kommerziellen SCEs wie Matlab, IDL oder O-Matrix zum Produktumfang.

4 Parallelverarbeitung in SCEs

Durch Pawletta wurde 1998 erstmals der Versuch unternommen, Methoden der SCE-basierten Parallelverarbeitung in einer Klassifikation zu systematisieren ([47]). Später entstanden spezifische Klassifikationen zur Systematisierung von Softwaresystemen, die die Matlab-basierte Parallelverarbeitung ermöglichen ([62, 71]). Im folgenden Kapitel wird eine neue Klassifikation zur SCE-basierten Parallelverarbeitung vorgestellt, die die bisher bekannten Klassifikationen in sich vereint.

Anschließend erfolgt die ausführliche Darstellung aller Klassen der neuentwickelten Systematik und die Präsentation identifizierter Softwaresysteme und Methoden. Das Kapitel schließt mit einer Betrachtung zu hybriden Ansätzen, die durch eine Kombination verschiedener Klassen erreicht werden können.

4.1 Klassifikation

Eine Parallelverarbeitung von SCE-Code erfolgt in den meisten Fällen unter Nutzung zusätzlicher Softwaresysteme, welche dem Nutzer parallele Programmiermodelle bereitstellen. Da die Entwicklung der SCE-basierten Parallelverarbeitung wesentlich durch Softwaresysteme beeinflusst wurde, wurden Klassifikationen entworfen, die die Ansätze der Parallelverarbeitung auf Grundlage solcher identifizieren. Da jedoch auch Parallelverarbeitungsansätze existieren, in denen kein dediziertes Softwaresystem notwendig ist oder für die bisher keines entwickelt wurde, kann eine ausschließlich auf Softwaresystemen basierende Klassifikation nicht das gesamte Spektrum der SCE-basierten Parallelverarbeitung abdecken.

Zur Klassifikation SCE-basierter Parallelverarbeitung finden sich in der Literatur drei Quellen: *Pawletta* ([47]), *Choy und Edelman* ([62]) sowie *Panuganti et al.* ([71]). Pawlettas Klassifikation leitet die Ansätze der Parallelverarbeitung anhand von Softwaresystemen ab, berücksichtigt jedoch auch Ansätze, die unabhängig von zusätzlicher Software sind. Dabei wird auf keine spezielle SCE fokussiert. Choys und Edelmans sowie Panugantis Klassifikationen leiten die Ansätze ausschließlich aus Softwaresystemen ab und betrachten nur die marktbeherrschende SCE Matlab. Trotz dieser Unterschiede lassen sich zwischen allen Klassifikationen Gemeinsamkeiten und Redundanzen finden.

Pawletta verwendet eine Klassifikation, bei der zwischen vier Kategorien von Parallelverarbeitungsansätzen unterschieden wird (s. Abb. 4.1):

Übersetzungsansatz: Der sequentielle SCE-Code wird für eine Parallelverarbeitungsplattform übersetzt und läuft dort parallel ab.

Kopplungsansatz: Die SCE ist mit einem entfernten Parallelverarbeitungssystem gekoppelt und bringt dort parallele Numerikroutinen zur Ausführung.

Parallel-SCE-Ansatz: Die SCE läuft selbst auf einem Parallelverarbeitungssystem ab und besitzt parallele Numerikroutinen.

Multi-SCE-Ansatz: Mehrere gekoppelte SCE-Instanzen arbeiten im Verbund und stellen gemeinsam ein Parallelverarbeitungssystem dar.

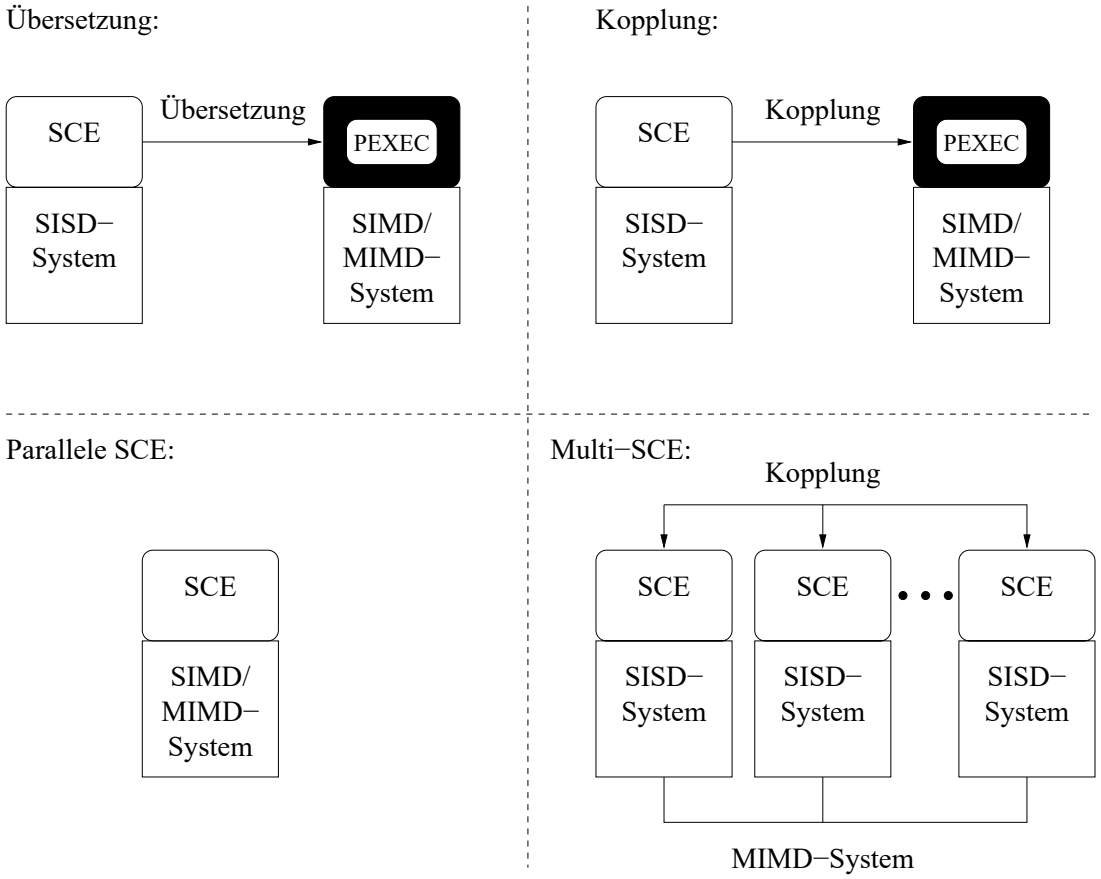


Abbildung 4.1: Klassifikation SCE-basierter Parallelverarbeitung nach Pawletta ([47])

Choy und Edelman, die sich speziell mit der SCE Matlab befassen, verwenden eine Klassifikation, die ebenfalls vier Kategorien von Ansätzen umfasst (s. Abb. 4.2):

Matlab Compiler: Der sequentielle Matlab-Code wird für eine Parallelverarbeitungsplattform übersetzt und läuft dort parallel ab.

Backend Support: Matlab ist mit einem entfernten Parallelverarbeitungssystem gekoppelt und bringt dort parallele Numerikroutinen zur Ausführung.

Embarrassingly Parallel: Mehrere gekoppelte Matlab-Instanzen arbeiten im Verbund, wobei eine Instanz den anderen übergeordnet ist. Die Programmierung erfolgt nach dem RPC-Modell.

Message Passing: Mehrere gekoppelte Matlab-Instanzen arbeiten im Verbund, wobei alle Instanzen gleichberechtigt sind. Die Programmierung erfolgt nach dem Message-Passing-Modell.

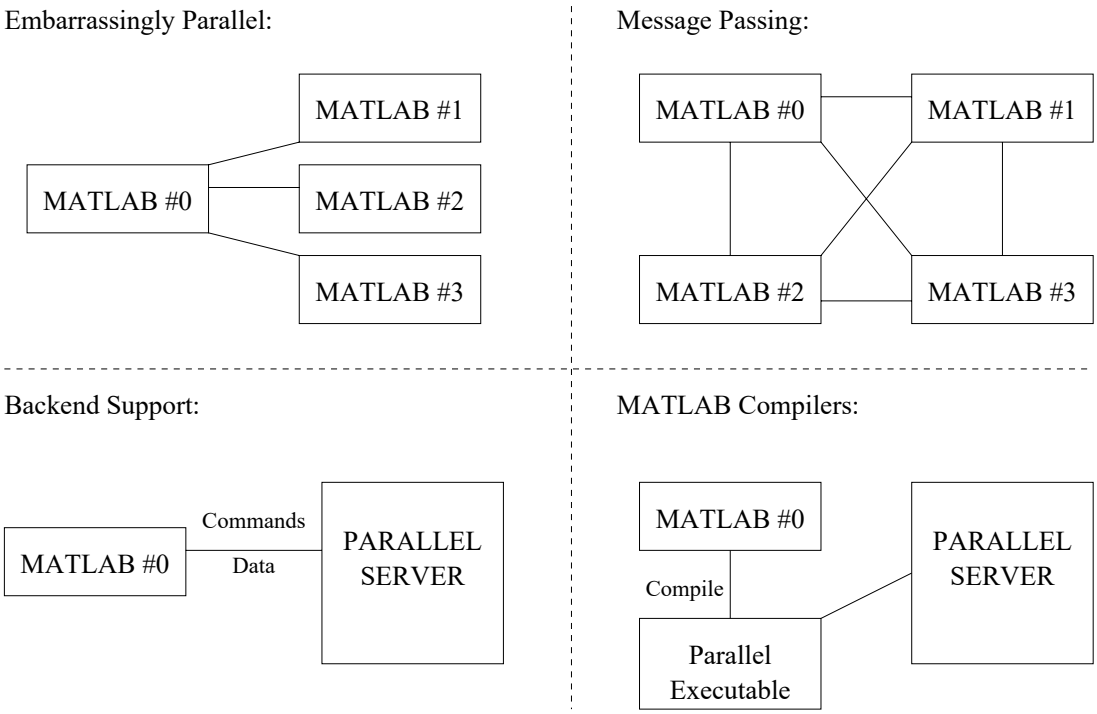


Abbildung 4.2: Klassifikation Matlab-basierter Parallelverarbeitung nach Choy und Edelman ([62])

Panuganti verwendet ebenfalls die von Choy und Edelman vorgeschlagene Klassifikation, ergänzt diese aber um eine weitere Klasse: *Parallel Global Address Space Model*. In dieser Klasse arbeiten mehrere gekoppelte Matlab-Instanzen im Verbund, wobei alle Instanzen gleichberechtigt sind. Die Programmierung erfolgt hier nach dem Shared-Memory-Modell.

Der Vergleich der Klassifikationen zeigt, dass die Klassen *Übersetzungsansatz* (Pawletta) und *Matlab Compiler* (Choy, Edelman) sowie *Kopplungsansatz* (Pawletta) und *Backend Support* (Choy, Edelman) einander entsprechen und in der Literatur zum Teil auf gleiche Softwaresysteme verwiesen wird. Darüber hinaus ist erkennbar, dass die Klassen *Embarrassingly Parallel* (Choy, Edelman), *Message Passing* (Choy, Edelman) und *Parallel Global Address Space Model* (Panuganti) als Unterklassen der Klasse *Multi-SCE* (Pawletta) angesehen werden können und in dieser lediglich unterschiedliche explizite

Programmiermodelle darstellen (s. Abschn. 2.2.2). Einzig die Klasse *Parallel-SCE* (Pawletta) entspricht weder einer Klasse eines der anderen Autoren noch ist sie in deren Klassen enthalten. Der Grund dafür ist, dass Pawletta die Klasse *Parallel-SCE* als allgemeines Konzept auffasst und beispielhaft die von Moler ([40]) diskutierten experimentellen parallelen Matlab-Versionen nennt. Choy und Edelman sowie Panuganti betrachten dagegen nur zusätzliche Softwaresysteme zur Erweiterung von Matlab, sodass dieser Ansatz unberücksichtigt bleibt. Die aufgeführten Umstände verdeutlichen die Notwendigkeit einer zusammenfassenden, neuen Klassifikation der SCE-basierten Parallelverarbeitung.

Als Grundlage für die im Folgenden vorgeschlagene neue Klassifikation dient die Anzahl der notwendigen SCE-Instanzen, wie bereits von Choy und Edelman verwendet. Choy und Edelman unterscheiden zwischen Ansätzen, die entweder keine Matlab-Instanz, genau eine Instanz oder mehr als eine Instanz erfordern. Die Anwendung dieser Systematik auf alle oben genannten Klassifikationen ist in Tabelle 4.1 dargestellt.

Anzahl Instanzen	Klassifikation Pawletta	Klassifikation Choy und Edelman	Klassifikation Panuganti et al.
0	Übersetzung	MATLAB Compiler	MATLAB Compiler
1	Kopplung, Parallel-SCE	Backend Support	Backend Support
2 oder mehr	Multi-SCE	Embarrassingly Parallel, Message Passing	Embarrassingly Parallel, Message Passing, Parallel Global Address Space Model

Tabelle 4.1: Anzahl notwendiger SCE-Instanzen

Ausgehend von Tabelle 4.1 beinhaltet die vorgeschlagene neue Klassifikation drei Klassen, die sich an der Anzahl der verwendeten Instanzen orientieren. Die Klassenbegriffe wurden, sofern möglich, an die SCE-übergreifende Klassifikation nach Pawletta angelehnt. Da die Klassen *Kopplungsansatz*, *Parallel-SCE-Ansatz* und *Backend Support* zu einer Klasse vereint werden, musste hier ein neuer treffender Oberbegriff gefunden werden. Die wesentliche Gemeinsamkeit der drei zusammengefassten Klassen ist hierbei die Tatsache, dass eine SCE-Instanz als Nutzerschnittstelle zu einem Parallelverarbeitungssystem dient, welches lokal (*Parallel-SCE-Ansatz*) oder entfernt (*Kopplungsansatz* bzw. *Backend Support*) installiert ist. Als Oberbegriff wurde daher die Bezeichnung *Frontend-Ansatz* gewählt. Die neue Klassifikation der SCE-basierten Parallelverarbeitung umfasst demnach die folgenden drei Kategorien (s. Abb. 4.3):

Übersetzungsansatz: Der sequentielle SCE-Code wird für eine Parallelverarbeitungsplattform übersetzt und läuft dort parallel ab.

Frontend-Ansatz: Die SCE stellt die Nutzerschnittstelle zu einem lokalen oder entfernten Parallelverarbeitungssystem dar und kann auf ihm parallele Numerikroutinen zur Ausführung bringen.

Multi-SCE-Ansatz: Mehrere gekoppelte SCEs arbeiten im Verbund und stellen gemeinsam ein Parallelverarbeitungssystem dar.

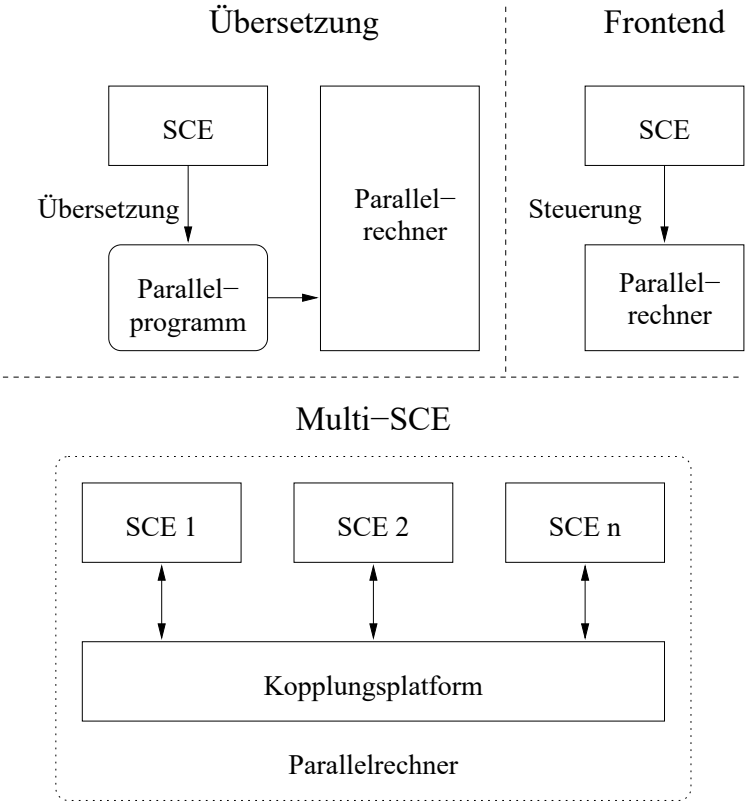


Abbildung 4.3: Neue Klassifikation SCE-basierter Parallelverarbeitung

In Analogie zu Tabelle 4.1 ist in Tabelle 4.2 die Zusammenfassung aller bisherigen Klassifikationen auf Basis der Anzahl notwendiger SCE-Instanzen dargestellt.

Neue Klasse	enthaltene Klassen Pawletta	enthaltene Klassen Choy und Edelman	enthaltene Klassen Panuganti et al.
Übersetzung	Übersetzung	MATLAB Compiler	MATLAB Compiler
Frontend	Kopplung, Parallel-SCE	Backend Support	Backend Support
Multi-SCE	Multi-SCE	Embarrassingly Parallel, Message Passing	Embarrassingly Parallel, Message Passing, Parallel Global Address Space Model

Tabelle 4.2: Neue Klassifikation und enthaltene Klassen nach Pawletta, Choy und Edelman sowie Panuganti et al.

4.2 Übersetzungsansatz

Beim Übersetzungsansatz werden sequentielle SCE-Programme oder -Programmteile zunächst in kompilierbaren Zwischencode einer skalaren beziehungsweise datenparallelen Programmiersprache übersetzt. Der Zwischencode wird anschließend entweder mit Hilfe eines parallelisierenden Compilers direkt in ein Parallelprogramm übersetzt oder nach der Übersetzung gegen parallele Numerikbibliotheken gelinkt. Der Vorteil des Übersetzungsansatzes ist, dass für eine Parallelisierung in der Regel keine Modifikation des SCE-Code notwendig ist, was dem Prinzip eines parallelisierenden Compilers entspricht (s. Abschn. 2.2.1.1). Zu den bekannten Softwaresystemen, die den Übersetzungsansatz unterstützen, zählen der *CONLAB Compiler*, *FALCON*, *Menhir* sowie *Otter* (s. Tab. 4.3).

System	SCE	erzeugter Zwischencode	veröffentlicht
CONLAB Compiler [39]	CONLAB (Matlab)	C mit Message-Passing-Routinen	1993
FALCON [43]	Matlab	Fortran 90	1996
Menhir [46]	Matlab	C oder Fortran 77 mit Parallelroutinen der linearen Algebra	1998
Otter [48]	Matlab	C mit Parallelroutinen der linearen Algebra und Signalverarbeitung	1998

Tabelle 4.3: Systeme nach dem Übersetzungsansatz

Eine Sonderstellung im Übersetzungsansatz nimmt der CONLAB Compiler ein, der eine Erweiterung des Systems CONLAB darstellt. CONLAB ist eine Entwicklungsumgebung für parallele Algorithmen und kann MIMD-Architekturen mit sowohl gemeinsamem als auch verteiltem Speicher simulieren. Der Sprachumfang von CONLAB entspricht dem von Matlab, ergänzt um Direktiven zur Parallelverarbeitung (Prozessverwaltung, Message-Passing- und Shared-Memory-Techniken). Der CONLAB Compiler übersetzt CONLAB-Programme (bzw. um Direktiven ergänzte Matlab-Programme) in C-Programme, die nach der Übersetzung in Maschinencode auf realer MIMD-Hardware lauffähig sind. Weil für eine parallele Abarbeitung des Programms explizit Anweisungen zur MIMD-Programmierung in den sequentiellen SCE-Code eingefügt werden müssen, erfolgt die Parallelisierung hier nicht implizit.

Die weiteren aufgeführten Systeme verbergen zwar die explizite Parallelität vor dem Anwender, führen jedoch ebenfalls keine automatische Parallelisierung durch. So wird für eine erfolgreiche Parallelisierung mittels FALCON ein parallelisierender Fortran-90-Compiler benötigt, der nicht zum Umfang des FALCON-Programmpakets gehört. Bei den Systemen Menhir und Otter ist die Parallelität dagegen bereits in den Numerikbibliotheken enthalten, wohingegen der von den Systemen erzeugte Zwischencode stets sequentiell bleibt.

4.3 Frontend-Ansatz

Beim Frontend-Ansatz dient die SCE als Nutzerschnittstelle zu einer Parallelverarbeitungsplattform. Von der SCE aus kann der Nutzer parallele Routinen auf der Plattform zur Ausführung bringen und deren Resultate innerhalb der SCE weiterverarbeiten. Die Plattform kann durch einen entfernten Rechner oder den lokalen Arbeitsplatzrechner repräsentiert werden. Im ersten Fall sind die parallelen Routinen nicht in den Funktionsumfang der SCE integriert, aus SCE-Sicht werden also *externe* parallele Routinen verwendet. Im zweiten Fall sind die parallelen Routinen Bestandteile des SCE-Berechnungsmoduls, sodass hier von *internen* parallelen Routinen gesprochen werden kann.

Bei der Verwendung des Frontend-Ansatzes erfolgt prinzipiell eine Programmierung mit parallelen Bibliotheken (s. Abschn. 2.2.1.3), das heisst es liegt ein implizites Programmiermodell vor. Dennoch muss vorliegender sequentieller SCE-Code teilweise modifiziert werden, um statt sequentieller Routinen parallele Routinen aufzurufen.

4.3.1 Externe parallele Routinen

Die Verwendung externer paralleler Routinen orientiert sich prinzipiell am Client-Server-Modell. Die SCE repräsentiert dabei den Client, während der entfernte Parallelrechner den Server darstellt. Auf der Seite des Clients werden teilweise auch Schnittstellen zu herkömmlichen Programmiersprachen (Fortran, C) bereitgestellt. Softwaresysteme, die eine SCE-Kopplung mit externen parallelen Routinen realisieren, sind *NetSolve*, *MATLAB*P* sowie das *PLAPACK Server Interface* (s. Tab. 4.4).

System	Client	parallele Routinen	Schnittstelle	Serverart	veröff.
Netsolve [67]	Matlab, Octave, Mathematica, C, Fortran, Excel	lineare Algebra, anwendungsspezifische Routinen	prozedural	zustandslos	1996
PSI [49]	Matlab	lineare Algebra	prozedural	zustandsbehaftet	1998
MATLAB*P [62]	Matlab	lineare Algebra, anwendungsspezifische Routinen	objektorientiert	zustandsbehaftet	2003

Tabelle 4.4: Systeme nach dem Frontend-Ansatz mit Kopplung externer paralleler Routinen

Hinsichtlich der angebotenen parallelen Routinen werden hauptsächlich Bibliotheken der linearen Algebra, wie zum Beispiel BLAS und ScaLAPACK durch den Server bereitgestellt. Darüber hinaus ist die Einbindung weiterer anwendungsspezifischer Bibliotheken möglich.

Die Nutzerschnittstelle innerhalb der SCE wird meist prozedural realisiert, wobei der Name der aufzurufenden externen Routine einer lokalen Prozedur als Stringparameter übergeben wird. Der Vorteil dieses Prinzips ist, dass serverseitig Routinen hinzugefügt werden können, ohne die Clientseite modifizieren zu müssen. Eine Ausnahme bildet das System MATLAB*P, bei dem extern gespeicherte Arrays durch Verknüpfung lokaler Daten mit der systemspezifischen Klasse `p` mittels Multiplikationsoperator `*` angelegt werden. Die Nutzerschnittstelle ist somit objektorientiert realisiert. Zur transparenten Weiterverarbeitung externer Arrays sind für die Klasse `p` Routinen der linearen Algebra für die externe Verarbeitung reimplementiert. Somit muss bei einer Erweiterung des serverseitigen Funktionsumfangs stets auch die Klasse `p` auf der Clientseite erweitert werden.

Bei einer SCE-Kopplung mit externen parallelen Routinen kann zwischen zustandslosen und zustandsbehafteten Servern unterschieden werden. Bei zustandslosen Servern werden alle Berechnungsdaten auf der Clientseite gehalten und nur im Fall einer externen Berechnung dem Server übermittelt. Ein zustandsbehafteter Server hält dagegen Daten zur Weiterverarbeitung in seinem Speicher und stellt sie dem Client bei Bedarf zur Verfügung.

4.3.2 Integrierte parallele Routinen

Bei der Verwendung integrierter paralleler Routinen beinhaltet das Berechnungsmodul der SCE parallele Numerikroutinen, die bei Bedarf ausgeführt werden. Die SCE kann dabei sowohl parallele numerische Standardroutinen als auch anwendungsspezifische parallele Routinen verwenden. Im Gegensatz zu anderen Ansätzen und ihren Unterklassen existieren für die Verwendung integrierter paralleler Routinen keine dedizierten zusätzlichen Softwaresysteme. Vielmehr wird diese Art der SCE-basierten Parallelverarbeitung durch SCE-Standardroutinen und individuelle Anwendungen abgedeckt.

Ein bekanntes Beispiel für die Integration von Standardroutinen stellen die von Moler ([40]) diskutierten parallelen Matlab-Versionen dar, mit denen The MathWorks Inc. in den 80er Jahren experimentierte. Dabei wurden sequentielle Routinen der linearen Algebra und Signalverarbeitung durch parallele Pendants für ein MIMD-System mit verteiltem Speicher ersetzt. Aufgrund der zu geringen Granularität lieferten die Experimente unbefriedigende Ergebnisse, sodass The MathWorks Inc. die Entwicklung paralleler Matlab-Versionen nicht weiter verfolgte. Heute existieren im Bereich der Standardroutinen SCE-integrierte Bibliotheken, die die SIMD-Kapazitäten moderner Prozessoren (MMX bzw. SSE) effektiv nutzen. Hierzu gehören Routinen der linearen Algebra (Matrix- und Vektoroperationen), Signalverarbeitung (FFT) sowie Vektoroperationen für trigonometrische und exponentielle Funktionen. Standardroutinen, die eventuell vorhandene MIMD-Kapazitäten (z.B. SMP) nutzen, sind bisher nur in IDL und Mathematica integriert.

Anwendungsspezifische parallele Routinen können durch die prinzipielle Erweiterbarkeit des SCE-Berechnungsmoduls einfach in den Funktionsumfang der SCE integriert werden. Hierbei muss lediglich die native Schnittstelle einer existierenden Routine an die SCE-eigene Schnittstelle durch Kapselungsfunktionen angepasst werden (s. Abschn. 3.2).

4.4 Multi-SCE-Ansatz

Der Multi-SCE-Ansatz ist nicht auf die Parallelverarbeitung beschränkt, sondern eignet sich für das gesamte Spektrum der SCE-basierten verteilten Verarbeitung (z.B. verteilte SCE-basierte Simulation). Im Rahmen der vorliegenden Arbeit werden allerdings ausschließlich Aspekte der Parallelverarbeitung, das heisst der verteilten Verarbeitung zum Zweck der Programmbeschleunigung in Zusammenhang mit Multi-SCEs betrachtet. Dazu wird zunächst das auf Pawletta (s. [47]) zurückgehende Multi-SCE-Konzept vorgestellt. Anschließend erfolgt die Präsentation identifizierter Multi-SCE-Realisierungen.

4.4.1 Multi-SCE-Konzept

Beim Multi-SCE-Ansatz werden mehrere konventionelle SCE-Instanzen mittels einer Kopplungsplattform verbunden. Der entstehende SCE-Verbund wird als Multi-SCE bezeichnet. Prinzipiell werden auf Softwareebene mehrere SISD-Systeme (einzelne SCE-Instanzen) zu einem MIMD-System (Multi-SCE) verknüpft.

Als Kopplungsplattform werden softwaretechnische Mittel der Interprozesskommunikation bezeichnet, die von der untergeordneten Hardware abstrahieren. Grundsätzlich kann zwischen drei Klassen von Kopplungsplattformen unterschieden werden:

Betriebssystemdienste: z.B. Sockets, Dateisystem

externe Middlewaredienste: z.B. PVM, MPI, ThreadMarks, Global arrays

SCE-interne Middlewaredienste: z.B. Matlab engine, proprietäre Dienste

Die SCE muss neben ihren Input- und Outputmodulen über eine Schnittstelle zur Kopplungsplattform verfügen. Diese Schnittstelle wird im Weiteren als *Low-Level-Interface* bezeichnet. Der Funktionsumfang des Low-Level-Interface kann primitive Kommunikations-, Synchronisations- und Instanziierungsoperationen umfassen.

Zur Programmierung paralleler Applikationen stehen dem Nutzer Schnittstellen zur Verfügung, die das Low-Level-Interface wesentlich vereinfachen oder hinsichtlich des Programmiermodells von ihm abstrahieren. Die Nutzerschnittstelle wird im Weiteren als *High-Level-Interface* bezeichnet. Der Funktionsumfang des High-Level-Interface umfasst Operationen zur Kommunikation, Synchronisation, Instanziierung und Partitionierung, die dem Programmierniveau der SCE angepasst sind. Der prinzipielle Aufbau einer Multi-SCE-Instanz ist in Abbildung 4.4 dargestellt.

Die Programmierung mittels High-Level-Interface folgt den in Abschnitt 2.2.2 diskutierten expliziten parallelen Programmiermodellen. Da das High-Level-Interface SCE-konforme Datentypen (Vektoren, Matrizen oder allg. Arrays) verarbeitet, werden die in Abschnitt 2.2.2 aufgeführten Programmiermodelle als *Array-Passing-Programmierung*, *Shared-Array-Programmierung*[1] und *Vektorielle RPC-Programmierung* bezeichnet.

Da beim Multi-SCE-Ansatz explizite parallele Programmiermodelle verwendet werden, muss im Fall einer Parallelisierung sequentieller SCE-Code modifiziert werden.

[1] bzw. *Global-Array-Programmierung*

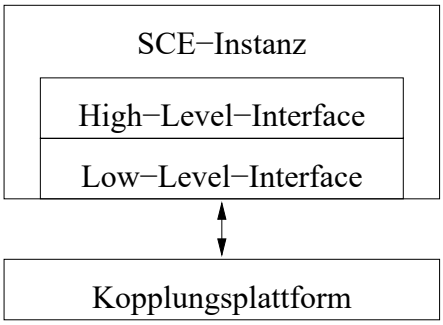


Abbildung 4.4: Prinzipieller Aufbau einer Multi-SCE-Instanz

Durch die SCE-spezifische interaktive Arbeitsweise können parallele Programme jedoch schneller als beim compilerbasierten Arbeiten entwickelt werden.

4.4.2 Multi-SCE-Realisierungen

Unter allen Ansätzen der SCE-basierten Parallelverarbeitung lassen sich die meisten Realisierungen dem Multi-SCE-Ansatz zuordnen. Dabei handelt es sich sowohl um von Einzelpersonen betriebene ad-hoc-Projekte als auch um längerfristige Forschungs- und Entwicklungsprojekte.

Im Rahmen der Recherche identifizierte Multi-SCE-Systeme sind in Tabelle 4.5 zusammengefasst dargestellt. Darin sind die verwendete Kopplungsplattform sowie die bereitgestellten Programmiermodelle aufgeführt. Die Programmiermodellabkürzungen entsprechen den in Abschnitt 2.2.2 diskutierten expliziten parallelen Programmiermodellen Message-Passing- (MP), Shared-Memory- (SHM) und RPC-Programmierung (RPC).

Tabelle 4.5 zeigt, dass seit dem Jahr 1999 eine bis heute anhaltende Dynamik im Bereich des Multi-SCE-Ansatzes besteht. So erschienen seit 1999 circa drei neue Multi-SCE-Systeme pro Jahr. Darüber hinaus wird sichtbar, dass drei Viertel der Multi-SCE-Realisierungen für die SCE Matlab entwickelt wurden, was durch die dominierende Stellung dieses Systems bedingt ist. Hinsichtlich der Kopplungsplattform sind Message-Passing-Bibliotheken wie PVM oder MPI-Implementierungen am häufigsten vertreten. Diese bieten gute Voraussetzungen für eine Integration in Multi-SCE-Systeme, da es sich um etablierte Middleware zur Parallelverarbeitung handelt. Multi-SCE-Systeme, die ein Message-Passing-System als Kopplungsplattform verwenden, stellen dieses Programmiermodell meist auch auf Ebene des High-Level-Interface bereit, das heisst das Multi-SCE-Softwaresystem realisiert lediglich Kapselungen der Message-Passing-Routinen. Im Bereich des High-Level-Interface sind Message-Passing- und RPC-Programmierung dominierend und jeweils in gleicher Anzahl vertreten. Die Häufigkeit der Message-Passing-Programmierung begründet sich in der breiten Etablierung dieses Programmiermodells. Die RPC-Programmierung wird dagegen oft wegen des leichten Zugangs zur Parallelverarbeitung bereitgestellt.

Die Tatsache, dass ein Großteil der Multi-SCE-Systeme für die kommerzielle SCE

System	SCE	Kopplungs-plattform	Programmier-modelle	veröff.
DP-Toolbox [42]	Matlab	PVM	MP, RPC	1995
PT Toolbox [41]	Matlab	PVM	MP	1995
MultiMATLAB [44]	Matlab	MPI	MP	1996
Paralize [45]	Matlab	Dateisystem	RPC	1997
PVM Toolbox [72]	Scilab	PVM	MP	1998
MATmarks [51]	Matlab	ThreadMarks	SHM	1999
PMI [50]	Matlab	Matlab engine	RPC	1999
PVM Toolbox [54]	Matlab	PVM	MP	1999
Multitask Tbx. [74]	Matlab	MPI	MP	2000
PLab [52]	Matlab	Sockets	RPC	2000
IDLPVM [55]	IDL	PVM	MP	2001
MatlabMPI [56]	Matlab	Dateisystem	MP	2001
MPI Toolbox [54]	Matlab	MPI	MP	2001
parmatlab [53]	Matlab	Sockets	RPC	2001
Beolab Toolbox [57]	Matlab	Matlab engine	RPC	2002
DistributePP [59]	Matlab	Dateisystem	RPC	2002
Parallelization Tk. [58]	Matlab	Matlab engine	RPC	2002
MPIDL [60]	IDL	MPI	MP	2003
Parallel [61]	Gauss	Sockets	RPC	2003
Parallel Octave [63]	Octave	MPI	MP	2003
DC-Toolbox [73]	Matlab	MPI, proprietär	MP, RPC	2004
Distributed Octave [64]	Octave	MPI	RPC	2004
MPI Toolbox [65]	Octave	MPI	MP	2004
Parallel Comp. Tk. [66]	Mathematica	proprietär	MP, SHM, RPC	2004
Matlab 2 Matlab [69]	Matlab	Sockets	RPC	2005
MDiCE [68]	Matlab	Sockets	RPC	2005
pMatlab [70]	Matlab	Dateisystem	SHM	2005
GAMMA [71]	Matlab	Global arrays	SHM	2006

Tabelle 4.5: Realisierungen des Multi-SCE-Ansatzes

Matlab entwickelt wurden, führt zwangsläufig zu lizenztechnischen Fragestellungen. So ist in einer Matlab-basierten Multi-SCE für jede SCE-Instanz eine gültige Lizenz notwendig, was die Betriebskosten einer Parallelverarbeitungsplattform somit stark erhöht. Tatsächlich sind die meisten Anwender Matlab-basierter Multi-SCEs bisher in Universitäten angesiedelt. Diese verfügen sowohl über eine hohe Anzahl an Matlab-Lizenzen als auch über entsprechende Parallelverarbeitungshardware.

Die vergleichsweise hohe Anzahl an Multi-SCE-Realisierungen lässt sich durch ein großes Interesse seitens der Anwender erklären, da durch die interpretative Verarbeitung eine deutliche Produktivitätssteigerung in der expliziten Parallelprogrammierung erreicht werden kann. Darüber hinaus sind mit diesem Ansatz die häufigsten ingenieur-

technischen Problemstellungen wie zum Beispiel Parameterstudien oder Optimierungsprobleme mit vergleichsweise geringem Aufwand parallelisierbar. Die vorliegende Arbeit konzentriert sich daher auf den Multi-SCE-Ansatz.

4.5 Hybride Ansätze

Neben den drei vorgestellten Ansätzen der SCE-basierten Parallelverarbeitung sind auch Kombinationen, das heisst hybride Ansätze möglich. Als Motivation für die Verwendung hybrider Ansätze kann eine Reduzierung der Lizenzkosten für kommerzielle SCEs angesehen werden. Hierbei ergeben sich vor allem aus der Kombination von Frontend-Ansatz und Multi-SCE-Ansatz sinnvolle Varianten.

Eine Möglichkeit der Kombination bietet die Anordnung beider Ansätze auf verschiedenen Hierarchieebenen. So können auf oberer Ebene mehrere SCEs nach dem Multi-SCE-Ansatz gekoppelt sein, während jede SCE des Verbundes parallele Numerikroutinen nach dem Frontend-Ansatz verwendet (s. Abb. 4.5). Diese hybride Variante bietet sich zum Beispiel bei Verwendung von Constellations (s. Abschn. 2.1.2) an.

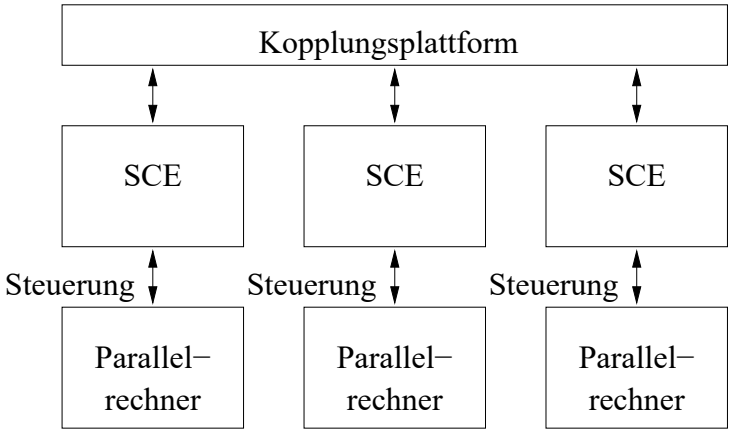


Abbildung 4.5: Hybrider Ansatz durch Hierarchisierung

Eine weitere Möglichkeit der Kombination von Ansätzen stellt die Überführung von Multi-SCE-Anwendungen in Frontend-Anwendungen dar. Dabei wird paralleler SCE-Code manuell in (expliziten) parallelen Code einer kompilierbaren Sprache übertragen um eine nutzerdefinierte parallele Routine zu schaffen, für die die SCE lediglich das Frontend zum Anwender darstellt (s. Abb. 4.6). Dieser Ansatz entspricht im Wesentlichen der Reimplementierung von SCE-Code (s. Abschn. 3.6). Da auch hier der Algorithmus bereits in SCE-Code vorliegt, erfordert die Übertragung auf den Frontend-Ansatz deutlich weniger Aufwand gegenüber der sofortigen Parallelprogrammierung in einer kompilierbaren Sprache.

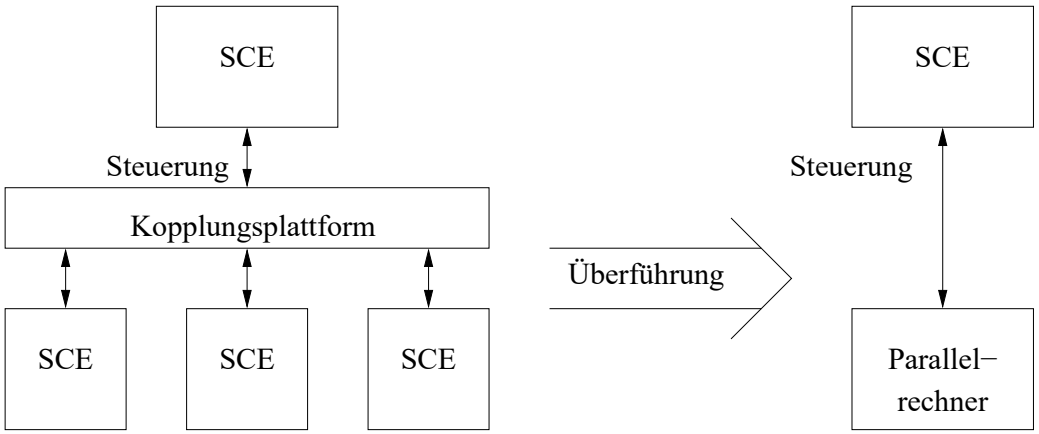


Abbildung 4.6: Hybrider Ansatz durch Überführung

5 Analyse existierender Multi-SCE-Systeme

Da unter allen SCE-basierten Parallelverarbeitungsansätzen der Multi-SCE-Ansatz für den breitesten Anwenderkreis von Interesse ist, wurden verschiedene Multi-SCE-Systeme einer Untersuchung unterzogen. Diese Untersuchungen dienen sowohl dem Vergleich der Systeme als auch der Identifikation von Ansätzen für Weiterentwicklungen. Die Untersuchungen wurden ohne Berücksichtigung spezifischer Anwendungshintergründe durchgeführt und betreffen sowohl qualitative als auch quantitative Merkmale der Multi-SCE-Systeme. Als qualitative Merkmale gelten hierbei die Anbindung der Kopplungsplattform, die anwenderfreundliche Realisierung des High-Level-Interface sowie die Eigenschaften des SCE-Verbundes. Für einen quantitativen Vergleich wurden Messungen zur Bestimmung der Kommunikationsleistung durchgeführt, welche einen wesentlichen Einfluss auf das Laufzeitverhalten paralleler Programme ausüben kann.

Ähnliche, wenn auch weniger umfangreiche Untersuchungen wurden bereits durch J. F. Baldomero durchgeführt. Zu diesen Untersuchungen wurden von Baldomero zwei Artikel veröffentlicht, in denen jeweils selbstentwickelte Multi-SCE-Systeme mit anderen frei verfügbaren Systemen verglichen werden. In einem 2001 erschienenen Artikel ([54]) findet ein Vergleich von Multi-SCE-Systemen für Matlab anhand ihrer Kopplungsplattformen sowie der Anzahl bereitgestellter Funktionen statt. Darüber hinaus erfolgt die Präsentation der selbstentwickelten Multi-SCE-Systeme *PVM Toolbox* und *MPI Toolbox* sowie ein quantitativer Vergleich der Systeme anhand einer Fallstudie (Wavelet Transformation). Im zweiten, 2004 veröffentlichten Artikel ([65]), findet in analoger Weise der Vergleich von Multi-SCE-Systemen für Octave statt. Ein quantitativer Vergleich erfolgt auch hier nur für zwei Systeme: die selbstentwickelte *MPI Toolbox* und *Parallel Octave* von Obara ([63]).

5.1 Untersuchte Systeme

Von den in Tabelle 4.5 aufgeführten Multi-SCE-Systemen konnten acht Systeme einer detaillierten Untersuchung unterzogen werden. Alle weiteren Systeme waren entweder nicht zugänglich oder ihre Entwicklung wurde eingestellt, sodass aktuelle SCE-Versionen nicht mehr unterstützt werden. In Tabelle 5.1 sind die untersuchten Systeme aufgeführt.

Das System *DP-Toolbox* ist eine zusätzliche Programmbibliothek für Matlab. Die Toolbox wird seit 2002 durch den Autor der vorliegenden Arbeit an der Hochschule Wismar gepflegt und kann über die DP-Toolbox-Homepage ([76]) bezogen werden. Die früheren Versionen der DP-Toolbox unterstützten sowohl die Message-Passing- als auch die

System	SCE	Programmier-modelle
DP-Toolbox v1.5 [90]	Matlab v7.1	MP
MatlabMPI v1.2 [56]	Matlab v7.1	MP
MPI Toolbox [54]	Matlab v7.1	MP
Beolab Toolbox [57]	Matlab v7.1	RPC
Parallelization Tk. v1.2 [58]	Matlab v7.1	RPC
DC-Toolbox v2.0 [73]	Matlab v7.1	MP, RPC
MPI Toolbox [65]	Octave v2.1	MP
PVM Toolbox [72]	Scilab v2.7	MP

Tabelle 5.1: Untersuchte Multi-SCE-Systeme

RPC-Programmierung (s. Tab. 4.5). Mit der Version 1.5 fand eine Anpassung der DP-Toolbox an die Matlab-Version 6 statt, bei der aus Stabilitätsgründen zunächst auf eine Unterstützung des RPC-Modells verzichtet wurde (s. Abschn. 6.1).

Das System *MatlabMPI* ist ebenfalls eine zusätzliche Programmbibliothek für Matlab. MatlabMPI wird durch J. Kepner am Lincoln Laboratory des Massachusetts Institute of Technology gepflegt und kann über seine Homepage ([80]) bezogen werden.

Bei den zwei Systemen, die als *MPI Toolbox* bezeichnet werden, handelt es sich um zusätzliche Programmbibliotheken für Matlab beziehungsweise Octave. Sie werden durch J. F. Baldomero an der Universität Granada gepflegt und sind über die jeweiligen Homepages der Systeme ([77, 78]) zu beziehen.

Die Systeme *Beolab Toolbox* und *Parallelization Toolkit* stellen ebenfalls zusätzliche Programmbibliotheken für Matlab dar. Diese beiden Systeme werden nicht kontinuierlich durch ihre Autoren T. Abrahamsson (Beolab) beziehungsweise E. Heiberg (Parallelization Tk.) weitergepflegt. Statt dessen wurden finale Versionen dieser Programmpakete im Matlab Central File Exchange, einem Verzeichnis zum Programmaustausch zwischen Matlab-Nutzern, zum Download bereitgestellt ([57, 58]).

Das System *DC-Toolbox* ist eine durch The MathWorks Inc. gepflegte und vertriebene zusätzliche Programmbibliothek für Matlab. Dieses System stellt in der Untersuchung das einzige kommerzielle Produkt dar. Eine Testversion der DC-Toolbox kann über die Homepage des Systems ([79]) bezogen werden.

Das System *PVM Toolbox*[1] ist im Gegensatz zu den anderen untersuchten Multi-SCE-Systemen keine zusätzliche Programmbibliothek, sondern ein bereits integrierter Bestandteil der SCE Scilab. Die Pflege des Systems erfolgt im Zuge der Weiterentwicklung von Scilab durch das INRIA, dem französischen nationalen Forschungsinstitut für Informatik und Automatisierungstechnik. Die PVM Toolbox kann zusammen mit Scilab über die Scilab-Homepage ([82]) bezogen werden.

Tabelle 5.1 zeigt, dass sechs Systeme für die kommerzielle SCE Matlab sowie jeweils ein System für die freien SCEs Scilab und Octave zur Verfügung standen. Hinsichtlich

[1] weitere Bezeichnungen: *//Scilab*, *Parallel Scilab* oder *Parallel Scilab using PVM*

der Programmiermodelle unterstützen drei Systeme die RPC-Programmierung, während die Message-Passing-Programmierung durch sechs Systeme unterstützt wird. Die *DC-Toolbox* stellt dabei als einziges untersuchtes System beide Modelle zur Verfügung. Die Shared-Memory-Programmierung wurde zum Zeitpunkt der Untersuchung durch kein verfügbares Multi-SCE-System unterstützt. Die Untersuchung eines entsprechenden Systems erfolgt im Rahmen der Präsentation neu- und weiterentwickelter Multi-SCE-Systeme in Kapitel 6.

5.2 Qualitative Merkmale

Die Untersuchung der qualitativen Merkmale von Multi-SCE-Systemen betrifft drei Aspekte: das Low-Level-Interface, das High-Level-Interface sowie den SCE-Verbund (s. Abb. 4.4). Bei der Betrachtung des Low-Level-Interface, das heisst der Schnittstelle zur Kopplungsplattform, wird insbesondere die Art der Plattformanbindung an die SCE untersucht. Das Ziel dieser Untersuchung ist die Analyse existierender Kopplungsansätze als Ausgangspunkt für Weiterentwicklungen. Bei Betrachtung des High-Level-Interface, das heisst der Programmierschnittstelle, wird die Realisierung programmiermodellspezifischer Funktionen untersucht. Das Ziel dieser Untersuchung ist der Vergleich der Anwenderfreundlichkeit der Systeme. Bei der Untersuchung des SCE-Verbundes stehen Aspekte der Instanziierung und Lebensdauer des Verbundes sowie des Mehrbenutzerbetriebs im Vordergrund. Darüber hinaus wird auch die Interaktivität von SCE-Instanzen betrachtet, die besonders in der Programmentwicklungsphase eine wichtige Rolle spielt.

5.2.1 Eigenschaften des Low-Level-Interface

Das Low-Level-Interface eines Multi-SCE-Systems hat den Zweck, Basisfunktionen zur Kommunikation beziehungsweise Synchronisation innerhalb der SCE bereitzustellen, und somit Grundvoraussetzungen der expliziten parallelen Programmierung zu erfüllen.

Viele SCEs stellen dem Nutzer bereits interne Funktionen zur Interprozesskommunikation bereit, die zum Beispiel das Arbeiten mit Sockets oder Dateien ermöglichen. Sollen Kopplungsplattformen verwendet werden, für die lediglich Schnittstellen auf dem Niveau kompilierbarer Sprachen zur Verfügung stehen, so muss die Anbindung durch Kapselungsfunktionen erfolgen (s. Abschn. 3.2). Die Kapselungsfunktionen stellen dabei lediglich das Bindeglied zwischen SCE und Kopplungsplattform bereit, garantieren aber nicht für die Verfügbarkeit der Plattform. Aus diesem Grund sind in einigen Multi-SCE-Systemen zusätzlich Laufzeitbibliotheken der verwendeten Kopplungsplattform enthalten. Tabelle 5.2 fasst die Anbindung der Kopplungsplattformen der untersuchten Systeme zusammen. Hier ist zu bemerken, dass die *DC-Toolbox* über zwei Anbindungen zu Kopplungsplattformen verfügt: eine proprietäre Plattform, die eine Parallelverarbeitung nach dem RPC-Modell ermöglicht und die Plattform *MPICH2*, die die Parallelprogrammierung nach dem Message-Passing-Modell erlaubt.

Anhand von Tabelle 5.2 wird deutlich, dass die Anbindung durch Kapselungsfunktionen ausschließlich durch Systeme erfolgt, die entweder Message-Passing-Bibliotheken

System	Kopplungs-plattform	Anbindung	Laufzeit-bibliotheken
DP-Toolbox (Matlab)	PVM v3.4	Kapselung	nicht enthalten
PVM Toolbox (Scilab)	PVM v3.4	Kapselung	enthalten
MPI Toolbox (Matlab)	LAM MPI v7.1	Kapselung	nicht enthalten
MPI Toolbox (Octave)	LAM MPI v7.1	Kapselung	nicht enthalten
DC-Toolbox MP (Matlab)	MPICH2 v1.0	Kapselung	enthalten
Beolab Toolbox (Matlab)	Matlab engine	Kapselung	enthalten
Parallelization Tk. (Matlab)	Matlab engine	Kapselung	enthalten
MatlabMPI (Matlab)	Dateisystem	SCE-intern	enthalten
DC-Toolbox RPC (Matlab)	proprietär	SCE-intern	enthalten

Tabelle 5.2: Anbindung von Kopplungsplattformen in Multi-SCE-Systemen

(LAM MPI, MPICH2 oder PVM) oder *Matlab engine* als Kopplungsplattform verwenden. Bei Verwendung von proprietären Plattformen (DC-Toolbox) oder Dateisystemen (MatlabMPI) besteht dagegen eine SCE-interne Anbindung an die Kopplungsplattform. Darüber hinaus ist erkennbar, dass die Systeme *DP-Toolbox* sowie *MPI Toolbox* (für Matlab und Octave) keine Laufzeitbibliotheken der Kopplungsplattform (PVM bzw. LAM MPI) bereitstellen, sodass diese zusätzlich zum Multi-SCE-System installiert werden müssen.

In den Systemen *Beolab Toolbox* und *Parallelization Toolkit* wird die Kopplungsplattform *Matlab engine* verwendet. Diese wurde ursprünglich nicht für den Einsatz in der Parallelverarbeitung konzipiert, sondern zur Steuerung von SCEs mittels kompilierbarer Programme. Aus diesem Grund ist zwar die Laufzeitbibliothek der Plattform in der SCE enthalten, es existiert jedoch keine SCE-Schnittstelle dafür, das heisst eine Kapselung der Funktionen muss vorgenommen werden. Zum Absetzen von Befehlen in gesteuerten Matlab-Instanzen, so genannten *Engines*, wird die Funktion `engEvalString` bereitgestellt. Diese Funktion weist ein blockierendes Verhalten auf, das heisst der steuernde Prozess blockiert für die Dauer der externen Befehlsabarbeitung. Eine parallele Abarbeitung von Befehlen in mehreren Instanzen ist somit nicht möglich. Die Systeme *Beolab Toolbox* und *Parallelization Toolkit* erreichen das notwendige nichtblockierende Verhalten durch die Ausführung von Lese- und Schreibzugriffen auf interne Dateideskriptoren. Diese Methode, die seitens The MathWorks Inc. nicht dokumentiert ist, wurde durch das System *PMI* (s. Tab. 4.5) eingeführt. Durch die starke Systemabhängigkeit der Methode ist sie nicht bei Windowsversionen von Matlab anwendbar. Grundsätzlich besitzt die Kopplungsplattform *Matlab engine* gegenüber anderen Plattformen den Vorteil, dass zusätzliche Programmbibliotheken des Multi-SCE-Systems nur in einer übergeordneten Matlab-Instanz vorhanden sein müssen, da der Datenaustausch sowie das Starten von SCE-Programmen über den Standard-Eingabekanal der entfernten SCE-Instanzen erfolgt (s. Abschn. 5.2.3).

5.2.2 Eigenschaften des High-Level-Interface

Ausschlaggebend für die Akzeptanz eines Multi-SCE-Systems ist die Anwenderfreundlichkeit der Schnittstelle zur Programmerstellung, das heisst des High-Level-Interface. Da je nach Programmiermodell unterschiedliche Programmieraufgaben explizit vorgenommen werden müssen, muss in den Anforderungen an das High-Level-Interface zwischen RPC- und Message-Passing-Programmierung differenziert werden.

Die Untersuchung RPC-basierter Systeme konzentriert sich auf die Gestaltung der RPC-Rufe. Diese Rufe repräsentieren programmiertechnische Umsetzungen der in Abschnitt 2.2.2.3 diskutierten RPC-Erweiterungen.

Bei der Untersuchung der Message-Passing-basierten Systeme wurde der Schwerpunkt auf den Abstraktionsgrad der Nutzerschnittstelle gegenüber etablierten Message-Passing-Schnittstellen (in Fortran bzw. C) gelegt und vor allem SCE-spezifische Erweiterungen dieser Schnittstellen untersucht. Als Erweiterung einer nativen Schnittstelle[2] gelten dabei Funktionalitäten von Multi-SCE-Systemen, die durch eine einfache Kapselung von Message-Passing-Funktionen nicht realisiert werden können und somit weiterentwickelte Ansätze darstellen. Die Untersuchung der Message-Passing-Systeme betrifft Funktionen der Kommunikation (Array-Passing) sowie kollektive Operationen.

5.2.2.1 Realisierung von RPC-Rufen

Die Realisierung der RPC-Rufe differiert in den untersuchten Multi-SCE-Systemen stark voneinander. Grundsätzlich kann zwischen folgenden qualitativen Kriterien unterschieden werden:

- Verwendete RPC-Variante
- Art der Datenzerlegung
- Art der Lastverteilung

Die nachfolgend diskutierten Kriterien sind für die untersuchten RPC-basierten Systeme in Tabelle 5.3 zusammengefasst.

Trotz der individuellen Unterschiede weisen alle Systeme eine Gemeinsamkeit auf, die den wesentlichen Unterschied gegenüber der compilerbasierten RPC-Programmierung kennzeichnet. Wie in Abschnitt 2.2.2.3 dargestellt, erfolgt der Aufruf einer entfernten Routine analog zum Aufruf einer lokalen Routine mit einem zusätzlichen Server-Parameter. Die SCE-basierte RPC-Programmierung unterscheidet sich davon, indem hier lediglich ein Kommando (RPC-Ruf) existiert, welches beliebige entfernte Routinen zur Ausführung bringen kann. Die entfernte Routine wird dabei über einen Stringparameter oder ein so genanntes Funktionshandle (vergleichbar mit einem Funktionszeiger) spezifiziert, diesem folgen die Eingabeparameter der Routine. In Analogie zu Abschnitt 2.2.2.3 lässt sich dieses Schema verallgemeinert wie folgt darstellen:

```
rückgabewerte[]=rpc(server[],funktion,parameter1[],parameter2[],...)
```

[2]Fortran oder C-Schnittstelle von PVM oder MPI

System	RPC-Variante	Datenzerlegung bei vektoriellem RPC	Lastverteilung
Beolab Toolbox (Matlab)	synchron vektoriell	3. Dimension	dynamisch
Parallelization Tk. (Matlab)	synchron skalar, asynchron skalar, asynchron vektoriell	manuell	manuell
DC-Toolbox (Matlab)	asynchron vektoriell, synchron vektoriell	Cell-Array	dynamisch

Tabelle 5.3: Realisierung von RPC-Rufen in Multi-SCE-Systemen

Hinsichtlich der RPC-Varianten realisieren die untersuchten Systeme eine oder mehrere der RPC-Erweiterungen *asynchron skalare RPC*, *synchron vektorielle RPC* sowie *asynchron vektorielle RPC*. Darüber hinaus ermöglicht das *Parallelization Toolkit* die Verwendung des klassischen RPC-Schemas, das heisst *synchron skalare RPC*, das für die Parallelverarbeitung jedoch ohne Bedeutung ist.

Die Zerlegung der zu verarbeitenden Daten bei vektoriellen RPCs, das heisst die Zerlegung in Teilaufgaben, wird durch die Systeme unterschiedlich realisiert. So zerlegt das System *Beolab Toolbox* ausschließlich dreidimensionale numerische Arrays. Ein Auftrag entspricht dabei einer Ebene der Datenstruktur und kann somit nur die Form einer Matrix besitzen. Dies schränkt die Menge der entfernt arbeitenden Prozeduren auf ausschließlich matrizenorientierte Funktionen ein. Durch die *DC-Toolbox* werden so genannte Cell-Arrays als Eingabeparameter des RPC-Rufes akzeptiert. Der Matlab-spezifische Datentyp Cell-Array kann ein multidimensionales Array aus Zellen beinhalten, wobei jede Zelle wiederum Matlab-Daten beliebigen Typs enthalten kann. Durch die Verwendung von Cell-Arrays beim vektoriellen RPC können entfernten Prozeduren somit beliebige Eingabeparameter zur Verfügung gestellt werden. Bei Verwendung des *Parallelization Toolkit* muss die Datenzerlegung bei vektoriellen RPCs durch den Nutzer erfolgen.

Die Art der Lastverteilung in RPC-Systemen entscheidet über die Zuordnung von Teilaufgaben zu Serverprozessen. Bei einer *statischen Lastverteilung* werden zu Beginn des RPC-Rufes alle Teilaufgaben deterministisch zugeordnet. Eine *dynamische Lastverteilung* ordnet dagegen jeweils einem freien Serverprozess eine Teilaufgabe zu, was zu einer nichtdeterministischen Verteilung führt. Eine dynamische Lastverteilung bedeutet zwar mehr Implementierungsaufwand im Multi-SCE-System, kann aber in Situationen, in denen die Anzahl der Teilaufgaben die der Server-Prozesse deutlich übersteigt, die Teilaufgaben stark schwankende Verarbeitungszeiten besitzen oder eine inhomogene Hardwareplattform vorliegt, zu deutlich geringeren parallelen Laufzeiten führen, wie Abbildung 5.1 anhand simulierter Programmlaufzeiten verdeutlicht. Von den untersuchten Multi-SCE-Systemen unterstützen die *Beolab Toolbox* sowie die *DC-Toolbox* die dynamische Lastverteilung. Das *Parallelization Toolkit* unterstützt keine Art der Lastverteilung, sodass eine Verteilung hier durch den Nutzer erfolgen muss.

Das *Parallelization Toolkit* nimmt unter den untersuchten RPC-Systemen eine Sonderstellung ein. Es besitzt wie andere RPC-Systeme eine Client-Server-Struktur, erlaubt

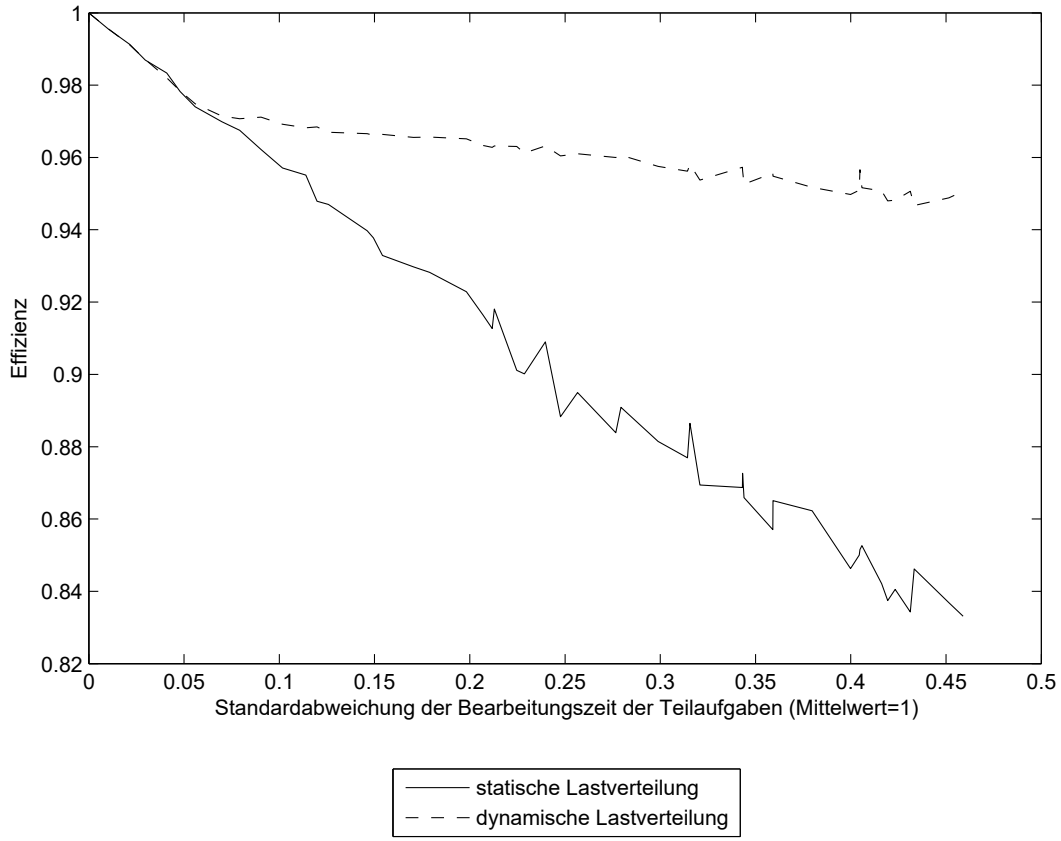


Abbildung 5.1: Effizienz von Lastverteilungen bei ungleichgewichtigen Teilaufgaben (durch Simulation ermittelt, 16 Serverprozesse, 256 Teilaufgaben)

ausschließlich Client-Server-Kommunikation und kann nur unabhängige Teilaufgaben parallelisieren. Obwohl diese Eigenschaften das Parallelization Toolkit als RPC-System charakterisieren, ist die Nutzerschnittstelle des Systems dem Message-Passing-Standard MPI angelehnt. Im Gegensatz zum etablierten Message-Passing kann hier jedoch nur einseitige Kommunikation, immer ausgehend von der Client-Seite, stattfinden. Zum Aufruf entfernter Prozeduren sind im System die Funktionen `MPI_Eval`, `MPI_IEval` und `MPI_BEval` implementiert, die wahlweise die Ausführung synchron skalarer RPCs, asynchron skalarer RPCs oder asynchron vektorieller RPCs ermöglichen. Diese Funktionen nehmen lediglich entfernt auszuführende Kommandos entgegen, das heisst ohne zusätzliche Eingabeparameter. Eine Verarbeitung nach dem RPC-Prinzip kann hier nur erfolgen, wenn die Eingabeparameter der entfernten Prozeduren vor dem Prozeduraufruf durch einseitige Kommunikationsroutinen vom Client an die Server übertragen werden. Analog dazu müssen die Rückgabewerte nach der Abarbeitung der entfernten Prozedur mittels einseitiger Kommunikation durch den Client vom Server abgerufen werden.

5.2.2.2 Realisierung des Array-Passing

Beim compilerbasierten Arbeiten werden Variablen ohne Informationen über ihren Typ oder ihre Dimension linear im Prozessspeicher abgelegt. Zum Senden einer Nachricht muss daher lediglich eine Speicheradresse sowie die Länge des zusammenhängenden Speicherbereichs angegeben werden. Für eine sinnvolle Weiterverarbeitung der Daten muss die Empfängerseite entweder a priori über Typ- und Dimensionsinformationen verfügen oder diese müssen explizit mit der Nachricht versendet werden.

Da die Programmiersprachen von SCEs Veränderungen von Datentyp und Dimension von Variablen während der Programmlaufzeit zulassen, werden diese Informationen immer im Verbund mit den eigentlichen Daten im Speicher abgelegt. Diese gemeinsame Speicherung erlaubt es, Datentyp- und Dimensionsinformationen implizit zusammen mit den Daten zu versenden, sodass aus Nutzersicht ein deutlich transparenteres Message-Passing möglich ist. Diese Anhebung des Programmierniveaus wird als *Array-Passing* (s. Abschn. 4.4.1) bezeichnet. Bis auf die Systeme *MPI Toolbox* für die SCEs Matlab und Octave unterstützen alle untersuchten Multi-SCE-Systeme das Array-Passing als Erweiterung der Message-Passing-Programmierung.

Die von Baldomero gepflegten Multi-SCE-Systeme *MPI Toolbox* für Matlab und Octave ([54, 65]) stellen hinsichtlich der Nutzerschnittstelle eine Besonderheit dar. Da diese Systeme nach Aussagen von Baldomero primär auf hohe Leistungsfähigkeit ausgerichtet sind, kapseln sie lediglich den vollständigen Funktionsumfang des MPI2-Standards in SCE-Funktionen.[3] Auf ein komfortables High-Level-Interface wird dabei verzichtet, sodass das Programmierniveau des Systems dem ursprünglichen, für kompilierbare Sprachen entwickelten, Standard entspricht. So muss zum Beispiel für Rückgabewerte von Empfangsoperationen zunächst manuell Speicher allokiert werden und diese anschließend rechtsseitig an die Funktion übergeben werden. Dies widerspricht grundsätzlich den gängigen SCE-Programmiertechniken (s. Abschn. 3.4). Da wie beim compilerbasierten Message-Passing Nachrichten ohne Typ- und Dimensionsinformation versendet werden, existiert keine Unterstützung des Array-Passing durch diese Systeme.

Beim compilerbasierten Arbeiten kann eine Nachricht beim Senden mit einem Integer-Wert (*message tag*) versehen werden, welcher auf Empfängerseite dem selektiven Empfang von Nachrichten dient. Dieses Prinzip der Nachrichtenmarkierung wurde im Multi-SCE-System *DP-Toolbox* dahingehend erweitert, dass Nachrichten hier durch einen Variablennamen spezifiziert und selektiert werden können. Die Ersetzung von Integer-Werten durch Variablennamen erhöht die Lesbarkeit des Codes deutlich. Die übrigen Message-Passing-unterstützenden Multi-SCE-Systeme erlauben dagegen lediglich die herkömmliche numerische Spezifikation. Die Unterstützung von Array-Passing sowie die Mittel zur Nachrichtenselektion sind in Tabelle 5.4 zusammengefasst dargestellt.

5.2.2.3 Realisierung kollektiver Operationen

Als kollektive Operationen werden Kommunikations- und Synchronisationsoperationen bezeichnet, an der eine Gruppe von Prozessen beteiligt ist. Kollektive Operationen stellen

[3] Die Messergebnisse in Abschnitt 5.3.2 bestätigen die hohe Leistungsfähigkeit des System.

System	Array-Passing	Message-Tag
DP-Toolbox (Matlab)	ja	String
PVM Toolbox (Scilab)	ja	Integer
MatlabMPI (Matlab)	ja	Integer
DC-Toolbox (Matlab)	ja	Integer
MPI Toolbox (Matlab)	nein	Integer
MPI Toolbox (Octave)	nein	Integer

Tabelle 5.4: Array-Passing-Unterstützung und Spezifikation von Nachrichten in Multi-SCE-Systemen

in der Message-Passing- und Shared-Memory-Programmierung ein wichtiges Hilfsmittel dar und können folgende Aufgaben erfüllen:

- Verteilung von Datenstrukturen unter Prozessen bzw. Zusammenfassung verteilter Datenstrukturen (Scatter/Gather)
- Gleichzeitiges Senden an mehrere Prozesse (Broadcast)
- Synchronisation von Prozesszuständen (Barrier Synchronisation)
- Anwendung globaler Operationen auf verteilte Datenstrukturen (Reduktion)

In Tabelle 5.5 ist die Unterstützung der nachfolgend diskutierten kollektiven Operationen in Message-Passing-basierten Multi-SCE-Systemen zusammengefasst dargestellt.

System	Array-Scattering und -Gathering	Broadcast	Barrier	SCE-Reduktion
DP-Toolbox (Matlab)	ja	ja	nein	nein
MatlabMPI (Matlab)	nein	ja	nein	nein
PVM Toolbox (Scilab)	nein	ja	ja	nein
MPI Toolbox (Matlab)	nein	ja	ja	nein
MPI Toolbox (Octave)	nein	ja	ja	nein
DC-Toolbox (Matlab)	nein	ja	ja	ja

Tabelle 5.5: Kollektive Operationen in Multi-SCE-Systemen

Die Verteilung und Zusammenfassung von Datenstrukturen mittels Scatter- und Gatheroperationen erfolgt in der compilerbasierten Message-Passing-Programmierung auf Grundlage eindimensionaler Arrays, die ausschließlich gleichförmig, das heisst in gleichgroße Teilarrays zerlegt werden können. Eine Erweiterung dieses Prinzips auf die SCE-basierte Verarbeitung wird durch das Multi-SCE-System *DP-Toolbox* verfolgt. So können mit Hilfe der DP-Toolbox mehrdimensionale Arrays entlang einer spezifizierbaren Dimension auf mehrere SCE-Instanzen verteilt werden. Ist eine gleichmäßige Verteilung

der Daten nicht möglich, so erreicht das System eine Verteilung, bei der sich die Anzahl der Dimensionselemente[4] der Teilarrays maximal um ein Element unterscheidet. In Anlehnung an das Array-Passing wird diese Methode im Folgenden als *Array-Scattering* beziehungsweise *Array-Gathering* bezeichnet.

Das gleichzeitige Senden an mehrere Prozesse mittels Broadcasting wird von allen untersuchten Systemen unterstützt. Das Broadcasting erfolgt durch die unterschiedlichen Systeme auf dem gleichen Niveau wie das Senden und Empfangen von Nachrichten. Das heisst, Systeme, die das Array-Passing unterstützen, erlauben auch beim Broadcasting das Senden und Empfangen von Daten inklusive ihrer Typ- und Dimensionsinformationen. Darüber hinaus ist bei keinem System eine SCE-spezifische Erweiterung des Broadcasting erkennbar.

Die Synchronisation von Prozesszuständen mit Hilfe der Barrier Synchronisation ist in den Systemen *DP-Toolbox* und *MatlabMPI* nicht möglich. Multi-SCE-Systeme, die die Barrier-Synchronisation unterstützen, realisieren wie beim Broadcasting keine SCE-spezifischen Erweiterungen dieser kollektiven Operation.

Bei der Anwendung globaler Operationen auf verteilte Datenstrukturen werden kommutative binäre Operationen, wie zum Beispiel Addition, Multiplikation oder Minimum- und Maximumbildung, auf alle Elemente einer verteilten Datenstruktur angewandt. Dabei fassen nach einer Baumstruktur jeweils zwei Prozesse ihre Daten zusammen. Das Ergebnis der kollektiven Operation wird im so genannten Wurzelprozess abgelegt. Neben der Verwendung vordefinierten Funktionen ist bei der nativen Message-Passing-Programmierung mit PVM und MPI auch die Benutzung nutzerdefinierter Funktionen möglich. Globale Operationen sind in der compilerbasierten Message-Passing-Programmierung etabliert, auf Ebene der Multi-SCE-Systeme jedoch kaum verbreitet. Einzig die *DC-Toolbox* erweitert das Prinzip globaler Operationen auf die Ebene von SCE-Funktionen. Analog zur Reduktion mittels kompilierter Funktionen, können hier verteilte Datenstrukturen mittels SCE-Funktionen, welche ebenfalls kommutative binäre Operationen ausführen müssen, zusammengefasst werden. Da das Prinzip eine Erweiterung gegenüber der nativen Message-Passing-Programmierung darstellt, wird es im Folgenden als *SCE-Reduktion* bezeichnet.

5.2.3 Eigenschaften des SCE-Verbundes

Neben den Charakteristika des High- und Low-Level-Interface sind die Eigenschaften des SCE-Verbundes wesentliche qualitative Merkmale eines Multi-SCE-Systems. Dabei sind die folgenden Aspekte von Bedeutung:

- SCE-Programmstart, Lebensdauer und Interaktivität des SCE-Verbundes
- Bedienung des SCE-Verbundes durch übergeordnete Instanzen
- Mehrbenutzerbetrieb im SCE-Verbund

[4] Dimensionselemente sind je nach Nummer der Dimension z.B. Zeilen, Spalten oder Ebenen

Die Systematik dieser im Folgenden diskutierten Eigenschaften des SCE-Verbundes fasst Abbildung 5.2 zusammen. In Tabelle 5.6 sind die nachfolgend diskutierten Eigenschaften des SCE-Verbundes zusammengefasst dargestellt.

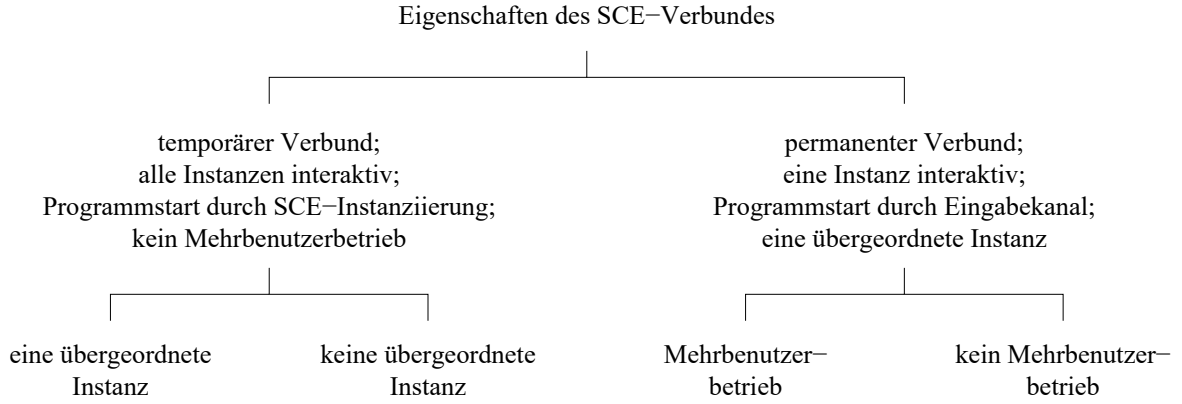


Abbildung 5.2: Systematik der Eigenschaften des SCE-Verbundes

System	Lebens-dauer	Programm-start durch	inter-aktive Ins-tanzen	Über-geord-nete Instanz	Mehr-benutzer-fähig
DP-Toolbox (Matlab)	temp.	SCE-Start	ja	ja	nein
PVM Toolbox (Scilab)	temp.	SCE-Start	ja	ja	nein
MatlabMPI (Matlab)	temp.	SCE-Start	ja	ja	nein
MPI Toolbox (Matlab)	temp.	SCE-Start	ja	nein	nein
MPI Toolbox (Octave)	temp.	SCE-Start	ja	nein	nein
Beolab Toolbox (Matlab)	perm.	Eingabekanal	nein	ja	nein
Parallelization Tk. (Matlab)	perm.	Eingabekanal	nein	ja	nein
DC-Toolbox (Matlab)	perm.	Eingabekanal	nein	ja	ja

Tabelle 5.6: Eigenschaften des SCE-Verbundes in Multi-SCE-Systemen

5.2.3.1 SCE-Programmstart, Lebensdauer des Verbundes und Interaktivität der Instanzen

Zur Ausführung paralleler SCE-Programme müssen Möglichkeiten existieren, in dedizierten Instanzen des SCE-Verbundes Programme oder Programmteile zu starten. Grundsätzlich können zwei Arten des Programmstarts in Multi-SCEs unterschieden werden:

1. Programmstart in bereits instanziierter SCE

2. Programmstart bei SCE-Instanziierung

Der Programmstart in einer instanziierten SCE setzt einen bestehenden SCE-Verbund voraus. Bei Systemen, die diese Art des Programmstarts verfolgen, kann der Verbund für die Dauer mehrerer paralleler Programmläufe existieren. Es kann daher in Bezug auf die Lebensdauer von einem *permanenten SCE-Verbund* gesprochen werden.

Der Programmstart bei SCE-Instanziierung setzt dagegen keinen bestehenden SCE-Verbund voraus. Bei Systemen, die diese Art des Programmstarts verfolgen, existiert der Verbund nur für die Dauer eines parallelen Programmlaufs, sodass hier bezüglich der Lebensdauer ein *temporärer SCE-Verbund* vorliegt. Die automatisierte Abarbeitung mehrerer aufeinanderfolgender Programme im gleichen Verbund ist hier nicht möglich.

Der Vorteil eines temporären SCE-Verbundes ist, dass jeder Programmlauf nicht von vorherigen Programmläufen beeinflusst wird (s. Abschn. 3.4), da die beteiligten SCE-Instanzen jeweils neu gestartet werden. Im permanenten Verbund ist eine derartige Beeinflussung, besonders durch Skript-Programme, theoretisch möglich. Nachteilig im temporären Verbund wirkt sich aus, dass die Instanziierung des SCE-Verbundes relativ viel Zeit beanspruchen kann, die zur Laufzeit des parallelen Programms gezählt werden muss. Im Vergleich dazu ist der Zeitaufwand des Programmstarts in einer permanenten SCE vernachlässigbar gering.

Die Art des Programmstarts und die Lebensdauer des SCE-Verbundes sind direkt mit der Interaktivität der Instanzen verknüpft. Besonders bei der Entwicklung von Multi-SCE-Programmen ist es hilfreich, wenn alle beteiligten Instanzen interaktiv bedienbar sind. Auf diese Weise können Fehler im parallelen Programm schnell erkannt und behoben werden. Darüber hinaus können im interaktiven Betrieb parallele Programme mit Hilfe eines Profilers effizient optimiert und somit Programmlaufzeiten weiter verkürzt werden.

In einem permanenten Verbund besitzen SCE-Instanzen in Analogie zur RPC-Programmierung ein festes Rollenverhalten. Die Anweisung zum Start eines Programms wird dabei von einer Client-Instanz an eine oder mehrere Server-Instanzen gerichtet. Da SCE-Instanzen nicht gleichzeitig Kommandos von mehren Eingabekanälen (in diesem Fall Interpreterprompt *und* Kanal zum Client) entgegennehmen können, ist die interaktive Bedienung der Server-Instanzen hier nicht möglich.

Die Instanzen eines temporären SCE-Verbundes besitzen dagegen kein festes Rollenverhalten. Die Anweisung zum Start eines Programms wird hier nicht über einen Eingabekanal von einer übergeordneten Instanz empfangen, sondern kann beim Start der SCE-Instanz einmalig spezifiziert werden. Da für den Programmstart somit kein Eingabekanal notwendig ist, sind alle Instanzen hier interaktiv bedienbar.

Die durchgeführten Untersuchungen haben ergeben, dass Systeme, die das RPC-Programmiermodell anbieten, auf einem permanenten SCE-Verbund beruhen. Multi-SCE-Systeme, die dagegen ausschließlich die Message-Passing-Programmierung anbieten, arbeiten nach dem Prinzip des temporären SCE-Verbundes. Die Ursache für diese eindeutige Zuordnung liegt in den Analogien des Rollenverhaltens der SCE-Instanzen zum Rollenverhalten von Prozessen im jeweiligen Programmiermodell (s. Abschn. 2.2.2). Diese Abbildung ist jedoch nicht zwingend, wie die *DC-Toolbox* beweist. Hier beruht das

System auf einem permanenten SCE-Verbund, unterstützt aber beide Programmiermodelle.

5.2.3.2 Übergeordnete Instanz

Als übergeordnete Instanz wird eine SCE-Instanz bezeichnet, von der aus Programme in anderen Instanzen des Verbundes gestartet werden können. In einer übergeordneten Instanz kann somit der Initialisierungsteil eines parallelen Programms gestartet werden, welcher weitere Programmteile in anderen Instanzen zur Ausführung bringt. Nach Abarbeitung des parallelen Programms können in der übergeordneten Instanz dessen Ergebnisse weiterverarbeitet werden.

In einem permanenten SCE-Verbund wird die übergeordnete Instanz stets durch die Client-Instanz repräsentiert, die auf den nichtinteraktiven Server-Instanzen Programme zur Ausführung bringt. In einem temporären Verbund ist dagegen eine übergeordnete Instanz nicht zwingend notwendig, es können vielmehr zwei Ebenen des Programmstarts unterschieden werden:

1. Betriebssystemebene (ohne übergeordnete Instanz)

2. SCE-Ebene (mit übergeordneter Instanz)

Beim Programmstart auf Betriebssystemebene erfolgt die Instanziierung des gesamten SCE-Verbundes und somit der Start des parallelen Programms (temporärer Verbund) aus der Umgebung des Betriebssystems, zum Beispiel mittels Kommandointerpreter. Es existiert somit keine dedizierte SCE-Instanz, in der der parallele Programmlauf vor- oder nachbereitet werden kann.

Beim Programmstart auf SCE-Ebene repräsentiert die übergeordnete Instanz bereits eine Instanz des Verbundes. Aus dieser Instanz heraus werden die übrigen SCE-Instanzen des Verbundes erzeugt und somit das parallele Programm gestartet. Auf diese Weise ist auch in Systemen, die auf einem temporären SCE-Verbund basieren, die Vor- und Nachbereitung paralleler Programme in einer dedizierten Instanz möglich.

Die Untersuchungen ergaben, dass lediglich die *MPI Toolbox* für Matlab sowie Octave die Arbeit mit übergeordneten Instanzen nicht unterstützt.

5.2.3.3 Mehrbenutzerbetrieb

Da ein permanenter SCE-Verbund für die Dauer mehrerer paralleler Programmläufe besteht, ist es möglich, mehreren Nutzern die Ausführung paralleler Programme auf dem Verbund zu erlauben. Dies kann in Analogie zu Parallelverarbeitungshardware zum Beispiel bei der Bereitstellung eines zentralen SCE-Parallelrechners innerhalb einer Institution sinnvoll sein. Weil der SCE-Verbund in diesem Fall als ständig verfügbare Ressource aufgefasst werden kann, werden eventuelle Nutzer von der Aufgabe der SCE-Instanziierung entlastet.

Von den untersuchten Multi-SCE-Systemen, die nach dem Prinzip des permanenten SCE-Verbundes arbeiten, erlaubt ausschließlich die *DC-Toolbox* den Mehrbenutzerbe-

trieb. Im Vergleich mit anderen Systemen weist die DC-Toolbox bemerkenswerte Eigenschaften auf, die die starke Fokussierung auf den Mehrbenutzerbetrieb unterstreichen. So ist ein *Jobmanager*-Prozess wesentlicher Bestandteil des SCE-Verbundes, der als Bindeglied zwischen *DC-Worker*-Instanzen und *DC-Toolbox*-Instanzen dient (s. Abb. 5.3).[5] Ein Jobmanager und mehrere DC-Worker bilden dabei die Einheit *DC-Engine*. Der Jobmanager verwaltet einerseits die Aufträge der Anwender, andererseits überwacht er die Tätigkeit der Worker und ist somit der zentrale Anmeldepunkt für alle beteiligten Instanzen. Die Worker-Instanzen verfügen darüber hinaus über ein *Checkpoint*-System, mit dem sich unterbrochene Aufträge, etwa auf Grund von Systemabsturz, wieder aufnehmen lassen.

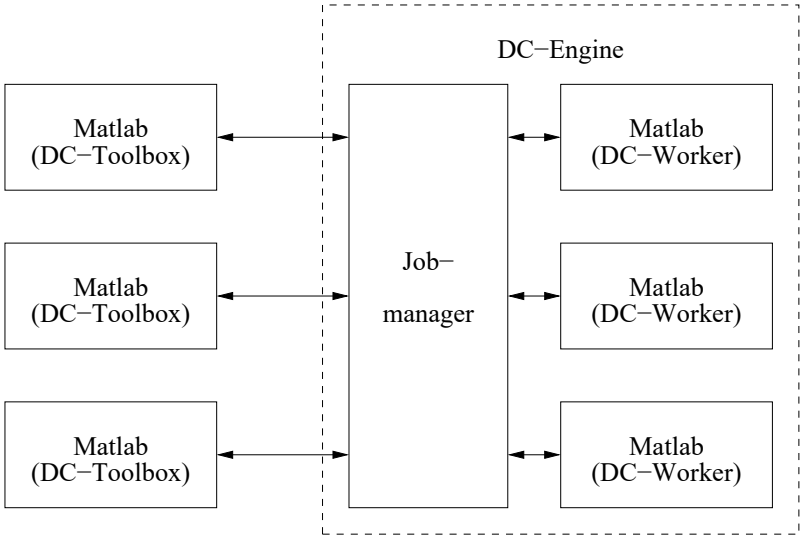


Abbildung 5.3: Struktur der DC-Toolbox (nach [73])

5.3 Quantitative Merkmale

Der quantitative Vergleich der untersuchten Multi-SCE-Systeme stützt sich auf Parameter der Kommunikationsleistung. Insbesondere in Workstation-Netzen sowie Cluster- und Gridplattformen stellt das Verbindungsnetzwerk, welches über die maximal erreichbare Kommunikationsleistung entscheidet, häufig eine stark limitierte Ressource in der parallelen Programmabarbeitung dar. Die optimale Ausnutzung des Netzwerkes kann daher den Erfolg einer Parallelisierung entscheidend beeinflussen.

Das Ziel der Untersuchungen war die Ermittlung von Kennwerten, die den quantitativen Vergleich von Multi-SCE-Systemen ermöglichen. Anhand dieser Kennwerte ist es außerdem möglich, den Einfluss verschiedener SCEs auf die Kommunikationsleistung

[5] Im Sinne des RPC-Modells entspricht eine DC-Worker-Instanz einem Server-Prozess und eine DC-Toolbox-Instanz einem Client-Prozess.

zu untersuchen, sowie die Effektivität der Anbindung der Kopplungsplattform durch Vergleiche mit direkten Messungen an der jeweiligen Plattform zu ermitteln. Zudem lassen die ermittelten Parameter qualitative Aussagen über die Performance von speziellen Anwendungsklassen (s. Abschn. 7.1) zu.

5.3.1 Messverfahren

Zur Bestimmung der Kommunikationsleistung wurde ein *Ping-Pong*-Verfahren mit zwei beteiligten Prozessen verwendet. Als Ping-Pong-Verfahren wird im Allgemeinen ein Messverfahren bezeichnet, bei dem eine Nachricht ausgehend von einem Masterprozess an einen oder mehrere Slaveprozesse gesendet wird. Nach Empfang der Nachricht senden die Slaveprozesse diese an den Masterprozess zurück. Die Zeitspanne zwischen dem Senden der ersten Nachricht und dem Empfang der letzten Nachricht im Masterprozess wird als *Roundtrip-Zeit* bezeichnet. Die Roundtrip-Zeit wird direkt durch die Kommunikationsleistung eines Systems beeinflusst und steigt mit Zunahme der Nachrichtengröße.

Wie bereits erwähnt, beschränken sich die durchgeführten Untersuchungen auf eine Anzahl von zwei Prozessen, das heisst einen Master- und einen Slaveprozess (1-zu-1-Kommunikation). Der Grund für die Einschränkung auf dieses einfache Kommunikationsschema ist das Ziel, möglichst grundlegende Kennwerte, ohne Fokussierung auf spezielle Anwendungshintergründe, zu erhalten.

5.3.1.1 Adaption an Programmiermodelle

Das beschriebene Ping-Pong-Verfahren eignet sich aufgrund der expliziten Kommunikationsoperationen (Senden/Empfangen) zunächst nur für Systeme, die das Message-Passing-Modell unterstützen. Um unter Verwendung anderer expliziter Programmiermodelle vergleichbare Ergebnisse zu erzielen, ist eine Anpassung des Verfahrens an diese Modelle notwendig.

Bei einer Adaption an das RPC-Modell besteht das Slave-Programm lediglich aus einer Dummy-Routine, die einen Eingabeparameter entgegennimmt und diesen als einzigen Ausgabeparameter zurückliefert. Durch das entfernte Starten der Routine durch den Masterprozess[6] und die Übergabe der Nachricht als Eingabeparameter wird ein zum Ping-Pong-Verfahren analoges Kommunikationsverhalten erreicht.

Eine Adaption an das Shared-Memory-Modell ist ebenfalls möglich. Auch wenn keines der untersuchten Multi-SCE-Systeme die Shared-Memory-Programmierung unterstützt, so ist die Erläuterung der Anpassung für Betrachtungen in Abschnitt 6.4.2 notwendig. Ein zum Ping-Pong-Verfahren analoges Kommunikationsverhalten kann hier durch explizite Synchronisationsmaßnahmen, wie zum Beispiel Barrier-Synchronisation, erreicht werden. Der Master-Prozess schreibt dabei die Nachricht auf eine gemeinsame Variable und aktiviert anschließend den wartenden Slaveprozess durch explizite Synchronisation. Anschließend führt der Slaveprozess einen lesenden und einen schreibenden Zugriff auf

[6] Als Prozess wird in diesem Zusammenhang eine SCE-Instanz mit einem laufenden SCE-Programm bezeichnet.

die gemeinsame Variable durch und aktiviert wiederum den wartenden Masterprozess, der die Nachricht aus der gemeinsamen Variable entnimmt.

In Abbildung 5.4 ist das Ping-Pong-Verfahren mit den diskutierten Anpassungen schematisch dargestellt.

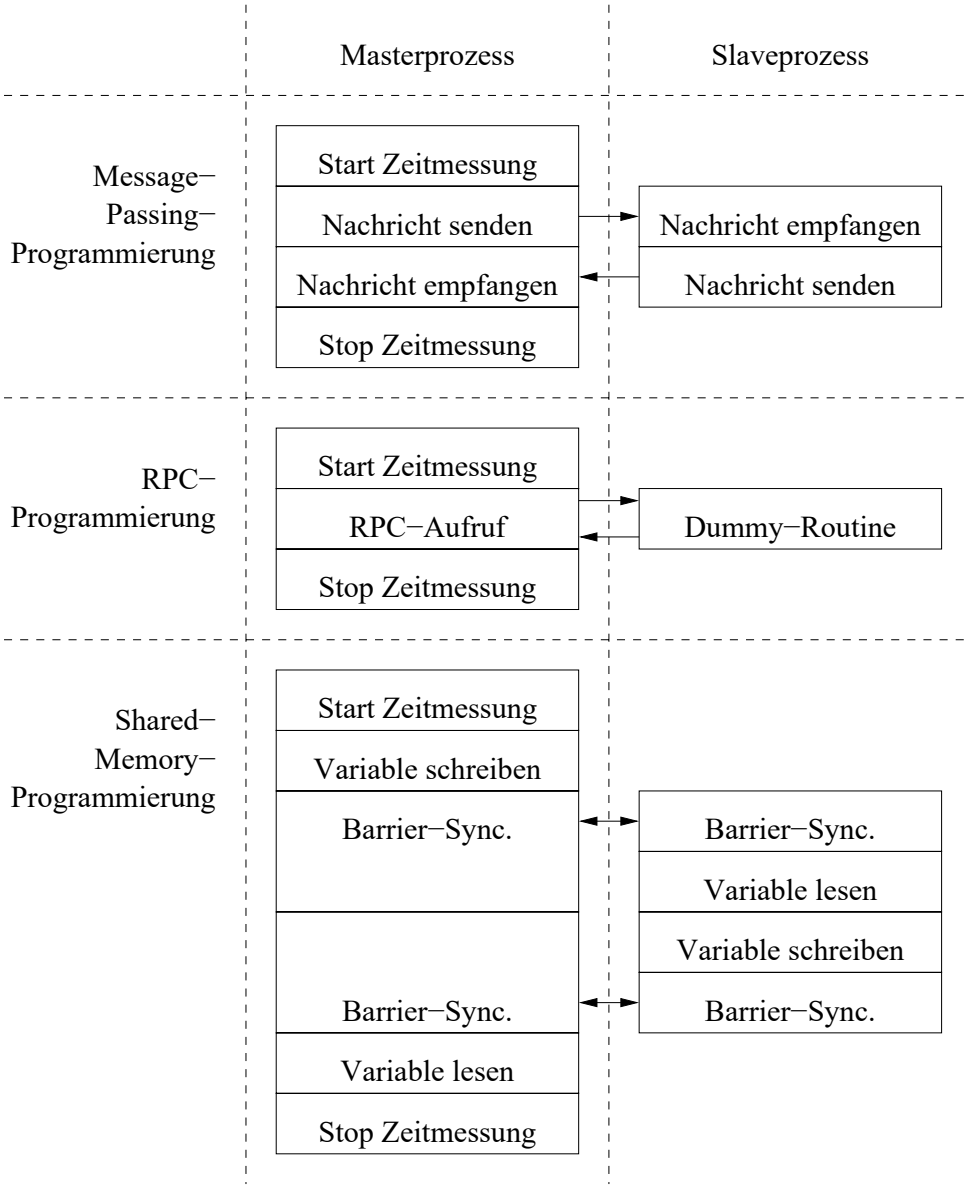


Abbildung 5.4: Ping-Pong-Verfahren mit expliziten parallelen Programmiermodellen

5.3.1.2 Modellansatz und Parameter

Eine Analyse der Messreihen (s. Abschn. A.1) ergab für die meisten Systeme eine annähernd lineare Abhängigkeit der Roundtrip-Zeit von der Nachrichtengröße. Um aus

den gemessenen Größen vergleichbare Kennwerte zu erhalten, wurde daher ein Modellansatz auf der Basis einer linearen Gleichung gewählt. Die zu ermittelnden Parameter entsprechen dabei der Latenzzeit und der Übertragungsrate des Systems:

$$t_{Roundtrip} = 2 * \left(\frac{Nachrichtengröße}{Übertragungsrate} + t_{Latenz} \right) \tag{5.1}$$

Die Latenzzeit repräsentiert hierbei die Zeit, die für das Senden einer leeren Nachricht benötigt wird und kann somit auch als Zugriffszeit auf ein Übertragungsmedium angesehen werden. Die Übertragungsrate bestimmt dagegen den Einfluss der Nachrichtengröße auf die Übertragungsdauer. Mit steigenden Nachrichtengrößen verliert die Latenzzeit für die Übertragungsdauer an Bedeutung, während der Einfluss der Übertragungsrate steigt. Beide Parameter sind herkömmliche Kennwerte der Kommunikationstechnik und können direkt mit den Leistungsangaben von Hardwareherstellern verglichen werden.

5.3.1.3 Messspektrum, Verringerung von Messfehlern und Betriebssystemeinflüssen

Da ein möglichst breites Spektrum an Nachrichtengrößen im Experiment untersucht werden sollte, wurden Messungen der Roundtrip-Zeit mit exponentiell steigenden Nachrichtengrößen vorgenommen.

Die Messung der Roundtrip-Zeit erfolgt durch zeitgebende Funktionen des Betriebssystems, die in SCE-Funktionen gekapselt sind. Wie jedes Messinstrument unterliegen diese Funktionen einem Messfehler. Da die Nachrichtengröße und somit auch die gemessene Roundtrip-Zeit exponentiell verläuft, kann der Einfluss des Messfehlers auf die Roundtrip-Zeit ebenfalls als exponentiell angesehen werden, das heisst Roundtrip-Zeiten bei geringen Nachrichtengrößen unterliegen einem weitaus höheren relativen Fehler als Roundtrip-Zeiten bei hohen Nachrichtengrößen.

Aus diesem Grund wurden die Operationen des Nachrichtentransfers wiederholt durchgeführt und die Zeitmessung außerhalb der wiederholten Operationen platziert. Die Roundtrip-Zeit ergibt sich somit aus dem Quotienten aus gemessener Zeit und Anzahl der Wiederholungen. Die Anzahl der Wiederholungen wurde dabei an die Größe der Nachricht angepasst, sodass die Zeitmessung stets in der Größenordnung von circa 30 Sekunden lag. Für die Ermittlung des notwendigen Wiederholungsfaktors wurde eine vorgelagerte Messung zur Abschätzung von Latenzzeit und Übertragungsrate vorgenommen. Die abgeschätzten Parameter fließen mittels Formel 5.1 in die Berechnung der zu erwartenden Roundtrip-Zeit und somit des notwendigen Wiederholungsfaktors ein. Durch dieses Verfahren konnte der absolute Messfehler dynamisch an die Nachrichtengröße und somit an die ermittelte Roundtrip-Zeit angepasst werden, sodass über den gesamten Messbereich der gleiche relative Fehler vorherrscht.

In Multitasking-Betriebssystemen können neben der Zeitmessung auch weitere Prozesse aktiv sein und die Messung durch eine erhöhte CPU-Auslastung beeinflussen. Diese betriebssystembedingten Einflüsse wurden durch die wiederholte Durchführung von Messungen und anschließende Mittelwertbildung verringert.

5.3.1.4 Bestimmung der Kennwerte

Die Ermittlung von Latenzzeit und Übertragungsrate ist durch eine lineare Regression nach der Methode der kleinsten Fehlerquadrate möglich. Durch diese Methode lassen sich die Parameter einer linearen Gleichung so bestimmen, dass die Summe der quadratischen Abweichungen zwischen den Messwerten y_i und den Funktionswerten $f(x_i)$ minimal wird:

$$\sum_{i=1}^{n}(y_i - f(x_i))^2 \rightarrow \text{Minimum} \tag{5.2}$$

Hierbei entspricht y_i der gemessenen Roundtrip-Zeit, x_i der jeweiligen Nachrichtengröße und $f(x_i)$ dem Funktionswert der linearen Gleichung 5.1. Für die Übertragungsrate und Latenzzeit ergibt sich somit:

$$\begin{pmatrix} \frac{1}{Übertragungsrate} \\ t_{Latenz} \end{pmatrix} = \frac{1}{2} * \begin{pmatrix} \sum_{i=1}^{n} x_i & n \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \end{pmatrix}^{-1} * \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n}(x_i * y_i) \end{pmatrix} \tag{5.3}$$

Aufgrund der im Experiment exponentiell ansteigenden Nachrichtengröße und somit ebenfalls exponentiell ansteigenden Roundtrip-Zeit konnte mit dem klassischen Regressionsverfahren eine hohe Abweichung zwischen gemessenen Roundtrip-Zeiten und mit der Regressionsgleichung berechneten Werten festgestellt werden. Diese Abweichungen sind besonders im Bereich kleiner Nachrichtengrößen zu erkennen, wie Abbildung 5.5 (gestrichelte Linie) am Beispiel der *DP-Toolbox* verdeutlicht.

Die Abweichungen im Bereich kleiner Nachrichtengrößen weisen auf eine unzureichend genaue Ermittlung der Latenzzeit hin. Die Ursache für die mangelnde Genauigkeit ist dabei das klassische Regressionsverfahren. Da das Verfahren den absoluten Fehler zwischen Messwert und berechnetem Wert minimiert, finden kleine absolute Fehler, wie sie im Bereich kleiner Nachrichtengrößen auftreten, weniger Berücksichtigung in der Parameterbestimmung, sodass die berechnete und reale Latenzzeit hier deutlich differieren.

Für eine genauere Bestimmung der Latenzzeit wurde daher auf ein modifiziertes Regressionsverfahren zurückgegriffen, welches die Summe der Quadrate des *relativen* Fehlers zwischen Messwert und Regressionsfunktion minimiert:

$$\sum_{i=1}^{n}\left(\frac{y_i - f(x_i)}{y_i}\right)^2 \rightarrow \text{Minimum} \tag{5.4}$$

Übertragungsrate und Latenzzeit ergeben sich hier aus:

$$\begin{pmatrix} \frac{1}{Übertragungsrate} \\ t_{Latenz} \end{pmatrix} = \frac{1}{2} * \begin{pmatrix} \sum_{i=1}^{n} \frac{x_i}{y_i^2} & \sum_{i=1}^{n} \frac{1}{y_i^2} \\ \sum_{i=1}^{n} \frac{x_i^2}{y_i^2} & \sum_{i=1}^{n} \frac{x_i}{y_i^2} \end{pmatrix}^{-1} * \begin{pmatrix} \sum_{i=1}^{n} \frac{1}{y_i^2} \\ \sum_{i=1}^{n} \frac{x_i}{y_i^2} \end{pmatrix} \tag{5.5}$$

Mit Hilfe dieses Verfahrens konnte die Latenzzeit der untersuchten Systeme deutlich genauer bestimmt werden, wie Abbildung 5.5 (durchgezogene Linie) zeigt.

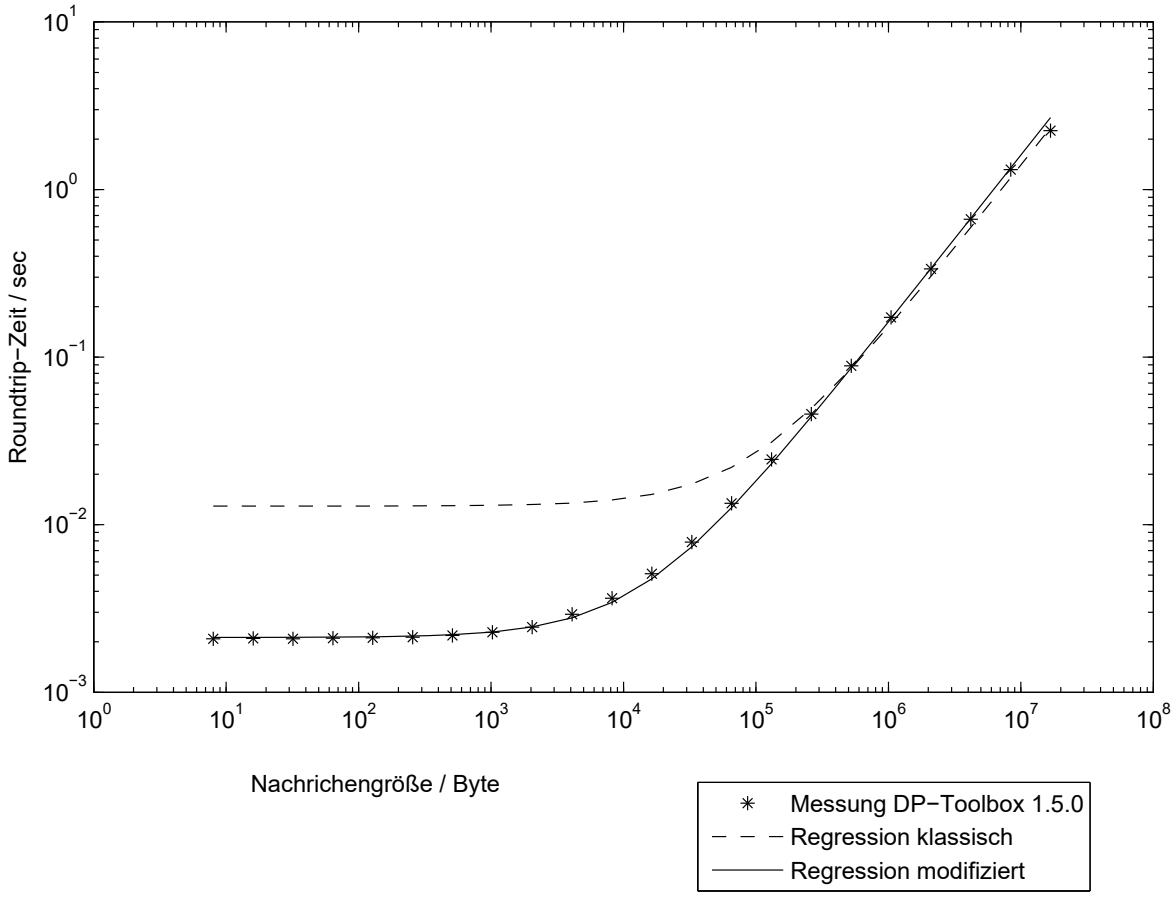


Abbildung 5.5: Roundtrip-Zeit der *DP-Toolbox* nach Messung, klassischer Regressionsrechnung und modifizierter Regressionsrechnung (doppelt logarithmische Darstellung)

5.3.2 Messergebnisse

Die Messungen wurden auf einem dedizierten Cluster durchgeführt. Auf Ebene der Hardware verfügte jeder Knoten über einen AMD-Athlon-Prozessor, getaktet mit 1500 MHz, sowie über 512 MB Arbeitsspeicher. Als Verbindungsnetzwerk wurde Gigabit Ethernet eingesetzt. Auf Softwareebene wurde ein Linux-Betriebssystem mit der Kernelversion 2.4.27 verwendet. Die ermittelten Latenzzeiten und Übertragungsraten sind in Tabelle 5.7 dargestellt. Die vollständige Ergebnisdarstellung in Diagrammform findet sich in Anhang A.1.

Zur Bewertung der Anbindungseffektivität der Kopplungsplattform an die SCE wurden auf der gleichen Hard- und Softwareplattform neben den Messungen mit Multi-SCE-Systemen auch direkte Messungen an den Kopplungsplattformen durchgeführt, die in [75] dokumentiert wurden. Diese Messungen fanden mittels kompilierter Programme statt, die direkten Zugriff auf native Schnittstellen der jeweiligen Plattform besaßen. Tabelle 5.8 fasst die Ergebnisse dieser Messungen zusammen.

System	Kopplungs-plattform	Latenz-zeit [ms]	Übertragungs-rate [MB/s]
MPI Toolbox (Matlab)	LAM MPI v7.1	0.09	36.7
MPI Toolbox (Octave)	LAM MPI v7.1	0.13	37.2
PVM Toolbox (Scilab)	PVM v3.4	0.16	12.0
DC-Toolbox (Matlab)	MPICH2 v1.0	0.33	34.1
DP-Toolbox (Matlab)	PVM v3.4	1.06	11.9
Parallelization Tk. (Matlab)	Matlab engine	25.34	37.2
MatlabMPI (Matlab)	Dateisystem	44.77	25.5
Beolab Toolbox (Matlab)	Matlab engine	81.78	37.1
DC-Toolbox (Matlab)	proprietär	535.23	2.9

Tabelle 5.7: Kommunikationsleistung von Multi-SCEs

Kopplungsplattform	Latenz-zeit [ms]	Übertragungs-rate [MB/s]
TCP-Sockets	0.04	46.6
MPICH2 v1.0	0.05	38.9
LAM MPI v7.1	0.05	37.6
PVM v3.4	0.10	31.6

Tabelle 5.8: Kommunikationsleistung von Kopplungsplattformen nach [75]

Der Vergleich der Multi-SCE-Systeme untereinander zeigt, dass Systeme, die Message-Passing-Bibliotheken als Kopplungsplattform einsetzen, die geringsten Latenzzeiten aufweisen. Diese liegen hier etwa im Bereich von 0.1 bis 1.1 Millisekunden. Systeme, die auf anderen Kopplungsplattformen basieren, besitzen dagegen deutlich höhere Latenzzeiten, beginnend bei 25 Millisekunden beim *Parallelization Toolkit*. Besonders auffallend ist die hohe Latenzzeit der *DC-Toolbox* unter Verwendung der proprietären Plattform (RPC-Schnittstelle, s. Abschn. 5.2.1), die im Bereich von einer halben Sekunde liegt.

Hinsichtlich der Übertragungsrate liegt der Großteil der untersuchten Systeme im Bereich von 12 bis 37 MB/s, unabhängig von der eingesetzten Kopplungsplattform. Die *DC-Toolbox* bildet mit einer Übertragungsrate von lediglich 3 MB/s hier die einzige Ausnahme.

Der Vergleich der Ergebnisse der Multi-SCE-Systeme mit den direkten Messungen aus Tabelle 5.8 zeigt, dass Multi-SCE-Systeme stets eine höhere Latenzzeit und eine geringere Übertragungsrate gegenüber dem direkten Zugriff auf die jeweilige Kopplungsplattform aufweisen. Der Grund dafür ist ein Overhead, der durch die Kapselung der Schnittstellen auf SCE-Ebene verursacht wird.

Die Systeme *MPI Toolbox* für Matlab und Octave weisen hierbei etwa die doppelte Latenzzeit auf, während die Übertragungsrate nur unwesentlich von der Messung mit der nativen Schnittstelle (LAM v7.1) abweicht. Die *DC-Toolbox* erreicht unter Verwen-

dung der Kopplungsplattform MPICH2 (Message-Passing-Schnittstelle, s. Abschn. 5.2.1) ebenfalls eine Übertragungsrate mit geringem Verlust gegenüber dem direkten Schnittstellenzugriff. Die Latenzzeit beträgt dagegen circa das Sechsfache der Latenzzeit mit direktem Zugriff. Im Fall der Systeme *DP-Toolbox* für Matlab und *PVM Toolbox* für Scilab, die auf der Plattform PVM basieren, beträgt dagegen die Übertragungsrate lediglich circa ein Drittel der Rate der nativen Schnittstelle. Hinsichtlich der Latenzzeit beider Systeme erreicht das System *PVM Toolbox* ähnliche Werte wie die direkte Messung, während die *DP-Toolbox* etwa die zehnfache Latenzzeit besitzt.

5.4 Zusammenfassung

In diesem Kapitel wurde eine qualitative und quantitative Analyse von Multi-SCE-Systemen vorgenommen. Dabei wurden acht Multi-SCE-Systeme untersucht, die einerseits verfügbar und andererseits auf aktuellen SCE-Versionen lauffähig waren. Von den acht Systemen unterstützten sechs die SCE Matlab und jeweils eines die SCEs Scilab und Octave. Bezüglich der Programmiermodelle wurde durch drei Systeme das RPC-Modell und durch sechs Systeme das Message-Passing-Modell unterstützt, wobei die *DC-Toolbox* beide Modelle unterstützte.

Der qualitative Vergleich der Systeme erfolgte anhand von Eigenschaften der in Abschnitt 4.4.1 diskutierten Schnittstellen von Multi-SCEs (Low-Level- und High-Level-Interface) sowie den Eigenschaften des SCE-Verbundes.

Auf Ebene des Low-Level-Interface (Schnittstelle zur Anbindung der Kopplungsplattform) wurde gezeigt, dass die Anbindung von Message-Passing-Systemen, das heisst externen Middlewarediensten, stets durch Kapselungsfunktionen realisiert wird. Die Anbindung der Plattform *Matlab engine* erfolgt für die untersuchten Systeme ebenfalls durch Kapselung, jedoch weist die native Schnittstelle dieser Plattform nur ein blockierendes, das heisst für die Parallelverarbeitung unbrauchbares Verhalten auf. Ein nichtblockierendes Verhalten wird durch die untersuchten Systeme nur über eine undokumentierte und stark systemabhängige Methode erreicht.

Auf Ebene des High-Level-Interface (Programmierschnittstelle) wurde deutlich, dass hinsichtlich des RPC-Programmiermodells alle in Abschnitt 2.2.2.3 diskutierten Erweiterungen des Modells in den untersuchten Systemen vertreten sind. Dabei verknüpfen einige Systeme die Prinzipien des vektoriellen RPC mit einer dynamischen Lastverteilung. Bezüglich des Message-Passing-Modells wurde gezeigt, dass die nativen Message-Passing-Schnittstellen in unterschiedlich starker Ausprägung dem SCE-Niveau angepasst sind. So wird das Prinzip des Array-Passing, das heisst das Versenden von Daten mit ihren Typ- und Dimensionsinformationen, durch die meisten Systeme realisiert. Besonders konsequent wird das Prinzip der Schnittstellenadaption durch die *DP-Toolbox* verfolgt, die als einziges System eine Nachrichtenspezifikation über Stringparameter sowie das Array-Scattering und -Gathering erlaubt.

Auf Ebene des SCE-Verbundes wurde erstmals eine Systematik für die diesbezüglichen Eigenschaften von Multi-SCE-Systemen entwickelt. Dabei wurde deutlich, dass für die untersuchten Systeme ein permanenter SCE-Verbund und interaktive SCE-Instanzen

einander ausschließen. Darüber hinaus wurde festgestellt, dass die *DC-Toolbox* auf dieser Ebene über besondere Eigenschaften (Mehrbenutzerbetrieb, Checkpoint-System) verfügt, die sie von anderen Systemen deutlich unterscheidet.

Der quantitative Vergleich der System stützte sich auf eine Untersuchung zur Kommunikationsleistung. Dazu wurde ein Ping-Pong-Verfahren mit einem 1-zu-1-Kommunikationsschema, das heisst mit zwei beteiligten Prozessen, verwendet. Dieses Verfahren wurde für alle expliziten Programmiermodelle (s. Abschn. 2.2.2) adaptiert, um vergleichbare Ergebnisse für die untersuchten Systeme zu erhalten. Die Ergebnisdarstellung erfolgte auf Basis von zwei verbreiteten Kenngrößen: Latenzzeit und Übertragungsrate. Die Ermittlung der Kenngrößen stützte sich auf einen linearen Modellansatz und eine entsprechende Regressionsrechnung. Aufgrund des konstanten relativen Messfehlers über einer exponentiell ansteigenden Messreihe musste für die exakte Ermittlung der Kenngrößen eine Anpassung des Regressionsverfahrens vorgenommen werden.

Die Ergebnisse der quantitativen Untersuchungen zeigten, dass Multi-SCE-Systeme mit einer Kopplung zu Message-Passing-Systemen die geringsten Latenzzeiten aufweisen. Besonders hohe Latenzzeiten (ca. 0.5 s) weist dagegen die *DC-Toolbox* unter Verwendung des RPC-Modells, das heisst der proprietären Kopplungsplattform auf. Hinsichtlich der Übertragungsrate liefert die *DC-Toolbox* in dieser Konfiguration ebenfalls die schlechtesten Ergebnisse.

Der Vergleich mit direktem Zugriff auf die Kopplungsplattformen zeigte, dass insbesondere für die Latenzzeit eine deutliche Differenz zu den untersuchten Systemen zu beobachten ist. Der Grund für diese Differenzen ist der resultierende Overhead bei der Anbindung von Kopplungsplattformen.

6 Weiterentwicklung des DP-Toolbox-Sets

Die Architektur der DP-Toolbox folgt seit dem Beginn ihrer Entwicklung einem Ansatz, bei dem die Funktionalitäten der Low-Level-, High-Level- oder SCE-Verbundebene durch dedizierte Teilsysteme (DPLOW, DPHIGH bzw. DPMM) abgedeckt werden. Die Menge aller Teilsysteme wird dabei als DP-Toolbox-Set bezeichnet. Der Vorteil dieses Ansatzes ist die prinzipielle Erweiterbarkeit des System auf unterschiedlichen Ebenen, wie Abbildung 6.1 verdeutlicht.

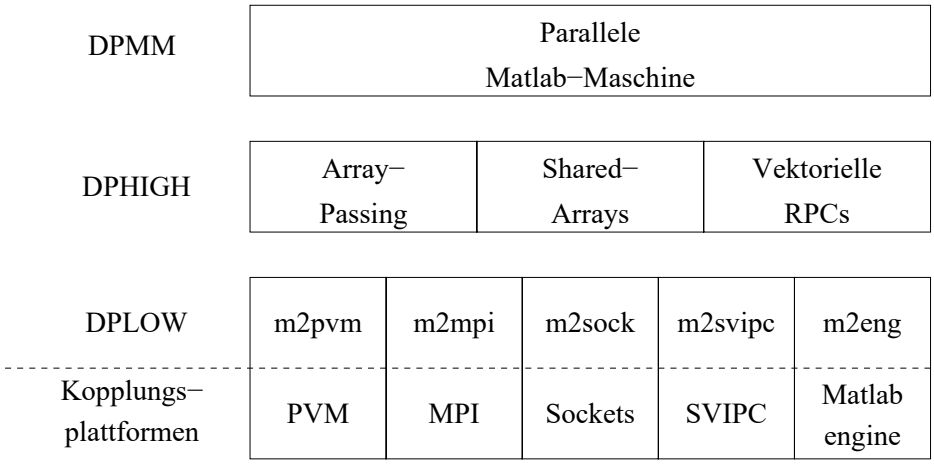


Abbildung 6.1: Architektur des DP-Toolbox-Sets

Die Entwicklung des DP-Toolbox-Sets erfolgte von 1995 bis 1999 durch S. Pawletta an der Universität Rostock. Während dieses Zeitraums entstanden die DP-Versionen 1.0 bis 1.4. Diese Toolboxversionen verfügten einerseits über einen hohen Funktionsumfang auf Ebene des Low-Level-Interface und andererseits über eine hohe Anzahl experimentell implementierter Funktionalitäten auf Ebene des High-Level-Interface und des SCE-Verbundes. Der Funktionsumfang erschwerte unerfahrenen Nutzern den Einstieg in das Softwaresystem, während der experimentelle Status einiger Funktionen die Toolbox für industrielle Anwender unattraktiv machte.

Im Jahr 2002 wurde die Entwicklung der Toolbox an der Hochschule Wismar durch den Autor der vorliegenden Arbeit wieder aufgenommen. Mit der DP-Version 1.5 erfolgte eine Anpassung des System an aktuelle Matlab-Versionen und eine erhebliche Reduktion des Funktionsumfangs auf Ebene des High-Level-Interface und des SCE-Verbundes. Das

primäre Ziel dieser Entwicklung war, die Toolbox für industrielle Anwender attraktiver zu gestalten.

Im Zuge der DP-Weiterentwicklung wurde deutlich, dass insbesondere für Forschungszwecke auf experimentelle Toolboxfunktionalitäten nicht vollständig verzichtet werden konnte. Aus diesem Grund erfolgte eine Aufspaltung der DP-Entwicklung in zwei Linien:

1. Weiterentwicklung der DP-Toolbox zu einem stabilen System für ingenieurtechnische Anwendungen (DP-1.7)

2. Weiterführung der experimentellen DP-Entwicklung zu Forschungs- und Lehrzwecken

Bei der Weiterentwicklung der DP-Toolbox zu einem System für ingenieurtechnische Anwendungen wurde, wie bereits bei der Entwicklung der DP-Version 1.5 begonnen (s. Abschn. 5.1), die Funktionalität der ursprünglichen Toolbox stark eingeschränkt und auf selten benötigte beziehungsweise experimentelle Funktionen verzichtet.

Die Weiterführung der experimentellen Entwicklung des DP-Toolbox-Sets erfolgte in Form neuartiger Teilsysteme auf Low- und High-Level-Ebene. Der Grundgedanke dieser Entwicklungen war, auf Ebene des Low-Level-Interface den Zugriff auf verschiedene Kopplungsplattformen zu ermöglichen und dem Anwender auf Ebene des High-Level-Interface alternative Programmiermodelle zur Verfügung zu stellen (s. Abb. 6.1). Die Teilsysteme sollten sich in die Architektur des DP-Toolbox-Sets eingliedern.

Auf Ebene des Low-Level-Interface entstanden vier neuartige Ansätze zur Anbindung von Kopplungsplattformen. Anhand der Vielzahl unterschiedlicher Low-Level-Interfaces zeigte sich, dass die Unterstützung alternativer Kopplungsplattformen durch ein einheitliches High-Level-Interface nur unter erheblichem Aufwand und unter Einschränkung der möglichen Programmiermodelle realisierbar ist. Die experimentellen Entwicklungen auf High-Level-Ebene konzentrierten sich daher auf die Realisierung einer neuartigen Nutzerschnittstelle für einen dedizierten Kopplungsansatz (MPI-2-Anbindung).

Im Rahmen der experimentellen Weiterentwicklung des DP-Toolbox-Sets entstand somit eine neuartige DP-Version mit einem MPI-2-basierten Low-Level-Interface und einem dazugehörigen High-Level-Interface mit alternativen Programmiermodellen. Darüber hinaus wurden drei weitere Prototypen zur Erprobung alternativer Kopplungsplattformen entwickelt, die über keine oder nur eingeschränkte Funktionalitäten der High-Level- und SCE-Verbundebene verfügen.

Das folgende Kapitel gibt eine Übersicht über das gesamte Spektrum der weitergeführten DP-Entwicklung. Es gliedert sich somit in vier Abschnitte:

1. Weiterentwicklung der DP-Toolbox für industrielle Anwendungen (DP-1.7)

2. Neuentwicklung einer MPI-2-basierten DP-Version (DP-MPI)

3. Weitere Prototypentwicklungen (DP-ME, DP-Java, DP-SVIPC)

4. Analyse der entwickelten Systeme (gemäß Kapitel 5)

Der Verlauf der bisherigen DP-Entwicklung inklusive der erfolgten Neu- und Weiterentwicklungen ist in Abbildung 6.2 dargestellt.

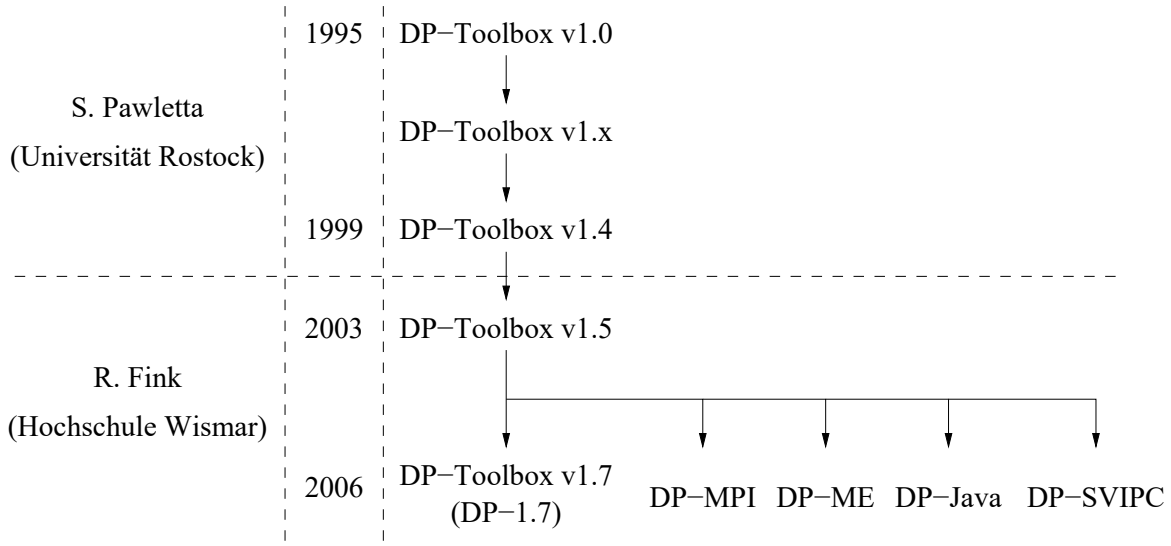


Abbildung 6.2: Zeitlicher Verlauf der DP-Entwicklung

6.1 Weiterentwicklung der DP-Toolbox für industrielle Anwendungen (DP-1.7)

Die Weiterentwicklung der DP-Toolbox zu einem stabilen System für ingenieurtechnische Anwendungen begann bereits im Rahmen der Entwicklung der DP-Version 1.5, bei der eine Anpassung an die Matlab-Version 6 vorgenommen wurde (s. Abschn. 5.1). So wurde die Unterstützung vorheriger Matlab-Versionen eingestellt und viele experimentelle Funktionalitäten, die vor allem die Ebene des SCE-Verbundes betrafen, aus der Toolbox entfernt.

Auf diesem Softwarestand setzt die im Folgenden dargestellte Weiterentwicklung zur DP-Toolbox-Version 1.7 auf. Als Entwicklungsrichtlinie wurde, beruhend auf der Parallelisierung industrieller Anwendungen und Anregungen industrieller Kooperationspartner, die folgenden Liste von Anforderungen an die Toolbox gestellt:

Ziel-SCE Matlab: Matlab ist die am weitesten verbreitete SCE und wegen der großen Anzahl von Toolboxen besonders im ingenieurtechnischen Bereich etabliert.

Kompatibilität mit Windows-Betriebssystemen: Windows-PCs sind die am weitesten verbreitete Plattform für Matlab.

Unterstützung des Message-Passing- und RPC-Programmiermodells: Beide Modelle sind für praktische Applikationen relevant.

Interaktive Instanzen: Interaktive Instanzen erleichtern die Fehlersuche und ermöglichen das Profiling paralleler Programme (s. Abschn. 5.2.3).

Temporärer SCE-Verbund, eine übergeordnete Instanz: Ein temporärer Verbund von SCE-Instanzen verhindert die Beeinflussung nacheinander ablaufender paralleler Programme. Das Arbeiten mit einer übergeordneten Instanz erlaubt die Vor- und Nachbereitung paralleler Programmläufe.

Hohe Stabilität, geringer Funktionsumfang: Das Multi-SCE-System soll erprobt sein. Durch die Beschränkung auf wesentliche Funktionen soll das Risiko eventueller Programmfehler minimiert und der Zugang zum System erleichtert werden.

Durch die DP-Toolbox-Version 1.5 und ihre Vorgänger wurden zahlreiche Kriterien bereits erfüllt. So wurde sie für die SCE Matlab konzipiert, besaß die geforderten Eigenschaften des SCE-Verbundes und wurde bereits von zahlreichen Anwendern erprobt.

Im Folgenden werden die Modifikationen der DP-Toolbox-Version 1.7 gegenüber der Version 1.5 auf den verschiedenen Systemebenen vorgestellt.

6.1.1 Low-Level-Interface

Das Low-Level-Interface der DP-Toolbox stellt eine SCE-Anbindung zur Kopplungsplattform PVM bereit. Dabei werden Funktionen der nativen C-PVM-Schnittstelle innerhalb einer Mex-Funktion gekapselt, sodass ein interaktiver Aufruf von Funktionen der PVM-Laufzeitbibliothek möglich ist. Durch das Low-Level-Interface werden Basisfunktionen der Kommunikation, Prozessidentifikation und der Prozessinstanziierung realisiert.

Die Version 1.5 der DP-Toolbox sah eine möglichst vollständige Kapselung der PVM-Bibliothek vor und besaß ein Low-Level-Interface mit einen Umfang von 48 Funktionen. Auf Ebene des Low-Level-Interface waren zusätzliche Pack- und Entpackroutinen vorgesehen, die Matlab-Arrays direkt in den PVM-Sendepuffer einfügen beziehungsweise direkt aus dem Empfangspuffer entnehmen konnten (`pvme_pkarray` bzw. `pvme_upkarray`). Die Funktionen waren in der Programmiersprache C implementiert und besaßen einen Umfang von circa 670 Codezeilen.

Das System DP-1.7 besitzt auf der Ebene des Low-Level-Interface lediglich 14 Funktionen, die alle notwendigen Basisfunktionalitäten bereitstellen. Für das Packen und Entpacken von Matlab-Arrays werden in der Version 1.7 Matlab-interne Funktionen verwendet (`mxSerialize` bzw. `mxDeserialize`), mit deren Hilfe beliebige Matlab-Daten in Bytefolgen transformiert und aus Bytefolgen rücktransformiert werden können. Für diese Funktionen ist lediglich die Entwicklung von Kapselungsroutinen mit einem Umfang von insgesamt 4 Codezeilen notwendig. Der PVM-Sendepuffer kann mit Hilfe dieser Funktionen direkt mit Bytefolgen gefüllt werden. Abbildung 6.3 verdeutlicht das Packen in den PVM-Sendepuffer in den Versionen 1.5 und 1.7 der DP-Toolbox. Das Entpacken aus dem PVM-Empfangspuffer verläuft analog dazu unter Verwendung der Funktionen `pvme_upkarray`, `pvm_upkbyte` und `mxDeserialize`.

Die Kompatibilität zu Windows-Betriebssystemen stellt auf Ebene des Low-Level-Interface für keine DP-Toolbox-Version ein Problem dar, da das PVM-Laufzeitsystem unter anderem für diese Betriebssysteme verfügbar ist.

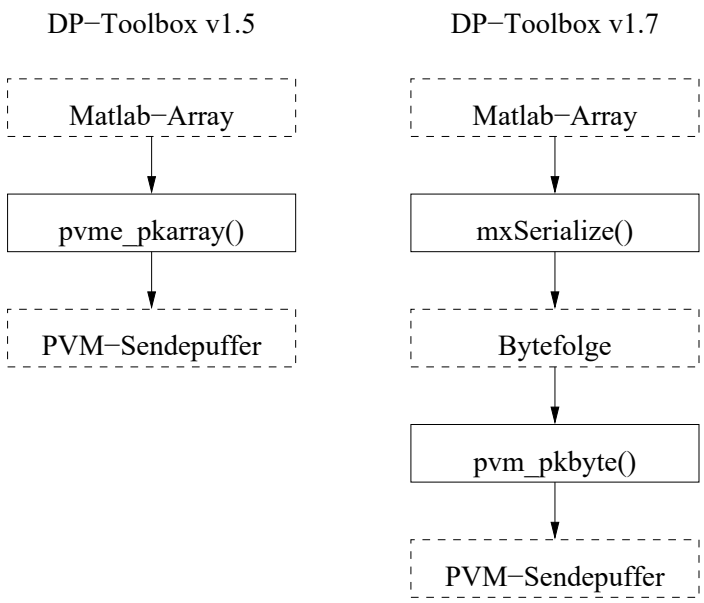


Abbildung 6.3: Packen von Matlab-Arrays in den DP-Versionen 1.5 und 1.7

6.1.2 High-Level-Interface

Das High-Level-Interface der DP-Toolbox bündelt Funktionen des Low-Level-Interface in Routinen, die an das hohe Programmierniveau der SCE angepasst sind. So kann das Verhalten der Routinen durch die Übergabe einer bestimmten Anzahl oder bestimmter Datentypen von Eingabeparametern bestimmt werden. Auf diese Weise ist es möglich, die Komplexität eines Funktionsaufrufs variabel zu gestalten und dem Niveau sowohl unerfahrener als auch professioneller Anwender gerecht zu werden. Im Gegensatz zum Low-Level-Interface fand keine Reduzierung der Anzahl der High-Level-Funktionen im System DP-1.7 statt.

Eine deutliche Reduzierung des Codeumfangs fand durch das Entfernen von Funktionalitäten zur stringbasierten Nachrichtenspezifikation statt (s. Abschn. 5.2.2). Im Sinne eines Multi-SCE-Systems mit hoher Stabilität und geringem Codeumfang stellten diese Funktionalitäten einen signifikanten Overhead dar. So besitzt die Sende-Funktion der DP-Version 1.5 einen Umfang von 213 Codezeilen, während die überarbeitete Sende-Funktion lediglich 43 Codezeilen umfasst. Die Nachrichtenspezifikation kann mit der Version 1.7 somit nur noch wie in der nativen PVM-Programmierung über Integer-Werte erfolgen.

Wie in Abschnitt 4.4.1 erwähnt, besitzt ein High-Level-Interface neben der Vereinfachung von Funktionsaufrufen die Aufgabe, vom Programmiermodell der Kopplungsplattform beziehungsweise des Low-Level-Interface zu abstrahieren und dem Anwender somit den Zugang zur SCE-basierten Parallelverarbeitung zu erleichtern.

In der Version 1.5 der DP-Toolbox wurde keine Abstraktion vom nativen Programmiermodell der Kopplungsplattform vorgenommen, sodass hier lediglich die Program-

mierung nach dem Message-Passing-Modell möglich war.

In der DP-Version 1.7 wurde dagegen, basierend auf der ursprünglichen Message-Passing-Schnittstelle, eine RPC-Schnittstelle nach dem Vorbild der *DC-Toolbox* implementiert. Die Schnittstelle umfasst dabei lediglich einen RPC-Ruf (`dpeval`), der die Server-Parameter, den Namen der entfernten Funktion und alle Teilaufgaben in Form von Cell-Arrays entgegennimmt. Der RPC-Ruf beinhaltet einen Master-Programmteil, der auf Seite des Clients abgearbeitet wird und einen Slave-Programmteil, der serverseitig zur Ausführung kommt. Die Schnittstelle realisiert dabei die RPC-Erweiterung *synchron vektorielles RPC* und arbeitet mit einer dynamischen Lastverteilung. Der Algorithmus, der die dynamische Lastverteilung auf Basis des Message-Passing-Programmiermodells ermöglicht, ist in Abbildung 6.4 dargestellt.

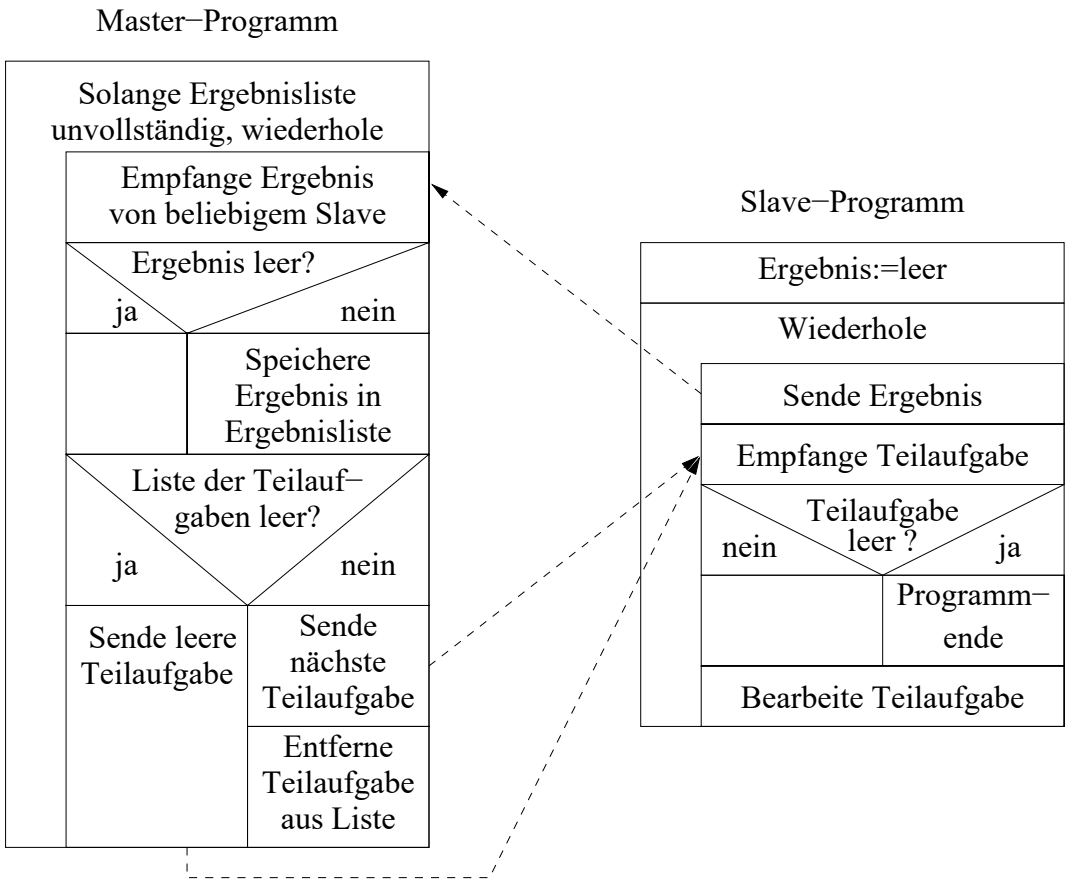


Abbildung 6.4: Algorithmus zur dynamischen Lastverteilung mittels Message-Passing-Programmierung

6.1.3 SCE-Verbund

Aufgrund der spezifischen Eigenschaften des SCE-Verbundes bei der DP-Toolbox ist die Arbeitsweise mit mehreren Instanzen stark an die klassische SCE-basierte Arbeitsweise

mit nur einer Instanz angelehnt. Die Toolbox unterstützt das Arbeiten mit einer übergeordneten Instanz in einem temporären SCE-Verbund. Somit kann der Anwender wie gewohnt am Arbeitsplatzrechner mit *einer* Matlab-Instanz arbeiten. Beim Start eines parallelen Programms in dieser Instanz wird eine Anzahl zusätzlicher Matlab-Instanzen gestartet, die ebenfalls das Parallelprogramm abarbeiten und anschließend wieder terminieren. Da die zusätzlichen Instanzen nicht durch zuvor abgelaufene Programme beeinflusst werden können, wird eine hohe Stabilität des SCE-Verbundes gewährleistet. Die erhöhte Programmstartzeit im temporären SCE-Verbund aufgrund von SCE-Instanziierungszeiten (s. Abschn. 5.2.3) stellt dabei nur bei kurzen Programmlaufzeiten einen Nachteil dar.

Das automatische Starten von Matlab-Instanzen erfolgt in der DP-Toolbox durch das PVM-Kommando `pvm_spawn`. Die Routine nimmt dabei unter anderem den Namen und Pfad des zu instanziierenden Programms, in diesem Falle Matlab, entgegen. Rückgabewerte der Funktion sind die Identifikationsnummern der gestarteten Prozesse, sodass eine Hierarchie aus startendem und gestarteten Prozessen entsteht. Die Windowsversion des PVM-Laufzeitsystems besitzt den Nachteil, dass hier lediglich nutzerdefinierte Parallelprogramme gestartet werden können, die in einem PVM-spezifischen Pfad hinterlegt sein müssen. Da dies für Programmsysteme wie Matlab, deren Programmdateien in komplexen Verzeichnisstrukturen hinterlegt sind, nicht möglich ist, kann eine native Matlab-Instanziierung unter Windows nicht erfolgen.

Durch die Verwendung eines temporären SCE-Verbundes sind in der DP-Toolbox alle Instanzen des Verbundes interaktiv bedienbar. Für den sinnvollen Einsatz interaktiver Instanzen müssen Ein- und Ausgabekanäle entfernter Instanzen auf einen lokalen Arbeitsplatzrechner „umgelenkt“ werden. Unter Verwendung Unix-ähnlicher Betriebssysteme kann zu diesem Zweck durch das X-Window-System eine Umlenkung der gesamten grafischen Oberfläche problemlos erfolgen. In Windows-Betriebssystemen besteht dagegen keine native Möglichkeit, Ein- und Ausgabekanäle umzulenken, sodass entfernte SCE-Instanzen nur direkt am entfernten Rechner bedienbar sind. Die Voraussetzung dafür ist jedoch die explizite Anmeldung eines Nutzers am entfernten Betriebssystem.

Die Version 1.5 der DP-Toolbox unterstützte lediglich Unix-ähnliche Betriebssysteme, sodass der SCE-Verbund durch das PVM-Laufzeitsystem instanziiert werden konnte. Die Bedienung aller Instanzen des Verbundes von einem Arbeitsplatzrechner konnte mit Hilfe des X-Window-Systems erfolgen.

Das System DP-1.7 unterstützt sowohl Windows- als auch Unix-ähnliche Betriebssysteme. Da sowohl die Instanziierung des Verbundes als auch die interaktive Bedienung von einem Arbeitsplatzrechner unter Verwendung von Windows problematisch ist, wird für diese Betriebssystemart lediglich ein SCE-Verbund auf dem lokalen Rechner unterstützt. Die Instanziierung und Hierarchisierung des Verbundes wird dabei unter Umgehung des PVM-Laufzeitsystems durchgeführt. Diese Kompromisslösung ist für Parallelrechner mit gemeinsamem physikalischen Speicher oder für das Testen paralleler Algorithmen auf herkömmlichen Workstations sinnvoll. Die Abarbeitung paralleler SCE-Programme in einem Verbund aus Windows-Rechnern ist dagegen nur unter Schwierigkeiten (Anmeldung, Matlab-Instanziierung und Programmstart müssen manuell erfolgen) möglich.

6.2 Neuentwicklung einer MPI-2-basierten DP-Version (DP-MPI)

Die Entwicklung dieses Systems ging aus der Erprobung neuartiger Ansätze zur Anbindung von Kopplungsplattformen hervor. Ausgangspunkt war die Kopplung eines MPI-2-Laufzeitsystems mit Matlab, die Bezeichnung für das System lautet daher *DP-MPI*. Darauf aufbauend wurde ein High-Level-Interface entwickelt, dass einerseits die Funktionalität und Semantik des MPI-2-Standards auf SCE-Ebene widerspiegelt und andererseits durch variable Eingabeparameter sehr effiziente Kommandos bereitstellt. Darüber hinaus bietet das High-Level-Interface die Shared-Memory-Programmierung als alternatives Programmiermodell an.

Wie in Abschnitt 6.1 werden im Folgenden die Eigenschaften des Systems auf Low-Level-, High-Level- und auf der Ebene des SCE-Verbundes präsentiert.

6.2.1 Low-Level-Interface

Der Ausgangspunkt für die Entwicklung des Low-Level-Interface von DP-MPI war die Erprobung verschiedener MPI-Laufzeitsysteme als Kopplungsplattform für ein Matlab-basiertes Multi-SCE-System. Dabei wurden die MPI-Implementierungen *MPICH*, *LAM* und *MPICH2* getestet. Die Implementierungen MPICH und LAM erwiesen sich in Zusammenhang mit Matlab als instabil. Unter Abschaltung der Java-Unterstützung für Matlab konnte für das MPI-Laufzeitsystem LAM ein stabiles Verhalten erreicht werden. Die Java-Unterstützung ist allerdings Voraussetzung für die grafische Matlab-Oberfläche, den Matlab-Editor sowie den Profiler, sodass eine Abschaltung die Produktivität des Anwenders wesentlich beeinträchtigen kann. Als einzige vollständig stabil anzubindende MPI-Implementierung erwies sich das System MPICH2, welches auch in der *DC-Toolbox* für Matlab als Kopplungsplattform eingesetzt wird (s. Abschn. 5.2.1). Da MPI einen Schnittstellenstandard darstellt, ist die Semantik der Funktionen in allen Implementierungen gleich. Somit ist ein Binden des Low-Level-Interface gegen alternative MPI-Implementierungen grundsätzlich möglich, sodass MPICH2 in diesem Sinne als „empfohlenes“ MPI-Laufzeitsystem angesehen werden kann.

Das Low-Level-Interface kapselt Funktionen des Standards in Mex-Funktionen, sodass diese interaktiv in Matlab aufrufbar sind. So wurde die Kapselung von Funktion zur Prozesserzeugung, zum Senden und Empfangen von Matlab-Arrays und Doublewerten, zur Prozessidentifikation und zur späten Prozesskopplung implementiert. Das direkte Senden und Empfangen von Matlab-Arrays wurde wie im System DP-1.7 durch die Transformation von Matlab-Arrays in Bytefolgen durch Matlab-eigene Routinen realisiert. Das Low-Level-Interface besitzt einen Umfang von 13 Kapselungs- und 7 Hilfsfunktionen.

Im Standard MPI besitzen Gruppenkennungen, so genannte *Kommunikatoren*, eine wesentliche Bedeutung. So bezieht sich zum Beispiel die Identifikationsnummer eines Prozesses immer auf einen speziellen Kommunikator. Kommunikator-Variablen sind Eingabeparameter aller wesentlichen MPI-Funktionen und sind insbesondere bei kollektiven Operationen von Bedeutung. Der Standard MPI-2 sieht keine Spezifikation des Daten-

typs von Kommunikatoren vor, sodass diese in MPI-Implementierungen durch verschiedene Datenstrukturen abgebildet werden können. Das Low-Level-Interface realisiert die Transformation von Kommunikator-Datentypen in Matlab-Datentypen unabhängig vom MPI-Laufzeitsystem. Dafür wird der Kommunikator mittels einer Hilfsfunktion in ein eindimensionales Matlab-Array aus Bytewerten kopiert. Auf diese Weise wird dem Anwender auf Low-Level-Ebene eine MPI-Programmierung ermöglicht, die dem nativen Standard in C oder Fortran entspricht.

6.2.2 High-Level-Interface

Das High-Level-Interface von DP-MPI vereinfacht die Routinen des Low-Level-Interface analog zur DP-Toolbox durch Funktionen mit variablen Eingabeparametern. Somit können auf Ebene der Nutzerschnittstelle Funktionen des nativen MPI-Standards vereinfacht aufgerufen werden. Die Bezeichnungen der Funktionen bleiben auf dieser Ebene an den Standard angelehnt.

6.2.2.1 Kollektive Operationen

Neben Vereinfachungen des MPI-Standards werden durch das High-Level-Interface kollektive Operationen bereitgestellt. So erfolgt das Array-Scattering beziehungsweise -Gathering sowie das Broadcasting in Analogie zum MPI-Standard. Für den Gather-Befehl bedeutet dies beispielsweise, dass alle beteiligten Instanzen den jeweiligen Befehl ausführen müssen, das Ergebnis der Operation aber nur in einer spezifizierten Wurzelinstanz zur Verfügung steht. Auf diese Weise können derartige Operationen in Parallelprogrammen sehr übersichtlich, weil ohne zusätzliche Verzweigungen, implementiert werden. Die Vereinfachung bei der Programmerstellung verdeutlicht Tabelle 6.1 in einer Gegenüberstellung des Array-Scattering mittels DP-1.7 und DP-MPI. Die Variable `A` wird dabei unter allen Prozessen aufgeteilt und die jeweiligen Teile in `B` hinterlegt. Die Liste der Prozess-Identifikationsnummern ist im Beispiel von DP-1.7 in der Variable `tid` hinterlegt.

DP-1.7	DP-MPI

`A=[1,2,3,4,5];`
`if dpparent==dpmyid % wenn Vaterprozess`
`  dpscatter(tid,A);`
`end`
`B=dprecv(dpparent);`
	`A=[1,2,3,4,5];`
	`B=MPI_Scatter(A);`

Tabelle 6.1: Array-Scattering mittels DP-1.7 und DP-MPI

6.2.2.2 Standard-Kommunikator

Neben den vom MPI-Standard vorgeschriebenen Kommunikatoren[1] wird ein weiterer Default-Kommunikator, `MPI_COMM_MATLAB`, durch das High-Level-Interface bereitgestellt. Dieser dient als Standardkommunikator für alle High-Level-Funktionen und ist die Voraussetzung für eine spätere Prozesserzeugung beziehungsweise -kopplung im temporären SCE-Verbund mit übergeordneter Instanz. In Analogie zum Kommunikator `MPI_COMM_WORLD`, der im MPI-Standard alle zeitgleich instanziierten Prozesse umfasst, beinhaltet `MPI_COMM_MATLAB` auf Matlab-Ebene alle Instanzen des Verbundes.

6.2.2.3 Shared-Memory-Schnittstelle

Aufbauend auf dem Message-Passing-Programmiermodell des MPI-Standards wurden auf Ebene des High-Level-Interface Funktionen zur Shared-Memory-Programmierung implementiert.

Jede gemeinsame Variable wird von einem Hintergrundprozess (Shared-Memory-Daemon bzw. shmd) verwaltet, der vom gesamten SCE-Verbund bei Deklaration einer Variable gestartet wird und eine Server-Programmschleife abarbeitet. Die clientseitige Kommunikation mit dem jeweiligen Hintergrundprozess erfolgt mit Hilfe eines individuellen Kommunikators. Für den einfachen Zugriff auf gemeinsame Variablen existiert eine Datenstruktur, die Kommunikatoren und Variablennamen zuordnet.

Der Zugriff auf gemeinsamen Speicher kann nur auf Basis vollständiger Variablen (nicht auf bestimmte Bereiche von Variablen) durch Lese- und Schreibkommandos erfolgen. Eine Matlab-Instanz kann dafür mit Hilfe des Kommandos `shmread` eine gemeinsame Variable in den lokalen Workspace kopieren. Durch das Kommando `shmwrite` werden analog dazu Variablen des Workspace in den gemeinsamen Speicher kopiert. Da der Hintergrundprozess jeweils nur einen Client bedienen kann, kann ein zeitgleicher Zugriff auf den gemeinsamen Speicher durch konkurrierende Matlab-Instanzen ausgeschlossen werden.

Für den Fall, dass zwischen Lese- und Schreibzugriff auf eine Variable deren Wert unverändert bleiben soll (z.B. wenn bestimmte Variablenbereiche durch eine Instanz modifiziert werden), stellt die Schnittstelle Funktionen für einen exklusiven Variablenzugriff bereit. Auf Seite des Daemon-Prozesses wird dafür die Kommunikation auf einen spezifischen Client eingeschränkt.

Die explizite Synchronisation zwischen den Matlab-Instanzen wird durch die vom MPI-Laufzeitsystem bereitgestellte Barrier-Routine realisiert.

6.2.3 SCE-Verbund

Die Eigenschaften des SCE-Verbundes wurden bei der Entwicklung von DP-MPI stark an die Eigenschaften des Systems DP-1.7 angelehnt. So wird auch hier die Arbeit mit einer übergeordneten Instanz und einem temporären SCE-Verbund aus interaktiven Instanzen unterstützt.

[1] `MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_NULL`

6.2.3.1 MPI_Comm_spawn im MPI-2-Standard

Das nachträgliche Starten von Instanzen wird in DP-MPI durch ein Spawn-Kommando realisiert. Die Funktion nimmt dafür den Namen eines ausführbaren Programms entgegen, seine Parameter und die Anzahl der zu startenden Programminstanzen. Im Gegensatz zu PVM, bei dem nach der Spawn-Operation alle PVM-Prozesse sofort über numerische Identifikatoren kommunizieren können, werden im MPI-Standard Interkommunikatoren zur Identifikation der startenden und gestarteten Prozesse verwendet. Um alle Prozesse in einem Kommunikator zusammenzufassen, muss nachträglich die Funktion `MPI_Intercomm_merge` ausgeführt werden, die einen gemeinsamen Kommunikator für startende und gestartete Prozesse erzeugt. Jede Kommunikationsoperation muss somit explizit mit dem gemeinsamen Kommunikator durchgeführt werden.

6.2.3.2 MPI_Comm_spawn in DP-MPI

Im System DP-MPI wurde auf High-Level-Ebene die Spawn-Funktion gegenüber dem MPI-Standard so modifiziert, dass statt beliebiger ausführbarer Programme Matlab-Instanzen mit spezifizierten Matlab-Programmen gestartet werden. Vor der Ausführung des SCE-Programms rufen gestartete und startende Instanzen die Funktion zur Zusammenfassung von Interkommunikatoren, `MPI_Intercomm_merge`, und legen deren Ergebnis im Standardkommunikator `MPI_COMM_MATLAB` ab. Durch die Verwendung dieses Standardkommunikators kann somit implizit im gesamten Matlab-Verbund kommuniziert werden. Unter anderem lassen sich so die Größe des Verbundes sowie die Identifikationsnummer (MPI-Rang) einer Instanz durch die Funktionen `MPI_Comm_size` und `MPI_Comm_rank` ohne weitere Angabe von Parametern ermitteln.

6.2.3.3 Implizites Spawning in DP-MPI

Neben der Adaption der Spawn-Operation wurde das Initialisierungskommando `MPI_Init` gegenüber dem Standard so modifiziert, dass optional ein Parameter angegeben werden kann, der die Anzahl der gewünschten Matlab-Instanzen spezifiziert. In diesem Fall wird innerhalb der Initialisierungsroutine der Name des übergeordneten Skriptes ermittelt und als Name des Parallelprogramms angenommen. Anschließend wird die gewünschte Anzahl Instanzen mittels `MPI_Comm_spawn` nach der in Abschnitt 6.2.3.2 beschriebenen Methode mit dem Parallelprogramm gestartet. In der übergeordneten Matlab-Instanz ist somit keine explizite Spawn-Operation notwendig, was die Übersichtlichkeit paralleler Programme deutlich erhöht. Tabelle 6.2 stellt die explizite und implizite Instanziierung anhand eines Minimalprogramms gegenüber. Der Start dieser Programme in einer übergeordneten Instanz bewirkt dabei das Starten von $nproc - 1$ Matlab-Instanzen, gefolgt von der Ausführung des gleichen Programms in allen Instanzen.

6.2.3.4 SCE-Verbund ohne übergeordnete Instanz

Eine weitere Möglichkeit der Instanziierung von MPI-Prozessen bietet das MPI-Hilfsprogramm `mpirun`. Mit diesem ist es möglich, mehrere MPI-Prozesse zeitgleich aus einer

Explizite Instanziierung	Implizite Instanziierung

`MPI_Init;`
`if MPI_Comm_rank==0 & MPI_Comm_size==1`
`  MPI_Comm_spawn(nproc-1,mfilename);`
`end`
`% Parallelcode`
`MPI_Finalize;`
	`MPI_Init(nproc);`
	`% Parallelcode`
	`MPI_Finalize;`

Tabelle 6.2: Explizite und implizite Instanziierung im System DP-MPI

Betriebssystemshell zu starten. Die Kommunikation der Prozesse kann dabei über den Standard-Kommunikator `MPI_COMM_WORLD` erfolgen. Diese Variante stellt beim compilerbasierten Arbeiten eine häufig verwendete MPI-Instanziierungsform dar. Eine Unterstützung dieser Instanziierungsform durch DP-MPI liefert somit für erfahrene MPI-Programmierer den idealen Einstieg in die SCE-basierte Parallelverarbeitung. Bezogen auf die Eigenschaften des SCE-Verbundes wird bei dieser Instanziierungsform ein temporärer SCE-Verbund ohne übergeordnete Instanz erzeugt. Da der Standardkommunikator `MPI_COMM_MATLAB` beim erstmaligen Aufruf eines DP-MPI-Kommandos mit `MPI_COMM_WORLD` identisch ist, ist die Abarbeitung der in Tabelle 6.2 dargestellten Programme auch bei dieser Art der Instanziierung möglich.

6.2.3.5 Windows-Unterstützung

Wie das System DP-1.7 unterstützt auch DP-MPI Windows-Betriebssysteme. Jedoch wird auch mit diesem System lediglich ein SCE-Verbund auf einem lokalen Rechner unterstützt. Da sich aus bisher ungeklärten Gründen mit der Windowsversion des MPICH2-Laufzeitsystems keine Matlab-Instanzen starten lassen, wurde auch hier die Instanziierung des Verbundes unter Umgehung des Laufzeitsystems durchgeführt. Die Bildung eines gemeinsamen Kommunikators erfolgt dabei über so genannte MPI-Portfunktionen, die an ein Client-Server-Verfahren angelehnt sind. Mit diesem Verfahren können unabhängige Prozesse zu einem MPI-Prozessverbund gekoppelt werden und in diesem wie gewohnt durch Kommunikatoren identifiziert werden. Dieses Verfahren der späten Instanzkopplung wird durch die *DC-Toolbox* ebenfalls verwendet.

6.3 Weitere Prototypentwicklungen

Im Rahmen der experimentellen Weiterentwicklung des DP-Toolbox-Sets entstanden neben dem voll entwickelten System DP-MPI drei weitere Prototypen, die primär der Erprobung neuartiger Anbindungen von Kopplungsplattformen dienten. Diese sind auf Ebene des High-Level-Interface nicht vollständig ausgebaut, konnten aber für eine Leistungsuntersuchung im Sinne von Kapitel 5 herangezogen werden.

6.3.1 Anbindung einer Thread-basierten Engine-Bibliothek (DP-ME)

Die Kopplungsplattform *Matlab engine* besitzt im Vergleich zu anderen Plattformen bemerkenswerte Eigenschaften. So verfügt die Engine-Bibliothek nativ über Funktionen zur Matlab-Instanziierung, zum entfernten Start von Matlab-Programmen sowie zur einseitigen Kommunikation über Matlab-Variablen. Somit ist im Gegensatz zu Systemen, die Message-Passing-Bibliotheken kapseln, hier keine explizite Adaption der Funktionen der Kopplungsplattform an die SCE-basierte Verarbeitung notwendig. Die native Engine-Bibliothek stellt dem Anwender das klassische, das heisst synchron skalare, RPC-Programmiermodell bereit. Für eine Verwendung der Bibliothek für Parallelverarbeitungszwecke muss dieses Modell gemäß Abschnitt 2.2.2.3 zu einem asynchronen beziehungsweise vektoriellen RPC-Modell erweitert werden. Den einfachsten Ansatz und die Voraussetzung für nachfolgende Erweiterungen in Form vektorieller RPCs stellt zunächst die Überwindung des blockierenden Verhaltens dar.

In Abschnitt 5.2.1 wurde erläutert, wie das blockierende Verhalten der Kopplungsplattform *Matlab engine* in bisherigen Engine-basierten Multi-SCE-Systemen umgangen wurde. Da hierbei in Interna der Engine-Bibliothek eingegriffen wird, handelt es sich um Lösungen, die stark vom verwendeten Betriebssystem abhängen und prinzipiell nicht unter Windows anwendbar sind.

Für die neuartige Anbindung der Engine-Bibliothek, im Folgenden als *DP-ME* bezeichnet, wurde diese so erweitert, dass das blockierende Verhalten durch die Verwendung von Threads[2] umgangen wird. Der gesamte Zugriff auf eine entfernte Matlab-Instanz ist dabei jeweils in einen Thread eingebettet, wobei die Kommunikation zwischen Hauptprogramm und Threads über gemeinsame Variablen erfolgt. Das Hauptprogramm stellt alle Funktionen der nativen Engine-Bibliothek bereit und besitzt darüber hinaus eine nichtblockierende Variante der Funktion `engEvalString`, eine Finalisierungsoperation für die nichtblockierende Funktion sowie eine Funktion zur Identifikation untätiger Instanzen, die die Implementierung einer dynamischen Lastverteilung auf höherer Ebene ermöglicht. Das Hauptprogramm ist durch die Mex-Schnittstelle, das heisst durch Kapselungsfunktionen, mit der übergeordneten Matlab-Instanz verbunden. Die Struktur des DP-ME-Systems verdeutlicht Abbildung 6.5.

Die Einbettung der Steuerung entfernter Instanzen in Threads erlaubt den betriebssystemunabhängigen Einsatz der DP-ME-Bibliothek in der Matlab-basierten Parallelverarbeitung. Der Verwendung unter Windows-Betriebssystemen steht jedoch im Wege, dass Instanzen hier nur auf *einem* dedizierten Rechner gestartet werden können. Für homogene Parallelverarbeitungsplattformen kann dieses Problem durch Eingriffe in die Betriebssystemkonfiguration, wie in [83] dokumentiert, umgangen werden.

[2]Thread: Leichtgewichtsprozess, d.h. ohne eigenes Code- und Datensegment

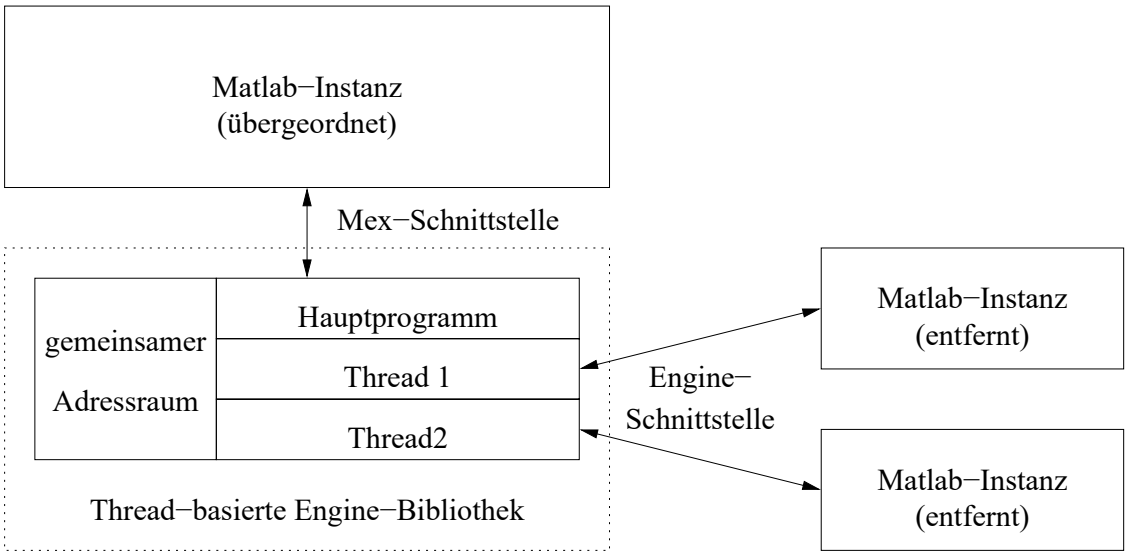


Abbildung 6.5: Struktur des Prototypsystems DP-ME

6.3.2 Anbindung einer Java-basierten Message-Passing-Bibliothek (DP-Java)

Die Anbindung von Message-Passing-Bibliotheken als Kopplungsplattform setzt in den meisten Fällen eine Kapselung von Funktionen auf Ebene compilerbasierter Sprachen voraus. Die entstehenden Kapselungsroutinen sind somit stark vom verwendeten Betriebssystem und der Verfügbarkeit der Laufzeitbibliotheken abhängig. In Abschnitt 5.2.1 wurde erwähnt, dass die Anbindung der Kopplungsplattformen entweder durch die Kapselung von Funktionen auf compilerbasierter Ebene oder durch die Verwendung SCE-interner Funktionen erreicht werden kann. Matlab bietet darüber hinaus die Möglichkeit, native Java-Schnittstellen direkt aufzurufen und somit Java-basierte Kopplungsplattformen ohne weitere betriebssystemabhängige Kapselungsfunktionen anzubinden.

Das Java-Laufzeitsystem stellt verschiedene Funktionspakete zur Interprozesskommunikation bereit. So besteht zum Beispiel die Möglichkeit der Stream-basierten Programmierung über Sockets, der Shared-Memory-Programmierung mittels Threads oder der RPC-Programmierung mittels RMI[3]. Für die Erprobung der Anbindung einer Java-basierten Kopplungsplattform wurde aufbauend auf der Socket-Schnittstelle eine minimale Message-Passing-Bibliothek in Java entwickelt. Anschließend wurden, basierend auf dieser alternativen und betriebssystemunabhängigen Kopplungsplattform, Teilfunktionalitäten der DP-Toolbox durch ein High-Level-Interface bereitgestellt. Das so entstandene Multi-SCE-System wird im Folgenden als *DP-Java* bezeichnet.

[3]RMI: Remote Method Invocation

6.3.2.1 Kopplungsplattform

Da die Java-basierte Message-Passing-Bibliothek ausschließlich für die Verwendung als Matlab-Kopplungsplattform entwickelt wurde, besitzt sie Eigenschaften, die sich von etablierten Message-Passing-Standards unterscheiden. So werden eindimensionale Arrays aus Double-Werten als einziger Nachrichtentyp unterstützt, die Bezeichnung der Java-Klasse, die die Bibliothek implementiert, lautet daher *VectorPassing*. Die Transformation von Matlab-Datentypen in Double-Vektoren wird auf Ebene des High-Level-Interface vorgenommen. Die Selektion von Nachrichten erfolgt wie in der ursprünglichen Version der DP-Toolbox durch Strings, jedoch wird dies hier bereits durch die Kopplungsplattform, das heisst durch die Klasse VectorPassing, realisiert. Darüber hinaus erfolgt die Identifikation von Prozessen durch einen String-Parameter, der einen Kommunikationsendpunkt durch Rechnernamen und Portnummer spezifiziert. Das Starten von Prozessen, das heisst eine Spawn-Operation, wird durch die Plattform nicht unterstützt.

Bei Instanziierung der Klasse VectorPassing werden zwei Java-Threads initialisiert, die Sende- beziehungsweise Empfangsoperationen ausführen. Die Kapselung der Sendeoperation innerhalb eines Threads hat den Zweck, das blockierende Verhalten von Schreibzugriffen auf Sockets zu umgehen und somit ein nichtblockierendes Sendeverhalten gemäß dem Message-Passing-Modell zu erreichen. Die Kapselung der Empfangsoperation in einem Thread dient dagegen dem ständigen Empfang von Nachrichten auf Socket-Ebene in einer Server-Schleife zum Zweck der Nachrichtenselektion durch das Hauptprogramm. Die Struktur der Kopplungsplattform verdeutlicht Abbildung 6.6.

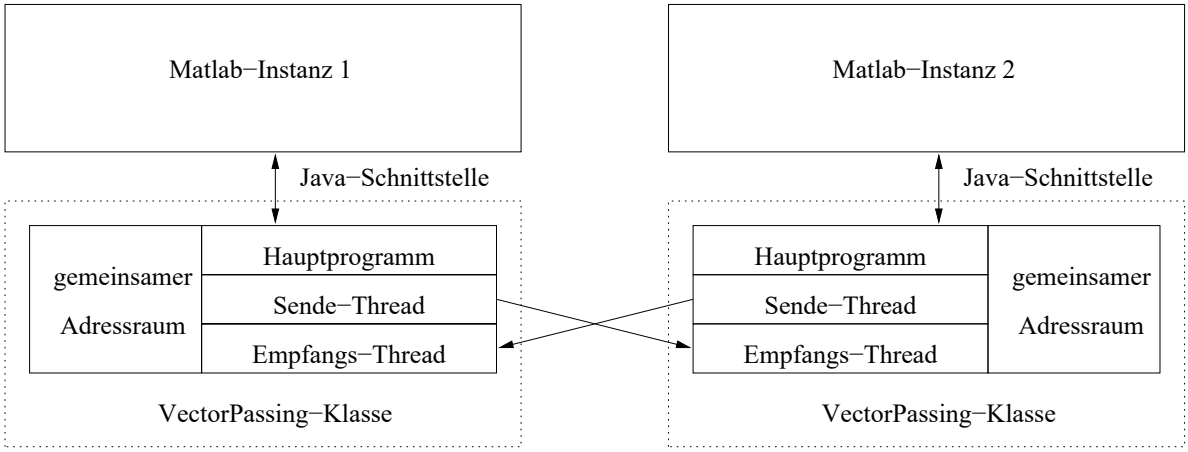


Abbildung 6.6: Struktur des Prototypsystems DP-Java

6.3.2.2 High-Level-Interface

Auf Ebene des High-Level-Interface wird eine Untermenge der DP-Toolbox-Funktionalität bereitgestellt. Die Transformation von Matlab-Datentypen in Nachrichten (in

diesem Fall Double-Vektoren) wird im Gegensatz zu allen bisher betrachteten Systemen durch Funktionen in SCE-Code durchgeführt, sodass auch auf dieser Systemebene die Unabhängigkeit vom Betriebssystem gewährleistet wird. Da die Klasse VectorPassing keine Funktionen zur Matlab-Instanziierung oder zum Matlab-Programmstart bereitstellt, mussten diese auf Ebene des High-Level-Interface durch Verwendung entfernter Shell-Kommandos (RSH) realisiert werden. Weil die Instanziierungsoperation in jedem Fall stark betriebssystemabhängig ist, stellt sie in diesem Rahmen lediglich eine Hilfsfunktion zur einfachen Erprobung von Funktionalitäten dar. Eine vollständig betriebssystemunabhängige Implementierung von SCE-basierten Spawn-Operationen ist aus den in Abschnitt 6.1.3 aufgeführten Gründen bisher unmöglich.

6.3.3 Anbindung von Shared-Memory-Betriebssystemdiensten (DP-SVIPC)

Da Shared-Memory-Architekturen durch die Verbreitung von Multikernprozessoren auch in ingenieurtechnischen Bereichen zunehmend an Bedeutung gewinnen, wurde die Anbindung einer Shared-Memory-Schnittstelle als Kopplungsplattform für Matlab-basierte Multi-SCEs erprobt. Diese Schnittstelle wird im Folgenden als *DP-SVIPC* bezeichnet.

Dafür wurde auf Techniken zur Interprozesskommunikation zurückgegriffen, die durch das Unix-Betriebssystem *System V* eingeführt wurden und daher als *System V interprocess communication mechanisms* (SVIPC) bezeichnet werden. Heute sind SVIPC-Schnittstellen Bestandteil vieler Unix-Derivate, unter anderem auch von Linux. Die Schnittstellen stellen Funktionen zur Verwaltung gemeinsamen Speichers, Semaphor-Operationen zur Prozesssynchronisation sowie Funktionen zur Verwendung von Nachrichtenwarteschlangen bereit.

Um mittels SVIPC auf gemeinsamen Speicher zuzugreifen, muss unter anderem dessen Größe durch die Funktion `shmget` spezifiziert werden. Während der Abarbeitung von `shmget` wird der gemeinsame Speicher somit bei Bedarf allokiert. Anschließend liefert die Funktion `shmat` die Adresse des ersten gemeinsamen Speicherelements. Unter Verwendung der Mex-Schnittstelle lässt sich diese Adresse mit Hilfe der Funktion `mxSetData` als Speicherort für die wesentlichen Daten (d.h. Daten ohne Typ- oder Dimensionsinformationen) einer Matlab-Variable spezifizieren. Somit kann eine Variable in mehreren Matlab-Instanzen als gemeinsame Variable deklariert werden, die im Gegensatz zur in Abschnitt 6.2.2 aufgeführten Methode (Shared-Memory abgebildet durch Message-Passing) nicht explizit in den Workspace beziehungsweise in den gemeinsamen Speicherbereich kopiert werden muss. Statt dessen ist eine Handhabung von gemeinsamen Variablen möglich, die dem Arbeiten mit globalen Variablen ähnelt.

Tests des DP-SVIPC-Systems haben dessen Funktionsfähigkeit nachgewiesen, allerdings birgt die verwendete Technik mehrere Risiken. So darf eine gemeinsame Variable nicht redimensioniert werden und ihren Typ nicht ändern. Darüber hinaus sind vollständige Zuweisungen an gemeinsame Variablen nach dem Schema `shm_A=B` nicht zulässig, auch wenn Typ und Dimension von `B` mit denen der gemeinsamen Variable übereinstimmen. Zulässig sind dagegen jegliche Zuweisungen von Teilfeldern, wie zum

Beispiel `shm_A(:)=B(:)`. Der Grund für diese Einschränkungen ist die Matlab-interne dynamische Speicherverwaltung, die bei Redimensionierung, Typänderung oder vollständiger Zuweisung einer Variable neuen Speicher zuweist und somit die durch `mxSetData` zugewiesene gemeinsame Speicheradresse überschreibt. Ein weiteres Risiko stellt die Tatsache dar, dass der einer Matlab-Variablen zugewiesene Speicher nicht durch Matlab selbst allokiert wurde, sodass beim Löschen der Variable oder der Freigabe nicht mehr benutzter Speicherbereiche (Garbage Collection) zwangsläufig Fehler beim Versuch der Deallokation auftreten.

Trotz der offensichtlichen Praxisuntauglichkeit dieser Anbindung konnte gezeigt werden, dass der Zugriff auf gemeinsamen Speicher in Matlab grundsätzlich möglich ist und der Zugriff auf gemeinsame Variablen in Analogie zur Benutzung globaler Variablen erfolgen kann.

Neben Funktionen zur Verwaltung gemeinsamen Speichers wurden im System DP-SVIPC Semaphor-Operationen der SVIPC-Schnittstelle gekapselt, mit deren Hilfe die notwendige Prozesssynchronisation realisiert werden konnte. Darüber hinaus wurde mit Hilfe von Semaphoren eine Funktion zur Barrier-Synchronisation mittels SVIPC in Matlab bereitgestellt. Die Funktionen zur Prozesssynchronisation erwiesen sich im Gegensatz zu den Speicherverwaltungsfunktionen als stabil.

6.4 Analyse neu- und weiterentwickelter Multi-SCE-Systeme

Die qualitativen und quantitativen Merkmale der im Rahmen der DP-Weiterentwicklung entstandenen Systeme und Prototypen werden zum Zweck der Vergleichbarkeit in den folgenden Abschnitten tabellarisch zusammengefasst. Um die Vergleichbarkeit mit bereits bestehenden Systemen zu gewährleisten, wurde auf eine starke Analogie zu den in Kapitel 5 dargestellten Tabellen geachtet.

6.4.1 Qualitative Merkmale

Die nachfolgenden Tabellen fassen die qualitativen Merkmale der entwickelten Systeme zusammen. Tabelle 6.3 stellt dabei zunächst die Ziel-SCE, die Kopplungsplattformen und die möglichen Programmiermodelle in Analogie zu Tabelle 4.5 gegenüber. Dabei wird deutlich, dass mit den neuentwickelten Systemen einerseits neuartige Anbindungen von Kopplungsplattformen erprobt wurden und andererseits das Spektrum aller expliziten parallelen Programmiermodelle abgedeckt wurde.

In Tabelle 6.4 sind die Kopplungsplattformen mit ihren vollständigen Bezeichnungen, die Form der Anbindung sowie der Status der Laufzeitbibliotheken (in SCE enthalten bzw. nicht enthalten) entsprechend Tabelle 5.2 zusammengefasst dargestellt. Hinsichtlich der Anbindungsform stellt dabei das System DP-Java eine Ausnahme dar, da dessen vollständig Java-basierte Kopplungsplattform weder durch Kapselungsfunktionen noch durch SCE-interne Funktionen angesprochen wird. Darüber hinaus repräsentiert der Prototyp DP-SVIPC eine Ausnahme in Bezug auf den Status der Laufzeitbibliotheken,

System	SCE	Kopplungs-plattform	Programmier-modelle
DP-1.7	Matlab	PVM	MP, RPC
DP-MPI	Matlab	MPI	MP, SHM
DP-ME	Matlab	Matlab engine	RPC
DP-Java	Matlab	Java MP-Bibliothek	MP
DP-SVIPC	Matlab	System V IPC	SHM

Tabelle 6.3: Neu- und weiterentwickelte Multi-SCE-Systeme (vgl. Tab. 4.5)

die hier statt durch die SCE oder das Multi-SCE-System durch das Betriebssystem bereitstellt werden.

System	Kopplungs-plattform	Anbindung	Laufzeit-bibliotheken
DP-1.7	PVM v3.4	Kapselung	nicht enthalten
DP-MPI	MPICH2 v1.0	Kapselung	nicht enthalten
DP-ME	Matlab engine	Kapselung	enthalten
DP-SVIPC	System V IPC	Kapselung	Betriebssystem
DP-Java	Java MP-Bibliothek	Java-Bindung	enthalten

Tabelle 6.4: Anbindung von Kopplungsplattformen in neu- und weiterentwickelten Multi-SCE-Systemen (vgl. Tab. 5.2)

Tabelle 6.5 fasst die Eigenschaften des High-Level-Interface von Systemen mit Message-Passing-Schnittstelle gemäß den Tabellen 5.4 und 5.5 zusammen. Hier wird deutlich, dass das Prinzip des Array-Scattering und -Gathering in allen entwickelten Systemen übernommen wurde. Andere Ansätze, wie zum Beispiel die Nachrichtenspezifikation über String-Parameter oder die Bereitstellung von Barrier-Operationen wurden dagegen nur teilweise realisiert.

System	Array-Passing	Message-Tag	Array-Scattering und -Gathering	Broadcast	Barrier	SCE-Reduktion
DP-1.7	ja	Integer	ja	ja	nein	nein
DP-MPI	ja	Integer	ja	ja	ja	nein
DP-Java	ja	String	ja	ja	nein	nein

Tabelle 6.5: Array-Passing, Spezifikation von Nachrichten und Gruppenoperationen in neu- und weiterentwickelten Multi-SCE-Systemen (vgl. Tab. 5.4 u. 5.5)

In Tabelle 6.6 ist die Realisierung von RPC-Rufen in Systemen, die das RPC-Programmiermodell erlauben, in Analogie zu Tabelle 5.3 zusammengefasst dargestellt. Diese

Tabelle zeigt, dass mit der RPC-Schnittstelle in DP-1.7 eine voll funktionsfähige RPC-Schnittstelle in Anlehnung an die entsprechende Schnittstelle der DC-Toolbox (s. Tab. 5.3) realisiert wurde. Der Prototyp DP-ME stellt dagegen lediglich primitive Funktionen auf Basis des asynchron skalaren RPC-Modells bereit, sodass hier die Zerlegung und Lastverteilung auf Anwendungsebene erfolgen muss.

System	RPC-Variante	Zerlegung vektorielle RPC	Lastverteilung
DP-1.7	synchron vektoriell	Cell-Array	dynamisch
DP-ME	asynchron skalar	manuell	manuell

Tabelle 6.6: Realisierung von RPC-Rufen in neu -und weiterentwickelten Multi-SCE-Systemen (vgl. Tab. 5.3)

Die Eigenschaften des SCE-Verbundes in den neu- und weiterentwickelten Systemen sind in Tabelle 6.7 zusammengefasst dargestellt. Hierbei wird deutlich, dass die Anbindung der Thread-basierten Engine-Bibliothek wie alle übrigen Engine-basierten Systeme einen permanenten SCE-Verbund unterstützt, in dem der Programmstart durch den Eingabekanal erfolgt. Wie bei den bereits bestehenden Systemen wird dies auch hier auf Kosten fehlender Interaktivität des SCE-Verbundes erreicht. Das System DP-SVIPC stellt auch hier eine Ausnahme dar, da durch diesen Prototyp keine Operationen zum Programmstart bereitgestellt werden. Der Programmstart muss hier manuell durch den Anwender vorgenommen werden, sodass zwangsläufig alle Instanzen des Verbundes interaktiv bedienbar sein müssen und keine übergeordnete Instanz existieren kann.

System	Lebensdauer	Programmstart durch	interaktive Instanzen	Übergeordnete Instanz	Mehrbenutzerfähig
DP-1.7	temporär	SCE-Start	ja	ja	nein
DP-MPI	temporär	SCE-Start	ja	ja	nein
DP-Java	temporär	SCE-Start	ja	ja	nein
DP-ME	permanent	Eingabekanal	nein	ja	nein
DP-SVIPC	-	manuell	ja	nein	nein

Tabelle 6.7: Eigenschaften des SCE-Verbundes in neu- und weiterentwickelten Multi-SCE-Systemen (vgl. Tab. 5.6)

6.4.2 Quantitative Merkmale

Für die entwickelten Multi-SCE-Systeme wurden mit dem in Abschnitt 5.3 vorgestellten Messverfahren die Kennwerte Übertragungsrate und Latenzzeit bestimmt. Eine ausführliche Ergebnisdarstellung in Form von Diagrammen erfolgt in Anhang A.2.

In Zusammenhang mit dem System DP-MPI wurde hier auch erstmals das an die Shared-Memory-Programmierung angepasste Messverfahren implementiert. Die Shared-Memory-Anbindung von DP-SVIPC konnte in diesem Zusammenhang nicht ausführlicher betrachtet werden, da diese speziell für den Einsatz auf Shared-Memory-Hardware vorgesehen ist. Ein Betrieb auf der in Abschnitt 5.3.2 beschriebenen Clusterplattform ist daher prinzipiell nicht möglich.

System	Kopplungs-plattform	Programmier-modell	Latenz-zeit [ms]	Übertragungs-rate [MB/s]
DP-MPI	MPICH2 v1.0	MP	0.28	20.6
DP-1.7	PVM v3.4	MP	0.58	12.1
DP-MPI	MPICH2 v1.0	SHM	1.57	17.4
DP-Java	Java MP-Bibliothek	MP	2.55	2.1
DP-ME	Matlab engine	RPC	27.48	33.8
DP-1.7	PVM v3.4	RPC	804.65	15.9

Tabelle 6.8: Kommunikationsleistung von neu- und weiterentwickelten Multi-SCEs (vgl. Tab. 5.7)

Tabelle 6.8 fasst die ermittelten Kennwerte der neu- und weiterentwickelten Systeme zusammen. Ein Vergleich der Ergebnisse mit Tabelle 5.7 zeigt die Stärken und Schwächen der neu- und weiterentwickelten Systeme.

So konnte die Latenzzeit der Message-Passing-Schnittstelle der DP-Version 1.7 auf etwa die Hälfte der Latenzzeit der Version 1.5 reduziert werden, sodass diese hier im Bereich von 0.5 Millisekunden liegt. Die Übertragungsrate der Schnittstelle besitzt gegenüber der DP-Version 1.5 dagegen noch die gleiche Größenordnung.

Die RPC-Schnittstelle des Systems DP-1.7 besitzt mit 0.8 Sekunden die höchste Latenzzeit unter allen untersuchten Systemen. Der Grund für diesen relativ hohen Wert ist die Tatsache, dass diese Schnittstelle auf einem temporären SCE-Verbund basiert, der nur für die Dauer eines RPC-Rufes instanziiert wird. Die hohe Latenzzeit ist somit im Wesentlichen auf die hohe Matlab-Instanziierungszeit zurückzuführen. Die Übertragungsrate der RPC-Schnittstelle liegt mit 16 MB/s etwa um 30 Prozent über der Rate der darunterliegenden Message-Passing-Schnittstelle. Da abstrahierende Programmschichten durch den zusätzlichen Overhead prinzipiell über eine geringere Kommunikationsleistung als die darunterliegenden Programmschichten verfügen müssten, stellt dieses Ergebnis eine bisher ungeklärte Anomalie dar.

Aufgrund der äquivalenten Kopplungsplattform und des nicht abstrahierenden Programmiermodells können die Message-Passing-Schnittstellen der Systeme DP-MPI (Tab. 6.8) und DC-Toolbox (Tab. 5.7) direkt miteinander verglichen werden. Dabei ist festzustellen, dass die Latenzzeiten beider Systeme etwa übereinstimmen. Bezüglich der Übertragungsrate erreicht das System DP-MPI dagegen nur 60 Prozent der Übertragungsrate der DC-Toolbox.

Die Shared-Memory-Schnittstelle des Systems DP-MPI weist dagegen die fünffache Latenzzeit der untergeordneten Message-Passing-Schnittstelle auf und erreicht etwa 85

Prozent ihrer Übertragungsrate. Hier bestätigt sich somit das Prinzip der geringeren Kommunikationsleistung auf höheren Programmebenen. Die Ursache für die stark erhöhte Latenzzeit ist dabei der Overhead, den eine erhöhte Anzahl an Sende- und Empfangsoperationen sowie die zusätzlichen Barrier-Operationen verursachen, um ein dem Ping-Pong-Messverfahren äquivalentes Kommunikationsschema zu ermöglichen.

Das System DP-ME weist Kennwerte auf, die in Höhe der Kennwerte bestehender Engine-basierter Multi-SCE-Systeme liegen. So entspricht die Latenzzeit etwa der Latenzzeit des *Parallelization Toolkit* (Tab. 5.7), während die Übertragungsrate 90 Prozent der Übertragungsrate der Systeme *Beolab Toolbox* und *Parallelization Toolkit* erreicht. Die Messwerte deuten somit auf einen relativ geringen Overhead der Thread-basierten Engine-Bibliothek gegenüber der Engine-Anbindung in existierenden Systemen hin.

Das System DP-Java weist eine Latenzzeit von 2.5 Millisekunden auf und liegt somit nur leicht über der Latenzzeit von Systemen, die etablierte Message-Passing-Systeme verwenden. Unter allen untersuchten Systemen weist DP-Java allerdings die geringste Übertragungsrate in Höhe von 2.1 MB/s auf. Die Ursache für die geringe Übertragungsrate konnte bisher nicht geklärt werden.

6.5 Zusammenfassung

In diesem Kapitel wurden die im Rahmen der Weiterentwicklung des DP-Toolbox-Sets entstandenen Systeme und Prototypen diskutiert. Dabei wurden zwei Entwicklungslinien verfolgt: die Weiterentwicklung der DP-Toolbox zu einem stabilen System für industrielle Anwendungen (DP-1.7) einerseits sowie die Weiterführung der experimentellen DP-Entwicklung andererseits. Im Rahmen der experimentellen Weiterentwicklung wurden mehrere alternative Ansätze zur Anbindung von Kopplungsplattformen erprobt, von denen einer (MPI-2-Anbindung) in die Entwicklung einer experimentellen DP-Version mit alternativen Programmiermodellen (DP-MPI) mündete. Weitere erprobte Ansätze zur Anbindung von Kopplungsplattformen waren die Thread-basierte Anbindung einer Engine-Bibliothek (DP-ME), die Anbindung einer Java-basierten Message-Passing-Bibliothek (DP-Java) sowie die Anbindung von Shared-Memory-Betriebssystemdiensten (DP-SVIPC). Alle entstandenen Systeme und Prototypen wurden einer mit Kapitel 5 vergleichbaren Leistungsuntersuchung unterzogen.

Die Weiterentwicklung der DP-Toolbox zu einem System für industrielle Anwendungen basierte auf einer Liste von Anforderungen, die unter anderem Matlab als Ziel-SCE, die Unterstützung von Windows-Betriebssystemen, einen interaktiven SCE-Verbund sowie einen reduzierten Funktionsumfang vorsahen. Gegenüber der DP-Version 1.5 wurde auf Ebene des Low-Level-Interface der Funktionsumfang von 48 auf 14 Funktionen reduziert sowie die Routinen zur Serialisierung von Matlab-Daten (670 Codezeilen) durch Verwendung Matlab-interner Funktionen (4 Codezeilen) ersetzt. Auf Ebene des High-Level-Interface wurde einerseits auf die stringorientierte Spezifikation von Nachrichten verzichtet und andererseits ein zusätzliches Programmiermodell in Form von vektoriellem RPC mit dynamischer Lastverteilung bereitgestellt. Auf Ebene des SCE-Verbundes konnte lediglich eine lokale Windows-Unterstützung realisiert werden, darüber hinaus

waren für die Erfüllung der weiteren Anforderungen (temporärer SCE-Verbund, eine übergeordnete Instanz, interaktive Instanzen) keine Modifikationen der Toolbox notwendig.

Die Grundlage für die Entwicklung des Systems DP-MPI war die stabile Anbindung einer MPI-2-Laufzeitbibliothek. Auf Ebene des Low-Level-Interface wurde die Anbindung verschiedener MPI-Systeme an Matlab erprobt, wobei sich herausstellte, dass lediglich mit dem System MPICH2, welches auch durch die DC-Toolbox verwendet wird, eine stabile Anbindung erreicht werden konnte. Auf Ebene des High-Level-Interface wurde ein weiterer Standard-Kommunikator eingeführt (`MPI_COMM_MATLAB`), der die Menge aller Instanzen des SCE-Verbundes spezifiziert und durch das High-Level-Interface als Default-Kommunikator verwendet wird. Auf diese Weise können kommunikatorbasierte Funktionen (das betrifft nahezu alle Funktionen von DP-MPI) mit reduzierten Parameterlisten aufgerufen werden, sodass sich hier eine wesentliche Vereinfachung gegenüber dem MPI-Standard ergibt. Darüber hinaus wird durch das High-Level-Interface von DP-MPI das Array-Scattering und -Gathering in starker Anlehnung an den MPI-Standard bereitgestellt, sodass die Verwendung dieser Funktionen im Gegensatz zum System DP-1.7 ohne weitere Programmverzweigungen erfolgen kann. Eine weitere Besonderheit des High-Level-Interface ist die Bereitstellung einer Schnittstelle zur Shared-Memory-Programmierung, die unter anderem Funktionen zum expliziten Schreiben und Lesen gemeinsamer Variablen bereitstellt. Die Eigenschaften des SCE-Verbundes sind in DP-MPI stark an das System DP-1.7 angelehnt. So bietet das System ebenfalls nur eine lokale Windows-Unterstützung und arbeitet mit einem temporären Verbund interaktiver Instanzen. Im Gegensatz zu DP-1.7 wird sowohl die Arbeit mit als auch ohne übergeordnete Instanzen unterstützt.

Die weiteren diskutierten Entwicklungen betrafen die alternative Anbindung von Kopplungsplattformen. Durch den Prototyp DP-ME wurde ein neuer Ansatz erprobt, das blockierende Verhalten von entfernten Matlab-Instanzen (*Matlab engines*) zu umgehen und somit das ursprünglich synchron skalare RPC-Modell zu einem asynchron skalaren Modell zu erweitern. Dieser neue Ansatz basiert auf der Kapselung der Engine-Bibliotheksrufe in Threads und ist im Gegensatz zu den bisherigen Implementierungen (s. Abschn. 5.2.1) unabhängig vom Betriebssystem. Beim System DP-Java stand die vollständige Plattformunabhängigkeit des Multi-SCE-Systems im Vordergrund. So liegt das gesamte Softwaresystem in Matlab-Code beziehungsweise Java-Bytecode vor und benötigt keine weiteren Laufzeitbibliotheken. Das Prototypsystem DP-SVIPC hat trotz der fehlenden Stabilität gezeigt, dass eine native Shared-Memory-Anbindung prinzipiell möglich ist und die Verwendung gemeinsamen Speichers in Analogie zur Verwendung globaler Variablen erfolgen kann.

Die Diskussion der qualitativen Merkmale der entwickelten Systeme erfolgte teilweise schon durch die vorhergehenden Abschnitte, sodass zum Ende des Kapitels lediglich eine tabellarische Zusammenfassung gemäß Abschnitt 5.2 vorgenommen wurde. Beim quantitativen Vergleich der Systeme anhand ihrer Kommunikationsleistung wurde deutlich, dass mit der Message-Passing-Schnittstelle der DP-Version 1.7 die Latenzzeit gegenüber der Version 1.5 um die Hälfte reduziert werden konnte. Die Latenzzeit der RPC-Schnittstelle wies dagegen mit circa 0.8 Sekunden den höchsten Wert unter al-

len untersuchten Systemen auf, bedingt durch den temporären SCE-Verbund und der damit verbundenen Matlab-Instanziierungszeit. Die Message-Passing-Schnittstelle des Systems DP-MPI stimmte etwa mit der Latenzzeit der entsprechenden Schnittstelle der DC-Toolbox überein, was auf die Verwendung der gleichen Kopplungsplattform zurückzuführen ist. Die darüberliegende Shared-Memory-Schnittstelle wies dagegen erwartungsgemäß eine geringere Kommunikationsleistung auf. Das System DP-ME zeigte etwa die gleiche Kommunikationsleistung wie existierende Engine-basierte Systeme, während das System DP-Java die geringste Übertragungsrate unter allen untersuchten Systemen aufwies. Das Prototypsystem DP-SVIPC konnte wegen der fehlenden Unterstützung für Systeme mit physikalisch verteiltem Speicher nicht für eine vergleichende Untersuchung herangezogen werden.

7 Anwendungsorientierte Untersuchungen

Das folgende Kapitel thematisiert die Anwendung von Multi-SCE-Systemen zur parallelen Abarbeitung ingenieurtechnischer SCE-Applikationen. Dabei sind zwei Faktoren von wesentlichem Interesse: der Aufwand für den „Umbau“ eines sequentiellen Programms in ein paralleles Programm (Parallelisierungsaufwand) sowie der Laufzeitgewinn des parallelen Programms gegenüber dem sequentiellen Pendant (Speedup). Diese Faktoren spiegeln den Aufwand und Nutzen der Entwicklung einer parallelen Applikation wider und müssen stets gegeneinander abgewogen werden. Die Möglichkeit der Abschätzung von Parallelisierungsaufwand und Laufzeitgewinn ist daher eine wesentliche Voraussetzung für die Etablierung von Multi-SCE-Systemen in ingenieurtechnischen Bereichen.

In diesem Kapitel findet ein anwendungsorientierter Vergleich von Multi-SCE-Systemen bezüglich Parallelisierungsaufwand und Laufzeitgewinn statt. Der Vergleich stützt sich auf sechs ingenieurtechnische Anwendungen, die mit drei ausgewählten Multi-SCE-Systemen parallelisiert wurden. Die drei Systeme wurden insbesondere wegen der Bereitstellung alternativer Programmiermodelle ausgewählt.

Darüber hinaus erfolgt ein direkter Vergleich der sechs Anwendungen, in dem qualitative Anwendungsmerkmale und quantitativ ermittelter Parallelisierungsaufwand und Laufzeitgewinn zueinander in Bezug gesetzt werden. Das Ziel dieses Vergleiches ist, allgemeingültige Regeln zur qualitativen Abschätzung von Parallelisierungsaufwand und Laufzeitgewinn abzuleiten.

Für diese Betrachtung erfolgt zunächst die Darstellung der im Sinne der Parallelverarbeitung relevanten Anwendungsmerkmale. Anschließend werden die Anwendungen einzeln vorgestellt, ihre Anwendungsmerkmale diskutiert und die individuellen Parallelisierungsergebnisse präsentiert. Abschließend erfolgt der direkte Vergleich der Anwendungen durch Gegenüberstellung ihrer qualitativen Merkmale und des quantitativ ermittelten Parallelisierungsaufwands und Laufzeitgewinns.

7.1 Merkmale ingenieurtechnischer Anwendungen

Im Folgenden werden drei Anwendungsmerkmale vorgestellt, die einen wesentlichen Einfluss auf den Parallelisierungsaufwand und Laufzeitgewinn paralleler SCE-Applikationen ausüben können.

7.1.1 Struktur des Ablaufverhaltens

Das Ablaufverhalten eines parallelen Programms beschreibt die zeitliche Abfolge paralleler und sequentieller Programmabschnitte sowie von Koordinationspunkten bei der Programmabarbeitung.

Typischerweise besitzt ein paralleles Programm ein Ablaufverhalten, in dem sequentielle und parallele Programmabschnitte im Sinne des Ahmdahlschen Gesetztes auftreten. Zu Beginn und Ende eines parallelen Abschnittes erfolgt eine implizite oder explizite Koordination der Prozesse durch Kommunikation beziehungsweise Synchronisation. Darüber hinaus ist je nach Programmiermodell auch innerhalb eines parallelen Abschnittes eine Koordination unter den Prozessen möglich. Das Ablaufverhalten lässt sich vereinfacht mit Hilfe eines grafischen Schemas darstellen (s. Abb. 7.1).

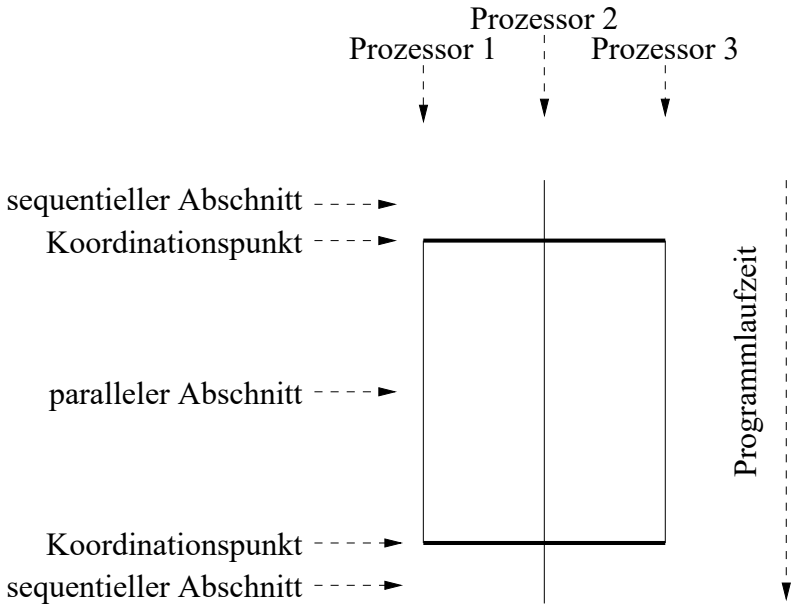


Abbildung 7.1: Grafische Darstellung des Ablaufverhaltens

Die Koordinationspunkte zu Beginn und Ende eines parallelen Abschnittes werden im Folgenden als *äußere Koordinationspunkte* bezeichnet, Koordinationspunkte innerhalb paralleler Abschnitte als *innere Koordinationspunkte*. Grundsätzlich sind somit drei Arten von Programmabschnitten möglich, deren zeitliche Abfolge das Ablaufverhalten eines parallelen Programms widerspiegelt:

- sequentielle Abschnitte
- parallele Abschnitte ohne innere Koordinationspunkte
- parallele Abschnitte mit inneren Koordinationspunkten

Die Parallelisierung verschiedener ingenieurtechnischer Anwendungen hat gezeigt, dass die in Abbildung 7.2 dargestellten Strukturen im Ablaufverhalten sehr häufig auftreten.

Man kann bezüglich des Ablaufverhaltens die folgenden vier Anwendungstypen unterscheiden:

Typ 1: ein paralleler Abschnitt ohne innere Koordinationspunkte

Typ 2: mehrere parallele Abschnitte ohne innere Koordinationspunkte

Typ 3: ein paralleler Abschnitt mit inneren Koordinationspunkten

Typ 4: mehrere parallele Abschnitte mit inneren Koordinationspunkten

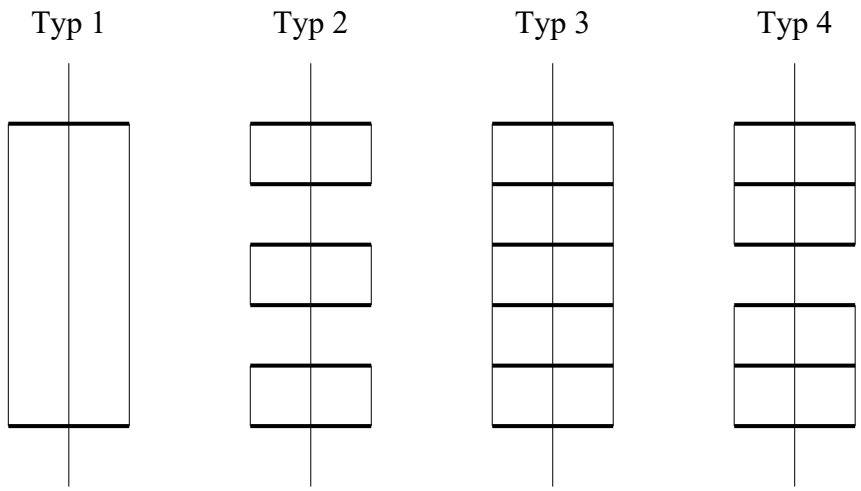


Abbildung 7.2: Typische Ablaufstrukturen ingenieurtechnischer Anwendungen

In [97] wurde gezeigt, dass parallele Optimierungs- und Simulationsanwendungen hinsichtlich der parallelisierten Programmebene (bezeichnet als Parallelisierungsebene) charakterisiert werden können. Dabei wurden die folgenden vier Parallelisierungsebenen identifiziert: unabhängige Optimierungsläufe, parallele Optimierungsmethoden, unabhängige Simulationsläufe und verteilte Modelle. Die hier vorgestellten typischen Ablaufstrukturen stellen dagegen eine abstraktere Systematik ohne spezifischen Anwendungshintergrund dar. Die typischen Ablaufstrukturen lassen sich eindeutig den in [97] diskutierten Parallelisierungsebenen zuordnen, wie Tabelle 7.1 zeigt.

7.1.2 Granularität

Wie in Abschnitt 2.3.1 erläutert, beschreibt die Granularität das Verhältnis von Rechenaufwand zu Kommunikationsaufwand innerhalb eines parallelen Programms. Bei der Beurteilung des zu erwartenden Laufzeitgewinns muss zunächst festgestellt werden, ob die Granularität einer Anwendung dem Verhältnis von Rechen- und Kommunikationsleistung der jeweiligen Parallelplattform entspricht.

Aussagen über die Granularität können tendenziell aus der Ablaufstruktur der Anwendung gewonnen werden. So besitzen Anwendungen mit Ablaufstrukturen gemäß Typ

Parallelisierungsebene	Ablaufstruktur
unabhängige Optimierungsläufe	Typ 1
parallele Optimierungsmethoden	Typ 2
unabhängige Simulationsläufe	Typ 1
verteilte Modelle	Typ 3

Tabelle 7.1: Zuordnung von Parallelisierungsebenen zu Ablaufstrukturen in Optimierungs- und Simulationsanwendungen

3 und 4, das heisst *mit* inneren Koordinationspunkten, häufig eine mittlere bis feine Granularität, während Anwendungen des Typs 1 und 2 meist als grobgranular einzustufen sind.

7.1.3 Anwendbarkeit paralleler Programmiermodelle

In Abschnitt 2.2.2.3 wurde bereits erläutert, dass das RPC-Programmiermodell mit seinen parallelen Erweiterungen nur für Probleme geeignet ist, bei denen innerhalb der Abarbeitung einer Teilaufgabe durch einen Prozess keine Koordination mit anderen Prozessen notwendig ist. Bezogen auf das Ablaufverhalten bedeutet dies, dass dieses Programmiermodell nur parallele Abschnitte *ohne* innere Koordinationspunkte (Typ 1 und 2, Abb. 7.2) zulässt. Der Grund dafür ist die Tatsache, dass Kommunikation und Synchronisation nur zu Beginn und Ende des entfernten Prozeduraufrufs, das heisst an den äußeren Koordinationspunkten stattfinden kann. Existieren dagegen parallele Abschnitte mit inneren Koordinationspunkten, so ist eine Lösung allein auf Basis des RPC-Modells nicht möglich.

Die etablierten parallelen Programmiermodelle Shared-Memory und Message-Passing lassen im Gegensatz dazu Ablaufstrukturen jeglichen Typs zu, da hier Kommunikation und Synchronisation zu jedem Zeitpunkt stattfinden kann.

Grundsätzlich kann daher zwischen RPC-fähigen und nicht RPC-fähigen Ablaufstrukturen unterschieden werden. Zu den RPC-fähigen Strukturen zählen Typ 1 und 2 in Abbildung 7.2, während Typ 3 und 4 nicht RPC-fähige Strukturen darstellen. Bei RPC-fähigen Anwendungen ist aufgrund der fehlenden inneren Koordinationspunkte grundsätzlich ein vergleichsweise geringer Parallelisierungsaufwand zu erwarten.

7.2 Anwendungsorientierter Vergleich von Multi-SCE-Systemen

Bei der anwendungsorientierten Untersuchung von Multi-SCE-Systemen waren sowohl der Parallelisierungsaufwand als auch das parallele Laufzeitverhalten von Interesse. Beide Kriterien wurden anhand unterschiedlicher Anwendungsszenarien und mit unterschiedlichen Multi-SCE-Systemen untersucht. Da das parallele Programmiermodell einen wesentlichen Einfluss auf den Parallelisierungsaufwand ausübt und dieser Einfluss insbe-

sondere bei Systemen mit alternativen Programmiermodellen von Interesse ist, wurden die folgenden Multi-SCE-Systeme in den Untersuchungen betrachtet:

- DC-Toolbox v2.0[1] (RPC- und Message-Passing-Programmierung)
- DP-1.7 (RPC- und Message-Passing-Programmierung)
- DP-MPI (Shared-Memory- und Message-Passing-Programmierung)

Die Untersuchung der Multi-SCE-Systeme fand anhand von vier etablierten Benchmarkproblemen und zwei realen ingenieurtechnischen Applikationen statt:

- Parameterstudie eines Feder-Masse-Systems
- Simulation gekoppelter Räuber-Beute-Systeme
- Numerisches Lösen einer partiellen Differentialgleichung
- Strömungssimulation mittels Lattice-Boltzmann-Verfahren
- Parameteroptimierung eines Abgasmodells
- Sicherheitstest eingebetteter Steuerungssoftware

Bei den ersten drei Applikationen handelt es sich um Benchmarkprobleme der SNE Comparison Parallel ([84]), die dem Vergleich paralleler Simulationstechniken dienen. Da die Definition der Benchmarkprobleme bereits 1994 erfolgte, verursachen die implementierten Applikationen auf heutiger Hardware kaum noch nennenswerte Rechenlasten. Aus diesem Grund erfolgte 2006 eine Neudefinition der SNE Comparison Parallel ([89]), die nunmehr die Strömungssimulation mittels Lattice-Boltzmann-Verfahren als eine der Aufgabenstellungen beinhaltet. Bei der Parameteroptimierung eines Abgasmodells sowie dem Sicherheitstest eingebetteter Steuerungssoftware handelt es sich um produktiv eingesetzte Entwurfsapplikationen der Automobilbranche.

Zur Untersuchung des Parallelisierungsaufwands wurden zwei Kennwerte herangezogen: das Verhältnis der Programmzeilenanzahl von paralleler zu sequentieller Programmvariante als Maß für den zusätzlichen Programmieraufwand sowie die Anzahl der genutzten Kommandos des jeweiligen Multi-SCE-Systems als Maß für den notwendigen Kenntnisstand des Programmierers in Bezug auf das jeweilige System. Für die Untersuchung des Laufzeitverhaltens wurden die etablierten Kennwerte Speedup und Effizienz verwendet. Da die Abhängigkeit des Speedups und der Effizienz von der Prozessorzahl in diesem Zusammenhang nur von untergeordnetem Interesse ist, wurden alle Laufzeituntersuchungen mit konstanten Prozessorzahlen durchgeführt.

Als Hard- und Softwareplattform für die Untersuchung des Laufzeitverhaltens diente ein Computercluster mit acht Knoten. Die Kopplung der Knoten erfolgte via Gigabit Ethernet. Von den acht Knoten waren jeweils vier mit einem Einzelprozessor (AMD

[1] im Folgenden auch als DC-2.0 bezeichnet

Athlon XP) und vier mit zwei SMP-Prozessoren (AMD Athlon MP) mit einer einheitlichen Taktrate von 1500 MHz ausgerüstet. Auf Ebene der Software wurde ein Linux-Betriebssystem mit Kernelversion 2.4.27 sowie Matlab in der Version 7.1 verwendet.

Der Programmcode der Benchmarkbeispiele in ihren sequentiellen und parallelen Varianten ist in Anhang B.1 bis B.4 aufgelistet. Für die realen Anwendungen muss aus Gründen der Geheimhaltung und des Programmumfangs auf die Darstellung des Codes verzichtet werden.

7.2.1 Parameterstudie eines Feder-Masse-Systems

Bei der Parameterstudie eines Feder-Masse-Systems wird der mittlere Bewegungsverlauf eines Feder-Masse-Systems innerhalb eines vorgegebenen Parameterbereiches stochastisch ermittelt.

7.2.1.1 Problembeschreibung

Das Feder-Masse-System wird beschrieben durch die Gleichung

$$\begin{aligned} 0 &= m\ddot{x}(t) + kx(t) + d\dot{x}(t) \\ \text{mit: } \dot{x}(0) &= 0, x(0) = 0.1, k = 9000, m = 450 \ . \end{aligned} \tag{7.1}$$

Der Dämpfungsfaktor d unterliegt einer Gleichverteilung im Intervall $[800; 1200]$. Gesucht ist der mittlere Bewegungsverlauf des Systems im Zeitintervall $[0; 2]$.

7.2.1.2 Sequentielle Implementierung

Die Lösung des Problems erfolgt näherungsweise durch numerische Integration. Als Integrationsverfahren wird das Runge-Kutta-Verfahren 4. Ordnung mit einer Schrittweite von $h = 0.001$ verwendet. Für die Verwendung des numerischen Integrationsverfahrens muss die Differentialgleichung 7.1 in ein System von Differentialgleichungen erster Ordnung überführt werden:

$$\begin{aligned} \dot{x_1}(t) &= -\frac{d}{m}x_1(t) - \frac{k}{m}x_2(t) \\ \dot{x_2}(t) &= x_1(t) \\ \text{mit: } x_1(0) &= \dot{x}(0) = 0 \ , \ x_2(0) = x(0) = 0.1 \ . \end{aligned} \tag{7.2}$$

Das Gleichungssystem kann in dieser Form in einer Ableitungsfunktion $\dot{\mathbf{x}} = f(\mathbf{x}, t)$ implementiert werden, die dem RK4-Algorithmus als zu integrierende Funktion übergeben wird. Das sequentielle Programm gliedert sich somit in die folgenden drei ineinander verschachtelten Teile: Hauptprogramm, RK4-Algorithmus und Ableitungsfunktion (s. Abb. 7.3).

Zur Berechnung des mittleren Bewegungsverlaufes wird im Hauptprogramm eine Schleife abgearbeitet, in der für 1000 Werte von d ein Simulationslauf durchgeführt wird. Zur Berechnung des mittleren Bewegungsverlaufes werden die Ergebnisse der Simulationsläufe aufsummiert und nach Abarbeitung der Schleife durch die Anzahl der Durchläufe dividiert.

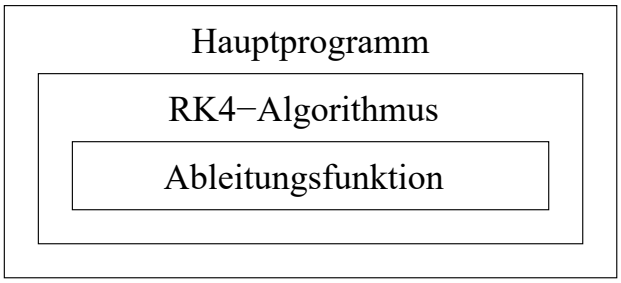


Abbildung 7.3: Programmteile der Parameterstudie eines Feder-Masse-Systems (sequentielle Implementierung)

7.2.1.3 Anwendungsmerkmale

Die parallel abzuarbeitenden Teilaufgaben werden in diesem Anwendungsbeispiel durch einzelne Simulationsläufe repräsentiert.

Da während der Abarbeitung einer Teilaufgabe keine Koordination mit anderen Instanzen notwendig ist und Teilaufgaben nur einmalig abzuarbeiten sind, liegt eine Ablaufstruktur mit *einem* parallelen Abschnitt *ohne* innere Koordinationspunkte (Typ 1, Abb. 7.2) vor. Die Granularität des Problems kann als grob eingestuft werden, da der verursachte Rechenaufwand deutlich gegenüber dem Kommunikationsaufwand zu Beginn und Ende des parallelen Abschnittes dominiert. Weil keine inneren Koordinationspunkte existieren, handelt es sich um ein RPC-fähiges Problem.

Da ein RPC-fähiges Problem vorliegt, ist der Parallelisierungsaufwand für diese Anwendung als gering einzustufen, während die grobe Granularität auf einen hohen Laufzeitgewinn hindeutet.

7.2.1.4 Parallele Implementierung

Das Anwendungsbeispiel wurde in den vier Varianten DP-1.7 (MP), DP-MPI (MP), DC-2.0 (RPC) sowie DP-1.7 (RPC) implementiert. Die Listings der Programmvarianten befinden sich in Abschnitt B.1 des Anhangs.

Im Fall der Message-Passing-Programmierung (MP) erfolgt zunächst der Start untergeordneter Instanzen sowie des parallelen Programms durch explizite (DP-1.7) beziehungsweise implizite (DP-MPI) Spawn-Operationen. Nachfolgend wird der Vektor der Dämpfungsparameter mit Hilfe von Array-Scattering-Funktionen unter allen beteiligten Instanzen aufgeteilt. Anschließend werden in jeder Instanz Simulationsläufe für die zugeordneten Werte von d durchgeführt und die Ergebnisse aufsummiert. Nach Abarbeitung der Schleife werden die Teilsummen durch Array-Gathering-Funktionen zusammengefasst und ihr Mittelwert gebildet. Da die Liste der zu bearbeitenden Teilaufgaben (einzelne Simulationsläufe) zu Beginn der Programmabarbeitung unter allen Instanzen aufgeteilt wird, handelt es sich hier um Lösungsvarianten mit statischer Lastverteilung.

Im Fall der RPC-Programmierung wird die Schleife im Hauptprogramm durch einen vektoriellen RPC-Ruf ersetzt. Für den RPC-Ruf wird die RK4-Routine als entfernt aus-

zuführende Funktion spezifiziert. Darüber hinaus erfolgt eine dem RPC-Ruf konforme Anpassung der RK4-Funktionsparameter (Ableitungsfunktion, Startzeit, Schrittweite, Anzahl Schritte, Anfangswerte, optionale Parameter) durch Cell-Arrays. Instanziierung und Programmstart auf untergeordneten Instanzen werden durch den RPC-Ruf gekapselt. Die Rückgabewerte des RPC-Rufes, die ebenfalls in Form von Cell-Arrays vorliegen, werden abschließend zum Zweck der Mittelwertbildung in Matrizenform überführt. Bei den RPC-basierten Varianten handelt es sich um Lösungen mit dynamischer Lastverteilung, da beide RPC-Rufe nach diesem Verteilungsprinzip arbeiten.

7.2.1.5 Ergebnisse bezüglich des Parallelisierungsaufwandes

Die Kennwerte zur Bewertung des Parallelisierungsaufwandes sind in Tabelle 7.2 zusammengefasst. Die Abkürzungen hinter den Bezeichnungen der Multi-SCE-Systeme beziehen sich in dieser und den nachfolgenden Tabellen des Kapitels auf das verwendete Programmiermodell.

Tabelle 7.2 zeigt, dass die Message-Passing-basierten Lösungsansätze gegenüber den RPC-basierten Lösungen einen höheren Programmieraufwand verursachten, was sowohl am höheren Codezeilenverhältnis als auch an der höheren Anzahl an Multi-SCE-Kommandos deutlich wird. Die Hauptgründe dafür sind einerseits die explizite Ausführung von Kommunikationsoperationen (Scatter/Gather) bei der Message-Passing-Programmierung und andererseits die Elimination der Hauptprogrammschleife im Fall der RPC-Programmierung, welche auch für das auffällige Codezeilenverhältnis von 0.97 im Fall des Systems DP-1.7 verantwortlich ist.

Beim Vergleich der Message-Passing-basierten Lösungen fällt auf, dass mit dem System DP-1.7 ein höherer Programmieraufwand gegenüber der DP-MPI-Variante notwendig war. Der Grund dafür ist die einerseits die an den MPI-Standard angelehnte Implementierung von Array-Scattering- und -Gathering-Funktionen in DP-MPI (s. Abschn. 6.2.2.1), die eine effektive Programmierung ohne zusätzliche Programmverzweigungen ermöglichen. Andererseits reduziert die implizite Instanziierung (s. Abschn. 6.2.3) durch DP-MPI den Programmieraufwand zusätzlich. Im Fall von DP-1.7 sind dagegen für Scatter- und Gatheroperationen jeweils zwei Kommandos notwendig (Scatter/Receive bzw. Send/Gather) sowie eine Unterscheidung zwischen Programmteilen für über- und untergeordnete Instanzen mittels Prozessidentifikation und anschließender Verzweigung (s. Tab. 6.1).

Der Vergleich der RPC-basierten Lösungen zeigt, dass bei Verwendung des Systems DC-2.0 neben dem RPC-Ruf zwei weitere Multi-SCE-Kommandos erforderlich waren. Der Grund dafür ist, dass jeder abgesetzte RPC-Ruf eine Speicherung seiner Parameter im DC-Jobmanager (s. Abschn. 5.2.3) zur Folge hat. Diese kumulative Speicherung führt zu steigenden Latenzzeiten bei der Ausführung nachfolgender RPC-Rufe. Um die parallele Programmlaufzeit zu jedem Durchlauf so gering wie möglich zu halten, wurden daher zwei zusätzliche Multi-SCE-Kommandos zum Auffinden und Löschen der Parameter des jeweils letzten RPC-Rufes eingefügt.

Multi-SCE-System	Anzahl Code-zeilen (LOC)	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Multi-SCE-Kommandos
sequentiell	32	-	-
DP-1.7 (RPC)	31	0.97	1
DC-2.0 (RPC)	32	1.00	3
DP-MPI (MP)	37	1.16	4
DP-1.7 (MP)	43	1.34	8

Tabelle 7.2: Parallelisierungsaufwand der Parameterstudie eines Feder-Masse-Systems

7.2.1.6 Ergebnisse bezüglich des Laufzeitverhaltens

In Tabelle 7.3 sind die Ergebnisse der Laufzeitmessungen zusammenfassend dargestellt. Unter Verwendung der Systeme DP-1.7 und DP-MPI konnten hier unabhängig vom Programmiermodell und der Art der Lastverteilung Speedupwerte nahe dem Idealwert von 12 und eine resultierende Prozessorauslastung von über 90 Prozent erreicht werden. Unter Verwendung des Systems DC-2.0 wurde dagegen lediglich ein Speedupwert von 9 erreicht, was durch die erhöhte Latenzzeit des Systems (s. Tab. 5.7) sowie den zusätzlichen Verwaltungsaufwand des DC-Jobmanagers zu erklären ist.

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	9075	-	-
DP-1.7 (RPC)	790	11.5	0.96
DP-MPI (MP)	813	11.2	0.93
DP-1.7 (MP)	819	11.1	0.92
DC-2.0 (RPC)	985	9.2	0.77

Tabelle 7.3: Laufzeitverhalten der Parameterstudie eines Feder-Masse-Systems (Prozessorzahl 12)

7.2.2 Simulation gekoppelter Räuber-Beute-Systeme

In diesem Anwendungsbeispiel wird die Populationsentwicklung von fünf gekoppelten Räuber-Beute-Systemen durch eine Einzelsimulation ermittelt.

7.2.2.1 Problembeschreibung

Ein System aus fünf Räuber-Beute-Populationen (v_1, v_2), (w_1, w_2), (x_1, x_2), (y_1, y_2) und (z_1, z_2) wird durch die folgenden Gleichungen beschrieben:

$$\begin{aligned}
\dot{v_1} &= a_v v_1 - b_v v_1 v_2 - c_v v_1^2 \\
\dot{v_2} &= -d_v v_2 + e_v v_1 v_2 - f_v v_2^2 + r_v \\
r_v &= v_2(g_v w_1 + h_v x_1 + j_v y_1 + k_v z_1) \\
\text{mit: } a_v &= 2, b_v = 0.5, c_v = 0.01, d_v = 0.2, e_v = 0.4, \\
f_v &= 0.2, g_v = 0.01, h_v = 0.02, j_v = 0.01, k_v = 0.03
\end{aligned} \tag{7.3}$$

$$\begin{aligned}
\dot{w_1} &= a_w w_1 - b_w w_1 w_2 - c_w w_1^2 + r_w \\
r_w &= w_1(-g_w v_2 + h_w x_2) \\
\dot{w_2} &= -d_w w_2 + e_w w_1 w_2 - f_w w_2^2 \\
\text{mit: } a_w &= 1, b_w = 0.5, c_w = 0.02, d_w = 0.1, e_w = 0.4, \\
f_w &= 0.04, g_w = 0.02, h_w = 0.04
\end{aligned} \tag{7.4}$$

$$\begin{aligned}
\dot{x_1} &= a_x x_1 - b_x x_1 x_2 - c_x x_1^2 + r_x \\
r_x &= -g_x x_1 v_2 \\
\dot{x_2} &= -d_x x_2 + e_x x_1 x_2 - f_x x_2^2 + s_x \\
s_x &= -h_x x_2 w_1 \\
\text{mit: } a_x &= 3, b_x = 0.9, c_x = 0.02, d_x = 0.2, e_x = 0.2, \\
f_x &= 0.04, g_x = 0.025, h_x = 0.1
\end{aligned} \tag{7.5}$$

$$\begin{aligned}
\dot{y_1} &= a_y y_1 - b_y y_1 y_2 - c_y y_1^2 + r_y \\
r_y &= y_1(-g_y v_2 + h_y z_2) \\
\dot{y_2} &= -d_y y_2 + e_y y_1 y_2 - f_y y_2^2 \\
\text{mit: } a_y &= 1, b_y = 0.8, c_y = 0.04, d_y = 0.2, e_y = 0.6, \\
f_y &= 0.07, g_y = 0.03, h_y = 0.025
\end{aligned} \tag{7.6}$$

$$\begin{aligned}
\dot{z_1} &= a_z z_1 - b_z z_1 z_2 - c_z z_1^2 + r_z \\
r_z &= -g_z z_1 v_2 \\
\dot{z_2} &= -d_z z_2 + e_z z_1 z_2 - f_z z_2^2 + s_z \\
s_z &= -h_z z_2 y_1 \\
\text{mit: } a_z &= 3, b_z = 0.7, c_z = 0.02, d_z = 0.5, e_z = 0.3, \\
f_z &= 0.04, g_z = 0.02, h_z = 0.04
\end{aligned} \tag{7.7}$$

Alle Systemzustände besitzen den Anfangswert 1 zum Zeitpunkt $t = 0$. Gesucht sind die Systemzustände v_1 bis z_2 zum Zeitpunkt $t = 100$.

7.2.2.2 Sequentielle Implementierung

Auch hier erfolgt eine näherungsweise Lösung des Problems durch numerische Integration. Dazu wird das RK4-Verfahren mit einer Schrittweite von $h = 0.01$ verwendet.

Die Gleichungen 7.3 bis 7.7 können ohne weitere Umformungen in einer Ableitungsfunktion implementiert werden, welche dem RK4-Algorithmus bereitgestellt wird. Es entsteht wiederum eine sequentielle Implementierung aus den drei ineinander verschachtelten Programmteilen Ableitungsfunktion, RK4-Algorithmus und Hauptprogramm (s. Abb. 7.3).

7.2.2.3 Anwendungsmerkmale

Die parallel abzuarbeitenden Teilaufgaben entsprechen bei diesem Anwendungsproblem der Simulation jeweils *eines* Räuber-Beute-Systems.

Während der Abarbeitung der Teilaufgaben ist eine Koordination mit anderen Instanzen notwendig, sodass eine Ablaufstruktur mit *einem* parallelen Abschnitt *mit* inneren Koordinationspunkten vorliegt. Da eine Koordination der Instanzen bei jedem Aufruf der Ableitungsfunktion stattfindet, der dazwischenliegende Rechenaufwand demgegenüber jedoch vergleichsweise gering ausfällt, kann dieses Anwendungsproblem als feingranular eingestuft werden. Aufgrund der inneren Koordinationspunkte handelt es sich um ein nicht RPC-fähiges Problem.

Der zu erwartende Parallelisierungsaufwand ist für das Anwendungsproblem als hoch einzustufen, da es sich um ein nicht RPC-fähiges Problem handelt. Darüber hinaus ist aufgrund der feinen Granularität nur ein geringer bis gar kein Laufzeitgewinn zu erwarten.

7.2.2.4 Parallele Implementierung

Für das Anwendungsbeispiel wurden die vier Lösungsvarianten DP-1.7 (MP), DP-MPI (MP), DC-2.0 (MP) sowie DP-MPI (SHM) implementiert. Die Listings der Programmvarianten sind in Abschnitt B.2 aufgeführt.

Im Fall der Message-Passing-Programmierung erfolgt unter Verwendung der Systeme DP-1.7 und DP-MPI im Hauptprogramm die Instanziierung untergeordneter Instanzen und der Start des Programms durch Spawn-Operationen. Unter Verwendung des Systems DC-2.0 erfolgt der Programmstart auf untergeordneten Instanzen dagegen durch spezielle Initialisierungskommandos für Message-Passing-Programme (z.B. `createParallelJob`). Der Simulationslauf wird anschließend auf allen Instanzen gleichzeitig abgearbeitet. Während des Simulationslaufs findet in der Ableitungsfunktion eine selektive Abarbeitung der Gleichungen 7.3 bis 7.7 statt, sodass jede beteiligte Instanz nur einen bestimmten Anteil der gesamten Ableitungsfunktion berechnet. Vor der Berechnung der Ableitungswerte findet dabei ein Austausch der Zustandsgrößen durch Sende- und Empfangsoperationen statt. Nach Ablauf der Simulation werden die Ergebnisse im Hauptprogramm durch eine Gather-Operation zusammengefasst.

Im Fall der Shared-Memory-Programmierung (SHM) erfolgt die Instanziierung im Hauptprogramm durch den impliziten Spawn-Mechanismus des DP-MPI-Subsystems.

Anschließend werden alle Zustandsgrößen (v_1 bis $z2$) als gemeinsame Variablen deklariert. Die Abarbeitung der Ableitungsfunktion findet auch hier selektiv statt. Vor Berechnung der Ableitungswerte erfolgt der Austausch der Zustandsgrößen durch ein Schreiben der lokalen Daten in den gemeinsamen Speicher und ein anschließendes Lesen der zusätzlich benötigten Zustandsgrößen aus dem gemeinsamen Speicher. Nach den Schreib- und Lesekommandos ist aus Synchronisationsgründen jeweils eine Barrier-Operation eingefügt. Nach Ablauf der Simulation werden die Ergebnisse durch das Hauptprogramm in einer zusätzlichen gemeinsamen Variable zusammengefasst.

7.2.2.5 Ergebnisse bezüglich des Parallelisierungsaufwandes

Die den Parallelisierungsaufwand betreffenden Kennwerte sind in Tabelle 7.4 aufgeführt. Zwischen den Lösungsvarianten sind hier, auch bezüglich des Programmiermodells, kaum signifikante Unterschiede festzustellen.

Die mit dem System DP-MPI implementierten Lösungsvarianten weisen mit beiden verwendeten Programmiermodellen das geringste Codezeilenverhältnis von circa 1.4 auf, bedingt durch die effiziente Verwendung von Gruppenoperationen und die implizite Prozesserzeugung. Die Verwendung der Shared-Memory-Schnittstelle erforderte dabei jedoch die höchste Anzahl an Multi-SCE-Kommandos, da hier zusätzlich die Initialisierung und Finalisierung des DP-MPI-Subsystems notwendig ist.

Mit einem Codezeilenverhältnis von 1.66 weist die Lösungsvariante unter Verwendung des Systems DC-2.0 den höchsten Parallelisierungsaufwand auf. Die Ursache dafür sind die umfangreichen Initialisierungs- und Finalisierungsschritte, die die Ausführung eines Message-Passing-basierten Programms unter Verwendung dieses Systems erfordert.

Multi-SCE-System	Anzahl Codezeilen (LOC)	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Multi-SCE-Kommandos
sequentiell	90	-	-
DP-MPI (MP)	128	1.42	6
DP-MPI (SHM)	130	1.44	10
DP-1.7 (MP)	139	1.54	7
DC-2.0 (MP)	149	1.66	9

Tabelle 7.4: Parallelisierungsaufwand der Simulation gekoppelter Räuber-Beute-Systeme

7.2.2.6 Ergebnisse bezüglich des Laufzeitverhaltens

Die Ergebnisse der Laufzeituntersuchungen sind in Tabelle 7.5 aufgeführt. Hierbei wird deutlich, dass mit keiner parallelen Lösungsvariante ein Laufzeitgewinn erreicht wurde. Die Ursache dafür ist die zu geringe Granularität des Problems bezüglich der verwendeten Hardwareplattform. Unter den insgesamt negativen Ergebnissen fällt auf, dass mit der Shared-Memory-basierten Lösung nur etwa die Hälfte des Speedups der übrigen

Lösungen erreicht wurde. Der Grund dafür ist die vergleichsweise hohe Latenzzeit dieser Schnittstelle (vgl. Tab. 6.8).

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	42.0	-	-
DP-MPI (MP)	140.3	0.30	0.06
DC-2.0 (MP)	155.1	0.27	0.05
DP-1.7 (MP)	186.0	0.23	0.05
DP-MPI (SHM)	298.6	0.14	0.03

Tabelle 7.5: Laufzeitverhalten der Simulation gekoppelter Räuber-Beute-Systeme (Prozessorzahl 5)

7.2.3 Numerisches Lösen einer partiellen Differentialgleichung

Bei diesem Anwendungsproblem wird die Bewegung eines Seils simuliert, welches an einem Ende fixiert ist und am anderen durch eine Schwingung angeregt wird.

7.2.3.1 Problembeschreibung

Die Bewegung des Seil wird durch eine partielle Differentialgleichung (eindimensionale Wellengleichung) beschrieben:

$$\begin{aligned} &u_{xx}(t,x) = a u_{tt}(x,t) \\ &\quad \text{mit: } u(0,t) = 0, u(L,t) = be^{-dt}\sin\omega t, u(x,0) = u_x(x,0) = 0 \\ &\qquad L = 10, a = 2, b = 1, d = 0.2, \omega = 1 \end{aligned} \tag{7.8}$$

Gesucht sind die Lösungskurven an den Stellen $x = 9L/10$, $x = 3L/4$, $x = L/2$ und $x = L/10$ im Zeitintervall $[0, 30]$.

7.2.3.2 Sequentielle Implementierung

Die Bewegung des Seils wird näherungsweise durch numerische Integration ermittelt. Da das System in diesem Fall durch eine partielle Differentialgleichung beschrieben wird, muss neben der zeitlichen auch eine örtliche Diskretisierung erfolgen. Für die örtliche Zerlegung wird die Seillänge L in $N = 800$ äquidistante Intervalle unterteilt und der Differentialquotient u_{xx} durch einen zentralen Differenzenquotienten ersetzt (*Method of Lines* bzw. *Methode finiter Differenzen*):

$$\begin{aligned} &k^2 a\ddot{u}_i(t) = u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)\ ,\ i = 1, \ldots, N-1 \\ &\quad \text{mit: } u_i(0) = \dot{u}_i(0) = 0, u_0(t) = u(0,t) = 0, \\ &\qquad u_N(t) = u(L,t) = be^{-dt}\sin\omega t, k = L/N \end{aligned} \tag{7.9}$$

Für die numerische Integration über der Zeit wird der RK4-Algorithmus mit einer Schrittweite von $h = 0.005$ verwendet.

Wie in Abschnitt 7.2.1.2 werden auch hier die Differentialgleichungen zweiter Ordnung in ein System von Differentialgleichungen erster Ordnung überführt:

$$\begin{aligned} \dot{u_{1_i}}(t) &= \frac{1}{k^2 a}(u_{2_{i-1}}(t) - 2u_{2_i}(t) + u_{2_{i+1}}(t)) \\ \dot{u_{2_i}}(t) &= u_{1_i}(t) \ , \ i = 1, \ldots, N-1 \\ \text{mit: } u_{2_0}(t) &= u_0(t) = 0 \ , \ u_{2_N}(t) = u_N(t) = be^{-dt} \sin \omega t \\ u_{1_i}(0) &= \dot{u_i}(0) = 0 \ , \ u_{2_i}(0) = u_i(0) = 0 \ . \end{aligned} \tag{7.10}$$

Nach der Umformung kann die Gleichung durch datenparallele Anweisungen effektiv in einer Ableitungsfunktion implementiert werden. Somit entsteht eine sequentielle Implementierung aus den drei ineinander verschachtelten Programmteilen Ableitungsfunktion, RK4-Algorithmus und Hauptprogramm (s. Abb. 7.3).

7.2.3.3 Anwendungsmerkmale

In diesem Anwendungsproblem entsprechen die parallel abzuarbeitenden Teilaufgaben der Simulation zusammenhängender Ortsbereiche.

Eine Koordination der Instanzen ist bei jedem Aufruf der Ableitungsfunktion notwendig, um Zustandsgrößen an den Rändern der Ortsbereiche auszutauschen. Die Anwendung besitzt somit eine Ablaufstruktur mit *einem* parallelen Abschnitt *mit* inneren Koordinationspunkten (Typ 3). Die Anwendung kann als feingranular eingestuft werden, da der Rechenaufwand zwischen den einzelnen Koordinationspunkten im Vergleich zum Kommunikationsaufwand an den Koordinationspunkten zu gering ausfällt. Durch die inneren Koordinationspunkte in der Ablaufstruktur handelt es sich um ein nicht RPC-fähiges Programm.

Da kein RPC-fähiges Problem vorliegt, ist auch bei dieser Anwendung ein hoher Parallelisierungsaufwand zu erwarten. Die feine Granularität deutet außerdem auf einen geringen Laufzeitgewinn hin.

7.2.3.4 Parallele Implementierung

Die parallele Lösung wurde in den drei Varianten DP-1.7 (MP), DP-MPI (MP) sowie DC-2.0 (MP) implementiert. Die Listings aller Programmvarianten befinden sich in Abschnitt B.3.

Die Instanziierung und der Programmstart auf untergeordneten Instanzen erfolgt bei Verwendung der Systeme DP-1.7 und DP-MPI wiederum durch Spawn-Operationen im Hauptprogramm. Im Fall des Systems DC-2.0 werden in Analogie zur parallelen Implementierung der *Simulation gekoppelter Räuber-Beute-Systeme* Initialisierungsfunktionen für den Programmstart verwendet. Durch das Hauptprogramm wird anschließend ein Vektor mit aufsteigenden Werten von 1 bis $N+1$ durch eine Scatter-Funktion unter den beteiligten Instanzen aufgeteilt. Somit können die Grenzen der zu simulierenden Ortsbereiche sowie die Indizes der betreffenden Stützstellen auf einfache Weise ermittelt werden.

Anschließend erfolgt in jeder Instanz der Aufruf des RK4-Algorithmus. Für jeden Aufruf der Ableitungsfunktion erfolgt eine Koordination der Instanzen, wobei jeweils die Zustandswerte von u an den Grenzen der Ortsbereiche zwischen den betreffenden Instanzen ausgetauscht werden. Dafür senden alle Instanzen zunächst an ihre „linke“ Nachbarinstanz und empfangen von ihrer „rechten“ Nachbarinstanz entsprechende Zustandswerte. Anschließend erfolgt das Senden und Empfangen in der umgekehrten Richtung. Nach Abarbeitung der Simulation werden durch das Hauptprogramm die Ergebnisse mittels Gather-Operation zusammengeführt. Zur Reduktion des Kommunikationsaufwands werden dabei nicht die vollständigen Simulationsergebnisse, sondern lediglich die Werte an den interessierenden Stützstellen zusammengefasst.

7.2.3.5 Ergebnisse bezüglich des Parallelisierungsaufwandes

In Tabelle 7.6 sind die für den Parallelisierungsaufwand charakteristischen Kennwerte zusammengefasst. Die unter Verwendung des Systems DC-2.0 entstandene Lösung zeigt dabei mit einem Codezeilenverhältnis von 2.18 den höchsten Parallelisierungsaufwand. Die Gründe dafür sind einerseits der hohe Initialisierungs- und Finalisierungsaufwand bei Verwendung des Systems DC-2.0 und andererseits die fehlenden Scatter- und Gatheroperationen, die hier auf Ebene des Anwendungsprogramms implementiert werden mussten.

Auch bei diesem Anwendungsbeispiel entstand mit dem System DP-MPI der geringste Parallelisierungsaufwand, wiederum bedingt durch effiziente Scatter- und Gatheroperationen sowie implizite Instanziierung.

Multi-SCE-System	Anzahl Codezeilen (LOC)	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Multi-SCE-Kommandos
sequentiell	40	-	-
DP-MPI (MP)	57	1.42	8
DP-1.7 (MP)	68	1.70	8
DC-2.0 (MP)	87	2.18	11

Tabelle 7.6: Parallelisierungsaufwand des numerischen Lösens einer partiellen Differentialgleichung

7.2.3.6 Ergebnisse bezüglich des Laufzeitverhaltens

Die Ergebnisse der Laufzeitmessungen sind in Tabelle 7.7 zusammengefasst dargestellt. Es ist erkennbar, dass auch mit diesem Anwendungsbeispiel kein Laufzeitgewinn erreicht werden konnte, sodass der Speedup aller Programmvarianten einen Wert unter eins erreicht. Die Ursache dafür ist der hohe Kommunikationsaufwand, verursacht durch die inneren Koordinationspunkte des Programms und folglich die zu geringe Granularität. Darüber hinaus sind zwischen den einzelnen Multi-SCE-Systemen hier keine signifikanten Unterschiede erkennbar.

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	49.8	-	-
DP-MPI (MP)	76.9	0.65	0.05
DP-1.7 (MP)	90.6	0.55	0.05
DC-2.0 (MP)	93.7	0.53	0.04

Tabelle 7.7: Laufzeitverhalten des numerisches Lösens einer partiellen Differentialgleichung (Prozessorzahl 12)

7.2.4 Strömungssimulation mittels Lattice-Boltzmann-Verfahren

Bei diesem Anwendungsbeispiel wird das Strömungsverhalten einer inkompressiblen Flüssigkeit durch Simulation nach der Lattice-Boltzmann-Methode ermittelt. Die Lattice-Boltzmann-Methode ist heute eine weit verbreitete Simulationsmethode im CFD-Bereich (*Computational Fluid Dynamics*), insbesondere weil sie gute Parallelisierungs- und Skalierungseigenschaften besitzt. Die Aufgabenstellung entstammt dem bekannten Cavity-Flow-Benchmarkproblem von *Hou et al.* ([86]).

7.2.4.1 Problembeschreibung

Eine inkompressible Flüssigkeit, welche in einer quadratischen Umrandung eingeschlossen ist, wird durch eine gleichförmige Bewegung am oberen Rand angeregt (s. Abb. 7.4). Die konstante Erregerströmung ist durch die Strömungsgeschwindigkeiten $u_{0x} = 0.1$ und $u_{0y} = 0$ gegeben, die Reynoldszahl besitzt den Wert $Re = 1000$. Die Anfangsgeschwindigkeit der Flüssigkeit ist gegeben durch $u_x = 0$ und $u_y = 0$, während die Dichte zum Startzeitpunkt den Wert $\rho = 1$ besitzt. Gesucht ist der normierte Geschwindigkeitsbetrag u/u_0 im Gleichgewichtszustand des Systems.

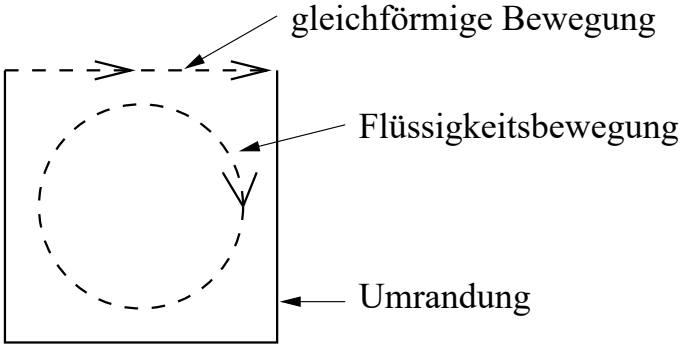


Abbildung 7.4: Problembeschreibung der Strömungssimulation

7.2.4.2 Sequentielle Implementierung

Wie bereits erwähnt erfolgt die Lösung des Problems numerisch durch das Lattice-Boltzmann-Verfahren. Dieses wurde aus dem Verfahren der Lattice-Gasautomaten abgeleitet, bei dem der Ortsbereich in ein Gitter zellulärer Automaten aufgeteilt wird. Auf diese Weise entsteht ein einfaches Partikelmodell, dessen Verhalten Rückschlüsse auf die makroskopischen Größen u und ρ zulässt. Im Lattice-Boltzmann-Verfahren werden die diskreten Zustände der Automaten durch reelwertige Aufenthaltswahrscheinlichkeiten ersetzt.

Die Simulation erfolgt durch die iterative Abarbeitung zweier Schritte: Kollision und Ausbreitung. Während des Kollisionsschrittes findet für jede Zelle eine Umwandlung ihrer Wahrscheinlichkeitswerte in makroskopische Größen statt, welche in die Berechnung neuer Aufenthaltswahrscheinlichkeiten anhand einer Kollisionsoperation einfließen. Während des Ausbreitungsschrittes werden die Wahrscheinlichkeitswerte unter benachbarten Zellen richtungsabhängig ausgetauscht. Die Iteration erfolgt in der Regel bis zum Erreichen eines Gleichgewichtszustandes.

Die Modellspezifikation basiert beim Lattice-Boltzmann-Verfahren auf der Systemgeometrie. Dafür werden einzelne Zellen entweder als Hinderniszellen (*wall cells*, W), Strömungszellen (*fluid cells*, F) oder Erregerzellen (*driving cells*, D) gekennzeichnet. Im zweidimensionalen Fall mit einer Gittergröße von 4-mal-4 Zellen lautet die Modellbeschreibung für dieses Anwendungsproblem durch eine Geometriematrix g wie folgt:

$$g_{4,4} = \begin{pmatrix} D & D & D & D \\ W & F & F & W \\ W & F & F & W \\ W & W & W & W \end{pmatrix} \tag{7.11}$$

Das in der sequentiellen Implementierung verwendete Gitter besitzt eine Größe von 257-mal-257 Zellen. Für eine Simulation bis zum Gleichgewichtszustand sind nach [86] 350000 Iterationsschritte notwendig. Da die sequentielle Implementierung für diese Anzahl an Iterationsschritten eine Rechenzeit von etwa 24 Stunden auf der Testplattform benötigt, wurde aus zeitlichen Gründen eine Begrenzung auf nur 2000 Iterationsschritte vorgenommen. Das sequentielle Programm wurde hier ohne weitere Modularisierung, das heisst als alleinstehendes Hauptprogramm, implementiert.

Für eine ausführliche Darstellung der sequentiellen Matlab-Implementierung sowie der Lattice-Boltzmann-Methode sei an dieser Stelle auf [98] verwiesen.

7.2.4.3 Anwendungsmerkmale

Als parallel abzuarbeitende Teilaufgabe kann hier die Simulation eines zusammenhängenden Teilbereiches des gesamten Gitters gesehen werden.

Da bei Berechnung des Ausbreitungsschrittes ein Werteaustausch zwischen benachbarten Zellen stattfindet, muss während der Abarbeitung der Teilaufgaben eine Koordination zwischen den Instanzen stattfinden. Somit liegt eine Ablaufstruktur mit *einem* parallelen Abschnitt *mit* inneren Koordinationspunkten vor (Typ 3). Da der Rechenaufwand zwischen den Koordinationspunkten hier vergleichsweise hoch ist, handelt es

sich um eine Anwendung mit mittlerer Granularität. Durch die Notwendigkeit innerer Koordinationspunkte ist die Anwendung nicht RPC-fähig.

Da ein nicht RPC-fähiges Problem vorliegt, ist der zu erwartende Parallelisierungsaufwand als hoch einzustufen. Aufgrund der mittleren Granularität ist ein geringer bis mittlerer Laufzeitgewinn zu erwarten.

7.2.4.4 Parallele Implementierung

Die parallele Implementierung umfasst die drei Varianten DP-1.7 (MP), DP-MPI (MP) sowie DC-2.0 (MP). Die Listings der Programmvarianten sind in Abschnitt B.4 aufgeführt.

In der parallelen Implementierung erfolgt die Instanziierung und der Programmstart unter Benutzung der Systeme DP-1.7 und DP-MPI durch Spawn-Operationen. Demgegenüber wird der Programmstart mittels DC-2.0 auch hier durch Initialisierungsfunktionen realisiert. Nachfolgend wird die Geometriematrix in vertikaler Richtung durch eine Scatter-Operation unter den beteiligten Instanzen aufgeteilt. Anschließend erfolgt die Abarbeitung der Simulationsschleife durch die jeweiligen Instanzen. Der Kollisionsschritt wird von jeder Instanz individuell ausgeführt, während zur Ausführung des Ausbreitungsschrittes ein Austausch der Verteilungswerte zwischen benachbarten Teilbereichen notwendig ist. In Analogie zu Abschnitt 7.2.3.4 findet hier zunächst ein Senden an die „untere“ und ein Empfangen von der „oberen“ Nachbarinstanz statt. Anschließend erfolgt der Werteaustausch in der entgegengesetzten Richtung. Nach Abarbeitung der Simulationsschleife erfolgt die Zusammenfassung der Ergebnisse durch eine Gather-Operation.

7.2.4.5 Ergebnisse bezüglich des Parallelisierungsaufwandes

In Tabelle 7.8 sind die Ergebnisse bezüglich des Parallelisierungsaufwandes zusammengefasst dargestellt. Der Vergleich der Codezeilenverhältnisse zeigt, dass mit dem System DP-MPI auch hier der geringste Parallelisierungsaufwand verursacht wurde, erklärbar durch die effizienten Scatter- und Gatheroperationen und die implizite Instanziierung.

Die unter Nutzung des Systems DP-1.7 implementierte Programmvariante zeigt wiederum das höchste Codezeilenverhältnis sowie die höchste Anzahl verwendeter Multi-SCE-Kommandos. Der Grund dafür ist auch hier der hohe Initialisierungs- und Finalisierungsaufwand des Message-Passing-Programms sowie die Implementierung von Scatter- und Gatheroperationen auf Ebene des Anwenderprogramms.

7.2.4.6 Ergebnisse bezüglich des Laufzeitverhaltens

Die Ergebnisse der Laufzeituntersuchungen sind in Tabelle 7.9 zusammengefasst. Die Ergebnisse zeigen, dass mit allen Multi-SCE-Systemen ein Laufzeitgewinn erreicht werden konnte. Die Effizienz lag dabei zwischen 64 und 68 Prozent, was die Charakterisierung als mittelgranulares Problem bestätigt. Unter Verwendung des Systems DP-MPI konnte hier mit einem Speedup von 8 (auf 12 Prozessoren) der höchste Laufzeitgewinn erreicht

Multi-SCE-System	Anzahl Codezeilen (LOC)	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Multi-SCE-Kommandos
sequentiell	65	-	-
DP-MPI (MP)	94	1.45	8
DP-1.7 (MP)	106	1.63	8
DC-2.0 (MP)	125	1.92	11

Tabelle 7.8: Parallelisierungsaufwand der Strömungssimulation mittels Lattice-Boltzmann-Verfahren

werden, wobei die Ergebnisse der übrigen Systeme nur geringfügig unter diesem Wert liegen.

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	458.0	-	-
DP-MPI (MP)	56.3	8.13	0.68
DP-1.7 (MP)	59.8	7.66	0.64
DC-2.0 (MP)	59.7	7.68	0.64

Tabelle 7.9: Laufzeitverhalten der Strömungssimulation mittels Lattice-Boltzmann-Verfahren (Prozessorzahl 12)

7.2.5 Parameteroptimierung eines Abgasmodells

Die Parameteroptimierung eines Abgasmodells basiert auf einer realen ingenieurtechnischen Applikation aus dem Automotive-Bereich. Der Anwendungshintergrund ist das in diesem Bereich bekannte Problem der Applikation der Übergangskompensationsfunktion.

7.2.5.1 Problembeschreibung

Bei Ottomotoren mit Saugrohreinspritzung kondensiert insbesondere bei niedrigen Motortemperaturen ein Teil des eingespritzten Kraftstoffes an der Saugrohrwand. Dieser Effekt wird als Wandfilmeffekt bezeichnet. Bei Laständerungen führt der Wandfilmeffekt zur Ausmagerung oder Anfettung des Kraftstoffgemischs und infolgedessen zu einem erhöhten Schadstoffausstoß. Der Effekt wird durch eine dynamische Funktion der Motorsteuerung kompensiert, wobei die Parameter der Kompensationsfunktion unter anderem von der aktuellen Last, der Drehzahl und der Temperatur des Motors beeinflusst werden. Die Ermittlung optimaler Einstellwerte erfolgt anhand von Messungen und daraus gewonnen Modellen zur Beschreibung des Wandfilmeffekts.

7.2.5.2 Sequentielle Implementierung

Im Fall des Beispielproblems liegen die für den Entwurf notwendigen motorspezifischen Messwerte für 72 Arbeitspunkte (verschiedene Temperaturen und Lastsituationen) in Form von Dateien vor. Zur Berechnung des Wandfilmeffekts steht ein Simulink-Modell zur Verfügung. Unter Verwendung der Matlab-Funktion `fminsearch` (Nelder-Mead-Methode) sollen für jeden Arbeitspunkt optimale Steuerungsparameter gefunden werden.

Die sequentielle Implementierung des Testproblems besitzt eine Programmstruktur, die für eine größere Klasse realer Anwendungen typisch ist: Das Hauptprogramm wurde als Matlab-Skript implementiert und arbeitet somit im Base-Workspace (s. Abschn. 3.5). Dadurch wird dem Simulink-Modell ein einfacher Zugriff auf alle relevanten Variablen des Hauptprogramms ermöglicht. Das Hauptprogramm arbeitet eine Schleife ab, in der durch Aufruf der Optimierungsroutine (`fminsearch`) jeweils die Optimierung eines Arbeitspunktes erfolgt. Der Optimierungsroutine wird eine Gütefunktion bereitgestellt. Diese nimmt aktuelle Steuerungsparameter entgegen, und führt für diese einen Simulationslauf durch. Die Ergebnisse fließen in die Berechnung des Gütekriteriums ein, welches durch die Gütefunktion an die Optimierungsroutine zurückgegeben wird.

Es besteht somit eine Programmstruktur aus den folgenden vier ineinander verschachtelten Programmteilen: Hauptprogramm, Optimierungsroutine, Gütefunktion und Simulationsanwendung. Dabei erfolgt auf fast allen Programmebenen ein Zugriff auf Variablen des Base- beziehungsweise Global-Workspace (s. Abb. 7.5).

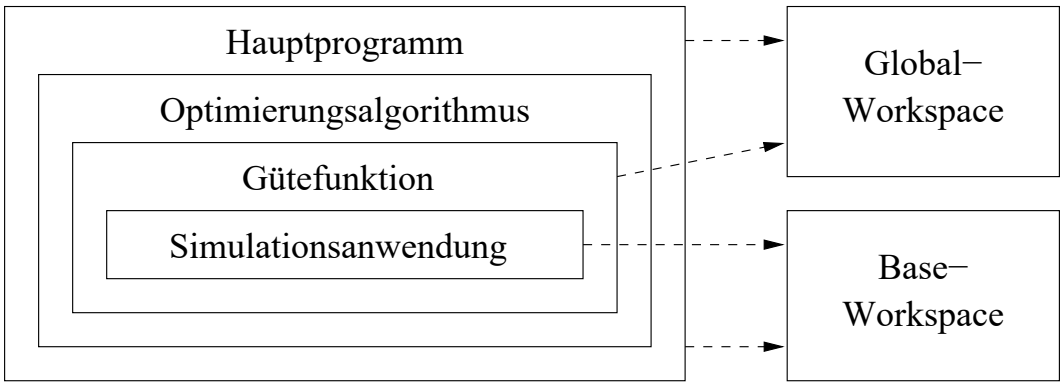


Abbildung 7.5: Parameteroptimierung eines Abgasmodells – sequentielle Struktur

7.2.5.3 Anwendungsmerkmale

Die individuellen Optimierungsläufe repräsentieren bei diesem Anwendungsproblem parallel abzuarbeitende Teilaufgaben.

Während der Abarbeitung der Optimierungsroutine ist keine Koordination mit anderen Instanzen notwendig, sodass eine Ablaufstruktur mit *einem* parallelen Abschnitt *ohne* innere Koordinationspunkte vorliegt (Typ 1). Da nur zu Beginn und Ende des

parallelen Abschnittes eine Kommunikation zwischen Instanzen stattfindet und der dazwischenliegende Rechenaufwand demgegenüber deutlich dominiert, liegt ein grobgranulares Problem vor. Durch das Fehlen innerer Koordinationspunkte ist die Anwendung RPC-fähig.

Aufgrund der RPC-Fähigkeit ist der zu erwartende Parallelisierungsaufwand als gering einzustufen, während die grobe Granularität einen hohen Laufzeitgewinn erwarten lässt.

7.2.5.4 Parallele Implementierung

Die parallele Implementierung umfasst die vier Varianten DP-1.7 (MP), DP-MPI (MP), DC-2.0 (RPC) sowie DP-1.7 (RPC).

Im Fall der Message-Passing-Programmierung erfolgt die Instanziierung sowie der anschließende Programmstart in untergeordneten Instanzen auch hier durch Spawn-Operationen zu Beginn des Hauptprogramms. Danach wird im Hauptprogramm die Liste der zu optimierenden Arbeitspunkte durch eine Scatter-Operation unter allen beteiligten Instanzen aufgeteilt. Anschließend führt jede Instanz die Optimierung der Arbeitspunkte ihrer individuellen Liste durch. Nach Abarbeitung der Hauptprogrammschleife erfolgt die Zusammenführung der Ergebnisse mittels Gather-Operation. Durch die fixierte Verteilung der Arbeitspunktliste liegt in diesem Fall eine statische Lastverteilung vor.

Im Fall der RPC-Programmierung kann bei diesem Anwendungsbeispiel die rechenintensive Routine der Hauptprogrammschleife (`fminsearch`) *nicht* durch einen RPC-Ruf der Optimierungsroutine ersetzt werden (vgl. Abschn. 7.2.1.4). Der Grund dafür ist die Kommunikation zwischen Hauptprogramm, Gütefunktion und Simulationsanwendung über Variablen des Basis- und Global-Workspace. Bei einer entfernten Abarbeitung der Optimierungsroutine stünden somit der Gütefunktion und Simulationsanwendung die durch das Hauptprogramm abgelegten Parameter nicht mehr zur Verfügung. Dieses Problem wurde durch die Implementierung einer dedizierten entfernten Routine gelöst, welche die entsprechenden Variablen im Base- und Global-Workspace ablegt und anschließend die Optimierungsroutine ruft. Auf Ebene des Hauptprogramms kann die Schleife über der Arbeitspunktliste somit durch einen RPC-Ruf ersetzt werden, welcher den Namen der dedizierten Routine sowie die Liste der Arbeitspunkte entgegennimmt. Die Instanziierung sowie der Start der entfernten Routinen wird hier durch den RPC-Ruf gekapselt. Die Lastverteilung erfolgt durch die Benutzung der RPC-Rufe in diesem Fall dynamisch.

7.2.5.5 Ergebnisse bezüglich des Parallelisierungsaufwandes

Die den Parallelisierungsaufwand charakterisierenden Kenngrößen sind in Tabelle 7.10 dargestellt. Dabei wird deutlich, dass die Message-Passing-basierten Lösungen ein geringeres Codezeilenverhältnis als die RPC-basierten Programmvarianten aufweisen. Der Grund dafür ist die notwendige Implementierung der dedizierten entfernten Routine, bedingt durch die komplexe Programmstruktur (s. Abb. 7.5). Anhand dieser Ergebnisse wird deutlich, dass eine klar strukturierte sequentielle Implementierung (d.h. ohne Verwendung globaler Variablen) eine nachfolgende RPC-basierte Parallelisierung deutlich

erleichtern kann. Andererseits zeigt dieses Anwendungsbeispiel, dass die Verwendung einer Message-Passing-Schnittstelle deutliche Vorteile bei derart komplexen Programmstrukturen besitzt, da somit auf allen beteiligten Instanzen das gleiche, auf der sequentiellen Vorlage basierende, Hauptprogramm abgearbeitet werden kann.

Anhand der Anzahl verwendeter Multi-SCE-Kommandos ist erkennbar, dass wie bei der *Parameterstudie eines Feder-Masse-Systems* die RPC-basierten Lösungen eine geringere Anzahl an Kommandos erfordern. So wird unter Verwendung des Systems DP-1.7 lediglich der RPC-Ruf benötigt, während das System DC-2.0 das Löschen der im Jobmanager gespeicherten Rufparameter durch zwei zusätzliche Kommandos erfordert.

Multi-SCE-System	Anzahl Codezeilen (LOC)	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Multi-SCE-Kommandos
sequentiell	91	-	-
DP-1.7 (RPC)	136	1.49	1
DC-2.0 (RPC)	137	1.51	3
DP-MPI (MP)	116	1.27	5
DP-1.7 (MP)	129	1.42	8

Tabelle 7.10: Parallelisierungsaufwand der Parameteroptimierung eines Abgasmodells

7.2.5.6 Ergebnisse bezüglich des Laufzeitverhaltens

Die laufzeitbezogenen Ergebnisse sind in Tabelle 7.11 dargestellt. Die Speedup-Werte zeigen, dass bei dieser Anwendung mit allen Lösungsvarianten ein vergleichsweise hoher Laufzeitgewinn erreicht werden konnte. Ein Vergleich bezüglich der Programmiermodelle zeigt, dass die Message-Passing-basierten Implementierungen gegenüber den RPC-basierten Lösungen geringere Laufzeitgewinne aufweisen. Der Grund dafür ist die unterschiedliche Art der Lastverteilung. Da bei diesem Anwendungsproblem die Abarbeitungszeit einzelner Teilaufgaben einer starken Streuung unterliegt, führt eine dynamische Lastverteilung gegenüber der statischen Verteilung hier zu höheren Laufzeitgewinnen (vgl. Abb 5.1).

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	6854	-	-
DP-1.7 (RPC)	666	10.3	0.86
DC-2.0 (RPC)	759	9.03	0.75
DP-1.7 (MP)	811	8.46	0.70
DP-MPI (MP)	845	8.11	0.68

Tabelle 7.11: Laufzeitverhalten der Parameteroptimierung eines Abgasmodells (Prozessorzahl 12)

7.2.6 Sicherheitstest eingebetteter Steuerungssoftware

Dieses Anwendungsbeispiel repräsentiert ebenfalls eine reale ingenieurtechnische Applikation aus dem Automotive-Bereich. Der Anwendungshintergrund ist der Sicherheitstest von Steuerungssoftware zur automatischen Abstandsregelung.

7.2.6.1 Problembeschreibung

Sicherheitsrelevante Steuerungssoftware im Automotive-Bereich muss heute hohen Anforderungen genügen. Für manuelle Sicherheitstests ergibt sich dabei häufig ein zu hoher Testaufwand. Dieses Anwendungsbeispiel verfolgt daher den Ansatz, evolutionäre Algorithmen für den Sicherheitstest von Steuerungssoftware zu verwenden.

Im speziellen Anwendungsfall wird die Steuerungssoftware einer automatischen Abstandsregelung nach dieser Methode getestet. Das zu testende System liegt dabei in Form eines Rechnermodells vor. Mittels evolutionärer Algorithmen findet eine gezielte Variation der Eingangsgrößen des Systems bis zum Erreichen eines kritischen Zustandes statt.

7.2.6.2 Sequentielle Implementierung

Die zu testende Steuerungssoftware sowie das dazugehörige Fahrzeugmodell wird wie in Abschnitt 7.2.5 durch ein Simulink-Modell abgebildet. Das Modell ist in eine Gütefunktion eingebettet, die ein Testszenario in Form einer Menge von Eingangsgrößen entgegennimmt, einen Simulationslauf für dieses Szenario durchführt und basierend auf den Simulationsergebnissen einen Gütewert als Abbildung des korrekten Steuerungsverhaltens zurückgibt. Die Gütefunktion ist in den genetischen Optimierungsalgorithmus der GEA-Toolbox ([87]) eingebettet, wobei die der Gütefunktion übergebenen Szenarien einzelne Individuen repräsentieren. Der Optimierungsalgorithmus wird durch ein Hauptprogramm konfiguriert und aufgerufen.

Es liegt somit wie in Abschnitt 7.2.5 eine Programmstruktur mit den vier ineinander verschachtelten Programmteilen Hauptprogramm, Optimierungsroutine, Gütefunktion und Simulationsanwendung vor. Der Unterschied der Struktur gegenüber Abbildung 7.5 ist, dass sowohl Gütefunktion als auch Simulationsanwendung hier lediglich auf lokale Variablen zugreifen. Eine detaillierte Beschreibung des Anwendungsproblems sowie der sequentiellen Implementierung findet sich in [88].

7.2.6.3 Anwendungsmerkmale

Die parallel abzuarbeitenden Teilaufgaben entsprechen in diesem Anwendungsfall den Gütewertberechnungen unterschiedlicher Individuen innerhalb eines Iterationsschrittes.

Während der Abarbeitung einer Teilaufgabe ist keine Koordination mit anderen Instanzen notwendig, während die Teilaufgaben mehrfach, das heisst zu jedem Iterationsschritt des Optimierungsalgorithmus, abzuarbeiten sind. Somit liegt eine Ablaufstruktur mit *mehreren* parallelen Abschnitten *ohne* innere Koordinationspunkte (Typ 2) vor. Die Granularität der Anwendung ist als grob einzustufen, da der Kommunikationsaufwand

zu Beginn und Ende paralleler Abschnitte gegenüber dem dazwischenliegenden Rechenaufwand vergleichsweise gering ausfällt. Hinsichtlich des Programmiermodells handelt es sich um ein RPC-fähiges Problem, da keine inneren Koordinationspunkte notwendig sind.

Da es sich bei der Anwendung um ein RPC-fähiges Problem handelt, ist der Parallelisierungsaufwand als gering einzuschätzen. Die grobe Granularität impliziert einen hohen Laufzeitgewinn, der jedoch aufgrund des hohen Anteils sequentieller Abschnitte (Typ 2) nach dem Ahmdahlschen Gesetz vermindert wird.

7.2.6.4 Parallele Implementierung

Die parallelen Lösungen wurden in den drei Varianten DP-1.7 (RPC), DP-MPI (MP) sowie DC-2.0 (RPC) implementiert.

In allen parallelen Programmvarianten fand eine Parallelisierung auf Ebene des Optimierungsalgorithmus statt, sodass sowohl im Hauptprogramm als auch in der Gütefunktion keine Multi-SCE-Kommandos verwendet werden. Für jede Programmvariante wurde eine entsprechende parallele Routine entwickelt, die den Namen einer Gütefunktion sowie eine Liste von Individuen entgegennimmt und entsprechend den Individuen eine Liste von Gütewerten zurückgibt. Alle Multi-SCE-Kommandos sind somit innerhalb dieser parallelen Routine gekapselt. Der Aufruf der parallelen Routine erfolgt durch den Optimierungsalgorithmus in jedem Iterationsschritt. Durch die Konfiguration des Optimierungsalgorithmus im Hauptprogramm kann je nach verwendetem Multi-SCE-System die Nutzung einer bestimmten parallelen Routine oder des herkömmlichen sequentiellen Algorithmus spezifiziert werden. Die Einbettung der parallelen Routine in den Optimierungsalgorithmus verdeutlicht Abbildung 7.6.

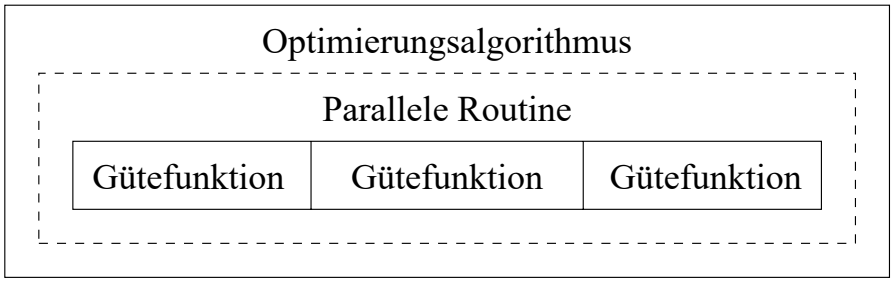


Abbildung 7.6: Parallelisierung auf Ebene des Optimierungsalgorithmus

Die Implementierung der parallelen Lösungen erfolgte hier teilweise mit unterschiedlicher Motivation. Die Lösungsvariante unter Verwendung des Sysems DC-2.0 entstand im Zuge einer offiziellen DC-Toolbox-Unterstützung in die GEA-Toolbox. Die Lösungsvarianten der Systeme DP-1.7 und DP-MPI wurden dagegen nur zu Vergleichszwecken implementiert, sodass ihre Unterstützung durch die GEA-Toolbox nur prototypisch und ohne größere Fehlerbehandlung erfolgt. Durch die starken Unterschiede in der Fehlerbehandlung ist der Parallelisierungsaufwand daher hier nicht vergleichbar, sodass sich die quantitative Untersuchung auf das Laufzeitverhalten beschränkt.

Im Fall der Message-Passing-Programmierung mit DP-MPI erfolgt die Instanziierung und der entfernte Programmstart durch die implizite Spawn-Operation. Anschließend wird die Liste der Individuen durch eine Scatter-Operation unter allen Instanzen aufgeteilt und der Name der Gütefunktion durch eine Broadcast-Operation an alle Instanzen gesendet. Es folgt der Aufruf der Gütefunktion für die Teilliste der Individuen in jeder Instanz. Abschließend erfolgt die Zusammenfassung der Gütewerte durch eine Gather-Operation und die Terminierung der untergeordneten Instanzen. Die Lastverteilung erfolgt dabei aufgrund der Scatter-Operation statisch.

Im Fall der RPC-Programmierung mit dem System DP-1.7 erfolgt zunächst eine Umwandlung der Liste der Individuen in ein dem RPC-Ruf konformes Cell-Array. Es folgt der Aufruf des RPC-Rufes mit der Gütefunktion als entfernt abzuarbeitender Routine. Die Instanziierung und Terminierung untergeordneter Instanzen erfolgt implizit durch den RPC-Ruf. Die als Cell-Array vorliegenden Rückgabewerte des RPC-Rufes werden zur Weiterverarbeitung in Matrizenform überführt. Durch die Verwendung des RPC-Rufes erfolgt eine dynamische Lastverteilung.

Im Fall der RPC-Programmierung mit dem System DC-2.0 wurde nicht wie in Abschnitt 7.2.1 und 7.2.5 ein einzelner vektorieller RPC-Ruf verwendet, sondern aus Gründen der besseren Fehlerbehandlung auf die weitaus komplexere objektorientierte Schnittstelle des Systems zurückgegriffen. Die Abarbeitung der parallelen Routine erfolgt sowohl auf der Client- als auch auf der Serverseite, wobei nach Identifikation anhand der Anzahl der übergebenen Individuen eine Verzweigung in Client- und Serverprogrammteil stattfindet. Auf der Clientseite wird zunächst ein Jobmanager gesucht und nach Auffinden des Jobmanagers ein Job zur Spezifikation des vektoriellen RPC-Rufes erzeugt. In einem Schleifendurchlauf wird nun jede individuelle Teilaufgabe mit der parallelen Routine als entfernt auszuführender Funktion spezifiziert. Die Aufrufparameter der Routine sind dabei der Name der Gütefunktion sowie die Parameter eines einzelnen Individuums. Der so spezifizierte RPC-Ruf wird dem Jobmanager durch eine nichtblockierende Funktion übergeben, nach deren Aufruf ein explizites Warten auf die Abarbeitung des Rufes erfolgt. Abschließend werden die in Form eines Cell-Arrays vorliegenden Ergebnisse explizit von Jobmanager angefordert und in Matrizenform überführt. Serverseitig erfolgt die Berechnung der spezifizierten Gütefunktion, eingebettet in eine komplexe Fehlerbehandlung. Die Lastverteilung erfolgt dynamisch durch den vektoriellen RPC-Ruf.

7.2.6.5 Ergebnisse bezüglich des Laufzeitverhaltens

Die Ergebnisse der Laufzeituntersuchungen sind in Tabelle 7.12 dargestellt. Die parallelen Implementierungen erreichten Speedupwerte im Bereich von 4.2 bis 4.7 bei einer Prozessorzahl von 12. Obwohl es sich um ein grobgranulares Problem handelt, wurden hier keine Laufzeitgewinne in der Nähe des idealen Speedups erreicht. Ein Grund dafür ist der große Anteil sequentieller Abschnitte, welcher den Speedup nach dem Ahmdahlschen Gesetz begrenzt. Ein weiterer Grund ist der Start der untergeordneten Instanzen, für den bei jedem Iterationsschritt mehrere Sekunden benötigt werden. Im Fall des Systems DC-2.0 entfällt zwar diese Instanziierungszeit, jedoch kompensiert die hohe Latenzzeit des Systems diesen Vorteil vollständig.

Ein Vergleich hinsichtlich der Lastverteilung zeigt, dass auch hier bei statischer Verteilung ein etwas geringerer Laufzeitgewinn erreicht wurde.

Multi-SCE-System	Laufzeit [s]	Speedup	Effizienz
sequentiell	1235	-	-
DC-2.0 (RPC)	262	4.72	0.39
DP-1.7 (RPC)	265	4.65	0.39
DP-MPI (MP)	293	4.22	0.35

Tabelle 7.12: Laufzeitverhalten des Sicherheitstests eingebetteter Steuerungssoftware (Prozessorzahl 12)

7.3 Vergleich der Anwendungen

Auf Basis des Datenmaterials aus dem anwendungsorientierten Vergleich von Multi-SCE-Systemen ist es möglich, die strukturellen Anwendungsmerkmale und den quantitativ ermittelten Parallelisierungsaufwand und Laufzeitgewinn zueinander in Bezug zu setzen. Dafür werden die mit verschiedenen Multi-SCE-Systemen und Programmiermodellen gewonnenen Größen gemittelt, sodass die quantitativen Werte für eine Menge von parallelen Programmvarianten gültig sind. In Tabelle 7.13 erfolgt die Gegenüberstellung von Anwendungsmerkmalen, Parallelisierungsaufwand und Laufzeitgewinn. Der Laufzeitgewinn ist dabei in Form der Effizienz dargestellt, da die verwendete Prozessorzahl zwischen den parallelisierten Anwendungen variiert und somit ein systemübergreifender Vergleich auf Basis des Speedups nicht möglich ist.

Anhand von Tabelle 7.13 wird deutlich, dass wie erwartet bei grobgranularen Anwendungen mit einem parallelen Abschnitt ohne innere Koordinationspunkte (Typ 1, Abb. 7.2) der geringste Programmieraufwand und der höchste Laufzeitgewinn auftritt. So weist die *Parameterstudie eines Feder-Masse-Systems* eine mittlere Effizienz von 0.9 auf, während bei der *Parameteroptimierung eines Abgasmodells* eine durchschnittliche Effizienz von 0.75 erreicht wird. Beide Anwendungen erfordern dabei die Kenntnis von durchschnittlich 4 Multi-SCE-Kommandos und weisen unter allen untersuchten Anwendungen die geringsten Codezeilenverhältnisse in Höhe von 1.1 beziehungsweise 1.4 auf.

Anwendungen, die eine Ablaufstruktur mit einem parallelen Abschnitt mit inneren Koordinationspunkten (Typ 3) besitzen, das heisst nicht RPC-fähig sind, erfordern dagegen einen höheren Parallelisierungsaufwand, wie die Anzahl der verwendeten Kommandos und die durchschnittlichen Codezeilenverhältnisse zeigen. Diese variieren für die betreffenden Anwendungen (*Simulation gekoppelter Räuber-Beute-Systeme*, *Numerisches Lösen einer partiellen DGL* und *Strömungssimulation mittels Lattice-Boltzmann-Verfahren*) zwischen 1.5 und 1.8.

Weiterhin wird deutlich, dass die *Strömungssimulation mittels Lattice-Boltzmann-Verfahren* trotz der inneren Koordinationspunkte eine durchschnittliche Effizienz von 0.65

Anwendung	Ablauf-struktur	Granu-larität	RPC-fähig	$\frac{LOC_{par}}{LOC_{seq}}$	Anzahl Kom-mandos	Effi-zienz
Parameterstudie eines Feder-Masse-Systems	Typ 1	grob	ja	1.12	4	0.90
Parameteroptimierung eines Abgasmodells	Typ 1	grob	ja	1.42	4	0.75
Strömungssimulation mittels LB-Verfahren	Typ 3	mittel	nein	1.67	9	0.65
Sicherheitstest eingebet-teter Steuerungssoftware	Typ 2	grob	ja	-	-	0.38
Simulation gekoppelter Räuber-Beute-Systeme	Typ 3	fein	nein	1.52	8	0.05
Numerisches Lösen einer partiellen DGL	Typ 3	fein	nein	1.77	9	0.05

Tabelle 7.13: Gegenüberstellung von Anwendungsmerkmalen, durchschnittlichem Parallelisierungsaufwand und durchschnittlichem Laufzeitgewinn

erreicht, bedingt durch den mittelgranularen Charakter der Anwendung. Der *Sicherheitstest eingebetteter Steuerungssoftware*, welcher als grobgranular und RPC-fähig eingestuft werden kann, erreicht demgegenüber aufgrund des hohen Anteils sequentieller Programmabschnitte einen geringeren Laufzeitgewinn in Höhe von 0.38.

7.4 Zusammenfassung

Dieses Kapitel befasste sich mit dem Einsatz von Multi-SCE-Systemen in ingenieurtechnischen Anwendungen. Die beiden Kriterien Parallelisierungsaufwand und Laufzeitgewinn, die in diesem Sinne den Aufwand und Nutzen einer parallelen Anwendung widerspiegeln, besaßen dabei eine zentrale Bedeutung. Es fand daher eine auf sechs ingenieurtechnischen Anwendungen basierende Untersuchung von Multi-SCE-Systemen statt, bei der drei Systeme mit alternativen Programmiermodellen hinsichtlich des Parallelisierungsaufwands und Laufzeitgewinns verglichen wurden. Darüber hinaus erfolgte der direkte Vergleich der untersuchten Anwendungen, bei dem ihre qualitativen Merkmale (Ablaufstruktur, Granularität sowie Anwendbarkeit von Programmiermodellen) dem durchschnittlichen Parallelisierungsaufwand und Laufzeitgewinn gegenübergestellt wurden.

Die anwendungsorientierte Untersuchung von Multi-SCE-Systemen zeigte, dass für die betrachteten Anwendungen der Laufzeitgewinn zwischen den Systemen sowie den verwendeten Programmiermodellen nicht signifikant variiert. So sticht bezüglich des Laufzeitverhaltens weder ein einzelnes Programmiermodell noch ein spezielles System im gesamten Spektrum der Anwendungen hervor. Wesentliche Unterschiede sind dagegen

beim Parallelisierungsaufwand zu beobachten. So zeigt die Anwendung *Parameterstudie eines Feder-Masse-Systems*, dass der Aufwand unter Verwendung des RPC-Modells wesentlich geringer ausfallen kann als beim Message-Passing-Modell. Die Anwendung *Simulation eines gekoppelten Räuber-Beute-Systems* zeigte daneben, dass die Message-Passing-Programmierung und die Shared-Memory-Programmierung etwa den gleichen Parallelisierungsaufwand erfordern.

Der direkte Vergleich der Multi-SCE-Systeme bezüglich der Effektivität ihrer angebotenen Programmiermodelle zeigt, dass die RPC-Programmierung mittels DC-2.0 einen höheren Parallelisierungsaufwand als mit dem System DP-1.7 erfordert. Als Ursache dafür wurde der zusätzliche Aufwand zur Verwaltung des DC-Jobmanagers identifiziert. Die Message-Passing-Programmierung erfordert bei Verwendung des Systems DC-2.0 ebenfalls den höchsten Parallelisierungsaufwand, bedingt durch die hohe Anzahl an Instanziierungs- und Finalisierungskommandos. Mit dem System DP-MPI erfolgte die Message-Passing-Programmierung dagegen mit dem geringsten Parallelisierungsaufwand. Als Hauptgrund dafür wurde die Möglichkeit der impliziten Instanziierung sowie die effektiven Scatter- und Gatheroperationen identifiziert. Ein systemübergreifender Vergleich der Shared-Memory-Programmierung konnte nicht erfolgen, da dieses Modell lediglich durch DP-MPI bereitgestellt wurde.

Die direkte Untersuchung der Anwendungen zeigte, dass bei grobgranularen RPC-fähigen Problemen der geringste Parallelisierungsaufwand sowie der höchste Laufzeitgewinn zu erwarten ist. Die Anwendung *Sicherheitstest eingebetteter Steuerungssoftware* verdeutlichte jedoch, dass bei einem hohem Anteil sequentieller Programmabschnitte auch Ausnahmen von dieser Regel auftreten können. Liegen dagegen nicht RPC-fähige Anwendungen vor, so tritt ein vergleichsweise hoher Parallelisierungsaufwand auf, während der Laufzeitgewinn nur in Ausnahmefällen befriedigende Werte erreicht.

Grundsätzlich zeigt dieses Kapitel, dass grobgranulare RPC-fähige Probleme sowie ein effizientes RPC-Programmiermodell die Voraussetzungen für den effektiven Einsatz von Multi-SCE-Systemen in ingenieurtechnischen Bereichen sind. Multi-SCE-Systemen, die vektorielles RPC mit dynamischer Lastverteilung anbieten, kommt daher in diesem Bereich eine besondere Bedeutung zu. Die Voraussetzung für eine einfache Parallelisierung nach dem RPC-Modell ist jedoch ein klar strukturiertes sequentielles Programm ohne Verwendung globaler Variablen. Liegen dagegen sequentielle Programme mit einer Struktur wie in Abbildung 7.5 vor, was für viele Matlab-Simulink-Anwendungen der Fall ist, so kann der Parallelisierungsaufwand mittels Message-Passing-Programmierung und entsprechenden Array-Scattering- und -Gathering-Routinen gegenüber der RPC-Programmierung deutlich geringer ausfallen.

8 Zusammenfassung

Die vorliegende Arbeit hatte das Ziel, die SCE-basierte Parallelverarbeitung in ingenieurtechnischen Bereichen weiter zu verbreiten. Dabei sollten insbesondere bestehende Softwaresysteme und Ansätze gesichtet und hinsichtlich ingenieurtechnischer Anwendungsfelder evaluiert werdend. Darüber hinaus sollten Erkenntnisse für die Anwendungsentwicklung gewonnen werden, die zum Beispiel den Aufwand und Nutzen der Parallelisierung bestimmter Anwendungsklassen abschätzbar machen.

Die Arbeit knüpfte in Teilen an die 1998 veröffentlichte Dissertation von Pawletta an, in der die parallele und verteilte Verarbeitung in SCEs sowie die Entwicklung und Anwendung der DP-Toolbox thematisiert wurde. Der von Pawletta vorgestellte Multi-SCE-Ansatz, der für ingenieurtechnische Anwender besonders interessant ist, wurde hierbei ausführlich betrachtet.

Zum Verständnis der weiteren Ausführungen erfolgte zunächst eine Einführung in die notwendigen Grundlagen der Parallelverarbeitung. Dabei wurden unter anderem die Ergebnisse zweier Untersuchungen zu Hardwareplattformen im Bereich kleiner Installationen (Low-End-Bereich) und des Supercomputing (High-End-Bereich) vorgestellt. Darüber hinaus wurden Aspekte der parallelen Programmierung erläutert und explizite sowie implizite Programmiermodelle vorgestellt. Insbesondere das RPC-Modell wurde dabei ausführlich betrachtet und als sehr geeignetes Modell für bestimmte Anwendungstypen identifiziert.

Bei den Betrachtungen bezüglich der Grundlagen von SCEs wurde unter anderem ein Vergleich von SCE-Programmen und Betriebssystemprozessen vorgenommen, der Unterschiede in Bezug auf den Sichtbarkeitsbereich von Daten sowie die Beeinflussung nachfolgend ablaufender Programme verdeutlicht. Daneben erfolgte eine Darstellung der SCE-basierten Programmierung sowie eine Betrachtung der Möglichkeiten zur Programmbeschleunigung in SCEs, zu denen unter anderem die SCE-basierte Parallelverarbeitung zählt.

Für das Gebiet der allgemeinen SCE-basierten Parallelverarbeitung erfolgte die Präsentation einer neuen Klassifikation für dedizierte Softwaresysteme und allgemeine Ansätze, in der die bisherigen Klassifikationen vereint wurden. Diese Klassifikation umfasst die drei Ansätze Übersetzungsansatz, Frontend-Ansatz sowie Multi-SCE-Ansatz. Nachfolgend wurden prinzipielle Techniken sowie identifizierte Softwaresysteme der jeweiligen Klassen vorgestellt. Dabei wurde deutlich, dass der Multi-SCE-Ansatz mit insgesamt 28 identifizierten Softwaresystemen eine dominierende Stellung einnimmt. Aus dieser Menge an Systemen erschienen 23 Systeme zwischen 1999 und 2006, also nach den Arbeiten von Pawletta. Der Vergleich identifizierter Multi-SCE-Systeme zeigte, dass hinsichtlich der Programmiermodelle die RPC- und Message-Passing-Programmierung dominierend sind, während bezüglich der Kopplungsplattform am häufigsten Message-Passing-

Systeme (PVM oder MPI-Implementierungen) verwendet werden. Darüber hinaus wurde deutlich, dass die meisten Multi-SCE-Systeme für Matlab entwickelt wurden.

Für eine detaillierte Untersuchung von Multi-SCE-Systemen wurden acht Systeme herangezogen, die verfügbar und auf aktuellen SCEs lauffähig waren. Darunter waren sechs Systeme für Matlab und jeweils ein System für die freien SCEs Scilab und Octave. Die Systeme unterstützen sowohl die RPC- als auch die Message-Passing-Programmierung, wobei die DC-Toolbox als einziges System die Parallelprogrammierung mit beiden Modellen erlaubt. Die Untersuchung gliederte sich in Betrachtungen zu qualitativen Merkmalen auf verschiedenen Systemebenen und quantitativen Kenngrößen der Kommunikationsleitung. Die Betrachtung der Merkmale auf High-Level-Ebene zeigte, dass für das Message-Passing-Modell durch mehrere Systeme Abstraktionen gegenüber der nativen Message-Passing-Programmierung vorgenommen wurden. Eine weit verbreitete Abstraktion stellt dabei das Array-Passing, das heisst das Senden und Empfangen von Daten mit ihren Typ- und Dimensionsinformationen, dar. Im Bereich der RPC-Programmierung wurde deutlich, dass dieses Modell in den untersuchten Systemen meist durch einen RPC-Ruf realisiert wird, der vektorielles RPC mit dynamischer Lastverteilung verknüpft. Die Untersuchung zur Kommunikationsleistung der Systeme zeigte unter anderem, dass bei Verwendung eines Message-Passing-Systems als Kopplungsplattform die geringsten Latenzzeiten auftreten, die jedoch stets über der Latenzzeit nativer Fortran- oder C-Bibliotheken liegen.

Die Fortführung der DP-Toolbox-Entwicklung war ein weiterer Bestandteil der Arbeit. Bei dieser Entwicklung wurden zwei Linien verfolgt: die Weiterentwicklung der DP-Toolbox zu einem System für industrielle Anwendungen sowie die Weiterführung der experimentellen DP-Toolbox-Entwicklung in Form verschiedener Prototypen. Die industrieorientierte DP-Version 1.7 erlaubt eine Programmierung nach dem Message-Passing- sowie dem RPC-Modell. Im Fall der Message-Passing-Programmierung wird dabei insbesondere das Array-Passing sowie das Array-Scattering und -Gathering unterstützt, während im Bereich der RPC-Programmierung das vektorielle RPC mit dynamischer Lastverteilung ermöglicht wird. Im Rahmen der Weiterführung der experimentellen DP-Entwicklung entstand unter anderem ein MPI-basiertes System mit vollständigem High-Level-Interface, welches sowohl die Message-Passing- als auch die Shared-Memory-Programmierung erlaubt. Insbesondere im Bereich der der Message-Passing-Programmierung wurden Ansätze der nativen MPI-Schnittstelle übernommen und SCE-spezifisch adaptiert, so dass hier im Vergleich zur DP-Version 1.7 wesentlich kompaktere Programme erstellt werden können.

Für den anwendungsorientierten Vergleich von Multi-SCE-Systemen wurden drei Systeme herangezogen, die insbesondere aufgrund der Unterstützung alternativer Programmiermodelle von Interesse waren. Dazu gehörten die DC-Toolbox, die weiterentwickelte DP-Toolbox (DP-1.7) sowie das experimentelle System DP-MPI. Die Untersuchung wurde anhand sechs ingenieurtechnischer Anwendungen durchgeführt, von denen vier als Benchmarkprobleme gelten, während zwei reale Entwurfsanwendungen der Automobilbranche darstellen. Als Vergleichskriterien wurde einerseits der Laufzeitgewinn, andererseits aber auch der Parallelisierungsaufwand unter Verwendung bestimmter Systeme herangezogen. Der anwendungsorientierte Vergleich der Systeme zeigte, dass hinsichtlich des

Laufzeitgewinns über das gesamte Anwendungsspektrum keine signifikanten Unterschiede zwischen den Toolboxen zu beobachten sind, während der Parallelisierungsaufwand zwischen den Systemen deutlich variiert. Ein programmiermodellbezogener Vergleich zeigte, dass unter Verwendung des RPC-Modells gegenüber dem Message-Passing- und Shared-Memory-Modell prinzipiell ein geringerer Parallelisierungsaufwand zu verzeichnen ist. Unter Verwendung der DC-Toolbox war dabei sowohl im Bereich der RPC- als auch Message-Passing-Programmierung der höchste Aufwand zu beobachten, während die DP-Toolbox eine besonders effektive RPC-Programmierung und DP-MPI eine sehr kompakte Message-Passing-Programmierung erlauben. Der direkte Vergleich der Anwendungen, bei dem die qualitativen Merkmale Ablaufverhalten, Granularität und Anwendbarkeit von Programmiermodellen dem durchschnittlichen Parallelisierungsaufwand und Laufzeitgewinn gegenübergestellt wurden, zeigte, dass mit grobgranularen RPC-fähigen Anwendungen der geringste Parallelisierungsaufwand und der höchste Laufzeitgewinn auftritt. Dafür stellt die vektorielle RPC-Programmierung in Verbindung mit dynamischer Lastverteilung das ideale Programmiermodell dar. Die Parallelisierung realer ingenieurtechnischer Anwendungen zeigte jedoch, dass bei Anwendungen mit komplexen Strukturen, wie zum Beispiel bei Benutzung globaler Variablen, die Message-Passing-Programmierung unter Verwendung von Array-Scattering- und -Gathering-Routinen gegenüber der RPC-Programmierung einen geringeren Parallelisierungsaufwand verursacht.

Es ist anzunehmen, dass zukünftige Entwicklungen auf dem Gebiet der SCE-basierten Parallelverarbeitung stark durch die aktuelle Hardwareentwicklung geprägt werden. Um die Vorteile aktueller Multikernprozessoren zu nutzen, werden SCE-Hersteller zunehmend numerische Bibliotheken mit Multithread-Unterstützung anbieten, wie das Beispiel von IDL zeigt. Das bedeutet, dass der Frontend-Ansatz, das heisst die Nutzung einer SCE als Nutzerschnittstelle zu einem Parallelverarbeitungssystem, in Zukunft eine wichtigere Rolle spielen wird. Darüber hinaus eröffnet die Entwicklung der Multikernprozessoren jedoch auch neue Möglichkeiten für die Verwendung des Multi-SCE-Ansatzes, insbesondere in Verbindung mit Arbeitsplatzrechnern. So wird die steigende Anzahl von Prozessorkernen die Einrichtung eines kompletten SCE-Verbundes auf der lokalen Workstation erlauben, wodurch der Aufwand für die Installation und Pflege einer Parallelverarbeitungsplattform, wie zum Beispiel eines Computerclusters, vollständig entfällt.

Literaturverzeichnis

zu Kapitel 2

[1] M. J. Flynn: Very High-Speed Computing Systems. In Proceedings of the IEEE, Vol. 54, Issue 12, Dezember 1966.

[2] G. M. Amdahl: Validity of the single processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings , Vol. 30, April 1967.

[3] S. S. Reddi, E. A. Feustel: A Conceptual Framework for Computer Architecture. In ACM Computing Surveys, Vol. 8, Issue 2, Juni 1976.

[4] M. Ben-Ari: Principles of Concurrent Programming. Prentice-Hall International, 1982.

[5] P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins: Data-Driven and Demand-Driven Computer Architecture. In ACM Computing Surveys, Vol. 14, Issue 1, März 1982.

[6] A. D. Birrell, B. J. Nelson: Implementing Remote Procedure Calls. In ACM Transactions on Computer Systems, Volume 2 Issue 1, Februar 1984.

[7] E. E. Johnson: Completing an MIMD multiprocessor taxonomy. In ACM SIGARCH Computer Architecture News, Vol. 16, Issue 3, Juni 1988.

[8] A. L. Ananda, B. H. Tay, E. K. Kohn: A Survey of Asynchronous Remote Procedure Calls. In ACM SIGOPS Operating Systems Review, Vol. 26, Issue 2, April 1992.

[9] T. L. Freeman, C. Phillips: Parallel Numerical Algorithms. Prentice Hall, 1992.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.

[11] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 1.0, Mai 1994. `http://www.mpi-forum.org/docs/mpi-10.ps`

[12] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), November 1996.

[13] M. J. Flynn, K. W. Rudd: Parallel Architectures. In ACM Computing Surveys, Vol. 28, No. 1, März 1996.

[14] T. Ungerer: Parallelrechner und parallele Programmierung. Spektrum Akademischer Verlag Heidelberg/Berlin, 1997.

[15] D. B. Skillicorn, D. Talia: Models and Languages for Parallel Computing. In ACM Computing Surveys, Vol. 30, No. 2, Juni 1998.

[16] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface. Version 1.0, Oktober 1998. `http://www.openmp.org/drupal/mp-documents/cspec10.pdf`

[17] T. Rauber, G. Rünger: Parallele und verteilte Programmierung. Springer-Verlag Berlin/Heidelberg, 2000.

[18] A. J. van der Steen, J. J. Dongarra: Overview of recent Supercomputers. Technical Report, September 2004. `http://www.top500.org/orsc/2004/`

[19] Advanced Micro Devices: AMD Core Math Library (ACML). Version 3.5.0, 2006. `http://developer.amd.com/assets/acml_userguide.pdf`

[20] J. J. Dongarra: Performance of Various Computers Using Standard Linear Equations Software. Technical Report, University of Tennese and Oak Ridge National Laboratory, November 2006. `http://www.netlib.org/benchmark/performance.ps`

[21] Intel Corporation: Intel C++ Compiler Optimizing Applications. Version 9.0, 2006. `ftp://download.intel.com/support/performancetools/c/linux/v9/optaps_cls.pdf`

[22] Intel Corporation: Intel Fortran Compiler Optimizing Applications. Version 9.0, 2006. `ftp://download.intel.com/support/performancetools/fortran/linux/v9/optaps_for.pdf`

[23] Intel Corporation: Intel Math Kernel Library Reference Manual. Version 9.0, 2006. `http://www.intel.com/software/products/mkl/docs/WebHelp/mklrefman.htm`

[24] Numerical Algorithms Group: The NAG Parallel Library Manual. Release 3, 2006. `http://www.nag.co.uk/numeric/FD/manual/html/FDlibrarymanual.asp`

[25] The Portland Group: PGI User's Guide - Parallel Fortran, C and C++ for Scientists and Engineers. Release 6.2, August 2006. http://www.pgroup.com/doc/pgiug.pdf

[26] TOP500 Team: TOP500 Report for June 2006. Juni 2006. http://www.top500.org/static/lists/2006/06/top500_statistics.pdf

zu Kapitel 3

[27] The MathWorks Inc.: MATLAB: User's Guide. 24 Prime Park Way, Natick, MA 01760, August 1992.

[28] D. H. Munro: Yorick: An Interpreted Language. Lawrence Livermore National Laboratory, University of California, 1994. http://www.maumae.net/yorick/doc/yorick.pdf

[29] P. Janhunen: Tela User's Guide. Version 1.21, Finnish Meteorological Institute, Februar 1995. http://www.space.fmi.fi/~pjanhune/tela/doc/usrguide.ps.gz

[30] J. W. Eaton: GNU Octave: A high-level interactive language for numerical computations. Edition 3 for Octave version 2.1.x, Februar 1997. http://www.gnu.org/software/octave/doc/interpreter/

[31] Scilab Group: Introduction to Scilab, User's Guide. November 1998. http://www.scilab.org/doc/intro/intro.pdf

[32] I. Searle: Rlab2 Reference Manual. Version 2.1, März 1999. http://rlab.sourceforge.net/html/rlab-ref.html

[33] R. Grothmann: Euler Documentation. Version 1.60, August 2002. http://mathsrv.ku-eichstaett.de/MGF/homes/grotqhmann/euler/eulerdoc.html

[34] Aptech Systems Inc.: GAUSS User Guide. Version 6.0, Dezember 2003. http://www.aptech.com/manuals/UserGuide6.0.pdf

[35] S. Wolfram: The Mathematica Book, 5th edition. Wolfram Media, 2003. http://documents.wolfram.com/mathematica/Mathematica_V5_Book.zip

[36] RSI Research Systems Inc.: Using IDL. IDL Version 6.1, Juli 2004. http://www.ittvis.com

[37] Harmonic Software Inc.: O-Matrix. Version 5.8, Januar 2005. http://www.omatrix.com/manual/

[38] The MathWorks Inc.: Getting Started with MATLAB. Version 7, März 2005. http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/getstart.pdf

zu Kapitel 4

[39] P. Drakenberg, P. Jacobson, B. Kågström: A CONLAB Compiler for a Distributed Memory Multicomputer. In Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, März 1993.

[40] C. Moler: Why there isn't a parallel MATLAB. Matlab News and Notes, Spring 1995.

[41] V. P. Pauca, J. Hollingsworth, K. Liu: User's Guide for the Parallel Toolbox for MATLAB. Technical Report, Wake Forest University, Mai 1995.

[42] S. Pawletta, W. Drewelow, P. Dünow, T. Pawletta, M. Süße: A MATLAB toolbox for distributed and parallel processing. 2nd International MATLAB Conference, Cambridge, Oktober 1995.

[43] L. DeRose, D. Padua: A MATLAB to Fortran 90 Translator and its Effectiveness. In Proceedings of the 10th International Conference on Supercomputing, Januar 1996.

[44] A. E. Trefethen, V. Menon, C. Chang, G. Czajkowski, C. Myers, L. N. Trefethen: MultiMATLAB: MATLAB on Multiple Processors. Technical Report, Cornell Theory Center, 1996.

[45] T. Abrahamsson: Paralize. MATLAB Central File Exchange, November 1997. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=211`

[46] S. Chauveau, F. Bodin: Menhir: An Environment for High Performance Matlab. In Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Mai 1998.

[47] S. Pawletta: Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen. Dissertation, Universität Rostock, Juni 1998.

[48] M. J. Quinn, A. Malishevsky, N. Seelam: Otter: Bridging the Gap between MATLAB and ScaLAPACK. In Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, Juli 1998.

[49] G. Morrow, R. van de Geijn: A Parallel Linear Algebra Server for Matlab-like Environments. In Proceedings of the 1998 ACM/IEEE conference on Supercomputing, November 1998.

[50] D. Lee: PMI. MATLAB Central File Exchange, März 1999. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=219`

[51] G. Almasi, C. Cascaval, D. A. Padua: MATmarks: A Shared Memory Environment for MATLAB Programming. In Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, August 1999.

[52] U. Kjems: PLab. Technical University of Denmark, November 2000. `http://bond.imm.dtu.dk/plab/`

[53] L. Andrade: parmatlab. MATLAB Central File Exchange, April 2001. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=217`

[54] J. F. Baldomero: Message Passing under MATLAB. Advanced Simulation Technologies Conference (ASTC), Seattle Washington, April 2001.

[55] Kilvarock Corp.: IDL to PVM interface. August 2001. `http://www.kilvarock.com/software/idltopvm.htm`

[56] J. Kepner: Parallel Programming with MatlabMPI. High Performance Embedded Computing Workshop, MIT Lincoln Laboratory, November 2001.

[57] T. Abrahamsson: Beolab Toolbox for v6.5. MATLAB Central File Exchange, Januar 2002. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=1216`

[58] E. Heiberg: MATLAB Parallelization Toolkit 1.20. MATLAB Central File Exchange, Januar 2002. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=1227`

[59] M. D. DeVore: DistributePP. MATLAB Central File Exchange, Februar 2002. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=1287`

[60] D. Mastrovito: MPIDL Overview. Princeton Plasma Physics Laboratory, Januar 2003. `http://nstx.pppl.gov/nstx/Software/Programming/mpidl_use_doc.html`

[61] M. Ford: Parallel V1.1: Multi workspace GAUSS plus networking. Forward Computing and Control Pty. Ltd., September 2003. `http://www.aptech.com/pdf/Parallel_1.1.pdf`

[62] R. Choy, A. Edelman: Parallel MATLAB: Doing it Right. Technical Report, Computer Science AI Laboratory, Massachusetts Institute of Technology, November 2003.

[63] T. Obara: Parallel Octave. Tohoku University, Dezember 2003. `http://www.higuchi.ecei.tohoku.ac.jp/octave/`

[64] J. D. Cole: Distributed Octave. Transient Research, April 2004. `http://www.transientresearch.com/d-octave/`

[65] J. F. Baldomero: MPI Toolbox for Octave. In VECPAR'04 - Sixth International Meeting on High Performance Computing for Computational Science, Valencia, Juni 2004.

[66] R. E. Maeder: Parallel Computing Toolkit Documentation. 2004. http://documents.wolfram.com/applications/parallel

[67] K. Seymour, A. YarKhan, S. Agrawal, J. Dongarra: NetSolve: Grid Enabling Scientific Computing. In Grid Computing: The New Frontier of High Performance Computing, Vol. 14, Elsevier, Advances in Parallel Computing, Mai 2005.

[68] D. Gruber, C. Kastinger, T. Mayerdorfer, S. Zorn-Pauli: MDiCE R2.0 - MultiMDiCE. Carinthia Tech Institute, Klagenfurt, Juli 2005. http://www.fh-kaernten.at/mdice/Download/MDiCE_R2_0_TechnischeDokumentation.pdf

[69] B. Phelan: Matlab 2 Matlab : Distributed Computing Toolbox. XTargets, November 2005. http://xtargets.com/cms/Tutorials/Matlab-Programming/Matlab-2-Matlab-Distributed-Computing-Toolbox.html

[70] H. Kim, J. Mullen: Introduction to Parallel Programming and pMatlab v0.7. MIT Lincoln Laboratory, Februar 2005. http://www.ll.mit.edu/pMatlab/files/pMatlab_intro.pdf

[71] R. Panuganti, M. M. Baskaran, D. E. Hudak, A. Krishnamurthy, J. Nieplocha, A. Rountev, P. Sadayappan: GAMMA: Global Arrays meets MATLAB. Technical Report, Ohio State University, Januar 2006.

[72] Scilab Group: PVM parallel toolbox. Scilab Documentation (4.0 Version), Februar 2006. http://www.scilab.org/product/man-eng/pvm/whatis.htm

[73] The MathWorks, Inc.: Distributed Computing Toolbox For Use with MATLAB. User's Guide, Version 2, März 2006. http://www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/distcomp.pdf

[74] J. Zollweg: Cornell Multitask Toolbox for MATLAB. Cornell Theory Center, 2006. http://www.tc.cornell.edu/Services/Support/Forms/cmtm

zu Kapitel 5

[75] G. Kindel: Weiterführende Leistungsanalyse eines Parallelverarbeitungssystems. Diplomarbeit, Hochschule Wismar, August 2005.

[76] R. Fink, S. Pawletta: The DP-Toolbox Home Page. März 2006. http://www.mb.hs-wismar.de/cea/dp

[77] J. F. Baldomero: LAM / MPI Parallel Computing under MATLAB. Mai 2006. `http://atc.ugr.es/javier-bin/mpitb_eng`

[78] J. F. Baldomero: LAM / MPI Parallel Computing under Octave. Juni 2006. `http://atc.ugr.es/javier-bin/mpitb`

[79] The MathWorks Inc.: Distributed Computing Toolbox. 2006. `http://www.mathworks.com/products/distribtb`

[80] J. Kepner: Parallel Programming with MatlabMPI. 2006. `http://www.ll.mit.edu/MatlabMPI`

[81] J. Kepner: MIT Lincoln Laboratory - Parallel Programming with MatlabMPI. 2006. `http://www.ll.mit.edu/MatlabMPI`

[82] INRIA: Scilab Home Page. 2006. `http://www.scilab.org`

zu Kapitel 6

[83] T. Weber: Realisierung eines Remote-Execution-Service für Matlab-Prozesse auf Basis von DCOM/WINDOWS. Studienarbeit, Hochschule Wismar, Januar 2006.

zu Kapitel 7

[84] F. Breitenecker, I. Husinsky, G. Schuster: Comparison of parallel simulation techniques. In Simulation News Europe, Issue 10, März 1994.

[85] H. Mehl: Methoden verteilter Simulation. Vieweg, Braunschweig/Wiesbaden, 1994.

[86] S. Hou, Q. Zou, S. Chen, G. D. Doolen, A. C. Cogley: Simulation of Cavity Flow by the Lattice Boltzmann Method. Journal of Computational Physics, Vol. 118, Issue 2, Mai 1995.

[87] H. Pohlheim: GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB. Documentation, Version 3.70, November 2005. `http://www.geatbx.com/docu/`

[88] H. Pohlheim, M. Conrad, A. Griep: Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences. In SAE 2005 Transactions Journal of Passenger Cars: Mechanical Systems, Februar 2006.

[89] F. Breitenecker, G. Höfinger, R. Fink, S. Pawletta, T. Pawletta: ARGESIM Benchmark on Parallel and Distributed Simulation. In Simulations News Europe, Special Issue: Parallel and Distributed Simulation Methods and Environments, Vol. 6, Issue 2, September 2006.

Veröffentlichungen des Autors

[90] R. Fink: Verteiltes und paralleles Rechnen mit Matlab. Technischer Bericht, Hochschule Wismar, Februar 2003.

[91] R. Fink: Vergleich paralleler Simulationstechniken. Studienarbeit, Hochschule Wismar, November 2003.

[92] R. Fink, S. Pawletta, T. Pawletta: A Matlab-based Solution to ARGESIM "Comparison of Parallel Simulation Techniques" using DP-Toolbox. In Simulation News Europe, Issue 38/39, Dezember 2003.

[93] R. Fink: Entwicklung einer neuen Benchmarksuite zur Evaluierung paralleler Simulationstechniken. Diplomarbeit, Hochschule Wismar, Januar 2004.

[94] R. Fink, S. Pawletta, M. Schultalbers: Matlab-based parallel optimization with integrated simulation. In 5th EUROSIM Congress on Modeling and Simulation, ESIEE Paris, September 2004.

[95] R. Fink, S. Pawletta, T. Pawletta: Untersuchung von Werkzeugen zur Parallelverarbeitung in Matlab und ähnlichen Systemen hinsichtlich ihrer Eignung für Simulationsanwendungen. In 18. Symposium Simulationstechnik, ASIM05, SCS Publishing House e.V., September 2005.

[96] R. Fink, S. Pawletta, T. Pawletta: Verteilte und parallele Verarbeitung in wissenschaftlich-technischen Berechnungsumgebungen. In 2. Workshop „Grid-Technologie für den Entwurf technischer Systeme", April 2006.

[97] R. Fink, S. Pawletta, B. Lampe, T. Pawletta: SCE based Parallel Processing and Applications in Simulation. In Simulations News Europe, Special Issue: Parallel and Distributed Simulation Methods and Environments, Vol. 6, Issue 2, September 2006.

[98] R. Fink: Implementing the 2D square lattice Boltzmann method in Matlab. Technischer Bericht, Hochschule Wismar, September 2006. `http://www.mb.hs-wismar.de/cea/lbm/implementing_lbm.pdf`

[99] R. Fink, S. Pawletta, B. Lampe, M. Schultalbers, T. Pawletta: Parallelverarbeitung mit Matlab – DC- und DP-Toolbox im Vergleich. In at-Automatisierungstechnik, 54 (2006) 10, Oktober 2006.

Abbildungsverzeichnis

Tabellenverzeichnis

A Kommunikationsleistung von Multi-SCE-Systemen – Messergebnisse

A.1 Existierende Systeme

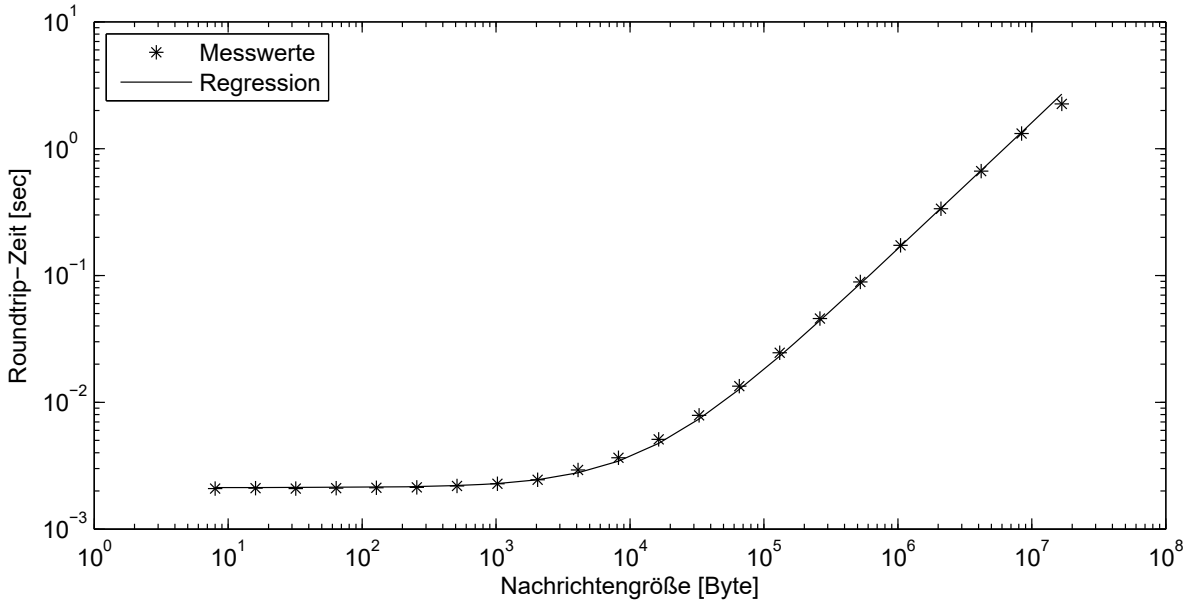


Abbildung A.1: Kommunikationsleistung DP-Toolbox v1.5 (Matlab, Message-Passing)

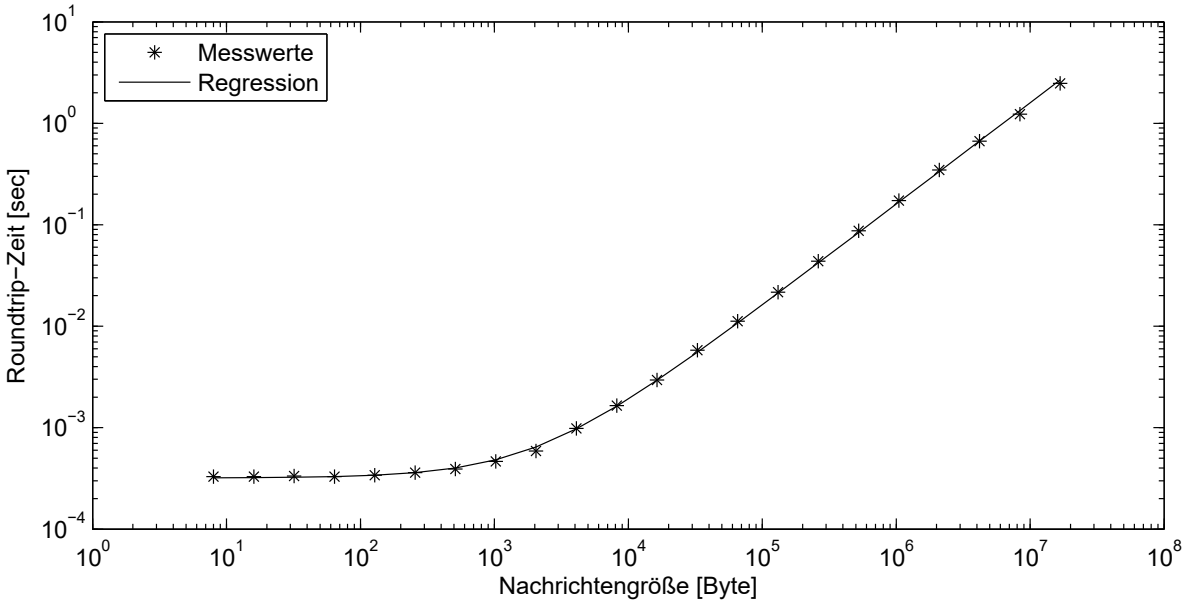


Abbildung A.2: Kommunikationsleistung PVM Toolbox (Scilab, Message-Passing)

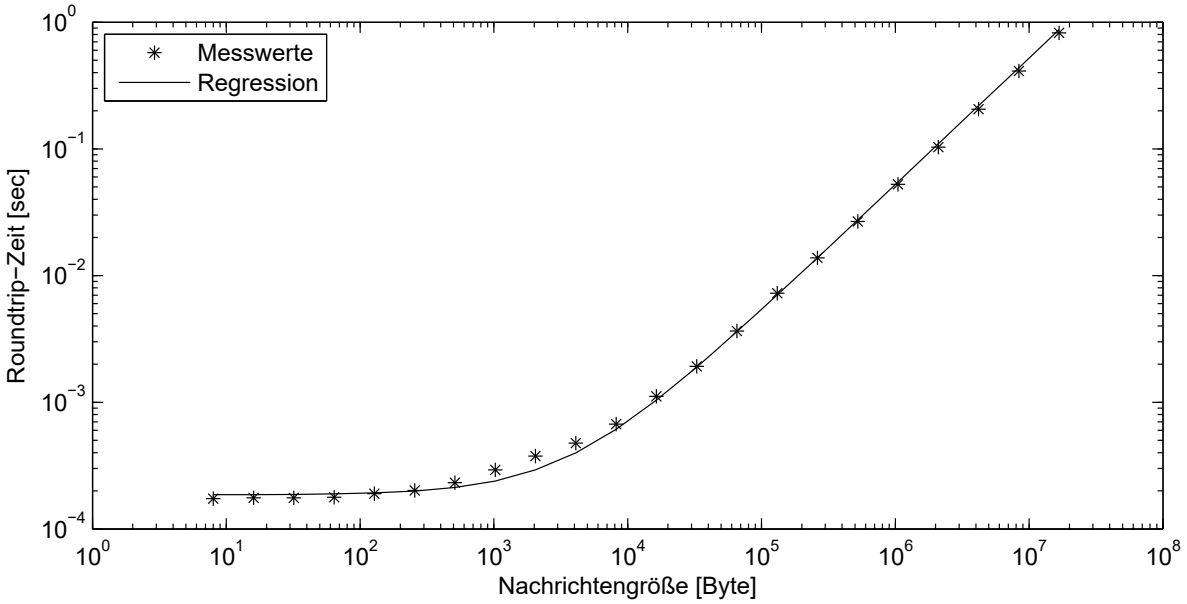


Abbildung A.3: Kommunikationsleistung MPI Toolbox (Matlab, Message-Passing)

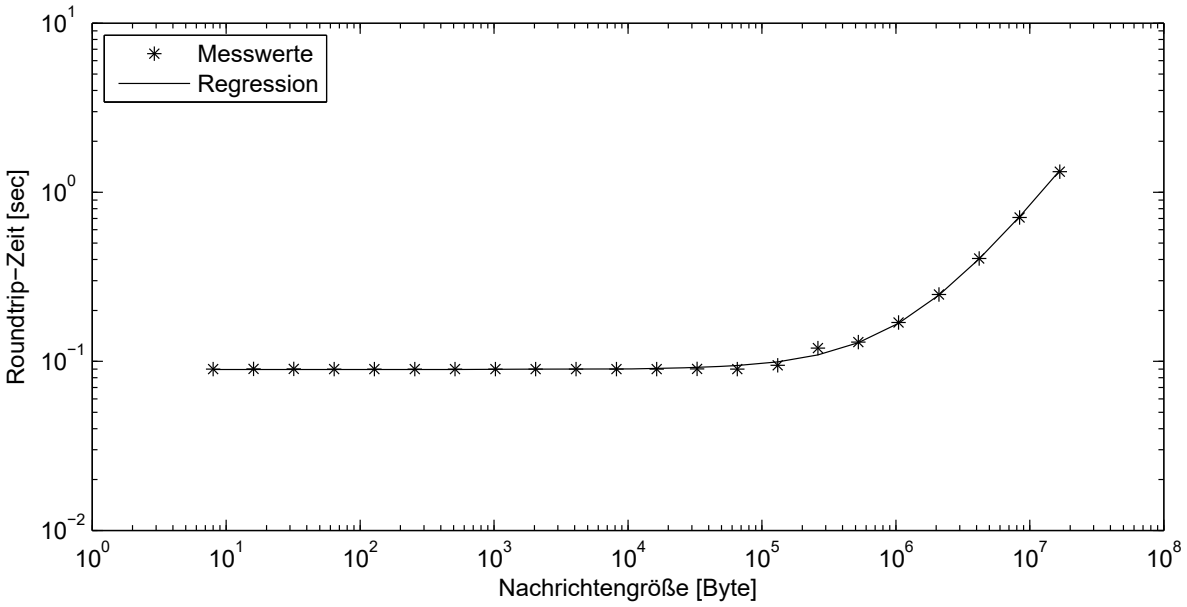


Abbildung A.4: Kommunikationsleistung MatlabMPI v1.2 (Matlab, Message-Passing)

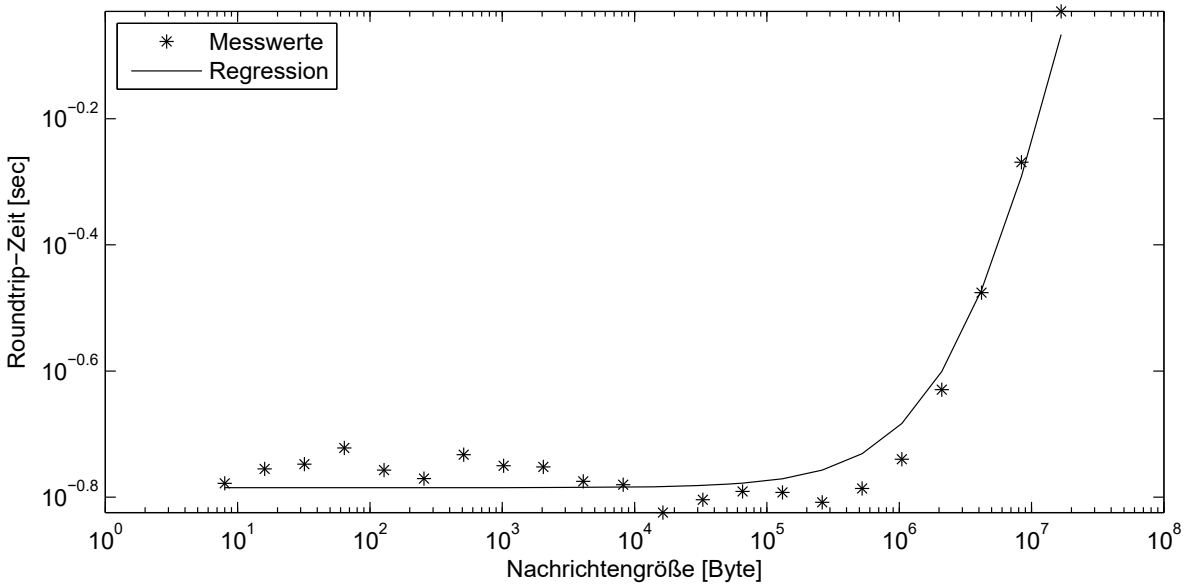


Abbildung A.5: Kommunikationsleistung Beolab Toolbox (Matlab, RPC)

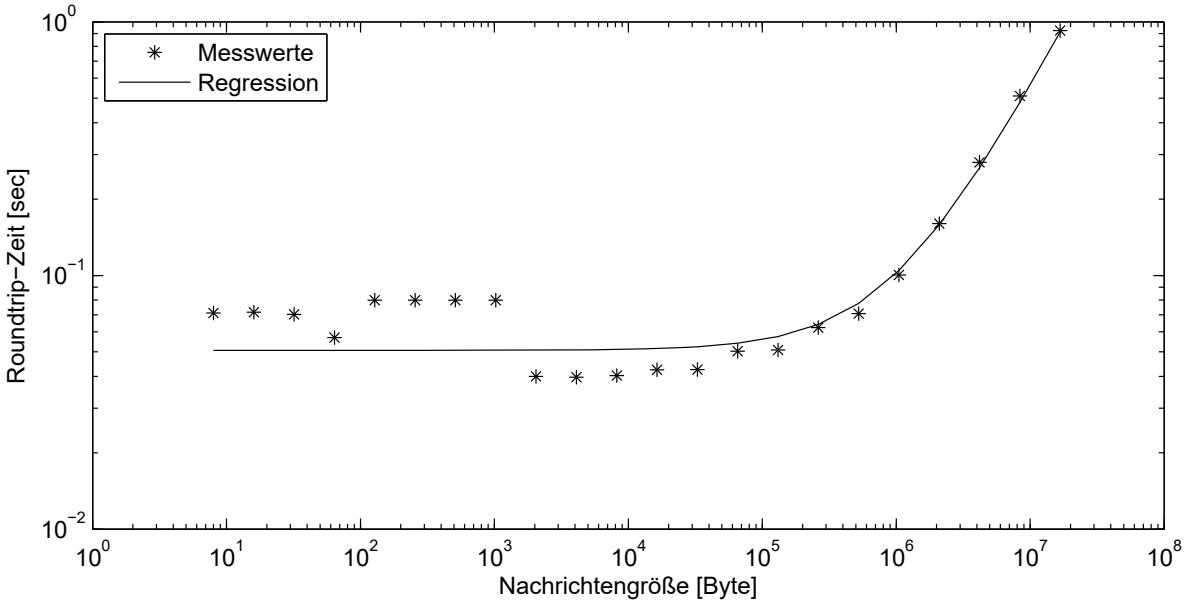


Abbildung A.6: Kommunikationsleistung Parallelization Toolkit (Matlab, RPC)

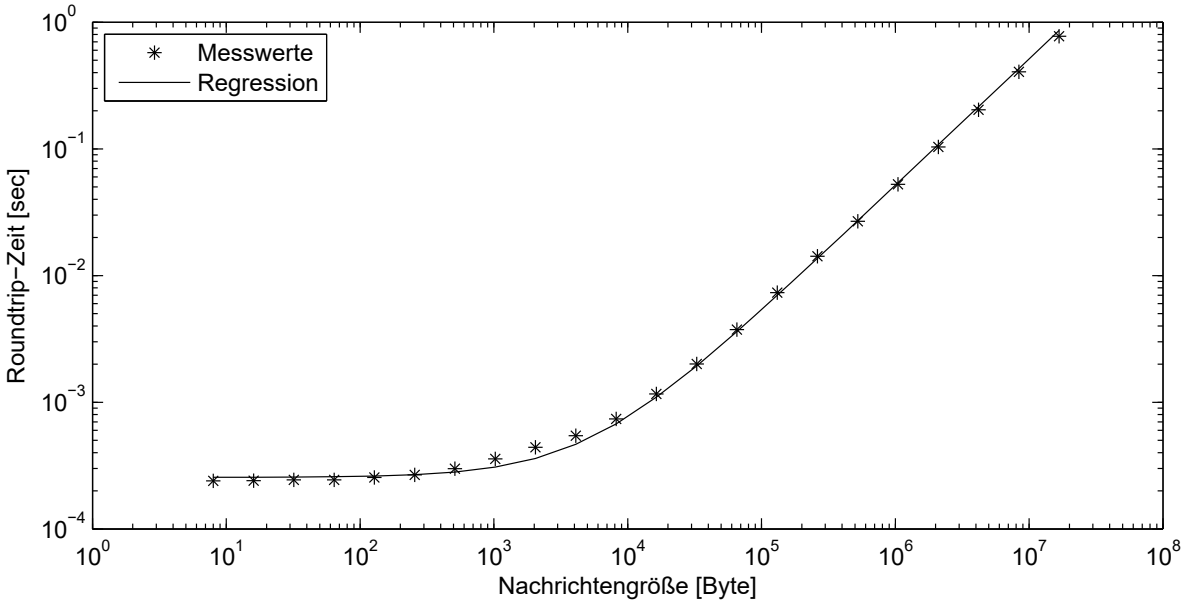


Abbildung A.7: Kommunikationsleistung MPI Toolbox (Octave, Message-Passing)

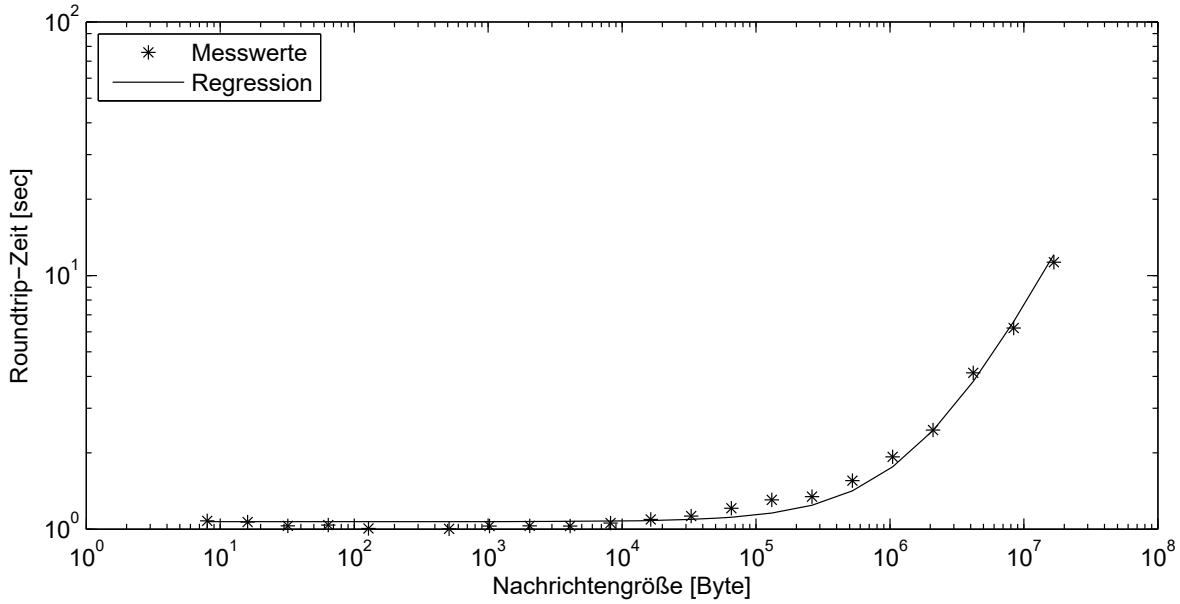


Abbildung A.8: Kommunikationsleistung DC-Toolbox v2.0 (Matlab, RPC)

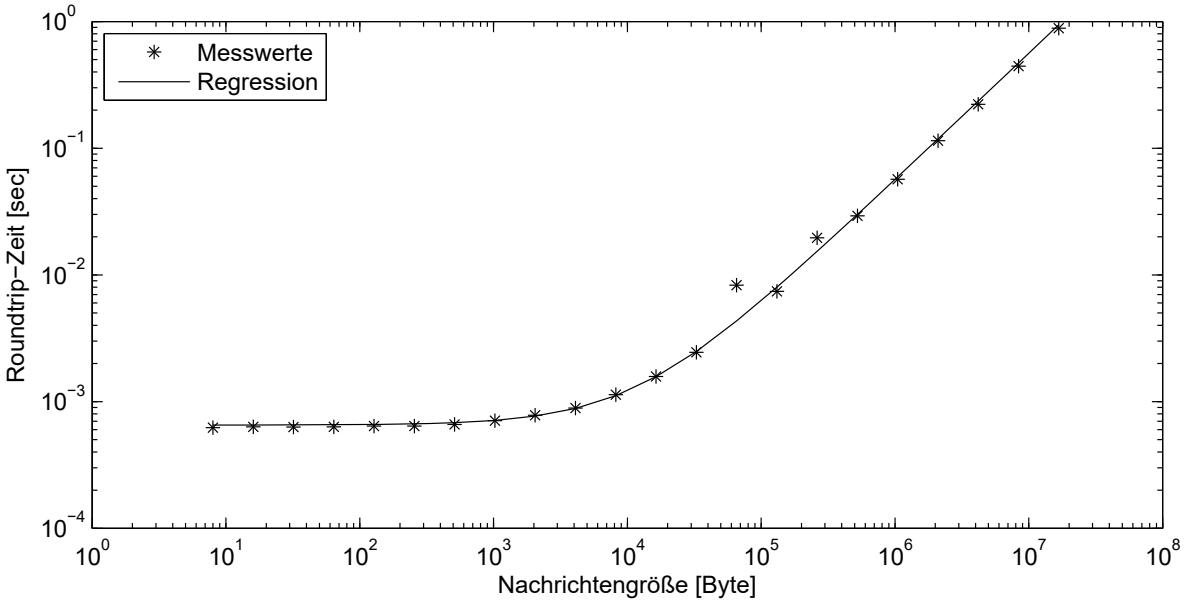


Abbildung A.9: Kommunikationsleistung DC-Toolbox v2.0 (Matlab, Message-Passing)

A.2 Entwickelte Systeme

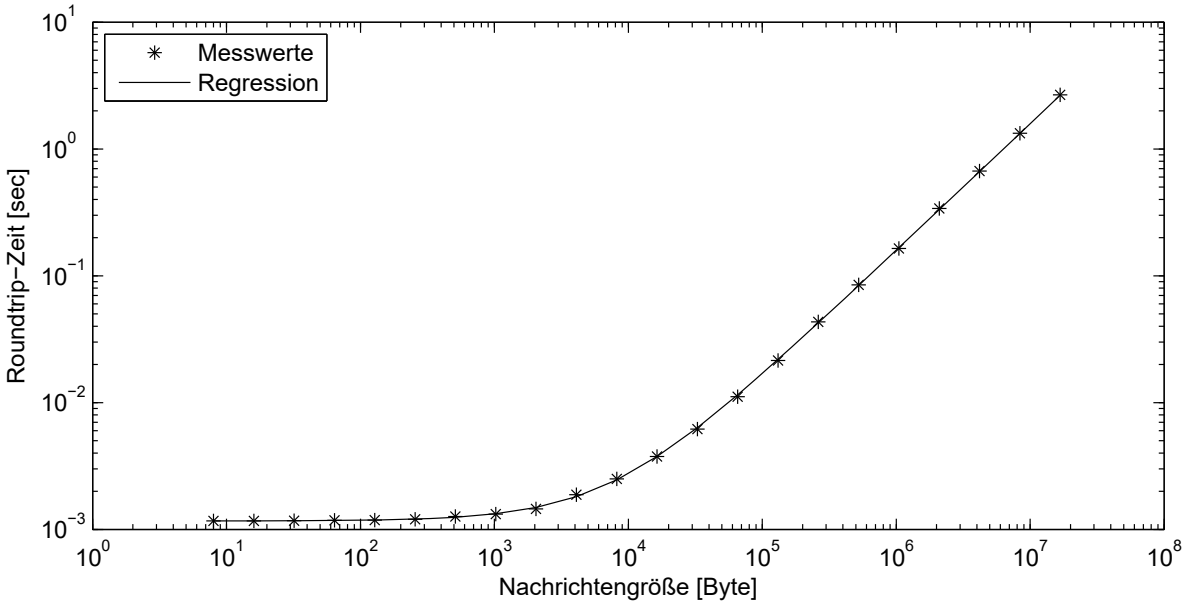


Abbildung A.10: Kommunikationsleistung DP-1.7 (Matlab, Message-Passing)

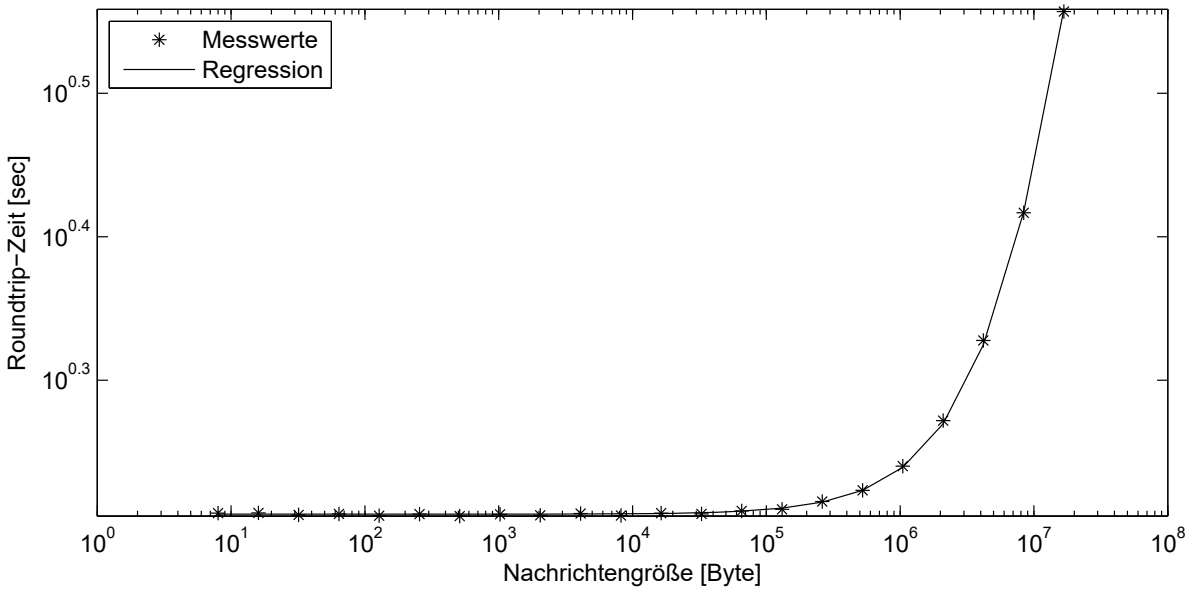


Abbildung A.11: Kommunikationsleistung DP-1.7 (Matlab, RPC)

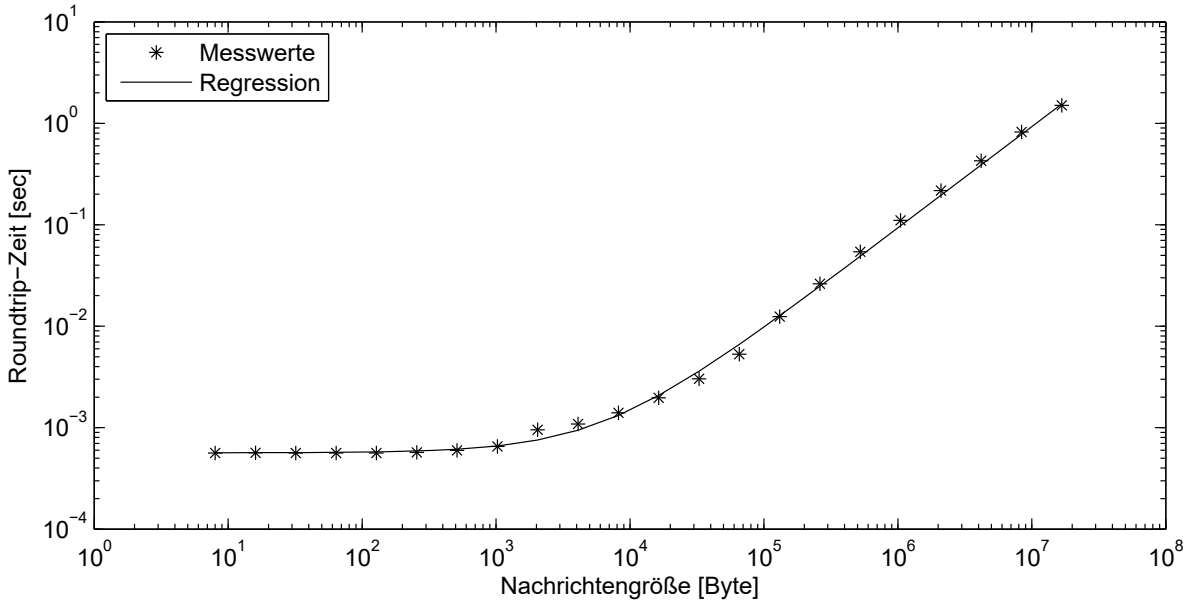


Abbildung A.12: Kommunikationsleistung DP-MPI (Matlab, Message-Passing)

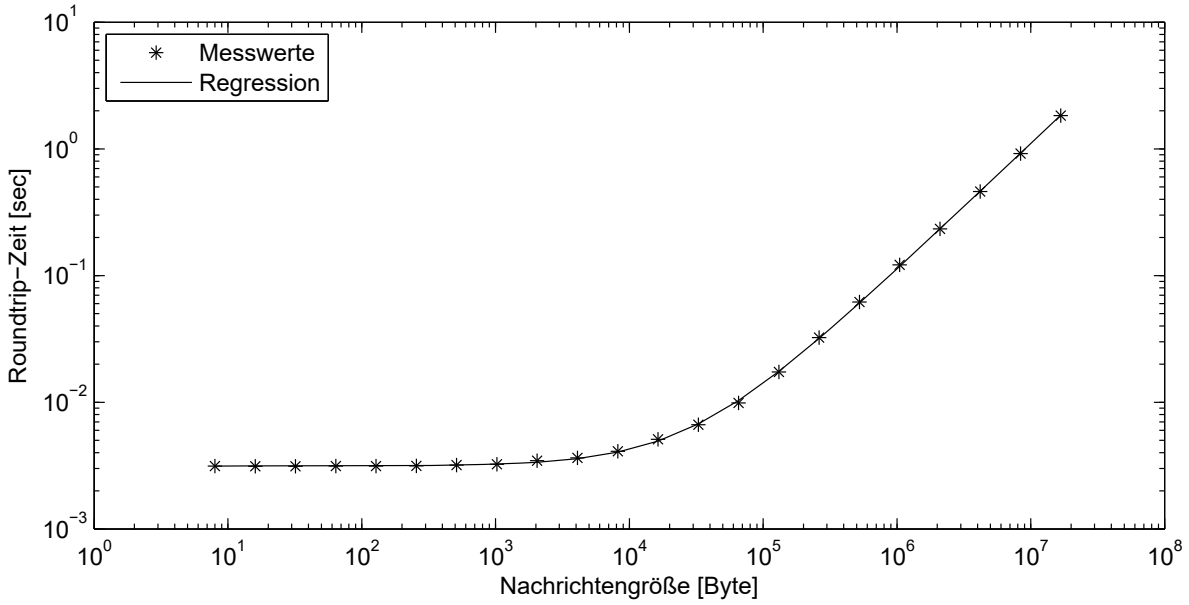


Abbildung A.13: Kommunikationsleistung DP-MPI (Matlab, Shared-Memory)

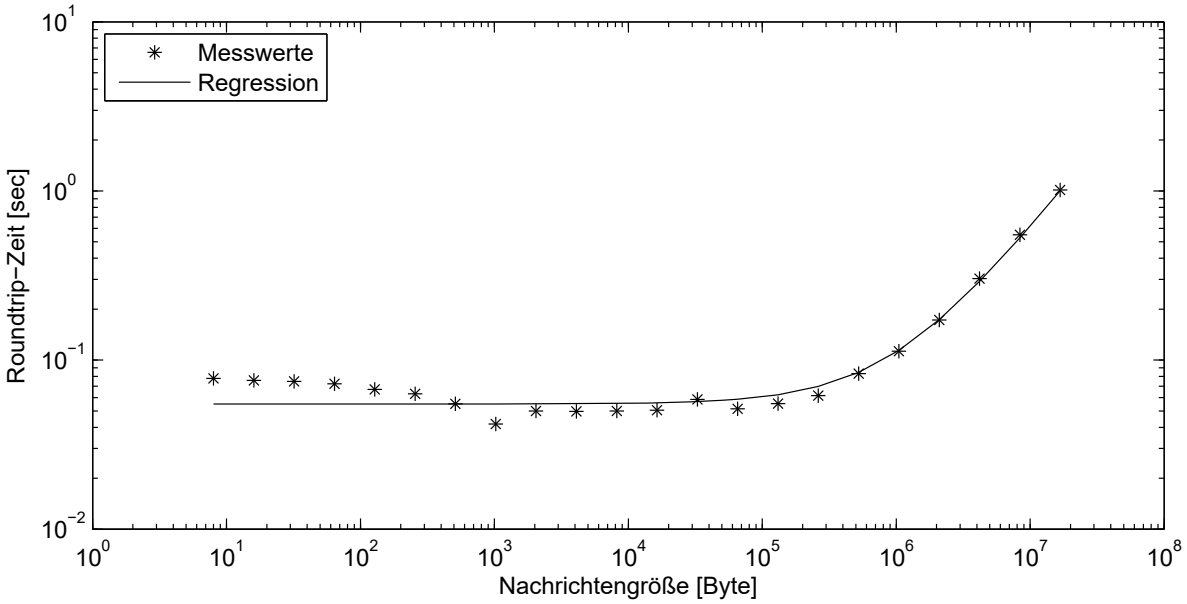


Abbildung A.14: Kommunikationsleistung DP-ME (Matlab, RPC)

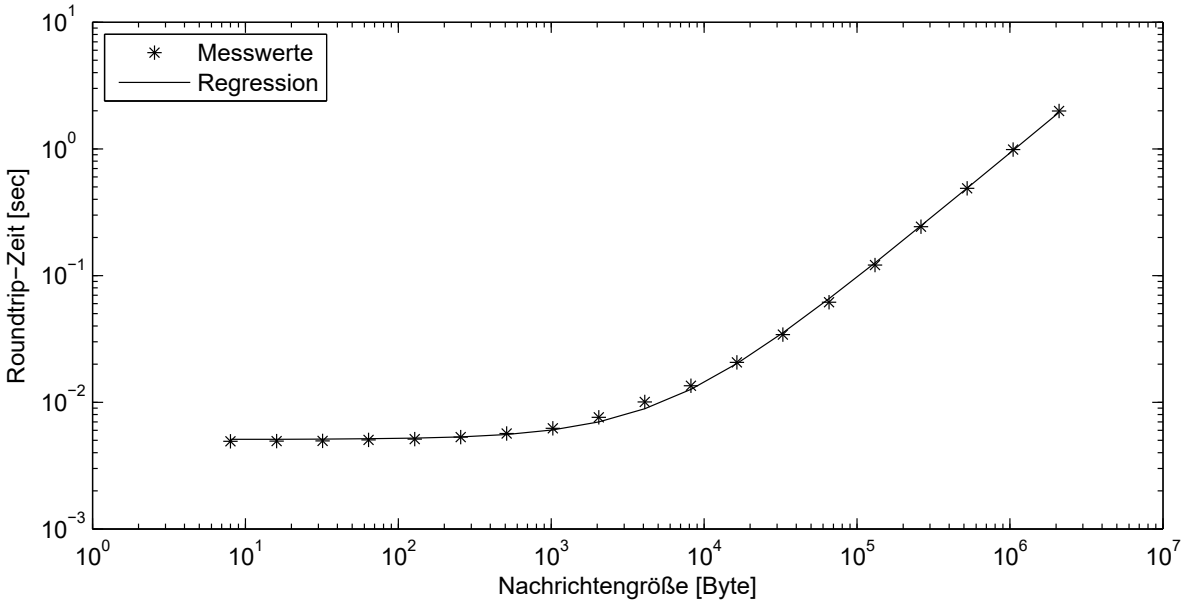


Abbildung A.15: Kommunikationsleistung DP-Java (Matlab, Message-Passing)

B Codebeispiele

B.1 Parameterstudie eines Feder-Masse-Systems

Hauptprogramm sequentiell

```
t0=0; % Startzeit der Simulation
h=0.001; % Schrittweite
steps=2000; % Anzahl Schritte
x0=[0,0.1]; % Anfangszustände

nruns=1000; % Anzahl Simulationsläufe
rand('state',0); % Initialisierung Zufallszahlengenerator
d=800+400*rand(nruns,1); % Erzeugung Dämpfungsparameter

xmean=zeros(steps+1,1); % Initialisierung des Ergebnisvektors
% Beginn Schleife über alle Dämpfungsparameter
for i=1:numel(d)
    x=rk4('mcs_equ',t0,h,steps,x0,d(i)); % Simulationslauf
xmean=xmean+x(:,2); % Summation der Simulationsergebnisse
end
% Ende Schleife über alle Dämpfungsparameter
xmean=xmean/numel(d); % Mittelwertbildung

save('result_mcs_seq.mat','xmean'); % Speicherung der Ergebnisse
```

RK4-Algorithmus (verwendet in Abschnitt B.1, B.2 und B.3)

```
function x=rk4(fun,t0,h,steps,x0,varargin)

x=zeros(steps+1,numel(x0)); % Initialisierung der Ergebnismatrix
x(1,:)=x0; % Einfügen der Anfangszustände
t=t0; % Simulationszeit initialisieren

% Beginn Simulationsschleife
for i=1:steps
    % Beginn Berechnung der Stufenvektoren k1-k4
    k1=h*feval(fun,t,x(i,:),varargin{:});
    k2=h*feval(fun,t+h/2,x(i,:)+k1/2,varargin{:});
    k3=h*feval(fun,t+h/2,x(i,:)+k2/2,varargin{:});
    k4=h*feval(fun,t+h,x(i,:)+k3,varargin{:});
    % Ende Berechnung der Stufenvektoren k1-k4
    x(i+1,:)=x(i,:)+(k1+2*k2+2*k3+k4)/6; % Berechnung neuer Zustände
    t=t0+i*h; % Zeitfortschaltung
end
% Ende Simulationsschleife
```

Ableitungsfunktion

```
function xdot=mcs_equ(t,x,D)

% Beginn Modellparameter
k=9000;
m=450;
% Ende Modellparameter

xdot=[0,0]; % Initialisierung des Ergebnisvektors

% Beginn Modellgleichungen
xdot(1)=-(D*x(1)+k*x(2))/m;
xdot(2)=x(1);
% Ende Modellgleichungen
```

Hauptprogramm parallel DP-1.7 (Message-Passing)

```
if dpparent==dpmyid % wenn übergeordnete Instanz
    % Beginn Masterprogramm
    nruns=1000; % Anzahl Simulationsläufe
    rand('state',0); % Initialisierung Zufallszahlengenerator
    d=800+400*rand(nruns,1); % Erzeugung Dämpfungsparameter

    hosts={'vprdpc1','vprdpc1','vprdpc2','vprdpc2',...
        'vprdpc3','vprdpc3','vprdpc4','vprdpc4',...
        'vprspc1','vprspc2','vprspc3','vprspc4'}; % beteiligte Rechner
    tid=dpspawn(hosts,mfilename); % Instanziierung der Slaveprogramme

    dpscatter(d,tid); % Verteilung der Dämpfungsparameter

    xmean=dpgather(tid)'; % Einsammeln der Ergebnisse
    xmean=sum(xmean,2)/nruns; % Mittelwertbildung

    dpexit; % Schließen des DP-Subsystems

    save('result_mcs_par_dp.mat','xmean'); % Speicherung der Ergebnisse
    % Ende Masterprogramm
else % wenn untergeordnete Instanz
    % Beginn Slaveprogramm
    t0=0; % Startzeit der Simulation
    h=0.001; % Schrittweite
    steps=2000; % Anzahl Schritte
    x0=[0,0.1]; % Anfangszustände

    d=dprecv(dpparent); % Empfangen der Dämpfungsparameter

    xmean=zeros(steps+1,1); % Initialisierung des Ergebnisvektors
    % Beginn Schleife über alle Dämpfungsparameter
    for i=1:numel(d)
        x=rk4('mcs_equ',t0,h,steps,x0,d(i)); % Simulationslauf
        xmean=xmean+x(:,2); % Summation der Simulationsergebnisse
    end
    % Ende Schleife über alle Dämpfungsparameter
    dpsend(xmean',dpparent); % Senden des Ergebnisvektors

    exit; % Instanz schließen
    % Ende Slaveprogramm
end
```

Hauptprogramm parallel DP-MPI (Message-Passing)

```
nhost=12; % Anzahl beteiligter Rechner
MPI_Init(nhost); % Initialisierung DP-MPI und Programminstanziierung

t0=0; % Startzeit der Simulation
h=0.001; % Schrittweite
steps=2000; % Anzahl Schritte
x0=[0,0.1]; % Anfangszustände

nruns=1000; % Anzahl Simulationsläufe
rand('state',0); % Initialisierung Zufallszahlengenerator
d=800+400*rand(nruns,1); % Erzeugung Dämpfungsparameter

d=MPI_Scatter(d); % Verteilung der Dämpfungsparameter

xmean=zeros(steps+1,1); % Initialisierung des Ergebnisvektors
% Beginn Schleife über alle Dämpfungsparameter
for i=1:numel(d)
    x=rk4('mcs_equ',t0,h,steps,x0,d(i)); % Simulationslauf
    xmean=xmean+x(:,2); % Summation der Simulationsergebnisse
end
% Ende Schleife über alle Dämpfungsparameter

xmean=MPI_Gather(xmean,2); % Einsammeln der Ergebnisse
xmean=sum(xmean,2)/nruns; % Mittelwertbildung

MPI_Finalize; % Schließen des DP-MPI-Subsystems

save('result_mcs_par_mpi.mat','xmean'); % Speicherung der Ergebnisse
```

Hauptprogramm parallel DC-2.0 (RPC)

```
t0=0; % Startzeit der Simulation
h=0.001; % Schrittweite
steps=2000; % Anzahl Schritte
x0=[0,0.1]; % Anfangszustände

nruns=1000; % Anzahl Simulationsläufe
rand('state',0); % Initialisierung Zufallszahlengenerator
d=800+400*rand(nruns,1); % Erzeugung Dämpfungsparameter

x_=dfeval('rk4',...
    cellstr(repmat('mcs_equ',nruns,1)),...
    num2cell(repmat(t0,nruns,1)),...
    num2cell(repmat(h,nruns,1)),...
    num2cell(repmat(steps,nruns,1)),...
    num2cell(repmat(x0,nruns,1),2),...
    num2cell(d),...
    'lookupurl','vprspc1',...
    'filedependencies',{'rk4.m','mcs_equ.m'}); % vektorieller RPC-Ruf

% Beginn Ergebnistransformation Cell-Array -> Double-Matrix
x=cell2mat(x_);
x=reshape(x(:,2),steps+1,nruns);
% Ende Ergebnistransformation Cell-Array -> Double-Matrix
xmean=mean(x,2); % Mittelwertbildung

jm=findResource('jobmanager','lookupurl','vprspc1'); % Jobmanager suchen
destroy(get(jm,'jobs')); % RPC-Job löschen

save('result_mcs_par_dc.mat','xmean'); % Speicherung der Ergebnisse
```

Hauptprogramm parallel DP-1.7 (RPC)

```
t0=0; % Startzeit der Simulation
h=0.001; % Schrittweite
steps=2000; % Anzahl Schritte
x0=[0,0.1]; % Anfangszustände

nruns=1000; % Anzahl Simulationsläufe
rand('state',0); % Initialisierung Zufallszahlengenerator
d=800+400*rand(nruns,1); % Erzeugung Dämpfungsparameter

hosts={'vprdpc1','vprdpc1','vprdpc2','vprdpc2',...
    'vprdpc3','vprdpc3','vprdpc4','vprdpc4',...
    'vprspc1','vprspc2','vprspc3','vprspc4'}; % beteiligte Rechner

x_=dpeval(hosts,'rk4',...
    cellstr(repmat('mcs_equ',nruns,1)),...
    num2cell(repmat(t0,nruns,1)),...
    num2cell(repmat(h,nruns,1)),...
    num2cell(repmat(steps,nruns,1)),...
    num2cell(repmat(x0,nruns,1),2),...
    num2cell(d)); % vektorieller RPC-Ruf

% Beginn Ergebnistransformation Cell-Array -> Double-Matrix
x=cell2mat(x_);
x=x(:,2:2:nruns*2);
% Ende Ergebnistransformation Cell-Array -> Double-Matrix
xmean=mean(x,2); % Mittelwertbildung

save('result_mcs_par_dp_rpc.mat','xmean'); % Speicherung der Ergebnisse
```

B.2 Simulation gekoppelter Räuber-Beute-Systeme

Hauptprogramm sequentiell

```
t0=0; % Startzeit der Simulation
h=0.01; % Schrittweite
steps=10000; % Anzahl Schritte
x0=ones(1,10); % Anfangszustände

x=rk4('rbs_equ_seq',t0,h,steps,x0); % Simulationslauf

x100=x(end,:); % Simulationsergebnisse ablegen

save('result_rbs_seq.mat','x100'); % Speicherung der Ergebnisse
```

Ableitungsfunktion sequentiell

```
function  Xdot=rbs_equ_seq(t,X)

v1=X(1);v2=X(2);w1=X(3);w2=X(4);x1=X(5);x2=X(6);y1=X(7);y2=X(8);...
    z1=X(9);z2=X(10); % Auslesen der Zustandsgrößen

% Beginn Modellparameter und -gleichungen System 1
av=2;bv=0.5;cv=0.01;dv=0.2;ev=0.4;fv=0.02;gv=0.01;hv=0.02;
jv=0.01;kv=0.03;
sv=v2*(gv*w1+hv*x1+jv*y1+kv*z1);
v1dot=av*v1-bv*v1*v2-cv*v1^2;
v2dot=-dv*v2+ev*v1*v2-fv*v2^2+sv;
% Ende Modellparameter und -gleichungen System 1

% Beginn Modellparameter und -gleichungen System 2
aw=1;bw=0.5;cw=0.02;dw=0.1;ew=0.4;fw=0.04;gw=0.02;hw=0.04;
rw=w1*(-gw*v2+hw*x2);
w1dot=aw*w1-bw*w1*w2-cw*w1^2+rw;
w2dot=-dw*w2+ew*w1*w2-fw*w2^2;
% Ende Modellparameter und -gleichungen System 2

% Beginn Modellparameter und -gleichungen System 3
ax=3;bx=0.9;cx=0.02;dx=0.2;ex=0.2;fx=0.04;gx=0.025;hx=0.1;
rx=-gx*x1*v2;
sx=-hx*x2*w1;
x1dot=ax*x1-bx*x1*x2-cx*x1^2+rx;
x2dot=-dx*x2+ex*x1*x2-fx*x2^2+sx;
% Ende Modellparameter und -gleichungen System 3
```

```
% Beginn Modellparameter und -gleichungen System 4
ay=1;by=0.8;cy=0.04;dy=0.2;ey=0.6;fy=0.07;gy=0.03;hy=0.025;
ry=y1*(-gy*v2+hy*z2);
y1dot=ay*y1-by*y1*y2-cy*y1^2+ry;
y2dot=-dy*y2+ey*y1*y2-fy*y2^2;
% Ende Modellparameter und -gleichungen System 4

% Beginn Modellparameter und -gleichungen System 5
az=3;bz=0.7;cz=0.02;dz=0.5;ez=0.3;fz=0.04;gz=0.02;hz = 0.04;
rz=-gz*z1*v2;
sz=-hz*z2*y1;
z1dot=az*z1-bz*z1*z2-cz*z1^2+rz;
z2dot=-dz*z2+ez*z1*z2-fz*z2^2+sz;
% Ende Modellparameter und -gleichungen System 5

Xdot=[v1dot,v2dot,w1dot,w2dot,x1dot,x2dot,y1dot,y2dot,...
    z1dot,z2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

Hauptprogramm parallel DP-1.7 (Message-Passing)

```
t0=0; % Startzeit der Simulation
h=0.01; % Schrittweite
steps=10000; % Anzahl Schritte

if dpparent==dpmyid % wenn übergeordnete Instanz
    hosts={'vprspc2','vprspc3','vprspc4','vprdpc1'}; % beteiligte Rechner
    tid=dpspawn(hosts,mfilename); % Instanziierung weiterer Programme
    tid(end+1)=dpmyid; % Identifikationsnummern zusammenfassen
    dpsend(tid,tid); % Identifikationsnummern senden
end
tid=dprecv(dpparent); % Identifikationsnummern empfangen
x0=ones(1,2); % Anfangszustände

x=rk4('rbs_equ_par_dp',t0,h,steps,x0,tid); % Simulationslauf

x100=x(end,:); % Simulationsergebnisse ablegen

dpsend(x100',dpparent); % Simulationsergebnisse senden
if dpparent==dpmyid % wenn übergeordnete Instanz
    x100=dpgather(tid)'; % Simulationsergebnisse einsammeln
    dpexit; % Schließen des DP-Subsystems
    save('result_rbs_par_dp.mat','x100'); % Speicherung der Ergebnisse
else % wenn untergeordenete Instanz
    exit; % Instanz schließen
end
```

Ableitungsfunktion parallel DP-1.7 (Message-Passing)

```
function  Xdot=rbs_equ_par_dp(t,X,tid)

if tid(1)==dpmyid % wenn Identifikationsnummer an erster Listenposition
    v1=X(1);v2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    w1=dprecv(tid(2));
    x1=dprecv(tid(3));
    y1=dprecv(tid(4));
    z1=dprecv(tid(5));
    dpsend(v2,tid(2:5));
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 1
    av=2;bv=0.5;cv=0.01;dv=0.2;ev=0.4;fv=0.02;gv=0.01;hv=0.02;...
```

```
        jv=0.01;kv=0.03;
    sv=v2*(gv*w1+hv*x1+jv*y1+kv*z1);
    v1dot=av*v1-bv*v1*v2-cv*v1^2;
    v2dot=-dv*v2+ev*v1*v2-fv*v2^2+sv;
    % Ende Modellparameter und -gleichungen System 1
    Xdot=[v1dot,v2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if tid(2)==dpmyid % wenn Identifikationsnummer an zweiter Listenposition
    w1=X(1);w2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    dpsend(w1,tid(1));
    v2=dprecv(tid(1));
    x2=dprecv(tid(3));
    dpsend(w1,tid(3));
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 2
    aw=1;bw=0.5;cw=0.02;dw=0.1;ew=0.4;fw=0.04;gw=0.02;hw=0.04;
    rw=w1*(-gw*v2+hw*x2);
    w1dot=aw*w1-bw*w1*w2-cw*w1^2+rw;
    w2dot=-dw*w2+ew*w1*w2-fw*w2^2;
    % Ende Modellparameter und -gleichungen System 2
    Xdot=[w1dot,w2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if tid(3)==dpmyid % wenn Identifikationsnummer an dritter Listenposition
    x1=X(1);x2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    dpsend(x1,tid(1));
    dpsend(x2,tid(2));
    v2=dprecv(tid(1));
    w1=dprecv(tid(2));
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 3
    ax=3;bx=0.9;cx=0.02;dx=0.2;ex=0.2;fx=0.04;gx=0.025;hx=0.1;
    rx=-gx*x1*v2;
    sx=-hx*x2*w1;
    x1dot=ax*x1-bx*x1*x2-cx*x1^2+rx;
    x2dot=-dx*x2+ex*x1*x2-fx*x2^2+sx;
    % Ende Modellparameter und -gleichungen System 3
    Xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end
```

```
if tid(4)==dpmyid % wenn Identifikationsnummer an vierter Listenposition
    y1=X(1);y2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    dpsend(y1,tid(1));
    v2=dprecv(tid(1));
    z2=dprecv(tid(5));
    dpsend(y1,tid(5));
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 4
    ay=1;by=0.8;cy=0.04;dy=0.2;ey=0.6;fy=0.07;gy=0.03;hy=0.025;
    ry=y1*(-gy*v2+hy*z2);
    y1dot=ay*y1-by*y1*y2-cy*y1^2+ry;
    y2dot=-dy*y2+ey*y1*y2-fy*y2^2;
    % Ende Modellparameter und -gleichungen System 4
    Xdot=[y1dot,y2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if tid(5)==dpmyid % wenn Identifikationsnummer an fünfter Listenposition
    z1=X(1);z2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    dpsend(z1,tid(1));
    dpsend(z2,tid(4));
    v2=dprecv(tid(1));
    y1=dprecv(tid(4));
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 5
    az=3;bz=0.7;cz=0.02;dz=0.5;ez=0.3;fz=0.04;gz=0.02;hz = 0.04;
    rz=-gz*z1*v2;
    sz=-hz*z2*y1;
    z1dot=az*z1-bz*z1*z2-cz*z1^2+rz;
    z2dot=-dz*z2+ez*z1*z2-fz*z2^2+sz;
    % Ende Modellparameter und -gleichungen System 5
    Xdot=[z1dot,z2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end
```

Hauptprogramm parallel DP-MPI (Message-Passing)

```
MPI_Init(5); % Initialisierung DP-MPI und Programminstanziierung

t0=0; % Startzeit der Simulation
h=0.01; % Schrittweite
steps=10000; % Anzahl Schritte
x0=ones(1,2); % Anfangszustände

x=rk4('rbs_equ_par_mpi',t0,h,steps,x0); % Simulationslauf

x100=x(end,:); % Simulationsergebnisse ablegen
x100=MPI_Gather(x100,2); % Simulationsergebnisse einsammeln

MPI_Finalize; % Schließen des DP-MPI-Subsystems

save('result_rbs_par_dp.mat','x100'); % Speicherung der Ergebnisse
```

Ableitungsfunktion parallel DP-MPI (Message-Passing)

```
function  Xdot=rbs_equ_par_mpi(t,X)

if MPI_Comm_rank==0 % wenn erster Prozessrang
    v1=X(1);v2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    w1=MPI_Recv(1);
    x1=MPI_Recv(2);
    y1=MPI_Recv(3);
    z1=MPI_Recv(4);
    MPI_Send(v2,1:4);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 1
    av=2;bv=0.5;cv=0.01;dv=0.2;ev=0.4;fv=0.02;gv=0.01;hv=0.02;...
        jv=0.01;kv=0.03;
    sv=v2*(gv*w1+hv*x1+jv*y1+kv*z1);
    v1dot=av*v1-bv*v1*v2-cv*v1^2;
    v2dot=-dv*v2+ev*v1*v2-fv*v2^2+sv;
    % Ende Modellparameter und -gleichungen System 1
    Xdot=[v1dot,v2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if MPI_Comm_rank==1 % wenn zweiter Prozessrang
    w1=X(1);w2=X(2); % Auslesen der Zustandsgrößen
```

```
    % Beginn Austausch der Zustandsgrößen
    MPI_Send(w1,0);
    v2=MPI_Recv(0);
    x2=MPI_Recv(2);
    MPI_Send(w1,2);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 2
    aw=1;bw=0.5;cw=0.02;dw=0.1;ew=0.4;fw=0.04;gw=0.02;hw=0.04;
    rw=w1*(-gw*v2+hw*x2);
    w1dot=aw*w1-bw*w1*w2-cw*w1^2+rw;
    w2dot=-dw*w2+ew*w1*w2-fw*w2^2;
    % Ende Modellparameter und -gleichungen System 2
    Xdot=[w1dot,w2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if MPI_Comm_rank==2 % wenn dritter Prozessrang
    x1=X(1);x2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    MPI_Send(x1,0);
    MPI_Send(x2,1);
    v2=MPI_Recv(0);
    w1=MPI_Recv(1);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 3
    ax=3;bx=0.9;cx=0.02;dx=0.2;ex=0.2;fx=0.04;gx=0.025;hx=0.1;
    rx=-gx*x1*v2;
    sx=-hx*x2*w1;
    x1dot=ax*x1-bx*x1*x2-cx*x1^2+rx;
    x2dot=-dx*x2+ex*x1*x2-fx*x2^2+sx;
    % Ende Modellparameter und -gleichungen System 3
    Xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if MPI_Comm_rank==3 % wenn vierter Prozessrang
    y1=X(1);y2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    MPI_Send(y1,0);
    v2=MPI_Recv(0);
    z2=MPI_Recv(4);
    MPI_Send(y1,4);
    % Ende Austausch der Zustandsgrößen
```

```
    % Beginn Modellparameter und -gleichungen System 4
    ay=1;by=0.8;cy=0.04;dy=0.2;ey=0.6;fy=0.07;gy=0.03;hy=0.025;
    ry=y1*(-gy*v2+hy*z2);
    y1dot=ay*y1-by*y1*y2-cy*y1^2+ry;
    y2dot=-dy*y2+ey*y1*y2-fy*y2^2;
    % Ende Modellparameter und -gleichungen System 4
    Xdot=[y1dot,y2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if MPI_Comm_rank==4 % wenn fünfter Prozessrang
    z1=X(1);z2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    MPI_Send(z1,0);
    MPI_Send(z2,3);
    v2=MPI_Recv(0);
    y1=MPI_Recv(3);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 5
    az=3;bz=0.7;cz=0.02;dz=0.5;ez=0.3;fz=0.04;gz=0.02;hz = 0.04;
    rz=-gz*z1*v2;
    sz=-hz*z2*y1;
    z1dot=az*z1-bz*z1*z2-cz*z1^2+rz;
    z2dot=-dz*z2+ez*z1*z2-fz*z2^2+sz;
    % Ende Modellparameter und -gleichungen System 5
    Xdot=[z1dot,z2dot]; % Ende Austausch der Zustandsgrößen
end
```

Hauptprogramm parallel DC-2.0 (Message-Passing)

```
function x100=rbs_par_dc(ident)

if nargin<1 % wenn in übergeordneter Instanz
    % Beginn Initialisierung Message-Passing-Programm
    jm=findResource('jobmanager','lookupurl','vprspc1');
    pj=createParallelJob(jm);
    createTask(pj,mfilename,1,{[]});
    set(pj,'minimumnumberofworkers',5,'maximumnumberofworkers',5,...
        'filedependencies',{mfilename,'rk4','rbs_equ_par_dc'});
    % Ende Initialisierung Message-Passing-Programm
    submit(pj); % paralleles Programm starten
    waitForState(pj,'finished'); % auf Programmende warten
    % Beginn Finalisierung Message-Passing-Programm
    oa=getAllOutputArguments(pj);
    destroy(pj);
    % Ende Finalisierung Message-Passing-Programm
    x100=oa{1}; % Simulationsergebnisse ablegen
    save('result_rbs_par_dc.mat','x100'); % Speicherung der Ergebnisse
    return; % Programmende in übergeordneter Instanz
end

t0=0; % Startzeit der Simulation
h=0.01; % Schrittweite
steps=10000; % Anzahl Schritte
x0=ones(1,2); % Anfangszustände

x=rk4('rbs_equ_par_dc',t0,h,steps,x0); % Simulationslauf

x100=x(end,:); % Simulationsergebnisse ablegen

% Beginn Simulationsergebnisse einsammeln (Allgather-Operation)
if labindex>1
    labSend(x100,1);
    x100=labReceive(1);
else
    for i=2:5
        x100=[x100,labReceive(i)];
    end
    for i=2:5
        labSend(x100,i);
    end
end
```

```
% Ende Simulationsergebnisse einsammeln (Allgather-Operation)
```

Ableitungsfunktion parallel DC-2.0 (Message-Passing)

```
function  Xdot=rbs_equ_par_dc(t,X)

if labindex==1 % wenn erster Prozessrang
    v1=X(1);v2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    w1=labReceive(2);
    x1=labReceive(3);
    y1=labReceive(4);
    z1=labReceive(5);
    labSend(v2,2:5);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 1
    av=2;bv=0.5;cv=0.01;dv=0.2;ev=0.4;fv=0.02;gv=0.01;hv=0.02;...
        jv=0.01;kv=0.03;
    sv=v2*(gv*w1+hv*x1+jv*y1+kv*z1);
    v1dot=av*v1-bv*v1*v2-cv*v1^2;
    v2dot=-dv*v2+ev*v1*v2-fv*v2^2+sv;
    % Ende Modellparameter und -gleichungen System 1
    Xdot=[v1dot,v2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if labindex==2 % wenn zweiter Prozessrang
    w1=X(1);w2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    labSend(w1,1);
    v2=labReceive(1);
    x2=labReceive(3);
    labSend(w1,3);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 2
    aw=1;bw=0.5;cw=0.02;dw=0.1;ew=0.4;fw=0.04;gw=0.02;hw=0.04;
    rw=w1*(-gw*v2+hw*x2);
    w1dot=aw*w1-bw*w1*w2-cw*w1^2+rw;
    w2dot=-dw*w2+ew*w1*w2-fw*w2^2;
    % Ende Modellparameter und -gleichungen System 2
    Xdot=[w1dot,w2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end
```

```
if labindex==3 % wenn dritter Prozessrang
    x1=X(1);x2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    labSend(x1,1);
    labSend(x2,2);
    v2=labReceive(1);
    w1=labReceive(2);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 3
    ax=3;bx=0.9;cx=0.02;dx=0.2;ex=0.2;fx=0.04;gx=0.025;hx=0.1;
    rx=-gx*x1*v2;
    sx=-hx*x2*w1;
    x1dot=ax*x1-bx*x1*x2-cx*x1^2+rx;
    x2dot=-dx*x2+ex*x1*x2-fx*x2^2+sx;
    % Ende Modellparameter und -gleichungen System 3
    Xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if labindex==4 % wenn vierter Prozessrang
    y1=X(1);y2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    labSend(y1,1);
    v2=labReceive(1);
    z2=labReceive(5);
    labSend(y1,5);
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 4
    ay=1;by=0.8;cy=0.04;dy=0.2;ey=0.6;fy=0.07;gy=0.03;hy=0.025;
    ry=y1*(-gy*v2+hy*z2);
    y1dot=ay*y1-by*y1*y2-cy*y1^2+ry;
    y2dot=-dy*y2+ey*y1*y2-fy*y2^2;
    % Ende Modellparameter und -gleichungen System 4
    Xdot=[y1dot,y2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if labindex==5 % wenn fünfter Prozessrang
    z1=X(1);z2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    labSend(z1,1);
    labSend(z2,4);
    v2=labReceive(1);
    y1=labReceive(4);
```

```
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 5
    az=3;bz=0.7;cz=0.02;dz=0.5;ez=0.3;fz=0.04;gz=0.02;hz = 0.04;
    rz=-gz*z1*v2;
    sz=-hz*z2*y1;
    z1dot=az*z1-bz*z1*z2-cz*z1^2+rz;
    z2dot=-dz*z2+ez*z1*z2-fz*z2^2+sz;
    % Ende Modellparameter und -gleichungen System 5
    Xdot=[z1dot,z2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end
```

Hauptprogramm parallel DP-MPI (Shared-Memory)

```
MPI_Init(5); % Initialisierung DP-MPI und Programminstanziierung
shmopen('v1','v2','w1','w2','x1','x2','y1','y2','z1',...
    'z2','x100'); % Deklaration gemeinsamer Variablen

t0=0; % Startzeit der Simulation
h=0.01; % Schrittweite
steps=10000; % Anzahl Schritte
x0=ones(1,2); % Anfangszustände

x=rk4('rbs_equ_par_mpi_shm',t0,h,steps,x0); % Simulationslauf

x100=x(end,:); % Simulationsergebnisse ablegen

shmgather('x100',x100,2); % Simulationsergebnisse einsammeln
x100=shmread('x100'); % Simulationsergebnisse auslesen

shmclose; % Entfernen gemeinsamer Variablen
MPI_Finalize; % Schließen des DP-MPI-Subsystems

save('result_rbs_par_dp.mat','x100'); % Speicherung der Ergebnisse
```

Ableitungsfunktion parallel DP-MPI (Shared-Memory)

```
function  Xdot=rbs_equ_par_mpi_shm(t,X)

if shmrank==0 % wenn erster Prozessrang
    v1=X(1);v2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    shmbarrier;
    shmwrite('v1',v1,'v2',v2);
    shmbarrier;
    [w1,x1,y1,z1]=shmread('w1','x1','y1','z1');
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 1
    av=2;bv=0.5;cv=0.01;dv=0.2;ev=0.4;fv=0.02;gv=0.01;hv=0.02;...
        jv=0.01;kv=0.03;
    sv=v2*(gv*w1+hv*x1+jv*y1+kv*z1);
    v1dot=av*v1-bv*v1*v2-cv*v1^2;
    v2dot=-dv*v2+ev*v1*v2-fv*v2^2+sv;
    % Ende Modellparameter und -gleichungen System 1
    Xdot=[v1dot,v2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

```
end

if shmrank==1 % wenn zweiter Prozessrang
    w1=X(1);w2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    shmbarrier;
    shmwrite('w1',w1,'w2',w2);
    shmbarrier;
    [v2,x2]=shmread('v2','x2');
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 2
    aw=1;bw=0.5;cw=0.02;dw=0.1;ew=0.4;fw=0.04;gw=0.02;hw=0.04;
    rw=w1*(-gw*v2+hw*x2);
    w1dot=aw*w1-bw*w1*w2-cw*w1^2+rw;
    w2dot=-dw*w2+ew*w1*w2-fw*w2^2;
    % Ende Modellparameter und -gleichungen System 2
    Xdot=[w1dot,w2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if shmrank==2 % wenn dritter Prozessrang
    x1=X(1);x2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    shmbarrier;
    shmwrite('x1',x1,'x2',x2);
    shmbarrier;
    [v2,w1]=shmread('v2','w1');
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 3
    ax=3;bx=0.9;cx=0.02;dx=0.2;ex=0.2;fx=0.04;gx=0.025;hx=0.1;
    rx=-gx*x1*v2;
    sx=-hx*x2*w1;
    x1dot=ax*x1-bx*x1*x2-cx*x1^2+rx;
    x2dot=-dx*x2+ex*x1*x2-fx*x2^2+sx;
    % Ende Modellparameter und -gleichungen System 3
    Xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if shmrank==3 % wenn vierter Prozessrang
    y1=X(1);y2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    shmbarrier;
    shmwrite('y1',y1,'y2',y2);
```

```
    shmbarrier;
    [v2,z2]=shmread('v2','z2');
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 4
    ay=1;by=0.8;cy=0.04;dy=0.2;ey=0.6;fy=0.07;gy=0.03;hy=0.025;
    ry=y1*(-gy*v2+hy*z2);
    y1dot=ay*y1-by*y1*y2-cy*y1^2+ry;
    y2dot=-dy*y2+ey*y1*y2-fy*y2^2;
    % Ende Modellparameter und -gleichungen System 4
    Xdot=[y1dot,y2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end

if shmrank==4 % wenn fünfter Prozessrang
    z1=X(1);z2=X(2); % Auslesen der Zustandsgrößen
    % Beginn Austausch der Zustandsgrößen
    shmbarrier;
    shmwrite('z1',z1,'z2',z2);
    shmbarrier;
    [v2,y1]=shmread('v2','y1');
    % Ende Austausch der Zustandsgrößen

    % Beginn Modellparameter und -gleichungen System 5
    az=3;bz=0.7;cz=0.02;dz=0.5;ez=0.3;fz=0.04;gz=0.02;hz = 0.04;
    rz=-gz*z1*v2;
    sz=-hz*z2*y1;
    z1dot=az*z1-bz*z1*z2-cz*z1^2+rz;
    z2dot=-dz*z2+ez*z1*z2-fz*z2^2+sz;
    % Ende Modellparameter und -gleichungen System 5
    Xdot=[z1dot,z2dot]; % Ablegen der abgeleiteten Zustandsgrößen
end
```

B.3 Numerisches Lösen einer partiellen Differentialgleichung

Hauptprogramm sequentiell

```
t0=0; % Startzeit der Simulation
h=0.005; % Schrittweite
steps=6000; % Anzahl Schritte
N=800; % Anzahl Ortsintervalle
x0=zeros(1,2*(N+1)); % Anfangszustände

x=rk4('pdgl_equ_seq',t0,h,steps,x0,N); % Simulationslauf

index=fix([N/10,N/4,N/2,N*3/4,N*9/10]); % Indizes der Ergebnisse
xindex=x(:,index); % Simulationsergebnisse ablegen

save('result_pdgl_seq.mat','xindex'); % Speicherung der Ergebnisse
```

Ableitungsfunktion sequentiell

```
function xdot=pdgl_equ_seq(t,x,N)

L=10;a=2;b=1;d=0.2;omega=1;k=L/N; % Modellparameter

% Beginn Auslesen der Zustandsgrößen
n=numel(x)/2;
x1=x(1:n);
x2=x(n+1:2*n);
% Ende Auslesen der Zustandsgrößen

LEFT=0; % Zustand der linken Grenze
RIGHT=b*exp(-d*t)*sin(omega*t); % Zustand der rechten Grenze

% Beginn Modellgleichungen
x1dot=x2;
x2dot=zeros(size(x2));
x2dot(1)=LEFT-2*x1(1)+x1(2);
x2dot(2:n-1)=x1(1:n-2)-2*x1(2:n-1)+x1(3:n);
x2dot(n)=x1(n-1)-2*x1(n)+RIGHT;
x2dot(:)=x2dot(:)/(k^2*a);
% Ende Modellgleichungen

xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

Hauptprogramm parallel DP-1.7 (Message-Passing)

```
t0=0; % Startzeit der Simulation
h=0.005; % Schrittweite
steps=6000; % Anzahl Schritte
N=800; % Anzahl Ortsintervalle

if dpparent==dpmyid % wenn übergeordnete Instanz
    hosts={'vprspc2';'vprspc3';'vprspc4';'vprdpc1';...
        'vprdpc1';'vprdpc2';'vprdpc2';'vprdpc3';...
        'vprdpc3';'vprdpc4';'vprdpc4'}; % weitere beteiligte Rechner
    tid=dpspawn(hosts,mfilename); % Instanziierung weiterer Programme
    tid(end+1)=dpmyid; % Identifikationsnummern zusammenfassen
    dpsend(tid,tid); % Identifikationsnummern senden
    dpscatter(1:N+1,tid); % Intervallindizes verteilen
end
tid=dprecv(dpparent); % Identifikationsnummern empfangen
Nrange=dprecv(dpparent); % Intervallindizes empfangen
x0=zeros(1,2*numel(Nrange)); % Anfangszustände

x=rk4('pdgl_equ_par_dp',t0,h,steps,x0,N,tid); % Simulationslauf

index=fix([N/10,N/4,N/2,N*3/4,N*9/10]); % Indizes der Ergebnisse
index=intersect(Nrange,index)-Nrange(1)+1; % Indextransformation
xindex=x(:,index); % Simulationsergebnisse ablegen

dpsend(xindex',dpparent); % Simulationsergebnisse senden
if dpparent==dpmyid % wenn übergeordnete Instanz
    xindex=dpgather(tid)'; % Simulationsergebnisse einsammeln
    dpexit; % Schließen des DP-Subsystems
    save('result_pdgl_par_dp.mat','xindex'); % Speicherung der Ergebnisse
else % wenn untergeordenete Instanz
    exit; % Instanz schließen
end
```

Ableitungsfunktion parallel DP-1.7 (Message-Passing)

```
function xdot=pdgl_equ_par_dp(t,x,N,tid)

L=10;a=2;b=1;d=0.2;omega=1;k=L/N; % Modellparameter

% Beginn Auslesen der Zustandsgrößen
n=numel(x)/2;
x1=x(1:n);
```

```
x2=x(n+1:2*n);
% Ende Auslesen der Zustandsgrößen

LEFT=0; % Zustand der linken Grenze
RIGHT=b*exp(-d*t)*sin(omega*t); % Zustand der rechten Grenze

% Beginn Austausch der Zustandsgrößen
POS=find(tid==dpmyid); % Prozessrang ermitteln
if POS~=1
    dpsend(x1(1),tid(POS-1)); % Senden an linken Nachbarn
end
if POS~=numel(tid)
    RIGHT=dprecv(tid(POS+1)); % Empfangen von rechtem Nachbarn
    dpsend(x1(end),tid(POS+1)); % Senden an rechten Nachbarn
end
if POS~=1
    LEFT=dprecv(tid(POS-1)); % Empfangen von linkem Nachbarn
end
% Ende Austausch der Zustandsgrößen

% Beginn Modellgleichungen
x1dot=x2;
x2dot=zeros(size(x2));
x2dot(1)=LEFT-2*x1(1)+x1(2);
x2dot(2:n-1)=x1(1:n-2)-2*x1(2:n-1)+x1(3:n);
x2dot(n)=x1(n-1)-2*x1(n)+RIGHT;
x2dot(:)=x2dot(:)/(k^2*a);
% Ende Modellgleichungen

xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

Hauptprogramm parallel DP-MPI (Message-Passing)

```
MPI_Init(12); % Initialisierung DP-MPI und Programminstanziierung

t0=0; % Startzeit der Simulation
h=0.005; % Schrittweite
steps=6000; % Anzahl Schritte
N=800; % Anzahl Ortsintervalle

Nrange=MPI_Scatter(1:N+1); % Intervallindizes verteilen
x0=zeros(1,2*numel(Nrange)); % Anfangszustände

x=rk4('pdgl_equ_par_mpi',t0,h,steps,x0,N); % Simulationslauf

index=fix([N/10,N/4,N/2,N*3/4,N*9/10]); % Indizes der Ergebnisse
index=intersect(Nrange,index)-Nrange(1)+1; % Indextransformation
xindex=x(:,index); % Simulationsergebnisse ablegen

xindex=MPI_Gather(xindex,2); % Simulationsergebnisse einsammeln

MPI_Finalize; % Schließen des DP-MPI-Subsystems

save('result_pdgl_par_mpi.mat','xindex'); % Speicherung der Ergebnisse
```

Ableitungsfunktion parallel DP-MPI (Message-Passing)

```
function xdot=pdgl_equ_par_mpi(t,x,N)

L=10;a=2;b=1;d=0.2;omega=1;k=L/N; % Modellparameter

% Beginn Auslesen der Zustandsgrößen
n=numel(x)/2;
x1=x(1:n);
x2=x(n+1:2*n);
% Ende Auslesen der Zustandsgrößen

LEFT=0; % Zustand der linken Grenze
RIGHT=b*exp(-d*t)*sin(omega*t); % Zustand der rechten Grenze

% Beginn Austausch der Zustandsgrößen
RANK=MPI_Comm_rank; % Prozessrang ermitteln
SIZE=MPI_Comm_size; % Anzahl beteiligter Prozesse ermitteln
if RANK~=0
    MPI_Send(x1(1),RANK-1); % Senden an linken Nachbarn
```

```
end
if RANK~=SIZE-1
    RIGHT=MPI_Recv(RANK+1); % Empfangen von rechtem Nachbarn
    MPI_Send(x1(end),RANK+1); % Senden an rechten Nachbarn
end
if RANK~=0
    LEFT=MPI_Recv(RANK-1); % Empfangen von linkem Nachbarn
end
% Ende Austausch der Zustandsgrößen

% Beginn Modellgleichungen
x1dot=x2;
x2dot=zeros(size(x2));
x2dot(1)=LEFT-2*x1(1)+x1(2);
x2dot(2:n-1)=x1(1:n-2)-2*x1(2:n-1)+x1(3:n);
x2dot(n)=x1(n-1)-2*x1(n)+RIGHT;
x2dot(:)=x2dot(:)/(k^2*a);
% Ende Modellgleichungen

xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

Hauptprogramm parallel DC-2.0 (Message-Passing)

```
function xindex=pdgl_par_dc(ident)

if nargin<1 % wenn in übergeordneter Instanz
    % Beginn Initialisierung Message-Passing-Programm
    jm=findResource('jobmanager','lookupurl','vprspc1');
    pj=createParallelJob(jm);
    createTask(pj,mfilename,1,{[]});
    set(pj,'filedependencies',{mfilename,'rk4','pdgl_equ_par_dc'});
    % Ende Initialisierung Message-Passing-Programm
    submit(pj); % paralleles Programm starten
    waitForState(pj,'finished'); % auf Programmende warten
    % Beginn Finalisierung Message-Passing-Programm
    oa=getAllOutputArguments(pj);
    destroy(pj);
    % Ende Finalisierung Message-Passing-Programm
    xindex=oa{1}; % Simulationsergebnisse ablegen
    save('result_pdgl_par_dc.mat','xindex'); % Speicherung der Ergebnisse
    return; % Programmende in übergeordneter Instanz
end

t0=0; % Startzeit der Simulation
h=0.005; % Schrittweite
steps=6000; % Anzahl Schritte
N=800; % Anzahl Ortsintervalle

% Beginn Verteilung der Intervallindizes (Scatter-Operation)
pos=1;
stride=floor((N+1)/numlabs)+1;
for i=1:numlabs
    if i-1==mod(N+1,numlabs);
        stride=stride-1;
    end
    Nrange=pos:pos+stride-1;
    if i==labindex
        break;
    end
    pos=pos+stride;
end
% Ende Verteilung der Intervallindizes (Scatter-Operation)

x0=zeros(1,2*numel(Nrange)); % Anfangszustände
```

```
x=rk4('pdgl_equ_par_dc',t0,h,steps,x0,N); % Simulationslauf

index=fix([N/10,N/4,N/2,N*3/4,N*9/10]); % Indizes der Ergebnisse
index=intersect(Nrange,index)-Nrange(1)+1; % Indextransformation
xindex=x(:,index); % Simulationsergebnisse ablegen

% Beginn Simulationsergebnisse einsammeln (Allgather-Operation)
if labindex~=1
    labSend(xindex,1);
    xindex=labReceive(1);
else
    for i=2:numlabs
        xindex=[xindex,labReceive(i)];
    end
    for i=2:numlabs
        labSend(xindex,i);
    end
end
% Ende Simulationsergebnisse einsammeln (Allgather-Operation)
```

Ableitungsfunktion parallel DC-2.0 (Message-Passing)

```
function xdot=pdgl_equ_par_dc(t,x,N)

L=10;a=2;b=1;d=0.2;omega=1;k=L/N; % Modellparameter

% Beginn Auslesen der Zustandsgrößen
n=numel(x)/2;
x1=x(1:n);
x2=x(n+1:2*n);
% Ende Auslesen der Zustandsgrößen

LEFT=0; % Zustand der linken Grenze
RIGHT=b*exp(-d*t)*sin(omega*t); % Zustand der rechten Grenze

% Beginn Austausch der Zustandsgrößen
if labindex~=1
    labSend(x1(1),labindex-1); % Senden an linken Nachbarn
end
if labindex~=numlabs
    RIGHT=labReceive(labindex+1); % Empfangen von rechtem Nachbarn
    labSend(x1(end),labindex+1); % Senden an rechten Nachbarn
end
if labindex~=1
```

```
    LEFT=labReceive(labindex-1); % Empfangen von linkem Nachbarn
end
% Ende Austausch der Zustandsgrößen

% Beginn Modellgleichungen
x1dot=x2;
x2dot=zeros(size(x2));
x2dot(1)=LEFT-2*x1(1)+x1(2);
x2dot(2:n-1)=x1(1:n-2)-2*x1(2:n-1)+x1(3:n);
x2dot(n)=x1(n-1)-2*x1(n)+RIGHT;
x2dot(:)=x2dot(:)/(k^2*a);
% Ende Modellgleichungen

xdot=[x1dot,x2dot]; % Ablegen der abgeleiteten Zustandsgrößen
```

B.4 Strömungssimulation mittels Lattice-Boltzmann-Verfahren

Hauptprogramm sequentiell

```
% begin experiment parameter settings
nx=257; % number of grid points in x direction
ny=nx; % number of grid points in y direction

iterations=2000; % number of iterations

geometry=ones(ny,nx); % wall cells (geometry==1)
geometry(2:ny-1,2:nx-1)=0; % fluid cells (geometry==0)
geometry(1,:)=2; % driving cells on the top boundary (geometry==2)

rho_0=1; % initial density
u_0=0.1; % driving velocity on the top boundary

Re=1000; % Reynolds number
viscosity=(ny-1)*u_0/Re; % kinematic viscosity (0.005<=viscosity<=0.2)
tau=(6*viscosity+1)/2; % relaxation time
% end experiment parameter settings

C=1;E=2;S=3;W=4;N=5;NE=6;SE=7;SW=8;NW=9; % help variables for directions
% directions are indiced as follows:
% 9 5 6
% 4 1 2
% 8 3 7

f=zeros(nx*ny,9); % distribution function values of each cell
feq=zeros(nx*ny,9); % equilibrium disribution function value
rho=zeros(nx*ny,1); % macroscopic density
ux=zeros(nx*ny,1); % macroscopic velocity in x direction
uy=zeros(nx*ny,1); % macroscopic velocity in y direction
usqr=zeros(nx*ny,1); % helper variable

% begin set initial distribution function values
f(:,C)=rho_0*4/9;
f(:,[E S W N])=rho_0/9;
f(:,[NE SE SW NW])=rho_0/36;
% end set initial distribution function values

FL=find(geometry==0); % create indices of all fluid cells
```

```
WALL=find(geometry==1); % create indices of all wall cells
DR=find(geometry==2); % create indices of all driving cells

for i=1:iterations

    % begin collision step ------------------------------------------

    % begin distribution function value transformation
    rho(:)=sum(f,2); % macroscopic density
    ux(:)=(f(:,E)-f(:,W)+f(:,NE)+f(:,SE)-f(:,SW)-f(:,NW))./rho; % x vel.
    uy(:)=(f(:,N)-f(:,S)+f(:,NE)+f(:,NW)-f(:,SE)-f(:,SW))./rho; % y vel.
    % end distribution function value transformation

    ux(DR)=u_0; % set x velocity for driving cells
    uy(DR)=0; % set y velocity for driving cells
    usqr(:)=ux.*ux+uy.*uy; % calculate helper variable value

    % begin equilibrium distribution function value calculation
    feq(:,C)=(4/9)*rho.*(1-1.5*usqr);
    feq(:,E)=(1/9)*rho.*(1+3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,S)=(1/9)*rho.*(1-3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,W)=(1/9)*rho.*(1-3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,N)=(1/9)*rho.*(1+3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,NE)=(1/36)*rho.*(1+3*(ux+uy)+4.5*(ux+uy).^2-1.5*usqr);
    feq(:,SE)=(1/36)*rho.*(1+3*(ux-uy)+4.5*(ux-uy).^2-1.5*usqr);
    feq(:,SW)=(1/36)*rho.*(1+3*(-ux-uy)+4.5*(-ux-uy).^2-1.5*usqr);
    feq(:,NW)=(1/36)*rho.*(1+3*(-ux+uy)+4.5*(-ux+uy).^2-1.5*usqr);
    % end equilibrium distribution function value calculation

    % begin wall cell f calculation (bounce back)
    f(WALL,[C E S W N NE SE SW NW])=f(WALL,[C W N E S SW NW NE SE]);
    % end wall cell f calculation (bounce back)

    % begin driving cell f calculation
    f(DR,:)=feq(DR,:); % distribution function value = equilibrium value
    % end driving cell f calculation

    % begin fluid cell f calculation
    f(FL,:)=f(FL,:)*(1-1/tau)+feq(FL,:)/tau;
    % end fluid cell f calculation

    % end collision step --------------------------------------------

    % begin propagation step ----------------------------------------
```

```
    f=reshape(f,[ny,nx,9]); % transform f for easy propagation

    % begin particle propagation
    f(:,2:nx,E)=f(:,1:nx-1,E);
    f(2:ny,:,S)=f(1:ny-1,:,S);
    f(:,1:nx-1,W)=f(:,2:nx,W);
    f(1:ny-1,:,N)=f(2:ny,:,N);
    f(1:ny-1,2:nx,NE)=f(2:ny,1:nx-1,NE);
    f(2:ny,2:nx,SE)=f(1:ny-1,1:nx-1,SE);
    f(2:ny,1:nx-1,SW)=f(1:ny-1,2:nx,SW);
    f(1:ny-1,1:nx-1,NW)=f(2:ny,2:nx,NW);
    % end particle propagation

    f=reshape(f,[nx*ny,9]); % re-transform f for next iteration step
    % end propagation step ------------------------------------------
end

% begin save
u=sqrt(ux.^2+uy.^2)/u_0; % calculate relative macroscopic velocity magn.
u=reshape(u,ny,nx); % reshape u to 2D for easy plotting
save('result_lbm_seq.mat','u'); % save u
% end save
```

Hauptprogramm parallel DP-1.7 (Message-Passing)

```
% begin experiment parameter settings
nx=257; % number of grid points in x direction
ny=nx; % number of grid points in y direction

iterations=2000; % number of iterations

geometry=ones(ny,nx); % wall cells (geometry==1)
geometry(2:ny-1,2:nx-1)=0; % fluid cells (geometry==0)
geometry(1,:)=2; % driving cells on the top boundary (geometry==2)

rho_0=1; % initial density
u_0=0.1; % driving velocity on the top boundary

Re=1000; % Reynolds number
viscosity=(ny-1)*u_0/Re; % kinematic viscosity (0.005<=viscosity<=0.2)
tau=(6*viscosity+1)/2; % relaxation time
% end experiment parameter settings

C=1;E=2;S=3;W=4;N=5;NE=6;SE=7;SW=8;NW=9; % help variables for directions
% directions are indiced as follows:
% 9 5 6
% 4 1 2
% 8 3 7

if dpparent==dpmyid % if in primary instance
    hosts={'vprspc2';'vprspc3';'vprspc4';'vprdpc1';...
        'vprdpc1';'vprdpc2';'vprdpc2';'vprdpc3';...
        'vprdpc3';'vprdpc4';'vprdpc4'}; % involved hosts
    tid=dpspawn(hosts,mfilename); % spawn secondary instances, store tids
    tid(end+1)=dpmyid; % add primary instance tid to all tids
    dpscatter(geometry,tid); % scatter geometry in y direction
    dpsend(tid,tid); % send all tids to all instances
end
geometry=dprecv(dpparent); % receive partial geometry
tid=dprecv(dpparent); % receive tid of all instances
ny=size(geometry,1); % gain new ny from geometry
POS=find(tid==dpmyid); % store position of domain part for easy access
SIZE=numel(tid); % store number of instances for easy access

f=zeros(nx*ny,9); % distribution function values of each cell
feq=zeros(nx*ny,9); % equilibrium disribution function value
rho=zeros(nx*ny,1); % macroscopic density
```

```
ux=zeros(nx*ny,1); % macroscopic velocity in x direction
uy=zeros(nx*ny,1); % macroscopic velocity in y direction
usqr=zeros(nx*ny,1); % helper variable

% begin set initial distribution function values
f(:,C)=rho_0*4/9;
f(:,[E S W N])=rho_0/9;
f(:,[NE SE SW NW])=rho_0/36;
% end set initial distribution function values

FL=find(geometry==0); % create indices of all fluid cells
WALL=find(geometry==1); % create indices of all wall cells
DR=find(geometry==2); % create indices of all driving cells

for i=1:iterations

    % begin collision step ------------------------------------------

    % begin distribution function value transformation
    rho(:)=sum(f,2); % macroscopic density
    ux(:)=(f(:,E)-f(:,W)+f(:,NE)+f(:,SE)-f(:,SW)-f(:,NW))./rho; % x vel.
    uy(:)=(f(:,N)-f(:,S)+f(:,NE)+f(:,NW)-f(:,SE)-f(:,SW))./rho; % y vel.
    % end distribution function value transformation

    ux(DR)=u_0; % set x velocity for driving cells
    uy(DR)=0; % set y velocity for driving cells
    usqr(:)=ux.*ux+uy.*uy; % calculate helper variable value

    % begin equilibrium distribution function value calculation
    feq(:,C)=(4/9)*rho.*(1-1.5*usqr);
    feq(:,E)=(1/9)*rho.*(1+3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,S)=(1/9)*rho.*(1-3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,W)=(1/9)*rho.*(1-3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,N)=(1/9)*rho.*(1+3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,NE)=(1/36)*rho.*(1+3*(ux+uy)+4.5*(ux+uy).^2-1.5*usqr);
    feq(:,SE)=(1/36)*rho.*(1+3*(ux-uy)+4.5*(ux-uy).^2-1.5*usqr);
    feq(:,SW)=(1/36)*rho.*(1+3*(-ux-uy)+4.5*(-ux-uy).^2-1.5*usqr);
    feq(:,NW)=(1/36)*rho.*(1+3*(-ux+uy)+4.5*(-ux+uy).^2-1.5*usqr);
    % end equilibrium distribution function value calculation

    % begin wall cell f calculation (bounce back)
    f(WALL,[C E S W N NE SE SW NW])=f(WALL,[C W N E S SW NW NE SE]);
    % end wall cell f calculation (bounce back)
```

```
% begin driving cell f calculation
f(DR,:)=feq(DR,:); % distribution function value = equilibrium value
% end driving cell f calculation

% begin fluid cell f calculation
f(FL,:)=f(FL,:)*(1-1/tau)+feq(FL,:)/tau;
% end fluid cell f calculation

% end collision step ------------------------------------------------

% begin propagation step --------------------------------------------
f=reshape(f,[ny,nx,9]); % transform f for easy propagation

% begin border exchange
northborder=[];
southborder=[];
if POS<SIZE
    dpsend(f(ny,:,:),tid(POS+1)); % send downwards
end
if POS>1
    northborder=dprecv(tid(POS-1)); % receive from upwards
    dpsend(f(1,:,:),tid(POS-1)); % send upwards
end
if POS<SIZE
    southborder=dprecv(tid(POS+1)); % receive from downwards
end
% end border exchange

% begin particle propagation
f(:,2:nx,E)=f(:,1:nx-1,E);
f(2:ny,:,S)=f(1:ny-1,:,S);
f(:,1:nx-1,W)=f(:,2:nx,W);
f(1:ny-1,:,N)=f(2:ny,:,N);
f(1:ny-1,2:nx,NE)=f(2:ny,1:nx-1,NE);
f(2:ny,2:nx,SE)=f(1:ny-1,1:nx-1,SE);
f(2:ny,1:nx-1,SW)=f(1:ny-1,2:nx,SW);
f(1:ny-1,1:nx-1,NW)=f(2:ny,2:nx,NW);

if ~isempty(northborder) % stream from north border
    f(1,:,S)=northborder(1,:,S);
    f(1,2:nx,SE)=northborder(1,1:nx-1,SE);
    f(1,1:nx-1,SW)=northborder(1,2:nx,SW);
end
if ~isempty(southborder) % stream from south border
```

```
        f(ny,:,N)=southborder(1,:,N);
        f(ny,2:nx,NE)=southborder(1,1:nx-1,NE);
        f(ny,1:nx-1,NW)=southborder(1,2:nx,NW);
    end
    % end particle propagation

    f=reshape(f,[nx*ny,9]); % re-transform f for next iteration step
    % end propagation step ------------------------------------------
end

% begin save
u=sqrt(ux.^2+uy.^2)/u_0; % calculate relative macroscopic velocity magn.
u=reshape(u,ny,nx); % reshape u to 2D for easy plotting
dpsend(u,dpparent); % send partial result to primary instance
if dpparent==dpmyid
    u=dpgather(tid); % collect partial results
    dpexit; % disconnect primary instance from PVM
    save('result_lbm_par_dp.mat','u'); % save u
else
    exit; % close secondary instances
end
% end save
```

Hauptprogramm parallel DP-MPI (Message-Passing)

```
MPI_Init(12); % connect instance to MPI and startup secondary instances

% begin experiment parameter settings
nx=257; % number of grid points in x direction
ny=nx; % number of grid points in y direction

iterations=2000; % number of iterations

geometry=ones(ny,nx); % wall cells (geometry==1)
geometry(2:ny-1,2:nx-1)=0; % fluid cells (geometry==0)
geometry(1,:)=2; % driving cells on the top boundary (geometry==2)

rho_0=1; % initial density
u_0=0.1; % driving velocity on the top boundary

Re=1000; % Reynolds number
viscosity=(ny-1)*u_0/Re; % kinematic viscosity (0.005<=viscosity<=0.2)
tau=(6*viscosity+1)/2; % relaxation time
% end experiment parameter settings

C=1;E=2;S=3;W=4;N=5;NE=6;SE=7;SW=8;NW=9; % help variables for directions
% directions are indiced as follows:
% 9 5 6
% 4 1 2
% 8 3 7

geometry=MPI_Scatter(geometry); % scatter geometry in y direction
ny=size(geometry,1); % gain new ny from geometry
RANK=MPI_Comm_rank; % store communicator rank for easy access
SIZE=MPI_Comm_size; % store communicator size for easy access

f=zeros(nx*ny,9); % distribution function values of each cell
feq=zeros(nx*ny,9); % equilibrium disribution function value
rho=zeros(nx*ny,1); % macroscopic density
ux=zeros(nx*ny,1); % macroscopic velocity in x direction
uy=zeros(nx*ny,1); % macroscopic velocity in y direction
usqr=zeros(nx*ny,1); % helper variable

% begin set initial distribution function values
f(:,C)=rho_0*4/9;
f(:,[E S W N])=rho_0/9;
f(:,[NE SE SW NW])=rho_0/36;
```

```
% end set initial distribution function values

FL=find(geometry==0); % create indices of all fluid cells
WALL=find(geometry==1); % create indices of all wall cells
DR=find(geometry==2); % create indices of all driving cells

for i=1:iterations

    % begin collision step ------------------------------------------

    % begin distribution function value transformation
    rho(:)=sum(f,2); % macroscopic density
    ux(:)=(f(:,E)-f(:,W)+f(:,NE)+f(:,SE)-f(:,SW)-f(:,NW))./rho; % x vel.
    uy(:)=(f(:,N)-f(:,S)+f(:,NE)+f(:,NW)-f(:,SE)-f(:,SW))./rho; % y vel.
    % end distribution function value transformation

    ux(DR)=u_0; % set x velocity for driving cells
    uy(DR)=0; % set y velocity for driving cells
    usqr(:)=ux.*ux+uy.*uy; % calculate helper variable value

    % begin equilibrium distribution function value calculation
    feq(:,C)=(4/9)*rho.*(1-1.5*usqr);
    feq(:,E)=(1/9)*rho.*(1+3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,S)=(1/9)*rho.*(1-3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,W)=(1/9)*rho.*(1-3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,N)=(1/9)*rho.*(1+3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,NE)=(1/36)*rho.*(1+3*(ux+uy)+4.5*(ux+uy).^2-1.5*usqr);
    feq(:,SE)=(1/36)*rho.*(1+3*(ux-uy)+4.5*(ux-uy).^2-1.5*usqr);
    feq(:,SW)=(1/36)*rho.*(1+3*(-ux-uy)+4.5*(-ux-uy).^2-1.5*usqr);
    feq(:,NW)=(1/36)*rho.*(1+3*(-ux+uy)+4.5*(-ux+uy).^2-1.5*usqr);
    % end equilibrium distribution function value calculation

    % begin wall cell f calculation (bounce back)
    f(WALL,[C E S W N NE SE SW NW])=f(WALL,[C W N E S SW NW NE SE]);
    % end wall cell f calculation (bounce back)

    % begin driving cell f calculation
    f(DR,:)=feq(DR,:); % distribution function value = equilibrium value
    % end driving cell f calculation

    % begin fluid cell f calculation
    f(FL,:)=f(FL,:)*(1-1/tau)+feq(FL,:)/tau;
    % end fluid cell f calculation
```

```
% end collision step ------------------------------------------

% begin propagation step ---------------------------------------
f=reshape(f,[ny,nx,9]); % transform f for easy propagation

% begin border exchange
northborder=[];
southborder=[];
if RANK<SIZE-1
    MPI_Send(f(ny,:,:),RANK+1); % send downwards
end
if RANK>0
    northborder=MPI_Recv(RANK-1); % receive from upwards
    MPI_Send(f(1,:,:),RANK-1); % send upwards
end
if RANK<SIZE-1
    southborder=MPI_Recv(RANK+1); % receive from downwards
end
% end border exchange

% begin particle propagation
f(:,2:nx,E)=f(:,1:nx-1,E);
f(2:ny,:,S)=f(1:ny-1,:,S);
f(:,1:nx-1,W)=f(:,2:nx,W);
f(1:ny-1,:,N)=f(2:ny,:,N);
f(1:ny-1,2:nx,NE)=f(2:ny,1:nx-1,NE);
f(2:ny,2:nx,SE)=f(1:ny-1,1:nx-1,SE);
f(2:ny,1:nx-1,SW)=f(1:ny-1,2:nx,SW);
f(1:ny-1,1:nx-1,NW)=f(2:ny,2:nx,NW);

if ~isempty(northborder) % stream from north border
    f(1,:,S)=northborder(1,:,S);
    f(1,2:nx,SE)=northborder(1,1:nx-1,SE);
    f(1,1:nx-1,SW)=northborder(1,2:nx,SW);
end
if ~isempty(southborder) % stream from south border
    f(ny,:,N)=southborder(1,:,N);
    f(ny,2:nx,NE)=southborder(1,1:nx-1,NE);
    f(ny,1:nx-1,NW)=southborder(1,2:nx,NW);
end
% end particle propagation

f=reshape(f,[nx*ny,9]); % re-transform f for next iteration step
% end propagation step -----------------------------------------
```

```
end

% begin save
u=sqrt(ux.^2+uy.^2)/u_0; % calculate relative macroscopic velocity magn.
u=reshape(u,ny,nx); % reshape u to 2D for easy plotting
u=MPI_Gather(u); % collect partial results
MPI_Finalize; % close secondary instances, disconnect from MPI
save('result_lbm_par_mpi.mat','u'); % save u
% end save
```

Hauptprogramm parallel DC-2.0 (Message-Passing)

```
function u=lbm_par_dc(ident)

if nargin<1 % if in primary instance
    % begin parallel program initialization
    jm=findResource('jobmanager','lookupurl','vprspc1');
    pj=createParallelJob(jm);
    createTask(pj,mfilename,1,{[]});
    set(pj,'filedependencies',{mfilename});
    % end parallel program initialization
    submit(pj); % start parallel program
    waitForState(pj,'finished'); % wait until program is finished
    % begin parallel program finalization
    oa=getAllOutputArguments(pj);
    destroy(pj);
    % end parallel program finalization
    u=oa{1}; % extract u
    save('result_lbm_par_dc.mat','u'); % save u
    return; % terminate primary instance program
end

% begin experiment parameter settings
nx=257; % number of grid points in x direction
ny=nx; % number of grid points in y direction

iterations=2000; % number of iterations

geometry=ones(ny,nx); % wall cells (geometry==1)
geometry(2:ny-1,2:nx-1)=0; % fluid cells (geometry==0)
geometry(1,:)=2; % driving cells on the top boundary (geometry==2)

rho_0=1; % initial density
u_0=0.1; % driving velocity on the top boundary

Re=1000; % Reynolds number
viscosity=(ny-1)*u_0/Re; % kinematic viscosity (0.005<=viscosity<=0.2)
tau=(6*viscosity+1)/2; % relaxation time
% end experiment parameter settings

C=1;E=2;S=3;W=4;N=5;NE=6;SE=7;SW=8;NW=9; % help variables for directions
% directions are indiced as follows:
% 9 5 6
% 4 1 2
```

```
% 8 3 7

% begin scatter geometry in y direction
pos=1;
stride=floor(ny/numlabs)+1;
for i=1:numlabs
    if i-1==mod(ny,numlabs);
        stride=stride-1;
    end
    range=pos:pos+stride-1;
    if i==labindex
        break;
    end
    pos=pos+stride;
end
geometry=geometry(range,:);
% end scatter geometry in y direction
ny=size(geometry,1); % gain new ny from geometry

f=zeros(nx*ny,9); % distribution function values of each cell
feq=zeros(nx*ny,9); % equilibrium disribution function value
rho=zeros(nx*ny,1); % macroscopic density
ux=zeros(nx*ny,1); % macroscopic velocity in x direction
uy=zeros(nx*ny,1); % macroscopic velocity in y direction
usqr=zeros(nx*ny,1); % helper variable

% begin set initial distribution function values
f(:,C)=rho_0*4/9;
f(:,[E S W N])=rho_0/9;
f(:,[NE SE SW NW])=rho_0/36;
% end set initial distribution function values

FL=find(geometry==0); % create indices of all fluid cells
WALL=find(geometry==1); % create indices of all wall cells
DR=find(geometry==2); % create indices of all driving cells

for i=1:iterations

    % begin collision step ----------------------------------------

    % begin distribution function value transformation
    rho(:)=sum(f,2); % macroscopic density
    ux(:)=(f(:,E)-f(:,W)+f(:,NE)+f(:,SE)-f(:,SW)-f(:,NW))./rho; % x vel.
    uy(:)=(f(:,N)-f(:,S)+f(:,NE)+f(:,NW)-f(:,SE)-f(:,SW))./rho; % y vel.
```

```
% end distribution function value transformation

ux(DR)=u_0; % set x velocity for driving cells
uy(DR)=0; % set y velocity for driving cells
usqr(:)=ux.*ux+uy.*uy; % calculate helper variable value

% begin equilibrium distribution function value calculation
feq(:,C)=(4/9)*rho.*(1-1.5*usqr);
feq(:,E)=(1/9)*rho.*(1+3*ux+4.5*ux.^2-1.5*usqr);
feq(:,S)=(1/9)*rho.*(1-3*uy+4.5*uy.^2-1.5*usqr);
feq(:,W)=(1/9)*rho.*(1-3*ux+4.5*ux.^2-1.5*usqr);
feq(:,N)=(1/9)*rho.*(1+3*uy+4.5*uy.^2-1.5*usqr);
feq(:,NE)=(1/36)*rho.*(1+3*(ux+uy)+4.5*(ux+uy).^2-1.5*usqr);
feq(:,SE)=(1/36)*rho.*(1+3*(ux-uy)+4.5*(ux-uy).^2-1.5*usqr);
feq(:,SW)=(1/36)*rho.*(1+3*(-ux-uy)+4.5*(-ux-uy).^2-1.5*usqr);
feq(:,NW)=(1/36)*rho.*(1+3*(-ux+uy)+4.5*(-ux+uy).^2-1.5*usqr);
% end equilibrium distribution function value calculation

% begin wall cell f calculation (bounce back)
f(WALL,[C E S W N NE SE SW NW])=f(WALL,[C W N E S SW NW NE SE]);
% end wall cell f calculation (bounce back)

% begin driving cell f calculation
f(DR,:)=feq(DR,:); % distribution function value = equilibrium value
% end driving cell f calculation

% begin fluid cell f calculation
f(FL,:)=f(FL,:)*(1-1/tau)+feq(FL,:)/tau;
% end fluid cell f calculation

% end collision step ------------------------------------------------

% begin propagation step --------------------------------------------
f=reshape(f,[ny,nx,9]); % transform f for easy propagation

% begin border exchange
northborder=[];
southborder=[];
if labindex<numlabs
    labSend(f(ny,:,:),labindex+1); % send downwards
end
if labindex>1
    northborder=labReceive(labindex-1); % receive from upwards
    labSend(f(1,:,:),labindex-1); % send upwards
```

```
    end
    if labindex<numlabs
        southborder=labReceive(labindex+1); % receive from downwards
    end
    % end border exchange

    % begin particle propagation
    f(:,2:nx,E)=f(:,1:nx-1,E);
    f(2:ny,:,S)=f(1:ny-1,:,S);
    f(:,1:nx-1,W)=f(:,2:nx,W);
    f(1:ny-1,:,N)=f(2:ny,:,N);
    f(1:ny-1,2:nx,NE)=f(2:ny,1:nx-1,NE);
    f(2:ny,2:nx,SE)=f(1:ny-1,1:nx-1,SE);
    f(2:ny,1:nx-1,SW)=f(1:ny-1,2:nx,SW);
    f(1:ny-1,1:nx-1,NW)=f(2:ny,2:nx,NW);

    if ~isempty(northborder) % stream from north border
        f(1,:,S)=northborder(1,:,S);
        f(1,2:nx,SE)=northborder(1,1:nx-1,SE);
        f(1,1:nx-1,SW)=northborder(1,2:nx,SW);
    end
    if ~isempty(southborder) % stream from south border
        f(ny,:,N)=southborder(1,:,N);
        f(ny,2:nx,NE)=southborder(1,1:nx-1,NE);
        f(ny,1:nx-1,NW)=southborder(1,2:nx,NW);
    end
    % end particle propagation

    f=reshape(f,[nx*ny,9]); % re-transform f for next iteration step
    % end propagation step ------------------------------------------
end

% begin save
u=sqrt(ux.^2+uy.^2)/u_0; % calculate relative macroscopic velocity magn.
u=reshape(u,ny,nx); % reshape u to 2D for easy plotting
% begin collect partial results (allgather)
if labindex~=1
    labSend(u,1);
    u=labReceive(1);
else
    for i=2:numlabs
        u=[u;labReceive(i)];
    end
    for i=2:numlabs
```

```
        labSend(u,i);
    end
end
% end collect partial results (allgather)
% end save
```

About the author ...

René Fink studierte Elektrotechnik mit der Vertiefungsrichtung Daten- und Informationstechnik an der Hochschule Wismar. Während seines praktischen Studiensemesters sowie seiner Diplomarbeit beschäftigte er sich bereits mit der Parallelverarbeitung in wissenschaftlich-technischen Berechnungsumgebungen, wie z.B. Matlab. Im Anschluss an sein Studium bearbeitete er an der Hochschule Wismar in Kooperation mit der Universität Rostock ein Promotions-projekt zur Untersuchung von Parallelverarbeitungsansätzen in wissenschaftlich-technischen Berechnungsumgebungen. Die Ergebnisse dieser Arbeit werden in diesem Band präsentiert. Momentan ist der Autor als Entwicklungsingenieur bei der IAV GmbH in Wismar tätig.

About this volume ...

Die Arbeit befasst sich mit der SCE-basierten Parallelverarbeitung. Als SCE wird dabei ein Softwaresystem zur interaktiven Entwicklung, Ausführung und Auswertung wissenschaftlich-technischer Berechnungsprogramme (wie z.B. Matlab, Scilab oder Octave) bezeichnet. Der aktuelle Entwicklungsstand der SCE-basierten Parallelverarbeitung wird wiedergegeben und eine neue Klassifikation für dieses Gebiet vorgeschlagen. Es wird gezeigt, dass aus der neuen Klassifikation insbesondere der Multi-SCE-Ansatz für ingenieurtechnische Bereiche interessant ist. Verfügbare Multi-SCE-Systeme werden anhand qualitativer und quantitativer Merkmale verglichen. Darüber hinaus erfolgt die Präsentation weiterentwickelter Multi-SCE-Systeme, die zum Teil explizit auf industrielle Anwendungsgebiete fokussieren. Eine anhand verschiedener ingenieurtechnischer Applikationen durchgeführte anwendungsbezogene Untersuchung zeigt beispielhaft Parallelisierungsaufwand und Laufzeitgewinn bei verschiedenen Anwendungstypen und Multi-SCE-Systemen.

Über diese Reihe …

Die Bände der ASIM - Reihe ***Fortschrittsberichte Simulation*** zeigen neueste Lösungsansätze, Methoden und Anwendungen der Simulation in Theorie und Praxis. Die Reihe umspannt Grundlagen und Anwendung der Simulation in einem immer breiter werdenden Spektrum, z. B. Ingenieurwissenschaften, Naturwissenschaften, Medizin, Ökonomie, Ökologie und Soziologie.
Fortschrittsberichte Simulation konzentrieren sich auf Monographien mit speziellem Charakter, wie z. B. Dissertationen und Habilitationen, Berichte zu ASIM-Workshops mit referierten Beiträgen, Berichte zu Forschungsprojekten, Handbücher zu Simulationswerkzeugen, Benchmarks, und ähnliches.
ASIM - Arbeitsgemeinschaft **SIM**ulation - die deutschsprachige Simulationsvereinigung, zugleich Fachausschuss der GI (Gesellschaft für Informatik), hat diese Reihe im ARGESIM / ASIM - Verlag als Ergänzung und Nachfolge zur ASIM- Reihe ***Fortschritte in der Simulationstechnik - Frontiers in Simulation*** ins Leben gerufen, um ein rasches und kostengünstiges Publikationsmedium für neue Entwicklungen in der Simulationstechnik anbieten zu können.