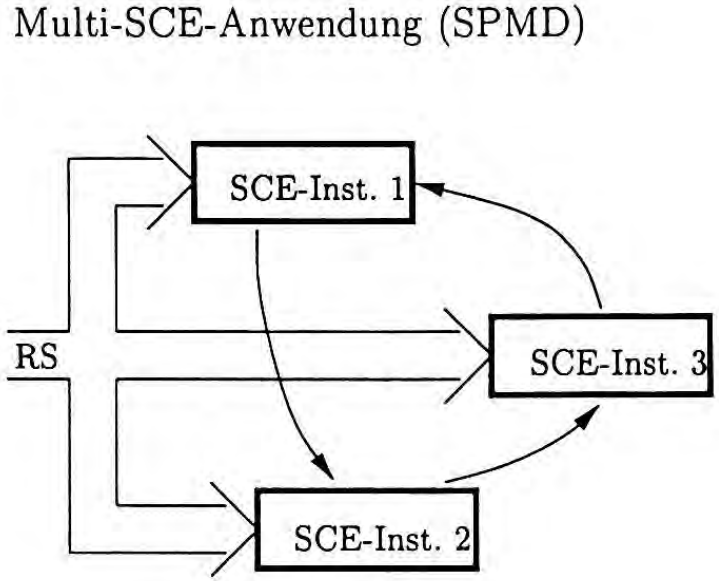
FORTSCHRITTSBERICHT
SIMULATION FBS 7



Sven Pawletta

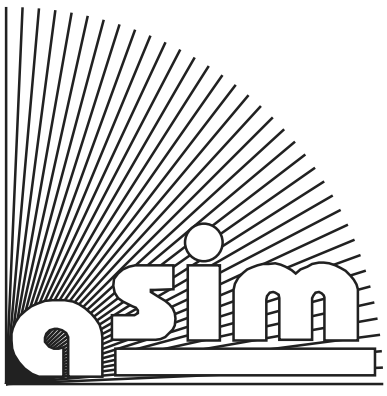
Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen

ARGESIM

ISBN Ebook 978-3-903347-07-6

ISBN Print 978-3-901608-57-5

DOI: 10.11128 / fbs.07



ARGESIM

Fortschrittsberichte Simulation

FBS Band 7

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

Sven Pawletta

Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen

ARGESIM / ASIM – Verlag, Wien, 2000
ISBN Print 978-3-901608-57-5

Ebook Reprint 2020
ISBN Ebook 978-3-903347-07-6
DOI: 10.11128/fbs.07

Fortschrittsberichte Simulation

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

Betreuer der Reihe:

Prof. Dr. G. Kampe (ASIM)
Fachhochschule Esslingen
Flandernstraße 101, D-73732 Esslingen,
Tel: +49-711-397-3741, Fax:+49-711-397-3763
Email: **kampe@ti.fht-esslingen.de**

Prof. Dr. D. P. F. Möller (ASIM)
Technische Informatiksysteme, Univ. Hamburg
Vogt-Kölln-Str. 30, D-22527 Hamburg
Tel: +49-40-42883-2428, Fax: +49-40-42883-2552
Email: **dietmar.moeller@informatik.uni-hamburg.de**

Prof. Dr. F. Breitenecker (ARGESIM / ASIM)
Technische Universität Wien
Wiedner Hauptstraße 8 - 10, 1040 Wien, Austria Tel:
+43-1-58801-10152, Fax: +43-1-58801-11499
Email: **Felix.Breitenecker@tuwien.ac.at**

FBS Band 7

Titel: Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen

Autor: Dr.-Ing. Sven Pawletta
Univ. Rostock, Inst. f. Automatisierungstechnik
Albert-Einstein Str. 2
D - 18051 Rostock
Email: **sven.pawletta@etechnik.uni-rostock.de**

Begutachter des Bandes:

Prof. Dr.-Ing. habil. Bernhard Lampe, Univ. Rostock
Prof. Dipl.-Ing. Dr. Felix Breitenecker, TU Wien
Dr.-Ing. habil. Peter Schwarz, Fraunhofer-Institut Dresden

ARGESIM / ASIM – Verlag, Wien, 2000
ISBN Print 978-3-901608-57-5

Ebook Reprint 2020
ISBN Ebook 978-3-903347-07-6
DOI: 10.11128/fbs.07

Erweiterung eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen

Dissertation

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

eingereicht an der Universität Rostock
Fakultät für Ingenieurwissenschaften

von

Dipl.-Ing. Sven Pawletta

Faule Straße 23
D-18055 Rostock

Rostock, im Juni 1998

Eingereicht am: 3. Juni 1998

Verteidigt am: 30. April 1999

Begutachtet von: Prof. Dr.-Ing. habil. Bernhard Lampe
Institut für Automatisierungstechnik, Universität Rostock

Prof. Dipl.-Ing. Dr. techn. Felix Breitenecker
Abteilung Simulationstechnik, Technische Universität Wien

Dr.-Ing. habil. Peter Schwarz
Fraunhofer-Institut für Integrierte Schaltungen Dresden

Kurzfassung

Interaktive Berechnungs- und Visualisierungssysteme erlangen seit ihrer Einführung in den frühen 80er Jahren eine immer größere Verbreitung.

Vor allem im automatisierungstechnischen Bereich konnten sie sich aufgrund ihrer interpretativen Arbeitsweise, der integrierten array-orientierten Programmiersprache und der Vielfalt der zur Verfügung gestellten numerischen Methoden sehr schnell als hocheffiziente Werkzeuge für die Prototyp-Entwicklung etablieren. Inzwischen haben diese Systeme in vielen wissenschaftlich-technischen Anwendungsgebieten die traditionelle Programmierung auf Basis kompilierbarer Sprachen weitgehend abgelöst.

Eine zweite Entwicklung von fundamentaler Bedeutung für das wissenschaftlich-technische Rechnen ist die Parallelverarbeitung. Aus Kosten- und Verfügbarkeitsgründen war ihr Einsatz lange Zeit auf das Spezialgebiet des Hochleistungsrechnens begrenzt. Seit Anfang der 90er Jahre stehen aber einem wachsenden Anwenderkreis für die Parallelverarbeitung geeignete Plattformen in Form von Mehrprozessorsystemen und vernetzten Computern zur Verfügung. Einer breiten Nutzung der Parallelverarbeitung steht aber noch entgegen, daß ihr Einsatz spezielle Werkzeuge und Kenntnisse erfordert.

Um dieses Problem zu lösen, ist es notwendig, die Techniken der Parallelverarbeitung und der interaktiven Berechnungs- und Visualisierungssysteme so zu kombinieren, daß die Vorteile beider Techniken optimal zum Tragen kommen, d.h. hohe Rechenleistung und einfache Bedienbarkeit. Bisher ist dies nur ungenügend gelungen.

In der vorliegenden Arbeit wird ein neuer Ansatz zur Lösung dieses Problems entwickelt, der sich vor allem dadurch auszeichnet, daß er mit geringem Aufwand auf Basis konventioneller Systeme realisierbar ist.

Anhand einer prototypischen Realisierung werden die Leistungsfähigkeit des Ansatzes untersucht und beispielhafte Anwendungen beschrieben.

Abstract

Since their introduction in the early eighties, interactive computation and visualization environments have become very popular.

Especially in the field of automatic control, they established themselves as highly efficient prototyping tools due to their interactive working method, the integrated array-oriented programming language and the variety of provided numerical routines. In the meantime those systems superseded to a large extend conventional compiler based programming techniques in many scientific and technical fields.

A second evolution of general importance in the field of scientific and technical computing is parallel processing. However, for costs and availability reasons the usability of parallel processing was limited to very high performance computing for a long time. This situation changed since the early nineties when platforms for parallel processing became available for an increasing user group. Nevertheless, parallel processing is not in broad use up to now, because its usage requires specialized tools and expert knowledge.

To overcome this problem it is necessary to combine the technologies of parallel processing and interactive computing and visualization systems in such a way, that the advantages of both technologies complement one another optimally – i.e. high performance and convenient operability. This only succeeded insufficiently up to now.

In the present work, a new approach is developed to solve that problem. The crucial advantage of this approach is, that it is feasible with small expenditure on the basis of conventional systems.

By means of a prototype implementation the ability of the approach is examined and application examples are described.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wissenschaftlich-technische Problemstellungen gehören zu den frühsten Computeranwendungen. Einerseits hat der ständig wachsende Bedarf an Rechenleistung und Speicherkapazität in diesem Bereich die Entwicklung der Computertechnik in den vergangenen 50 Jahren maßgeblich initiiert und andererseits übten Entwicklungen im Bereich der Computerarchitekturen starken Einfluß auf das wissenschaftlich-technische Rechnen selbst aus. Aufgrund dieser engen Verknüpfung kann die Entwicklung der beiden Bereiche nicht voneinander losgelöst betrachtet werden. Freeman und Phillips unterscheiden in [29] zwei grundlegende Phasen.

Die erste Phase (1950-1980) ist gekennzeichnet durch eine enorme Leistungssteigerung der Computerhardware (alle fünf Jahre eine Verzehnfachung der Leistung) und eine relative Konstanz der grundlegenden Programmiertechnik. Letzteres war möglich, weil die Mehrzahl der realisierten Computerarchitekturen auf einem gemeinsamen Verarbeitungsprinzip – *dem Von-Neumann-Modell* (s. [3]) – bzw. auf Erweiterungen dieses Prinzips beruhten. Für die Programmierung stellte das Von-Neumann-Modell eine hervorragende Abstraktion realer Compterarchitekturen dar, weil es die meisten Variationen und Entwicklungen auf der Architekturebene für den Nutzer transparent machte. Damit war es möglich, Programme weitgehend unabhängig von einer konkreten Computerarchitektur zu erstellen[1]. Obwohl das Von-Neumann-Modell ein rein *sequentielles Verarbeitungsprinzip* darstellt, war es durchaus geeignet, auch als Basis für die Programmierung der in der ersten Phase entwickelten parallelen Computerarchitekturen zu dienen, da die Parallelität dieser Architekturen (bit-parallel, synchrone Datenparallelität, Pipelining) nicht Gegenstand der Programmierung war bzw. durch Compiler weitgehend transparent gemacht wurde.

Im wissenschaftlich-technischen Bereich wurde die Computertechnik in der ersten Phase vorwiegend zur Lösung numerischer Probleme eingesetzt, wobei die Programmierung, zumindest in der zweiten Hälfte dieser Phase, durch höhere prozedurale Programmiersprachen (Fortran, C u.a.) und vorgefertigte numerische Programmbibliotheken (PACK-Bibliotheken, NAG, IMSL u.a.) gekennzeichnet war.

[1] Gemeint ist hier die konzeptionelle Architekturunabhängigkeit, nicht Portabilität im Sinne von Binär- oder Quelltextkompatibilität.

Der Beginn der zweiten Phase (ca. 1980) wird nach Freeman und Phillips durch das Aufkommen der *Multiprozessorsysteme* markiert. Weil diese Systeme die Uniprozessor-Architekturen der ersten Phase nicht ersetzen, sondern diese als elementare Komponenten enthalten, gibt es in der zweiten Phase zwei grundlegende Entwicklungen auf der Architekturebene: Uniprozessor- und Multiprozessorsysteme. Da die Parallelverarbeitung in Multiprozessorsystemen *asynchron* erfolgt, ist das Von-Neumann-Modell als alleinige Architekturabstraktion – und damit als Basis für die Programmierung solcher Systeme – nicht mehr geeignet. Im Unterschied zu Uniprozessor- müssen Multiprozessorsysteme *explizit parallel* programmiert werden. Damit können Programme konzeptionell nicht mehr unabhängig von einer Zielarchitektur erstellt werden.

Diese Entwicklung in der Computertechnik führte im wissenschaftlich-technischen Bereich zu einer stärkeren Differenzierung zwischen dem Hochleistungsrechnen und allgemeinen Anwendungen. Zwar unterschied sich die in den beiden Teilbereichen zur Verfügung stehende Computertechnik hinsichtlich der Leistungsfähigkeit und der damit einhergehenden Kosten bereits in der ersten Phase voneinander, da die Unterschiede zwischen den Architekturen aber nur geringe Auswirkungen auf die Programmiertechnik hatten, war der gesamte Bereich des wissenschaftlich-technischen Rechnens auf dieser Ebene noch homogen. Dagegen erfordern die Multiprozessorsysteme in der zweiten Phase, die bis zum Beginn der 90er Jahre aus Kostengründen nur dem Bereich des Hochleistungsrechnens zur Verfügung standen, völlig andere Programmiertechniken als die im allgemeinen Gebrauch befindlichen Uniprozessorsysteme.

Dementsprechend spielt in der zweiten Phase im Bereich des Hochleistungsrechnens die parallele Programmierung eine zentrale Rolle. Innerhalb der Informatik wurden dafür anspruchsvolle Programmierkonzepte entwickelt und in Form völlig neuer paralleler Programmiersprachen bereitgestellt. Dennoch werden bis heute, vor allem für numerische Anwendungen, vorwiegend um „parallele Features“ erweiterte, traditionelle Techniken eingesetzt (parallele Fortran- und C-Dialekte in Verbindung mit parallelen Numerikbibliotheken, z.B. BLAS).

Der allgemeine wissenschaftlich-technische Anwendungsbereich ist dagegen durch einschneidende Veränderungen der Programmiertechnik gekennzeichnet. Zum Ende der ersten Phase wurde die Leistungsfähigkeit der Uniprozessorsysteme bereits so groß, daß auf Kosten der Effizienz Aspekte wie Komfortabilität und Benutzerfreundlichkeit stärker in den Mittelpunkt gerückt werden konnten. Als Resultat dieser Entwicklung entstanden Entwicklungsumgebungen für wissenschaftlich-technische Anwendungen, in denen numerische oder symbolische Berechnungen interaktiv ausgeführt werden konnten. Während sich für die Systeme, die das symbolische Rechnen unterstützen (z.B. Maple, Mathematica, MuPAD; s. [37, 89, 34]), der Oberbegriff *Computeralgebra-Systeme* (CAS) durchgesetzt hat, werden die auf numerischen Methoden basierenden Systeme (z.B. MATLAB, Octave, Scilab; s. [82, 25, 74]) häufig nur nach ihrem bekanntesten Vertreter als MATLAB-ähnliche Systeme bezeichnet. MATLAB (*matrix laboratory*) selbst wird von seinen Entwicklern in [82] allgemein als *scientific and technical computing environment* charakterisiert. Im Rahmen der vorliegenden Arbeit soll diese Bezeichnung und das davon abgeleitete Kürzel SCE als Oberbegriff für alle Systeme dieser Klasse verwendet werden.

Das Grundanliegen der ersten SCEs bestand darin, dem Nutzer einen bequemen interaktiven Zugriff auf die Routinen von Numerikbibliotheken zu ermöglichen, um so einerseits die Einarbeitung in numerische Problemlösungsstrategien zu erleichtern und andererseits die Phase der Erstellung und Inbetriebnahme von Algorithmen zu effektivieren (*rapid prototyping*). Für die anschließende Anwendung in der Produktionsphase sahen die ursprünglichen Konzepte der SCEs eine Übersetzung der Prototypen in kompilierbare Programmiersprachen vor. Die wachsende Leistungsfähigkeit der Uniprozessorsysteme sowie die immer ausgefeiltere interne Technik der SCEs erlaubte es jedoch zunehmend, diese Systeme auch für die Produktionsphase zu verwenden. Darüber hinaus wurde der Methodenumfang ständig erweitert. So stellen moderne SCEs neben der numerischen Basisfunktionalität vor allem leistungsfähige Visualisierungsmethoden und umfangreiche fachspezifische Algorithmensammlungen zur Verfügung. Komplexe Anwendungen können mit Hilfe von integrierten, arrayorientierten Programmiersprachen erstellt werden, die so ausdrucksstark sind, daß Operationen in einer Programmzeile kodiert werden können, für die in klassischen Programmiersprachen wie Fortran oder C einige Dutzend Programmzeilen notwendig wären. Aufgrund dieser Eigenschaften haben SCEs die traditionelle Programmierung auf der Basis kompilierbarer Sprachen in vielen wissenschaftlich-technischen Anwendungsbereichen weitgehend abgelöst.

Seit Beginn der 90er Jahre schwächt sich die Trennung zwischen Hochleistungsrechnen und allgemeinen wissenschaftlich-technischen Anwendungen in Bezug auf die verfügbaren Computerarchitekturen wieder allmählich ab, da einerseits die ersten Multiprozessorsysteme in Massen produziert werden, und dadurch stark im Preis verfallen, und andererseits die Technologie der lokalen Computervernetzung (*local area networks* - LANs) so leistungsfähig geworden ist, daß auch die weit verbreiteten Computercluster zur Parallelverarbeitung benutzt werden können (*network computing*). Da sowohl Multiprozessorsysteme als auch Computercluster asynchrone Parallelverarbeitungssysteme sind, gehören sie zur Klasse der MIMD-Architekturen (*multiple instructions multiple data*). Ihre Programmierung beruht deshalb auf den gleichen Prinzipien.

Damit ist im Bereich des wissenschaftlich-technischen Rechnens folgende Situation entstanden:

- Auf der Seite des Hochleistungsrechnens existieren langjährige Erfahrungen zur Parallelverarbeitung auf der Basis von MIMD-Systemen. Die dort eingesetzte Programmiertechnik ermöglicht eine effiziente Nutzung der Hardwareressourcen, sie befindet sich aber hinsichtlich Komfortabilität und Benutzerfreundlichkeit im wesentlichen noch auf dem Niveau der ersten Phase.
- Für den allgemeinen Anwendungsbereich stehen mit den SCEs moderne und sehr effiziente Programmiertechniken – jedoch ausschließlich für Uniprozessorsysteme – zur Verfügung. Aufgrund des stetig steigenden Leistungsbedarfs und der immer besseren Verfügbarkeit besteht aber auch in diesem Bereich die dringende Notwendigkeit, MIMD-Systeme im großen Umfang einzusetzen.

Es liegt auf der Hand, daß ein breiter Einsatz von MIMD-Systemen im allgemeinen wissenschaftlich-technischen Anwendungsbereich nicht nach dem Vorbild des Hochleistungsrechnens erfolgen kann, weil dies zu einem Rückschritt in der Programmier-

technik führen würde. Vielmehr wäre eine Zusammenführung von SCE-basierter Anwendungsentwicklung und Parallelverarbeitung wünschenswert. Daß die Informatik hierfür fertige Lösungen „produziert“ ist jedoch, wie bereits die Entwicklung der sequentiellen SCEs in den 80er Jahren zeigte, nicht zu erwarten. So gründet sich der außerordentliche Erfolg der existierenden SCEs vor allem darauf, daß sie das Ergebnis einer konsequent anwendungsorientierten Umsetzung von theoretischen Grundlagen und Methoden aus der Informatik sind. In ähnlicher Weise werden auch für die Parallelverarbeitung geeignete SCEs aus den primären Anwendungsbereichen heraus entstehen. Einen Beitrag dazu soll die vorliegende Arbeit leisten.

Sie basiert auf mehrjährigen Forschungen zur verteilten und parallelen Verarbeitung in der Automatisierungstechnik, wobei der Schwerpunkt auf regelungstechnischen Problemstellungen lag. Da dieses Fachgebiet einerseits durch einen hohen Rechenleistungsbedarf und verteilte Problemstrukturen gekennzeichnet ist (aufwendige iterative Offline-Berechnungen, Echtzeitanforderungen, verteilte Prozeßsteuerungen etc.) und andererseits bereits wesentlichen Anteil an der Entwicklung der sequentiellen SCEs hatte, ist es prädestiniert, auch zur Entwicklung von parallelen SCE-Konzepten beizutragen.

Im regelungstechnischen Anwendungsbereich ist die Parallelverarbeitung häufig mit einer verteilten Verarbeitung gekoppelt bzw. in diese eingebettet. Den Beziehungen zwischen paralleler und verteilter Verarbeitung ist deshalb ein eigener Abschnitt im Grundlagenteil der Arbeit gewidmet. Da eine umfassende Behandlung beider Aspekte im Rahmen dieser Arbeit nicht möglich ist, wird über den erwähnten Abschnitt hinaus die verteilte Verarbeitung nicht näher betrachtet. Es sei jedoch bemerkt, daß die gewonnenen Erkenntnisse und Ergebnisse größtenteils sowohl für parallele als auch für verteilte Problemstellungen nutzbar sind.

In einem einführenden Kapitel werden die allgemeinen hardware- und softwaretechnischen Grundlagen der Parallelverarbeitung, die aus der Informatik entlehnt sind und auf denen diese Arbeit aufbaut, dargestellt.

Die anschließenden Kapitel zu Message-Passing-Systemen und sequentiellen SCEs analysieren den aktuellen Stand der Technik auf den Gebieten, die durch diese Arbeit näher zusammengebracht werden sollen, d.h. der Parallelverarbeitung auf MIMD-Systemen, wobei der Schwerpunkt auf Systemen ohne gemeinsamen Speicher liegt, und der SCE-basierten Entwicklung wissenschaftlich-technischer Anwendungen. Darauf aufbauend, werden im Kapitel 5 verschiedene grundsätzliche Wege zur Verbindung von Parallelverarbeitung und SCE-Anwendungen untersucht und bewertet.

In den Kapiteln 6, 7 und 8 werden das Konzept, eine prototypische Realisierung sowie beispielhafte Anwendungen eines neuen Ansatzes – *dem Multi-SCE-Ansatz* – dargestellt.

Abschließend erfolgt in den Kapiteln 9 und 10 eine Betrachtung der Bezüge zu anderen Arbeiten und zu möglichen Weiterentwicklungen des vorgestellten Multi-SCE-Ansatzes.

Zur Einordnung der vorliegenden Arbeit sei bemerkt, daß ihre Intention nicht darin besteht, einen Beitrag zur parallelen Verarbeitung als Teilgebiet der Informatik zu leisten. Vielmehr soll aufbauend auf den durch die Informatik bereitgestellten Grundlagen ein Fortschritt hinsichtlich einer vereinfachten Nutzung der Parallelverarbeitung im Bereich des wissenschaftlich-technischen Rechnens erzielt werden.

Kapitel 2

Grundlagen der Parallelverarbeitung

Wie in der Einleitung bereits erläutert wurde, war die Parallelverarbeitung aufgrund der hohen Kosten für parallele Hardware über einen langen Zeitraum ausschließlich dem Hochleistungsrechnen vorbehalten und stellte damit ein eigenständiges Spezialgebiet innerhalb der Informatik dar.

Durch den stetigen Preisverfall der Multiprozessorsysteme in den letzten Jahren und die durch verbesserte Netzwerktechnologien möglich gewordene Nutzung von Computerclustern als Parallelverarbeitungsplattform spielt die parallele Verarbeitung heute in vielen Anwendungsgebieten auf unterschiedlichsten Ebenen eine zunehmende Rolle. Darüber hinaus kam es in jüngster Zeit in Teilbereichen zu einer Verschmelzung von paralleler und verteilter Verarbeitung. Damit hat sich aus dem ehemaligen, relativ scharf abgegrenzten Spezialgebiet ein breit gefächertes und nur noch schwer zu überschauendes allgemeines Fachgebiet entwickelt.

In diesem Kapitel sollen deshalb zuerst die parallele und verteilte Verarbeitung zueinander in Beziehung gesetzt und gegeneinander abgegrenzt werden. Danach erfolgt eine Einführung in die grundlegenden Prinzipien paralleler Rechnerarchitekturen, Konzepte zu deren Programmierung sowie Methoden zur Leistungsbewertung. Abschließend werden die verschiedenen Ansätze hinsichtlich ihrer Eignung für die Parallelverarbeitung im Rahmen von wissenschaftlich-technischen Berechnungs- und Visualisierungsumgebungen bewertet.

2.1 Parallel versus Verteilt

Je nach Sichtweise handelt es sich bei der parallelen und verteilten Verarbeitung um *verschiedene Teilgebiete* der Informatik oder um *ein grundlegendes Prinzip* der rechnergestützten Problemlösung.

Da es aufgrund der historischen Entwicklung keine allgemein akzeptierte Klassifikation gibt, die die Zweiseitigkeit dieser Problematik inhaltlich und begrifflich exakt erfaßt, kann die zugrunde liegende Sichtweise bei Arbeiten in diesem Bereich nur aus dem Kontext erkannt werden. Dadurch wird der erste Zugang über die relevante Informatik-Literatur dem fachgebietsfremden Anwender außerordentlich erschwert.

Das gemeinsame Grundprinzip der parallelen und verteilten Verarbeitung wird vor allem in vergleichenden Betrachtungen deutlich, die die sequentielle Verarbeitung einbeziehen. So existieren nach Carriero und Gelernter ([16]) zwei Strategien der rechnergestützten Problemlösung:

1. die sequentielle Lösung eines Problems und
2. die örtliche und/oder zeitliche Zerlegung eines Problems in Teilaufgaben.

Die rein sequentielle Verarbeitung wird dabei als unnatürliche Strategie angesehen, weil sie allein durch Restriktionen konventioneller Rechnerarchitekturen begründet ist.

Dagegen entspricht die zweite Strategie der natürlichen Struktur realer Problemstellungen. Von zentraler Bedeutung bei dieser Strategie ist, daß die Bearbeitung der Teilaufgaben, über ihre örtliche und/oder zeitliche Separierung hinweg, koordiniert verläuft. Carriero und Gelernter bezeichnen sie deshalb auch als *koordinierte Verarbeitung*.

Demzufolge stellt jede verteilte Verarbeitung eines Problems eine örtlich und zeitlich koordinierte Verarbeitung von Teilaufgaben dar, wobei die parallele Verarbeitung ein Spezialfall der verteilten Verarbeitung ist, bei der örtlich getrennte Teilaufgaben zeitlich simultan bearbeitet werden. Eine Zusammenfassung dieser Betrachtungsweise in Form einer Klassifikation ist in Abbildung 2.1 dargestellt.

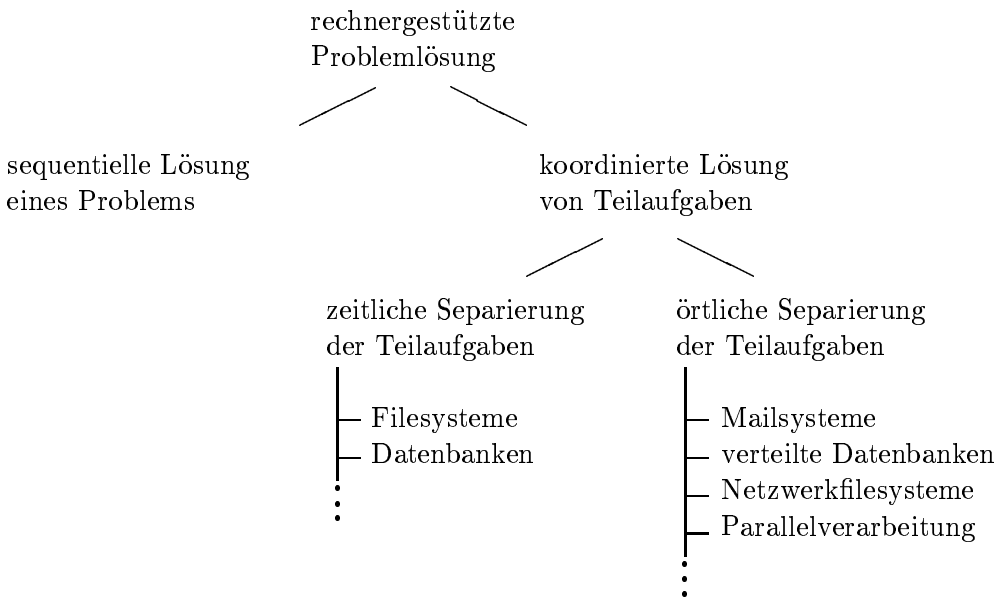


Abbildung 2.1: Klassifikation rechnergestützter Problemlösungsstrategien

Wie bereits erwähnt, finden solche Klassifikationen nur eine geringe Akzeptanz, was zu einem großen Teil darin begründet liegt, daß sie neue Begriffe wie Koordination, konkurrente Verarbeitung ([4]) u.a. einführen.

Während die Klassifikation in Abbildung 2.1 die enge Verbindung von paralleler und verteilter Verarbeitung widerspiegelt (parallele Verarbeitung als Spezialfall der verteilten Verarbeitung), ist die Differenziertheit zweier nebeneinander stehender Teilgebiete daraus nicht erkennbar. Letzteres wird deutlich, wenn man die unterschiedlichen, in Abbildung 2.2 aufgeführten Motivationen für die parallele und verteilte Verarbeitung betrachtet. Dabei stellen die Motivationspunkte gleichzeitig die Vorteile der parallelen bzw. verteilten Verarbeitung gegenüber der sequentiellen Verarbeitung dar.

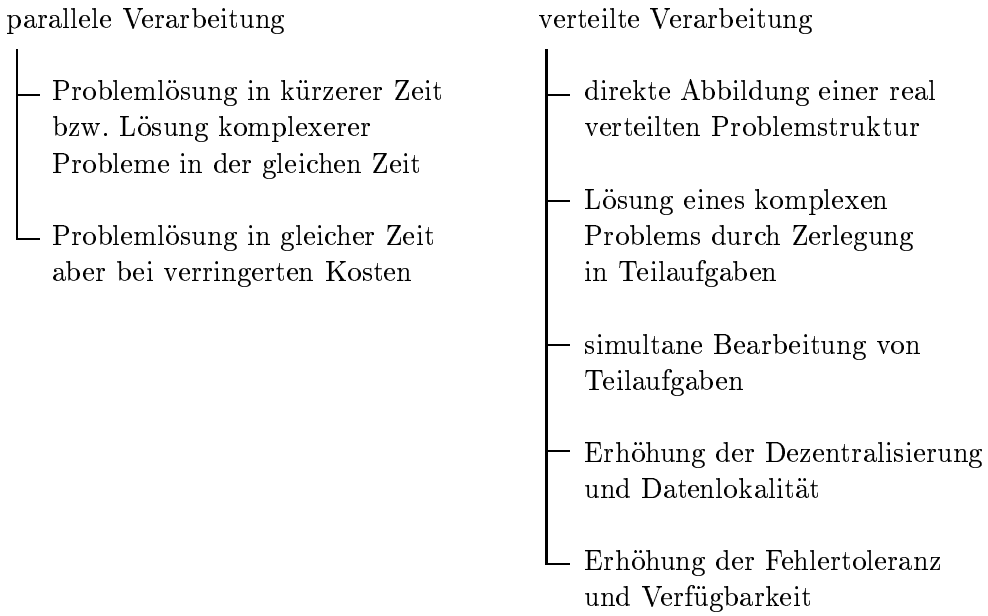


Abbildung 2.2: Motivationen für die parallele und verteilte Verarbeitung

Den unterschiedlichen Motivationen bzw. Vorteilen entsprechend, haben die parallele und verteilte Verarbeitung verschiedene Ursprünge.

So stammen die theoretischen Grundlagen der Parallelverarbeitung vor allem aus dem Bereich der Betriebssystementwicklung und das primäre Anwendungsgebiet war über lange Zeit das Hochleistungsrechnen auf der Basis eng gekoppelter[1] paralleler Rechnerarchitekturen. Die Schlüsselprobleme in diesem Bereich sind die Kommunikation und Synchronisation zwischen den Verarbeitungseinheiten. Die Toleranz gegenüber Ausfällen einzelner Verarbeitungseinheiten in eng gekoppelten Systemen ist dagegen von geringem Interesse, da solche Ereignisse sehr selten auftreten und deshalb in der Regel als fatale Fehler behandelt werden.

[1]Kopplung über gemeinsamen Speicher; siehe Abschnitt 2.2.1.2

Die Ursprünge der verteilten Verarbeitung gehen auf die Rechnervernetzung (lose gekoppelte Systeme) zurück. Anders als bei der parallelen Verarbeitung steht hier die Toleranz sowohl gegenüber Ausfällen ganzer Komponenten des lose gekoppelten Systems als auch gegenüber Kommunikationsfehlern (z.B. verlorengegangene oder korrupte Nachrichten) im Mittelpunkt.

Seit Anfang der 90er Jahre gibt es den Trend, auch lose gekoppelte Systeme für die Parallelverarbeitung zu nutzen. Da hierbei sowohl die traditionellen Probleme der parallelen als auch der verteilten Verarbeitung eine Rolle spielen, kommt es in diesem Bereich zu einer Verschmelzung beider Teilgebiete. Dadurch ergibt sich hier ein besonderer Bedarf nach einem gemeinsamen Oberbegriff für die parallele und verteilte Verarbeitung. Obwohl die Entwicklung in diesem Bereich noch außerordentlich dynamisch ist, zeichnet sich bereits ab, daß sich auch hier kein wohldefinierter, d.h. durch eine sinnvolle Klassifikation begründeter Oberbegriff durchsetzen wird. Neben anderen wird in der aktuellen Literatur häufig der Begriff *network computing* benutzt.

Die in der vorliegenden Arbeit eingesetzten Konzepte und Methoden stammen größtenteils aus dem Bereich des Network-Computing.

2.2 Parallele Rechnerarchitekturen

In der Literatur erfolgt der Einstieg in die Klassifikation von Rechnerarchitekturen in der Regel über den von Flynn in [26] vorgeschlagenen Ansatz.

Flynn abstrahiert die sequentielle Verarbeitung als die Anwendung eines Instruktionsstromes SI (*single instruction stream*) auf einen Datenstrom SD (*single data stream*) in einem Verarbeitungselement PE (*processing element*). Formal lassen sich durch Vervielfachung des Verarbeitungselementes und Kombination von einzelnen und mehrfachen Instruktions- und Datenströmen (s. Abb. 2.3) vom **SISD**-Modell drei Parallelverarbeitungsschemen ableiten:

SIMD: Ein Instruktionsstrom wird auf mehrere Datenströme angewandt (*datenparallel*). Damit existiert wie bei der sequentiellen Verarbeitung nur *ein* Kontrollfluß, d.h. die Verarbeitungselemente müssen in einer sinnvollen Realisierung im Gleichtakt arbeiten (*synchrone Parallelität*).

MIMD: Jedem Datenstrom ist ein separater Instruktionsstrom zugeordnet. Da dadurch mehrere, unabhängige Kontrollflüsse existieren, ist ein Gleichtakt der Verarbeitungselemente nicht mehr notwendig (*asynchrone Parallelität*).

MISD: Mehrere Instruktionsströme werden auf einen Datenstrom angewandt. Die Interpretation dieses Verarbeitungsprinzips ist in der Literatur umstritten ([27, 7, 28]).[2]

Der Vorteil der Klassifikation nach Flynn gegenüber zahlreichen anderen Klassifizierungsansätzen (z.B. [28, 75, 39, 76, 85]) besteht darin, daß eine Differenzierung anhand weniger grundlegender Verarbeitungsmodelle möglich ist. Bei der Anwendung auf reale Rechnerarchitekturen tritt jedoch das Problem auf, daß in diesen

[2]Einige Autoren sehen die MISD-Klasse sogar als leer an ([29, 87]), d.h. keine existierende Rechnerarchitektur kann ihr zugeordnet werden.

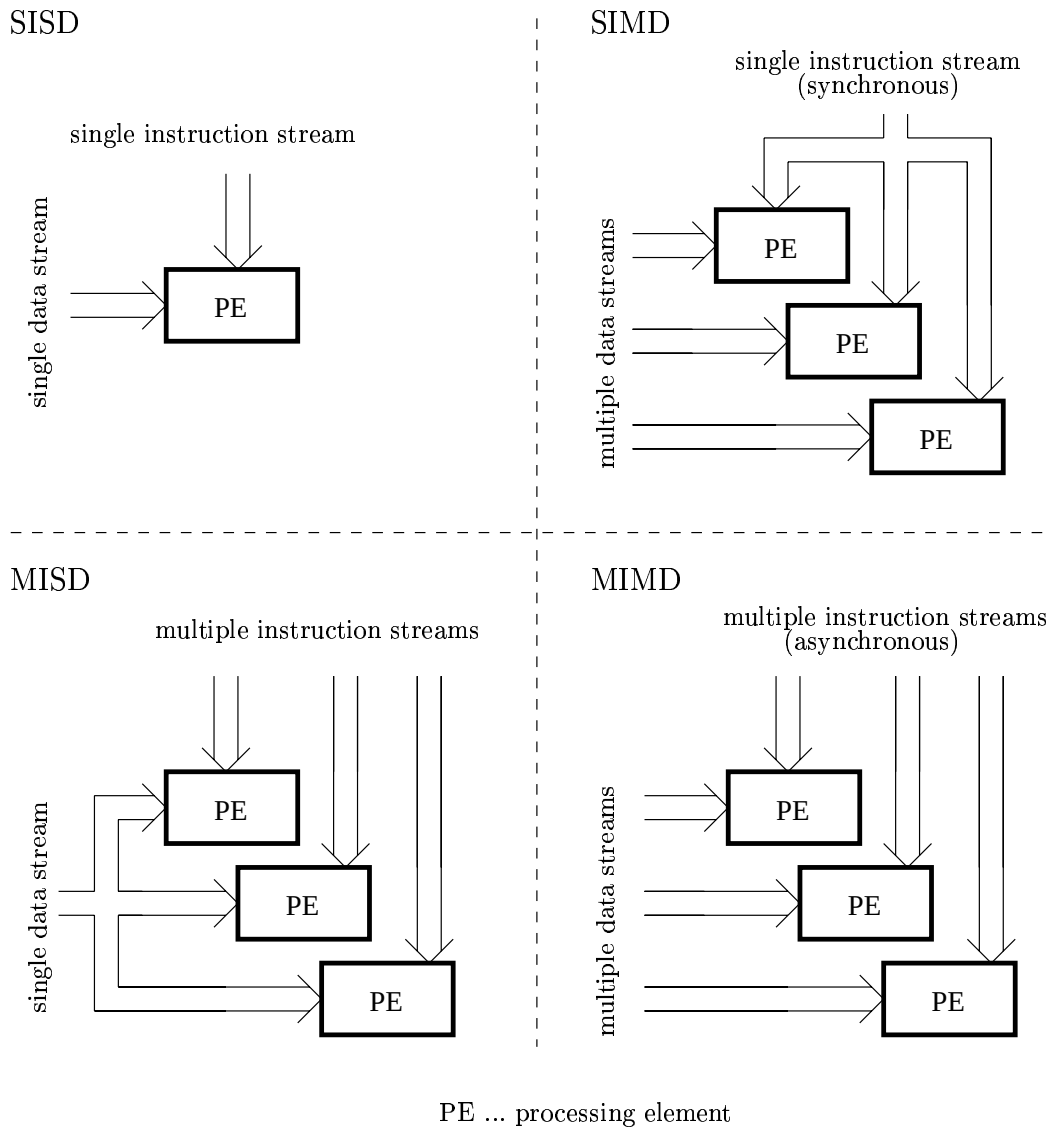


Abbildung 2.3: Kombinationen von Instruktions- und Datenströmen nach Flynn

häufig verschiedene Parallelisierungstechniken miteinander kombiniert werden. Bei genauer Betrachtung sind solche Systeme nach der Flynn'schen Klassifikation oftmals hybrid.

Eine Einordnung von Rechnerarchitekturen hinsichtlich der verwendeten Parallelisierungstechniken wird von Moldovan in [56] vorgenommen. Ausgangspunkt ist dabei die Unterteilung in zwei Hauptgruppen:

Von-Neumann-basierte Architekturen, die sich als natürliche Erweiterung des klassischen Von-Neumann-Modells ([3]) erklären lassen, und

spezielle Architekturen, die auf grundsätzlich anderen Verarbeitungsprinzipien beruhen.

Bei den speziellen Architekturen handelt es sich entweder um experimentelle Ansätze (z.B. Daten-Fluß-, Reduktions-Maschinen) oder um anwendungsspezifische Lösungen (z.B. Hardwareimplementationen von neuronalen Netzwerken). Sie sind somit für allgemeine wissenschaftlich-technische Problemstellungen nicht geeignet und sollen deshalb in dieser Arbeit nicht weiter berücksichtigt werden.

2.2.1 Erweiterungen des Von-Neumann-Modells

Das Von-Neumann-Modell beschreibt ein sequentielles Verarbeitungsprinzip auf Basis der drei Komponenten Speicher, Prozessor und Ein-/Ausgabeeinheit (s. Abb. 2.4). Als natürliche Erweiterungen dieses Prinzips können

- die Einführung von Parallelität innerhalb des Prozessors,
- die Verwendung mehrerer Prozessoren innerhalb eines Rechners sowie
- die Kopplung mehrerer Rechner

angesehen werden.

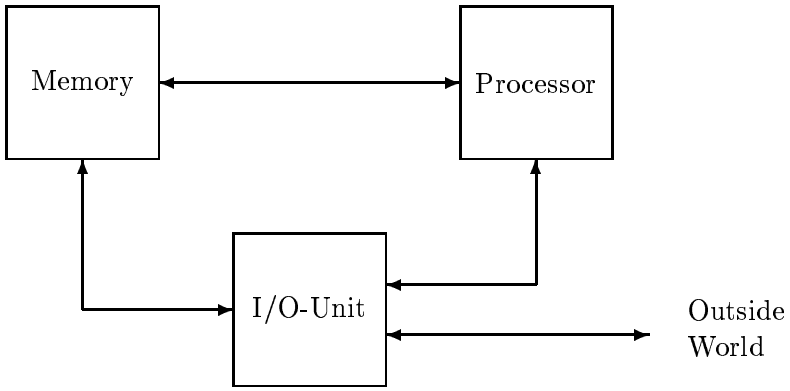


Abbildung 2.4: Von-Neumann-Modell nach [56]

2.2.1.1 Parallelität innerhalb des Prozessors

Im Von-Neumann-Modell besteht der Prozessor aus einem Steuerwerk (*control unit,* CU), das einen Instruktionsstrom dekodiert, und einem Rechenwerk (*arithmetic logical unit,* ALU), welches Operationen über einem Datenstrom ausführt (Abb. 2.5).

Mit Hilfe spezieller bzw. mehrfacher Steuer- und Rechenwerke können innerhalb eines Prozessors verschiedene Formen der Parallelverarbeitung realisiert werden:

Bit-Parallelität: Durch Implementation einer bit-parallelen Arithmetik kann eine ALU Operationen über *Datenwörtern* ausführen. Da die parallel verarbeiteten Bits zu einem Datenwort gehören und nicht verschiedene Datenströme repräsentieren, handelt es sich hierbei nach wie vor um eine SISD-Verarbeitung.

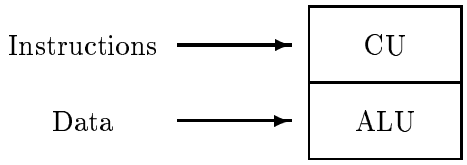


Abbildung 2.5: Prinzipieller Aufbau eines Prozessors nach dem Von-Neumann-Modell

Pipelining: Wird die CU und/oder ALU mehrstufig ausgelegt, kann durch überlagerte Ausführung von Instruktionen (*instruction pipelining*) bzw. von arithmetischen Operationen (*arithmetic pipelining*) eine parallele Verarbeitung erreicht werden. Die Einordnung der Pipeline-Verarbeitung in die Flynn'sche Klassifikation ist umstritten.[3]

Arrayverarbeitung: Durch Verwendung mehrerer ALUs, die unter einem gemeinsamen Steuerwerk (*array control unit,* ACU) betrieben werden, sind parallele Operationen über mehrdimensionalen Daten ausführbar. Da die Verarbeitung über die ACU durch einen einzigen Instruktionsstrom gesteuert wird, führen alle ALUs identische Operationen über verschiedenen Datenströmen synchron aus. Damit sind Arrayprozessoren eindeutig der SIMD-Klasse zu zuordnen.

Vektorpipelining: Bei der Verarbeitung von mehrdimensionalen Daten kann Parallelität nicht nur durch Verwendung von Arrayprozessoren erreicht werden, sondern auch durch Nutzung des arithmetischen Pipelineprinzips. So können Vektoroperationen in eine Anzahl von gleichartigen, skalaren Elementoperationen zerlegt und in einer mehrstufigen ALU überlagert ausgeführt werden (*pipelined vector processor*). Da es sich hierbei um eine spezielle Anwendung des allgemeinen Pipelineprinzips handelt, ist die Einordnung derartiger Prozessoren in die Flynn'sche Klassifikation ebenfalls umstritten.

2.2.1.2 Mehrere Prozessoren innerhalb eines Rechners

Die Verwendung mehrerer Prozessoren in einem Rechner wird als *Multiprozessoransatz* bezeichnet. Im Unterschied zur Parallelverarbeitung innerhalb eines Prozessors, führt der Multiprozessoransatz zwangsläufig zu einem asynchronen System mit mehreren Instruktionsströmen (Kontrollflüssen) und mehreren Datenströmen. Die Parallelverarbeitung in Multiprozessorsystemen erfolgt damit immer nach dem MIMD-Modell.

Um MIMD-Systeme sinnvoll anwenden zu können, müssen diese über Methoden verfügen, die primär asynchrone Arbeit der Prozessoren unter bestimmten Bedingungen zu synchronisieren und Daten zwischen den Prozessoren auszutauschen. In

[3] Bräunl und Fountain ([7, 28]) interpretieren Pipelining beispielsweise als MISD-Verarbeitung. Flynn führt dagegen in [27] „den Pipeline-Prozessor" als einen der drei Grundtypen „des SIMD-Prozessors" auf.

Multiprozessorsystemen können diese Methoden durch zwei unterschiedliche Techniken bereitgestellt werden:

Systeme mit gemeinsamen Speicher: Alle Prozessoren haben über eine Verbindungsstruktur (z.B. ein Bussystem) Zugriff auf einen gemeinsamen Speicherbereich. Die Synchronisation und Kommunikation zwischen den Prozessoren kann damit unmittelbar über Speicherzugriffe erfolgen. Multiprozessorsysteme mit gemeinsamen Speicher werden deshalb auch als *eng gekoppelte MIMD-Systeme* bezeichnet.

Systeme ohne gemeinsamen Speicher: Synchronisations- und Kommunikationsoperationen werden über ein Netzwerk, welches die Prozessoren untereinander verbindet, realisiert (*lose gekoppelte MIMD-Systeme*).

In der Literatur werden zur Unterscheidung der beiden Techniken häufig die Begriffe *System mit gemeinsamen Speicher*[4] und *System mit lokalem bzw. verteiltem Speicher*[5] benutzt. Diese Terminologie ist jedoch nicht widerspruchsfrei, da in realen Multiprozessorsystemen, die zur Synchronisation und Kommunikation einen gemeinsamen Speicher verwenden, in der Regel auch lokale Speichermodule (z.B. Cache-Speicher) vorhanden sind.

2.2.1.3 Gekoppelte Rechner

Die dritte Möglichkeit das Von-Neumann-Modell zu erweitern besteht darin, mehrere Rechner über ihre Ein-/Ausgabeeinheiten und ein externes Netzwerk miteinander zu verbinden, wodurch wiederum ein MIMD-System entsteht.

Da in gekoppelten Rechnersystemen kein gemeinsamer physischer Speicher realisierbar ist, stellen sie grundsätzlich lose gekoppelte MIMD-Systeme dar.

2.2.2 Reale Rechnersysteme

Unabhängig vom unterstützten Verarbeitungsmodell oder von der eingesetzten Parallelisierungstechnik kann man reale Rechnersysteme hinsichtlich ihrer Verfügbarkeit in die Kategorien *Standard-* oder *Spezialtechnik* einordnen. Unter Standardtechnik sollen dabei Systeme verstanden werden, die in Massen produziert werden und damit als allgemein verfügbar gelten können. Die komplementäre Kategorie Spezialtechnik enthält dementsprechend Systeme, die in geringen Stückzahlen hergestellt werden und nur einem eingeschränkten Anwenderkreis zur Verfügung stehen.

Wie aus der Tabelle 2.1 zu entnehmen ist, werden im Bereich der Standardtechnik nur skalare Prozessoren eingesetzt. Die Bezeichnung „skalar“ sollte dabei so interpretiert werden, daß in diesen Prozessoren im Vergleich zu Vektor- und Arrayprozessoren keine „nennenswerte“ Parallelverarbeitung stattfindet. Tatsächlich werden aber in modernen Prozessoren verschiedene Parallelisierungstechniken (Pipelining, Arrayverarbeitung) eingesetzt, womit diese im Sinne der Flynn'schen Klassifikation

[4] engl. *shared memory*
[5] engl. *distributed* bzw. *local memory*

Standardtechnik:		
	skalare Prozessoren:	bit-parallele Arithmetik (4 bis 64 Bit); Instruktionspipelining; in der oberen Leistungsklasse Multi-Pipelines und Arrayverarbeitung mit wenigen parallelen ALUs (< 20)
	Rechnersysteme:	Personalcomputer und Workstations
	Einprozessor-Maschinen	mit o.g. Prozessoren
	Mehrprozessor-Maschinen	Multiprozessorsysteme, meist mit gemeinsamen Speicher und geringer Anzahl o.g. Prozessoren (< 20)
	gekoppelte Rechnersysteme:	vernetzte Ein- und Mehrprozessor-Maschinen
	LAN-Cluster	auf Basis von Standard-Netzwerktechnologien (meist 10 Mbit/s)
	dedizierte Cluster	auf Basis von Hochgeschwindigkeits-Netzwerken (100 Mbit/s und mehr)
Spezialtechnik:		
	Prozessoren:	
	skalare Prozessoren:	bit-parallele Arithmetik (bis 256 Bit); Instruktions- und arithmetisches Pipelining; Multi-Pipelines
	Vektorprozessoren	auf Basis von Vektorpipelining
	Arrayprozessoren	mit bis zu einigen zehntausend Verarbeitungseinheiten (massiv parallel)
	Rechnersysteme:	
	Parallelrechner	Multiprozessorsysteme auf Basis von skalaren und/oder Vektorprozessoren; mit gemeinsamen Speicher (Anz. Prozessoren < 20) oder ohne gemeinsamen Speicher (bis zu mehreren tausend Prozessoren)
	SIMD-Rechner	auf Basis ein oder mehrerer Arrayprozessoren

Tabelle 2.1: Reale Rechnersysteme

hybrid sind. Da die Parallelverarbeitung für den Nutzer transparent erfolgt, kann aber weiterhin das SISD-Modell als Abstraktion verwendet werden.

Einem breiten Anwenderkreis stehen damit neben SISD-Systemen (Einprozessor-Maschinen) nur MIMD-Systeme mit relativ wenigen Verarbeitungseinheiten in Form von Mehrprozessor-Maschinen und Computerclustern zur Verfügung.

Im Bereich der Spezialtechnik kommen sowohl skalare Prozessoren als auch Vektor- und Arrayprozessoren zum Einsatz. Während Arrayprozessoren eindeutig der SIMD-

Klasse zugeordnet werden können, ist die Einordnung des Vektorpipelinings, wie bereits im Abschnitt 2.2.1.1 erwähnt, umstritten.

Auf der Ebene kompletter Systeme handelt es sich entweder um MIMD-Systeme (bestehend aus Vektor- oder skalaren Prozessoren) oder um SIMD-Systeme (bestehend aus einem Arrayprozessor). Rechner mit mehreren Arrayprozessoren bilden eine MIMD-Struktur, in der jede Verarbeitungseinheit ein SIMD-System ist.

2.3 Programmierung paralleler Rechnerarchitekturen

In den vorangegangenen Abschnitten wurde gezeigt, daß es eine ganze Reihe von Ansätzen gibt, Parallelverarbeitung in eine Rechnerarchitektur einzuführen, und daß die exakte Klassifikation von realen Rechnersystemen relativ problematisch ist, weil einerseits verschiedene Verfahren in Kombination miteinander verwendet werden und andererseits die Einordnung von einzelnen Verfahren, wie z.B. das Pipelining, umstritten ist.

Für die Programmierung spielen diese Probleme, insbesondere wenn man davon ausgeht, daß sie nicht in einer Maschinensprache, sondern in einer höheren Programmiersprache erfolgt, aber nur eine untergeordnete Rolle. So ist es für die Erstellung eines Programms weitgehend unerheblich, ob der Prozessor über eine bit-parallele Arithmetik verfügt oder Instruktionen durch Pipelining überlagert ausführen kann. Selbst die Kodierung von vektoriellen Operationen – sofern dies durch die Programmiersprache unterstützt wird – ist prinzipiell unabhängig von deren Ausführung auf einer konkreten Hardware (z.B. elementweise parallele Ausführung auf einem Arrayprozessor, überlagerte Ausführung auf einem Vektorprozessor oder elementweise sequentielle Ausführung auf einem skalaren Prozessor).

Von entscheidender Bedeutung ist dagegen, ob durch ein Programm *ein einzelner Kontrollfluß* (SISD/SIMD-Programmierung) oder *mehrere Kontrollflüsse* (MIMD-Programmierung) zu kodieren sind.

2.3.1 SISD/SIMD-Programmierung

Da die Verarbeitung sowohl in SISD- als auch in SIMD-Architekturen von jeweils einem einzigen Instruktionsstrom gesteuert wird, ist die Erstellung von Programmen für beide Plattformen eng miteinander verflochten. Grundsätzlich lassen sich die Ansätze implizite und explizite daten-parallele Programmierung sowie automatische Vektorisierung unterscheiden.

2.3.1.1 Implizite daten-parallele Programmierung

Der Standard der impliziten daten-parallelen Programmierung wird durch Fortran 90 ([54]) definiert. Da Parallelität in Fortran 90 ausschließlich implizit über Vektoroperationen ausgedrückt wird (s. Beispielprogramm in Abb. 2.6) kann eine Übersetzung mittels Compiler sowohl für SISD- als auch für SIMD-Plattformen erfolgen.

```
INTEGER S_PROD
INTEGER, DIMENSION(100) :: A, B, C
...
C = A * B
S_PROD = SUM(C)
```

Abbildung 2.6: Berechnung des Skalarproduktes in Fortran 90 (nach [7])

Bei der Übersetzung für SISD-Plattformen werden sämtliche Vektoroperationen in sequentielle Elementoperationen umgewandelt. Für SIMD-Plattformen muß der Compiler die Elemente der Datenarrays selbständig auf die vorhandenen Verarbeitungseinheiten verteilen, damit die Vektoroperationen daten-parallel ausgeführt werden können.

2.3.1.2 Explizite daten-parallele Programmierung

In Fortran 90 geschriebene Programme haben den Vorteil, daß sie für alle SISD- und SIMD-Plattformen portabel sind. Nachteilig ist allerdings, daß der Anwender bei der impliziten daten-parallelen Programmierung eventuell vorhandene spezielle Eigenschaften eines SIMD-Rechners nicht direkt für eine effizientere Programmausführung ausnutzen kann.

Aus diesem Grund bieten die Hersteller von SIMD-Rechnern in der Regel eigene Fortran-Dialekte mit zusätzlichen expliziten parallelen Sprachkonstrukten an, mit denen beispielsweise Vektorelemente nach bestimmten Anordnungen auf die Verarbeitungseinheiten verteilt, zwischen diesen rotiert und verschoben werden können.

Nähere Informationen zu expliziten daten-parallelen Programmiersprachen wie Fortran-Plus, CMFortran und Fortran D sowie Beispielprogramme sind in [7] enthalten.

2.3.1.3 Automatische Vektorisierung

Aufgrund der langen Verfügbarkeit von SISD-Systemen existiert heute ein großer Bestand an sequentieller Software, die hauptsächlich in Fortran 77 und C implementiert wurde. Zumindest Programme in Fortran 77 lassen sich aufgrund der Abwärtskompatibilität von Fortran-90-Compilern auch für SIMD-Plattformen übersetzen. Da es aber in sequentiellen Sprachen wie Fortran 77 keine Vektoroperationen gibt, können so übersetzte Programme keinen Nutzen aus der auf SIMD-Plattformen möglichen daten-parallelen Verarbeitung ziehen. Damit besteht noch für geraume Zeit ein Bedarf zur automatischen Vektorisierung von sequentiellen Programmen.

Das Ziel der Vektorisierung ist die Umwandlung von sequentiellen Schleifen und Selektionen in Vektoroperationen. Bevor eine Vektorisierung erfolgen kann, müssen eventuell vorhandene Datenabhängigkeiten erkannt und nach Möglichkeit durch Veränderung der Ausführungsreihenfolge aufgelöst werden. Formalisierte Me-

thoden zur Erkennung und Auflösung von Datenabhängigkeiten sowie zur Vektorisierung werden in [7] beschrieben.

Es existieren zwar einige Compiler mit Vektorisierungsfähigkeiten (z.B. VAST-2, s. [50]), allerdings arbeiten diese nur halbautomatisch. Damit hängt die Qualität des erzeugten Programmkodes stark von der immer noch notwendigen Interaktion mit dem Nutzer ab. In [7] wird sogar bezweifelt, daß vollautomatische Vektorisierung jemals mit zufriedenstellender Qualität möglich sein wird.

2.3.2 MIMD-Programmierung

In einer MIMD-Architektur wird jeder Prozessor durch einen separaten Instruktionsstrom gesteuert, d.h. daß auf jedem Prozessor ein autonomes SISD-Programm ablaufen kann.[6] Tatsächlich können MIMD-Systeme so programmiert und genutzt werden, mit Parallelverarbeitung im Sinne der parallelen Lösung *eines* Problems hat dies jedoch noch nichts zu tun.

Damit mehrere SISD-Programme im Rahmen eines parallelen MIMD-Programms ein Problem in kooperativer Weise parallel lösen können, müssen sie sich untereinander durch Interaktionen koordinieren. Mögliche Interaktionsformen sind:

Instantiierung: Beim Start eines MIMD-Programms müssen die zugehörigen SISD-Programme als Prozesse instantiiert werden. Unter Umständen sind weitere Instantiierungen im Verlauf der Programmabarbeitung notwendig.

Kommunikation: Die Prozesse eines MIMD-Programms können in der Regel nicht völlig autonom voneinander arbeiten, sondern müssen untereinander Daten austauschen.

Synchronisation: Da die Abarbeitung verschiedener Prozesse auf einer MIMD-Plattform asynchron erfolgt, müssen Prozesse ihren Kontrollfluß synchronisieren können, wenn dies aufgrund der Programmlogik erforderlich ist (z.B. zur Kommunikation).

Die Methoden zur Realisierung von Interaktionen zwischen Prozessen werden durch das Betriebssystem bzw. durch zusätzliche Systemsoftware bereitgestellt.

Die Ausführung von MIMD-Programmen muß nicht gezwungenermaßen auf einer MIMD-Hardware stattfinden, sondern kann auch auf virtuellen MIMD-Plattformen erfolgen. Dazu werden häufig SISD-Systeme in Verbindung mit Multitasking- oder Multithreading-Systemen eingesetzt, welche die nur einmal vorhandene Ressource „Prozessor“ mit Hilfe eines Schedulers zwischen mehreren konkurrierenden Tasks (Prozesse) bzw. Threads (Leichtgewichtsprozesse) aufteilen. Damit können MIMD-Programme auf solchen Plattformen quasi-parallel ausgeführt werden.

Die Erstellung von MIMD-Programmen erfolgt entweder durch explizite parallele Programmierung oder durch automatische Parallelisierung von sequentiellen

[6] Enthält eine MIMD-Architektur Prozessoren die eine daten-parallele Verarbeitung unterstützen (Vektor- oder Arrayprozessoren), können in entsprechender Weise auch mehrere SIMD-Programme ablaufen.

Programmen. Ein allgemeiner Ansatz zur impliziten parallelen Programmierung wie bei SIMD-Plattformen existiert für MIMD-Systeme nicht.

2.3.2.1 Explizite Programmierung von MIMD-Systemen

Im Mittelpunkt der expliziten parallelen Programmierung stehen die zwischen den Prozessen eines MIMD-Programms notwendigen Kommunikations- und Synchronisationsoperationen. Im Abschnitt 2.2.1.2 wurde erwähnt, daß MIMD-Architekturen bezüglich der Realisierung solcher Operationen in Systeme mit oder ohne gemeinsamen Speicher eingeteilt werden. Dementsprechend lassen sich auf der Ebene der Programmierung zwei korrespondierende Paradigmen unterscheiden:

speicher-orientiert[7]**:** Die Kommunikation wird über globale Variablen, auf die alle beteiligten Prozesse Zugriff haben, realisiert. Zur Vermeidung von fehlerhaften Daten oder Blockierungen müssen gleichzeitige Zugriffe mehrerer Prozesse auf gemeinsame Speicherbereiche durch Synchronisationsmaßnahmen (z.B. Semaphore oder Monitore, s. [20, 38, 11]) verhindert werden. Die Synchronisation zwischen den Prozessen ist damit bei der speicher-orientierten MIMD-Programmierung in erster Linie eine Voraussetzung für die Kommunikation.

nachrichten-orientiert[8]**:** Der Datenaustausch zwischen den Prozessen erfolgt durch Senden und Empfangen von Nachrichten. Synchronisation ist beim nachrichtenorientierten Datenaustausch nicht die primäre Voraussetzung, sondern eine Folge der Kommunikation. So kann in einem Prozeß eine bestimmte Nachricht erst empfangen werden, wenn sie vom sendenden Prozeß tatsächlich abgeschickt wurde.

Die Zuordnung des speicher- bzw. nachrichten-orientierten Programmierparadigmas zu MIMD-Systemen mit bzw. ohne gemeinsamen Speicher ist so zu verstehen, daß es sich um das jeweils *native* Paradigma der Architekturklasse handelt. Da sich beide Verfahren aufeinander abbilden lassen (virtueller gemeinsamer Speicher oder Nachrichtenaustausch über gemeinsamen Speicher, s. [7, 81]), sind sie prinzipiell unabhängig von der konkreten Architektur eines MIMD-Systems verwendbar.

Darüber hinaus können durch Erhöhung des Abstraktionsgrades weitere parallele Programmiermodelle abgeleitet werden, in denen die Kommunikation und Synchronisation für den Anwender völlig transparent erfolgt (z.B. Linda [16], Remote-Procedure-Calls [18]).

Für die Erstellung von MIMD-Programmen existiert eine größere Zahl von Entwicklungsumgebungen, die sich in drei verschiedene Gruppen einteilen lassen:

Sequentielle Sprachen in Verbindung mit parallelen Toolkits:
Als Programmiersprachen werden meist Fortran 77 oder C verwendet. Die für Prozeßinteraktionen erforderlichen Methoden werden durch parallele Toolkits

[7] engl. *shared memory programming*
[8] engl. *message passing programming*

als Bestandteil des Betriebssystems oder als eigenständige Programmbibliotheken bereitgestellt und über System- bzw. externe Funktionsaufrufe eingebunden. Umfangreiche Zusammenstellungen von parallelen Toolkits für verschiedene Programmiermodelle befinden sich in [86] und [17].

Um parallele Features erweitere sequentielle Sprachen: In der Mehrzahl handelt es sich um plattformspezifische Fortran-Dialekte der Hersteller von MIMD-Rechnern (z.B. Encore Parallel Fortran, Cray Fortran; s. [29]). Neben den gewöhnlichen sequentiellen Ausdrucksmöglichkeiten enthalten diese Sprachen zusätzliche parallele Konstrukte (`PARALLEL, DO ALL,` u.a.).

Parallele Sprachen: Einige Programmiersprachen wurden von vornherein für die parallele Programmierung entworfen. Es gibt also keine sequentiellen Vorläufer. Die Verbreitung der meisten parallelen Sprachen ist gering. Bekanntere Vertreter sind Ada ([5, 78]) und Occam ([42, 13]).

2.3.2.2 Automatische Parallelisierung

Wie bereits im Abschnitt 2.3.1.3 für SIMD-Plattformen beschrieben, besteht auch für MIMD-Systeme der Wunsch, vorhandene sequentielle Programme möglichst automatisch zu parallelisieren.

Im Unterschied zur Vektorisierung werden bei der Parallelisierung Programmschleifen nicht zu Vektoroperationen zusammengefaßt, sondern die einzelnen Durchläufe einer Programmschleife auf verschiedene Prozessoren verteilt. Dazu müssen ebenfalls eventuell vorhandene Datenabhängigkeiten erkannt und nach Möglichkeit aufgelöst werden.

Eine vollautomatische Generierung parallelen Kodes aus sequentiellen Programmquellen mit zufriedenstellender Qualität ist ähnlich problematisch wie die automatische Vektorisierung. Nutzerinteraktionen sind deshalb auch beim Einsatz von automatisch parallelisierenden Compilern (z.B. EPF-Compiler, s. [29]) notwendig.

2.4 Methoden zur Leistungsbewertung paralleler Systeme

Bei der Bewertung von parallelen Systemen muß grundsätzlich zwischen der Leistungsfähigkeit der Hardware und der Leistungsfähigkeit der eingesetzten Algorithmen unterschieden werden.

Wichtige Bewertungskriterien für eine parallele Hardware sind deren *Rechen-* und *Kommunikationsleistung*. Hersteller von Rechnersystemen geben hierfür gewöhnlich maximale und damit nur theoretisch erreichbare Ausführungs- und Transferraten an.[9]

[9] Rechenleistung in Mips und Mflops (*millions of instructions* bzw. *floating point operations per second*); Kommunikationsleistung in Mbytes/s (*megabytes per second*)

Für eine realistische Bewertung sind jedoch anwendungstypische Leistungsdaten, die mit Benchmark-Programmen (z.B Whetstone, Dhrystone, LINPACK u.a., s. [88]) ermittelt wurden, geeigneter.

Darüber hinaus muß beachtet werden, daß die *effektive Rechen-* und *Kommunikationsleistung*, d.h. die Leistung, die zur Abarbeitung eines Programms tatsächlich zur Verfügung steht, von verschiedenen Einflußfaktoren, wie zum Beispiel dem Overhead der Systemsoftware und der Systembelastung durch andere Nutzer bzw. gleichzeitig laufende Anwendungen, abhängt und stark schwanken kann.

Unabhängig von konkreten Leistungsdaten kann eine qualitative Einordnung von Rechnersystemen bezüglich des Verhältnisses von Kommunikationsleistung zu Rechenleistung, wie in Abbildung 2.7 dargestellt, erfolgen, da dieses Verhältnis für unterschiedliche Parallelverarbeitungsmodelle signifikant verschieden ist.

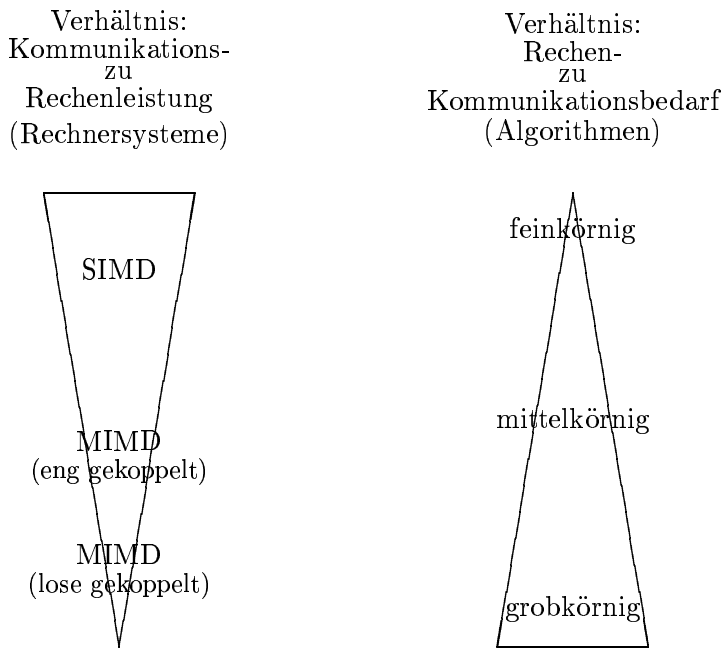


Abbildung 2.7: Qualitative Einordnung von Rechnersystemen und Algorithmen

In analoger Weise kann die Anzahl von Rechen- und Kommunikationsoperationen in parallelen Algorithmen zueinander in Beziehung gesetzt werden (*Granularität*). Aus der Gegenüberstellung der beiden Verhältnisse ist ersichtlich, daß SIMD-Plattformen nur für fein- bis mittelkörnige, und MIMD-Plattformen nur für grob- bis mittelkörnige Algorithmen geeignet sind.

Der tatsächliche Nutzen der Parallelverarbeitung für eine konkrete Problemstellung läßt sich anhand des Gewinns der parallelen Lösung gegenüber einer gleichwertigen sequentiellen Lösung messen. Je nach Zielstellung muß dabei zwischen dem Parallelitätsgewinn (*Speedup*) und dem Skalierungsgewinn (*Scaleup*) unterschieden werden.

Der Speedup gibt an, um wieviel mal schneller ein Problem auf p Prozessoren gelöst werden kann, als auf nur einem Prozessor und ist definiert als Verhältnis der Ausführungszeit des sequentiellen Algorithmus T_s und der Ausführungszeit des parallelen Algorithmus T_p:

$$S_p = \frac{T_s}{T_p}.$$

Da der Implementation eines parallelen Algorithmus immer ein gewisser Overhead inhärent ist, gilt bei praktischen Anwendungen

$$T_s < T_1$$

und damit

$$S_1 < 1.$$

Soll ausschließlich der parallele Algorithmus hinsichtlich des Parallelitätsgewinns untersucht werden, so muß der *algorithmische Speedup*

$$\bar{S}_p = \frac{T_1}{T_p}$$

bestimmt werden.

Idealerweise steigt der algorithmische Speedup linear mit der Anzahl der Prozessoren an. Praktisch kann ein solcher linearer Speedup ($\bar{S}_p = p$) jedoch nicht erreicht werden, da ein zu lösendes Problem immer aus einem parallelisierbaren und einem sequentiellen (nichtparallelisierbaren) Teil besteht. Diese Tatsache wird als *Amdahls Gesetz* ([2]) bezeichnet. Danach gilt für die Ausführung eines parallelen Programms mit einem sequentiellen Anteil s auf p Prozessoren:

$$T_p = sT_1 + (1-s)\frac{T_1}{p}$$

und damit

$$\bar{S}_p = \frac{T_1}{T_p} = \frac{p}{1+s(p-1)}.$$

Ein Maß für die *Effizienz* eines parallelen Algorithmus ist das Verhältnis von tatsächlichem und idealem Speedup:

$$E_p = \frac{\bar{S}_p}{p}.$$

Gemäß Amdahls Gesetz gilt für den Wertebereich der so definierten Effizienz[10]:

$$\frac{1}{p} \leq E_p \leq 1.$$

Ist das Ziel der Parallelverarbeitung nicht die Lösung eines Problems in kürzerer Zeit, sondern die Lösung komplexerer Probleme in der gleichen Zeit, wird anstelle des

[10]Häufig wird die Effizienz auch in Prozentzahlen angegeben ($E_p = 1 \equiv 100\%$).

Parallelisierungs- der Skalierungsgewinn zur Leistungsbewertung benutzt. Bei dieser Betrachtungsweise ist die Ausführungszeit eines parallelen Algorithmus nicht nur von der Anzahl der verwendeten Prozessoren, sondern auch von der Problemkomplexität abhängig. So ist der Scaleup für ein Problem der Komplexität n auf p Prozessoren gegenüber einem Problem der Komplexität m ($m < n$) auf einem Prozessor wie folgt definiert:

$$\text{Wenn} \quad T_1(m) = T_p(n), \quad \text{dann beträgt der Scaleup:} \quad SC_p = \frac{n}{m}.$$

Die Komplexität ist nicht näher definierbar, da sie problemabhängig ist (z.B. verwendete Rechengenauigkeit, Größe des untersuchten Datenbereiches u.ä.).

Weitere Maße und Methoden zur Leistungsbewertung werden u.a. in [45] und [56] diskutiert.

2.5 Zusammenfassung

Parallelverarbeitung beruht grundsätzlich auf der Ausnutzung von synchroner oder asynchroner Parallelität. In diesem Sinne kann das Gebiet der Parallelverarbeitung in zwei große Teilbereiche gegliedert werden, deren wesentliche Charakteristika in Tabelle 2.2 zusammenfassend dargestellt sind.

	Parallelverarbeitung	
	synchron	**asynchron**
Hardware	SIMD-Systeme	MIMD-Systeme
Programmierung	daten-parallele Vektoroperationen; ein Kontrollfluß	asynchrone Prozesse; mehrere Kontrollflüsse
Algorithmen	fein- bis mittelgranulär	grob- bis mittelgranulär

Tabelle 2.2: Synchrone versus asynchrone Parallelverarbeitung

Aus technischer Sicht sind beide Teilbereiche der Parallelverarbeitung für allgemeine wissenschaftlich-technische Anwendungen relevant, da diese einerseits zu einem großen Teil aus feingranulären vektoriellen Operationen bestehen und andererseits häufig grobgranuläre Problemstrukturen aufweisen (z.B. Monte-Carlo-Studien, Evolutionsalgorithmen u.a.).

Der breiten Nutzung der synchronen Parallelverarbeitung steht aber entgegen, daß zur Zeit noch keine SIMD-Plattformen für einen größeren Anwenderkreis zur Verfügung stehen.

MIMD-Plattformen können dagegen, aufgrund der immer stärkeren Verbreitung von Mehrprozessor-Maschinen und Computerclustern, heute als allgemein verfügbar

gelten. Einschränkend muß jedoch bemerkt werden, daß der auf Mehrprozessor-Maschinen erzielbare Parallelitätsgrad noch sehr gering ist, da diese in der Regel nur zwei, vier oder acht Prozessoren besitzen. Bei der Anwendung von Computerclustern kann zwar ein höherer Parallelitätsgrad erreicht werden, dafür ist die Programmierung aber im wesentlichen auf das nachrichten-orientierte Paradigma eingeschränkt, da aufgrund der relativ geringen Kommunikationsleistung eine effiziente Implementierung eines virtuellen gemeinsamen Speichers äußerst schwierig ist.

Vorläufig ist damit für einen größeren Anwenderkreis vor allem die Nutzung der asynchronen Parallelverarbeitung nach dem nachrichten-orientierten Paradigma innerhalb von wissenschaftlich-technischen Berechnungs- und Visualisierungssystemen (SCEs) von Interesse.

Kapitel 3

Message-Passing-Systeme

Message-Passing-Systeme sind Softwaresysteme, die die explizite Programmierung von MIMD-Architekturen nach dem nachrichten-orientierten Paradigma unterstützen. Sie gehören zur Gruppe der parallelen Toolkits und sind üblicherweise als Programmbibliotheken realisiert, die in konventionelle Programmiersprachen eingebunden werden (C, Fortran etc.). Portable Implementationen eignen sich für den gesamten Bereich der MIMD-Architekturen (Computercluster, Multiprozessorsysteme mit und ohne gemeinsamen Speicher) und sind damit besonders für die Erstellung plattformunabhängiger Anwendungen geeignet.

Aufgrund der großen Anzahl von existierenden Systemen (siehe [86, 17]) ist es außerordentlich schwer, den Stand der Technik in diesem Bereich zu identifizieren. Zur Selektion der wichtigsten Systeme schlägt Mattson in [52] folgende Kriterien vor:

- *Supported - either formally or informally.*
- *In heavy use outside the group that created them.*
- *Available on multiple parallel architectures.*

Nach diesen Kriterien ermittelt er als Standard-Systeme: P4, PVM, TCGMSG und C-Linda (s. [14, 31, 36, 16]). Bereits unmittelbar nach Beendigung seiner Untersuchungen (1995) räumt Mattson ein, daß auch erste Implementierungen des offiziellen Message-Passing-Standards (MPI, s. [77]) die angesetzten Kriterien erfüllen.

Bei der von Mattson vorgenommenen Auswahl muß weiterhin berücksichtigt werden, daß sich seine Untersuchungen nicht nur auf Message-Passing-Systeme beschränkten, sondern auch andere parallele Toolkits berücksichtigt wurden. So unterstützt beispielsweise das System C-Linda nicht das nachrichten-orientierte Paradigma, sondern ein assoziatives Speichermodell.

Da im Rahmen der vorliegenden Arbeit in erster Linie Message-Passing-Systeme von Interesse sind, wurden für eine nähere Untersuchung die Systeme P4, TCGMSG, PVM und der Standard MPI ausgewählt.

Die Entwicklungsgeschichte von P4 reicht über den Vorgänger m4 bis ins Jahr 1984 zurück. Seit 1989 wird das System von Butler (University of North Florida) und Lusk (Argonne National Laboratory) gepflegt.

Das System TCGMSG wurde am Argonne National Laboratory aus dem P4-Vorgänger m4 entwickelt und wird heute am Pacific Northwest Laboratory weitergepflegt.

Die Entwicklung von PVM erfolgt seit 1989 an der Emory University und am Oak Ridge National Laboratory. In den Jahren 1993 und 1994 erfuhr PVM mit der Version 3.3 aufgrund der Vielzahl von unterstützten MIMD-Plattformen weltweit sehr große Verbreitung und stellt seitdem einen "Quasi-Standard" dar.

MPI ist das Ergebnis eines Standardisierungsprozesses, der im April 1992 begann. Nach der offiziellen Verabschiedung des Standards in der Version 1.0 (kurz: MPI-1) im Juni 1994 wurden verschiedene Implementierungen, teilweise unter Verwendung existierender Message-Passing-Systeme, realisiert (z.B. MPICH auf Basis von PVM oder P4, s. [21]). Da der Standard eine Grundlage für hochportable Implementierungen bieten soll, wurden diesbezüglich kritische Teilbereiche, wie z.B. die Prozeßverwaltung, in MPI-1 noch nicht berücksichtigt. Die Definition solcher Aspekte soll nach Sammlung ausreichender Erfahrungen in weiteren Versionen des MPI-Standards erfolgen[1].

Im folgenden werden die wesentlichen Eigenschaften der genannten Message-Passing-Systeme sowie des Standards MPI-1 untersucht und anschließend eine vergleichende Bewertung vorgenommen.

3.1 Prozeßverwaltung

Die Prozeßverwaltung von Message-Passing-Systemen wird maßgeblich durch die Art und Weise der Prozeßinstantiierung und -identifikation bestimmt.

3.1.1 Prozeßinstantiierung

Im Gegensatz zu SISD-Programmen sind beim Starten von MIMD-Programmen verschiedene Szenarien möglich (s. Abb. 3.1).

Um ein SISD-Programm zu starten, muß dieses als einzelner Prozeß instantiiert werden. Der Start eines MIMD-Programms erfordert dagegen die Instantiierung mehrerer Prozesse. Wie dies konkret erfolgen muß, wird maßgeblich durch das vom MIMD-Programm implizierte Prozeßmodell bestimmt.

Nach dem SPMD-Modell (*single program multiple data*) besteht ein MIMD-Programm zur Laufzeit aus mehreren identischen Prozessen (auf der Basis einer einzigen Programmdatei), deren Anzahl über die gesamte Laufzeit konstant ist. Der Start eines SPMD-Programms kann damit durch eine *statische Instantiierung* der Prozesse erfolgen.

Das allgemeinere MPMD-Modell (*multiple programs multiple data*) läßt dagegen ein Ensemble aus unterschiedlichen Prozessen (auf der Basis verschiedener Programmdateien) zu. Sollen alle Prozesse während der gesamten Laufzeit des Programms existent sein, so kann dies ebenfalls durch statische Prozeßinstantiierung realisiert werden.

[1] Die Verabschiedung von MPI-2 ist für 1997 geplant.

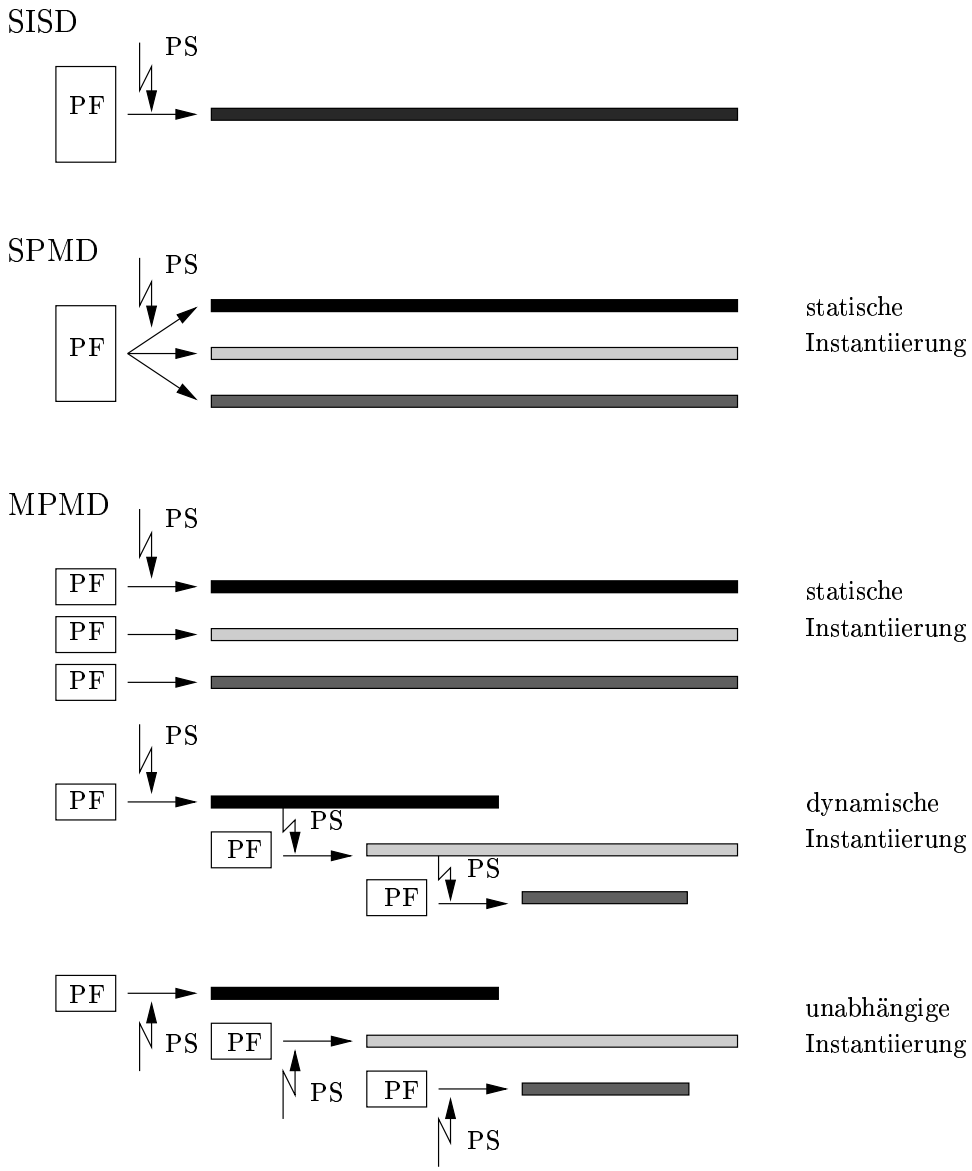


Abbildung 3.1: Prozeßinstantiierung bei SISD, SPMD und MPMD-Programmen

Ist die Anzahl der Prozesse während der Programmabarbeitung variabel, d.h. das Programm selbst kann zur Laufzeit weitere Prozesse erzeugen, handelt es sich um eine *dynamische Instantiierung*.

Letztendlich können auch Prozesse ein MPMD-Ensemble bilden, die völlig unabhängig voneinander erzeugt wurden (*unabhängige Instantiierung*).[2]

Keines der analysierten Message-Passing-Systeme bietet eine konsistente Unterstützung aller Instantiierungsmethoden. So sehen P4 und TCGMSG nur die statische Instantiierung von Prozessen vor. Damit sind diese Systeme nur im eingeschränkten Maße für MPMD-Programme geeignet. Der Standard MPI-1 definiert keine Funktionen zur Prozeßverwaltung, basiert aber konzeptionell ebenfalls auf der statischen Prozeßinstantiierung. Bei PVM hängen die möglichen Varianten davon ab, ob der Start eines MIMD-Programms von einer Standard- oder von einer Spezial-Shell (PVM-Shell) aus erfolgt. In jedem Falle wird die dynamische und die unabhängige Instantiierung von Prozessen unterstützt. Eine statische Instantiierung ist nur für SPMD-Programme von der PVM-Shell aus möglich.

Der Aufruf von SISD-Programmen mit Hilfe von Standard-Shells erfolgt für den Nutzer völlig transparent, d.h. es muß nur der Name der gewünschten Programmdatei angegeben werden. Die eigentliche Prozeßinstantiierung realisiert das Shell-Programm mit Hilfe von Betriebssystemfunktionen.

MIMD-Programme mit dynamischer oder unabhängiger Instantiierung (nur PVM) können ebenfalls direkt von einer Standard-Shell aus gestartet werden, weil durch die Shell jeweils nur ein einzelner Prozeß instantiiert werden muß.

Problematischer ist das Starten von MIMD-Programmen mit statischer Instantiierung (P4, TCGMSG und PVM), da Standard-Shells nicht in der Lage sind, mittels einer Operation mehrere Prozesse zu instantiieren. Dieses Problem wird in den analysierten Message-Passing-Systemen unterschiedlich gelöst:

P4 sieht eine zusätzliche Beschreibungsdatei vor, in der entweder die Anzahl der Prozesse eines SPMD-Programms bzw. alle zu einem MPMD-Programm gehörigen Programmdateien zu spezifizieren sind. Der Programmstart erfolgt durch Instantiierung eines „ersten Prozesses“ über eine Standard-Shell. Innerhalb des Prozesses wird durch Aufruf einer Initialisierungsfunktion die Steuerung an das P4-Laufzeitsystem übergeben, welches die Beschreibungsdatei einliest und daraufhin die restlichen Prozesse instantiiert.

TCGMSG verwendet ebenfalls eine Beschreibungsdatei. Aus einer Standard-Shell heraus wird aber nicht ein „erster Prozeß“ des MIMD-Programms gestartet, sondern ein spezielles TCGMSG-Startprogramm, welches die Beschreibungsdatei einliest und dann sämtliche Prozesse des MIMD-Programms instantiiert.

PVM unterstützt, wie bereits erwähnt, keine direkte statische Instantiierung von MIMD-Programmen aus einer Standard-Shell heraus. Mit Hilfe der PVM-Shell können aber von einer Programmdatei mehrere Prozesse gestartet wer-

[2] Die unabhängige Instantiierung ist vor allem für die verteilte Verarbeitung bedeutsam, weniger für die Parallelverarbeitung.

den, womit die statische Instantiierung von SPMD-, jedoch nicht von MPMD-Programmen möglich ist.

Neben der Prozeßanzahl (SPMD) bzw. den zugehörigen Programmdateien (MPMD) sind für die Instantiierung von MIMD-Programmen noch weitere Informationen erforderlich. So muß spezifiziert werden, wo die Prozesse zu instantiieren sind (*process to processor mapping*). Im Zusammenhang mit dem Prozeß-Mapping sind meist weitere Informationen bezüglich der Konfiguration der MIMD-Plattform erforderlich.

Das Prozeß-Mapping kann explizit oder transparent erfolgen. P4 und TCGMSG unterstützen nur explizites Prozeß-Mapping, d.h. der Nutzer muß eine eindeutige Zuordnung von Prozessen zu Prozessoren spezifizieren. PVM ermöglicht sowohl explizites als auch transparentes Prozeß-Mapping. Bei transparentem Prozeß-Mapping erfolgt die Prozeß-Prozessor-Zuordnung automatisch durch das Message-Passing-System. Intelligente Algorithmen können Zuordnungen ermitteln, die eine möglichst gleichmäßige Auslastung der Prozessoren ergeben (Lastbalancierung).

Die Konfigurationsmöglichkeiten der MIMD-Plattform sind je nach Architektur sehr unterschiedlich. So erlauben Multiprozessorsysteme oftmals die Allokation der vorhandenen Prozessoren in Teilmengen oder die Modifikation der Topologie des Verbindungsnetzwerkes. Bei der Konfiguration von Computerclustern geht es dagegen vor allem darum, welche Rechner des Clusters zu einer MIMD-Plattform zusammengefaßt werden sollen. Unabhängig von den konkreten Konfigurationsparametern einer MIMD-Architektur, kann man zwischen statischer (P4, TCGMSG) und dynamischer Konfiguration (PVM) unterscheiden. Bei der statischen Konfiguration müssen alle Parameter vor bzw. beim Start eines MIMD-Programms festgelegt werden. Die dynamische Konfiguration ermöglicht dagegen die Änderung von Parametern durch das MIMD-Programm selbst.

Dynamische Konfiguration eröffnet vor allem für MIMD-Plattformen auf Basis von Computerclustern die Möglichkeit zur Realisierung von fehler- und lasttoleranten Applikationen, d.h. Programme können ausgefallene oder stark belastete Rechner durch Modifikation der MIMD-Konfiguration „meiden“ bzw. durch eventuell vorhandene „Ausweichrechner“ ersetzen.

3.1.2 Prozeßidentifikation

Prozeßinteraktionen auf der Basis des nachrichten-orientierten Kommunikationsmodells setzen eine eindeutige Identifizierbarkeit aller Prozesse innerhalb eines MIMD-Programms voraus.

Die einfachste Methode zur Prozeßidentifikation ist deren Numerierung. Da bei statischer Instantiierung die Anzahl der Prozesse über die gesamte Programmlaufzeit bekannt und konstant ist, kann hier eine geordnete Prozeßnumerierung erfolgen. Dieses Verfahren wird bei den Systemen P4 und TCGMSG sowie beim Standard MPI-1 eingesetzt. Den Prozessen werden jeweils Nummern im Bereich von 0 bis Prozeßanzahl minus 1 zugewiesen.

Bei dynamischer bzw. unabhängiger Instantiierung ist ebenfalls eine eindeutige Identifizierung der Prozesse durch Numerierung möglich. Aufgrund der veränderlichen

Anzahl von Prozessen kann diese jedoch nicht geordnet erfolgen. Eine Prozeßnumerierung ohne Ordnungsrelation hat aber andererseits den Vorteil, daß eine Prozeßnummer nicht nur zur eindeutigen Identifikation, sondern auch zur Kodierung von prozeßspezifischen Informationen genutzt werden kann. In PVM, wo dieses Verfahren eingesetzt wird, kodiert eine Prozeßnummer neben anderen Informationen beispielsweise den Rechner, auf dem der Prozeß läuft.

PVM und MPI-1 unterstützen im Zusammenhang mit kollektiven Operationen noch weitere Ordnungskonzepte (s. Abschn. 3.3.4).

3.2 Nachrichtenbasierte Kommunikation

Das Anliegen der nachrichtenbasierten Kommunikation ist der Transport von Daten aus dem Adreßraum eines Senders in den Adreßraum eines Empfängers. In Abbildung 3.2 ist zu erkennen, daß eine solche Datentransport-Operation aus den drei grundlegenden Teiloperationen:

- Überführung von Datenobjekten in ein Nachrichtenobjekt,
- Übergabe eines Nachrichtenobjekts (*message passing*) und
- Überführung eines Nachrichtenobjekts in Datenobjekte

besteht.

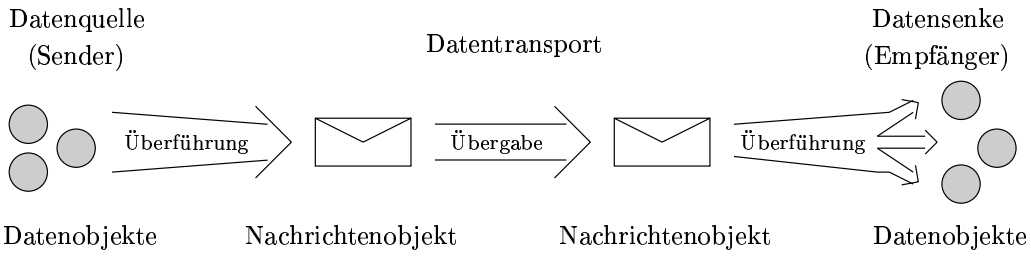


Abbildung 3.2: Transport von Daten durch Nachrichtenübergabe

Da sich die Repräsentation von Datenobjekten gleichen Typs zwischen verschiedenen Architekturen unterscheiden kann (z.B. bei heterogenen Computerclustern als MIMD-Plattform), ist unter Umständen eine vierte Teiloperation zur Konvertierung von Datenrepräsentationen erforderlich.

3.2.1 Übergabe von Nachrichtenobjekten

Alle untersuchten Message-Passing-Systeme stellen für die Nachrichtenübergabe Kommunikationsoperationen in Form von Sende- und Empfangsfunktionen bereit.

Obwohl das nachrichten-orientierte Kommunikationsmodell neben der Übergabe einer Nachricht von einem Sender an einen Empfänger auch eine Übergabe an mehrere Empfänger zuläßt, wird letztere Variante (*multicast*) nur von PVM unterstützt.

In P4, TCGMSG und MPI-1 existiert zwar eine dem Multicast ähnliche Broadcast-Funktion (s. Abschn. 3.3.1), diese basiert aber nicht unmittelbar auf dem nachrichtenorientierten Kommunikationsmodell, sondern auf dem abstrakteren Modell kollektiver Operationen.

3.2.1.1 Sequentielle und nebenläufige Kommunikationsoperationen

Die Kommunikationsoperationen eines Prozesses werden auf den meisten Plattformen wie gewöhnliche Operationen sequentiell ausgeführt. Einige spezielle MIMD-Plattformen (z.B. Intel iPSC und DELTA) unterstützen aber auch eine nebenläufige Ausführung von Kommunikations- und Rechenoperationen.

Im Standard MPI-1 werden sequentielle Kommunikationsoperationen als *blockierende* und nebenläufige als *nicht-blockierende* Operationen bezeichnet. Diese Begriffe können, insbesondere wenn sie auf korrespondierende Funktionsimplementationen angewandt werden, leicht zu Fehlinterpretationen führen. Beispielsweise bedeutet der Aufruf einer blockierenden Kommunikationsfunktion nicht zwangsläufig, daß dadurch auch der aufrufende Prozeß blockiert wird. Dies hängt allein davon ab, ob die aufgerufene Funktion eine lokale oder nicht-lokale Semantik hat.

Die vorliegende Arbeit geht von folgenden Begriffsinterpretationen aus:

Lokale Operationen bzw. Funktionen können innerhalb eines Prozesses vollständig ausgeführt und unabhängig von externen Bedingungen oder Ereignissen beendet werden. Sie können den ausführenden Prozeß nicht blockieren.

Nicht-lokale Operationen bzw. Funktionen sind mit Aktivitäten in anderen Prozessen verknüpft. Ob und wann sie vollständig ausgeführt und beendet werden können hängt von externen Bedingungen und Ereignissen ab. Damit ist unter Umständen eine Blockierung des ausführenden Prozesses möglich.

Blockierende Kommunikationsoperationen werden wie gewöhnliche Operationen innerhalb eines Prozesses sequentiell ausgeführt. Die Implementation erfolgt durch Funktionen mit lokaler oder nicht-lokaler Semantik. Die Rückkehr einer blockierenden Kommunikationsfunktion signalisiert, daß die gewünschte Operation ausgeführt und beendet wurde.

Nicht-blockierende Kommunikationsoperationen werden innerhalb eines Prozesses zu anderen Operationen nebenläufig ausgeführt. Die Implementation von nicht-blockierenden Kommunikationsoperationen erfolgt durch jeweils eine Post- und eine Completion-Funktion. Während Post-Funktionen immer eine lokale Semantik besitzen, können Completion-Funktionen lokalen oder nicht-lokalen Charakter haben. Die Rückkehr eines Post-Funktionsaufrufs signalisiert lediglich, daß die Ausführung einer Kommunikationsoperation veranlaßt wurde (*request posting*). Erst ein rückkehrender Completion-Funktionsaufruf garantiert, daß die Operation vollständig ausgeführt und beendet wurde (*request completion*).

MPI-1 und TCGMSG unterstützen sowohl blockierende als auch nicht-blockierende Kommunikationsoperationen. P4 und PVM stellen dem Anwender dagegen nur blockierende Kommunikationsoperationen zur Verfügung.

3.2.1.2 Sendeoperationen

Traditionell wird bei der nachrichtenbasierten Kommunikation zwischen *synchroner* und *asynchroner Nachrichtenübergabe* unterschieden ([29]). Bei näherer Betrachtung zeigt sich aber, daß die Synchronisationsoption nur dem Sender, nicht aber dem Empfänger zur Verfügung steht. Ist die vollständige Ausführung einer Sendeoperation an die gleichzeitige Ausführung einer Empfangsoperation geknüpft, muß sich der Sender bei einer Nachrichtenübergabe mit dem Empfänger synchronisieren. Werden dagegen Sendeoperationen völlig unabhängig von Empfangsoperationen ausgeführt, ist die Abarbeitung im Sender vom Empfänger entkoppelt, d.h. asynchron zum Empfänger. Umgekehrt ist zumindest eine schwache Synchronisation[3] des Empfängers mit dem Sender unvermeidlich, da eine Empfangsoperation nur abgeschlossen werden kann, wenn auch eine Sendeoperation ausgeführt wurde. Es ist daher treffender, zwischen synchronen und asynchronen Sendeoperationen zu unterscheiden.

Synchrone Sendeoperationen haben den Vorteil, daß sie sich sehr effizient implementieren lassen, da Nachrichtenobjekte direkt aus dem Adreßraum des Senders in den Adreßraum des Empfängers kopiert werden können. Der Nachteil bei diesem Verfahren ist, daß synchrone Sendeoperationen immer eine nicht-lokale Semantik besitzen und damit potentiell den Sender blockieren können.[4]

Asynchrone Sendeoperationen können dagegen vollständig ausgeführt werden, auch wenn aktuell kein Empfangsprozeß zur Nachrichtenübergabe bereit ist. In diesem Fall können Nachrichtenobjekte aber nicht mehr direkt aus dem Adreßraum des Senders in den Adreßraum des Empfängers kopiert werden, sondern müssen bis zur Ausführung der Empfangsoperation zwischengespeichert werden. Die dafür erforderliche Pufferverwaltung macht die Implementation von sicheren und zugleich effizienten asynchronen Sendeoperationen außerordentlich problematisch.

Wird die Pufferverwaltung statisch implementiert, kann die Zwischenspeicherung zwar effizient erfolgen, es besteht aber die Gefahr einer Ressourcenerschöpfung. Eine dynamische Pufferverwaltung kann dieses Problem in gewissen Grenzen entschärfen, ist dafür aber mit einem erhöhten Overhead verbunden. Für die Implementation von asynchronen Sendeoperationen werden deshalb zwei Strategien, eventuell in Kombination miteinander, angewandt:

- Die Pufferverwaltung wird über eine Schnittstelle dem Anwender zugänglich gemacht. Damit ist es möglich Programme zu implementieren, die eine Ressourcenerschöpfung definiert behandeln bzw. von vornherein einen skalierbaren Ressourcenbedarf haben.

[3] Mit schwacher Synchronisation ist hier gemeint, daß ein Empfänger auch bei asynchroner Nachrichtenübergabe nie den Sender „überholen“ kann.

[4] Dies gilt auch für nicht-blockierende synchrone Sendeoperationen. Hier hat die Completion-Funktion eine nicht-lokale Semantik (s. Abschnitt 3.2.1.1).

- Die Sendefunktionen werden so implementiert, daß sie bei ausreichenden Speicherressourcen asynchron arbeiten und bei erschöpften Ressourcen automatisch auf synchronen Betrieb umschalten.

Aufgrund dieser Praxis ist die traditionelle Einteilung nach synchroner bzw. asynchroner Arbeitsweise für eine hinreichend genaue Charakterisierung realer Implementationen nicht ausreichend. In MPI-1 werden deshalb insgesamt vier Betriebsarten (*communication modes*) für Sendefunktionen definiert.

Im *Synchronous-* und im *Buffered-Mode* ist eine ausschließlich synchrone bzw. asynchrone Arbeitsweise vorgeschrieben. Eine Sendefunktion im Buffered-Mode darf demnach auch bei erschöpften Speicherressourcen nicht in einen synchronen Betrieb umschalten, sondern muß ein solches Ereignis als Fehlerzustand behandeln.

Der *Standard-Mode* erlaubt eine völlig transparente Arbeitsweise der Sendefunktion, d.h. nach ihrer Rückkehr ist lediglich gesichert, daß die Sendeoperation ausgeführt wurde. Damit können Sendefunktionen im Standard-Mode sowohl synchron als auch asynchron arbeiten. Im Unterschied zum Buffered-Mode stellt eine eventuelle Ressourcenerschöpfung im Standard-Mode kein Fehlerereignis dar.

Die vierte Betriebsart *Ready-Mode* ermöglicht eine besonders effiziente Ausführung der gesamten Nachrichtenübergabeoperation. Beim Aufruf einer Sendefunktion im Ready-Mode muß garantiert sein, daß der Empfangsprozeß bereits zur Nachrichtenübergabe bereit ist, d.h. schon die Empfangsfunktion aufgerufen hat. Bei einer solchen Konstellation kann das Message-Passing-System ein Übergabeprotokoll mit sehr geringem Overhead für den Nachrichtentransfer zwischen Sender und Empfänger verwenden.

Der Standard MPI-1 sieht für jede Betriebsart die Implementation einer separaten Sendefunktion vor und definiert eine Schnittstelle zur Pufferverwaltung, über die die Speicherressourcen für asynchrone Sendeoperationen (Buffered-Mode) einstellbar sind.

Die untersuchten Message-Passing-Systeme unterstützen jeweils nur eine Untermenge der in MPI-1 für Sendeoperationen definierten semantischen Varianten:

TCGMSG stellt nur eine einzige Sendefunktion bereit. Verschiedene Betriebsarten sind durch den Anwender nicht wählbar. Die tatsächliche Arbeitsweise der Sendefunktion hängt von den Eigenschaften des von der jeweiligen Plattform zur Verfügung gestellten Transportmechanismus ab. Unterstützt dieser eine Pufferung, arbeitet die Sendefunktion im Rahmen der zur Verfügung stehenden Ressourcen asynchron - ansonsten synchron.

PVM sieht ebenfalls keine explizite Einstellung bestimmter Betriebsarten vor, besitzt aber eine eigene Pufferverwaltung mit Anwenderschnittstelle. Allerdings wird die Pufferverwaltung von einigen Sendefunktionen nicht benutzt.

P4 stellt jeweils separate Funktionen für synchrone und asynchrone Sendeoperationen sowie eine Schnittstelle zur Pufferverwaltung bereit.

Damit unterstützen TCGMSG und PVM nach dem MPI-Standard nur den Standard-Mode und P4 nur den Synchronous- und Buffered-Mode.

3.2.1.3 Empfangs- und Probeoperationen

Wie bereits im vorangegangenen Abschnitt erwähnt wurde, ergeben sich für Empfangsoperationen aus der Synchronisationsproblematik keine alternativen semantischen Varianten. Da Empfangsoperationen nur ausgeführt und beendet werden können, wenn tatsächlich Nachrichten zur Übergabe von anderen Prozessen bereitgestellt werden, handelt es sich bei ihnen immer um nicht-lokale Operationen. Dementsprechend besteht beim Empfang von Nachrichten potentiell die Gefahr der Blockierung des ausführenden Prozesses[5].

Um dieses Problem abzuschwächen stellen alle untersuchten Message-Passing-Systeme Probeoperationen zur Verfügung, mit denen getestet werden kann, ob eine Nachricht zum Empfang vorliegt oder nicht. In MPI-1, P4 und PVM liefern sie darüber hinaus, im Falle einer vorhandenen Nachricht, zusätzliche nachrichtenspezifische Informationen, wie zum Beispiel den Absender und die Länge der Nachricht.

MPI-1 und PVM enthalten neben diesen Basisoperationen, der nicht-lokalen Empfangs- und der lokalen Probeoperation, noch weitere abgeleitete Varianten.

So wird in PVM zwischen *unbedingten* und *bedingten Empfangsoperationen* unterschieden. Die unbedingten Empfangsoperationen entsprechen exakt der besprochenen nicht-lokalen Basisvariante. Bedingte Empfangsoperationen besitzen dagegen eine lokale Semantik, d.h. sie werden auch beendet, wenn keine Nachricht zum Empfang vorliegt.

MPI-1 sieht eine zweite Probeoperation mit nicht-lokaler Semantik vor. Bei dieser geht es nicht mehr darum, zu testen, ob eine Nachricht zum Empfang vorliegt, sondern allein um die nachrichtenspezifischen Informationen[6].

Während Sendeoperationen immer auf eine bestimmte Nachricht, die an einen oder mehrere festgelegte Empfänger übergeben werden soll, angewendet werden, gibt es bei Empfangs- und Probeoperationen die Möglichkeit, Nachrichten in Abhängigkeit vom Absender und/oder von bestimmten Markierungen selektiv zu behandeln.

Zur Absenderselektion wird bei allen untersuchten Message-Passing-Systemen die Prozeßnummer des Senders benutzt (s. Abschn. 3.1.2). Mit Hilfe eines speziellen NULL-Identifikators kann die Absenderselektion deaktiviert werden.

Für die Markierung von Nachrichten sehen alle Systeme einen Integer-Identifikator (*message tag*) vor, der vom Sender gesetzt und beim Empfang bzw. Proben zur Selektion benutzt wird. Außer in TCGMSG kann auch diese Selektion mit einem NULL-Tag deaktiviert werden.

MPI-1 sieht neben der Nachrichtenmarkierung mit Integer-Identifikatoren noch die Verknüpfung einer Nachricht mit einer Kommunikationsdomäne vor. Für die nachrichtenbasierte Kommunikation erbringen Kommunikationsdomänen gegenüber

[5]Dies gilt auch für nicht-blockierende (nebenläufige) Empfangsoperationen. Hier hat die Completion-Funktion eine nicht-lokale Semantik (s. Abschnitt 3.2.1.1).

[6]In MPI wird die lokale Basisoperation als nicht-blockierende Probe und die nicht-lokale Variante als blockierende Probe bezeichnet, was sehr irreführend ist, da diese Begriffe zur Unterscheidung der sequentiellen und nebenläufigen Ausführung von Kommunikationsoperationen benutzt werden (s. Abschnitt 3.2.1.1). In diesem Sinne sind aber beide MPI-Probeoperationen blockierend, d.h. sie werden sequentiell abgearbeitet.

der Nachrichtenmarkierung mit Integer-Identifikatoren aber keine signifikanten Vorteile.

3.2.2 Überführung von Datenobjekten in ein Nachrichtenobjekt

Wie bereits erwähnt ist das Anliegen der nachrichtenbasierten Kommunikation der Transport von Datenobjekten zwischen den Adreßräumen verschiedener Prozesse. Innerhalb eines Prozesses wird ein Datenobjekt durch eine Referenz und einen Datentyp eindeutig spezifiziert. Die Referenz - üblicherweise eine Speicheradresse - gibt an, wo sich das Datenobjekt im Adreßraum befindet. Der Datentyp bestimmt, wie ein Datenobjekt zu interpretieren ist.

Message-Passing-Systeme können nicht ohne weiteres Datenobjekte beliebigen Typs oder mehrere Datenobjekte auf einmal transportieren, sondern nur einzelne Nachrichtenobjekte, wobei unter einem Nachrichtenobjekt ein durch das Message-Passing-System direkt transportierbares Datenobjekt verstanden werden soll.

Nach dem MPI-Standard und in allen analysierten Message-Passing-Systemen stellen nur zusammenhängende Bytefolgen definierter Länge Datenobjekte dar, die direkt transportiert werden können.

Für den Transport eines Datenobjektes, das intern nicht durch eine zusammenhängende Bytefolge repräsentiert wird (z.B. alle Elemente eines Arrays mit gradzahligem Index, dynamische Strukturen u.ä.) existieren prinzipiell zwei Möglichkeiten:

- Das Datenobjekt wird in ein Nachrichtenobjekt konvertiert, d.h. in ein Datenobjekt mit zusammenhängender Bytefolge.
- Der Anwender muß eine genaue Strukturbeschreibung des Datenobjekts zur Verfügung stellen, anhand derer das Message-Passing-System das Datenobjekt direkt als Nachrichtenobjekt behandeln kann.

Während die erste Variante aufgrund des notwendigen Kopiervorgangs bei jedem Datentransport Overhead erzeugt, muß der Aufwand zur Erstellung der Strukturbeschreibung bei der zweiten Variante für gleichartige Datenobjekte nur einmal betrieben werden. Ändert sich die Struktur der zu transportierenden Datenobjekte jedoch dynamisch, ist die Strukturbeschreibung jeweils neu zu erstellen.

TCGMSG und P4 unterstützen keine der beiden Varianten direkt. Da die erste Variante aber auch völlig unabhängig vom Message-Passing-System realisiert werden kann, ist der Transport von Datenobjekten mit nicht-zusammenhängenden Bytefolgen prinzipiell möglich. PVM und MPI unterstützen beide Varianten.

Der gleichzeitige Transport von mehreren separaten Datenobjekten erfolgt in analoger Weise, da es auch hier um die Übertragung von nicht-zusammenhängenden Bytefolgen geht.

3.2.3 Überführung eines Nachrichtenobjektes in Datenobjekte

Um ein Nachrichtenobjekt auf Seiten des Empfängers wieder in Datenobjekte zu überführen, ist die Kenntnis der ursprünglichen Datentypen erforderlich. Diese Typinformationen können zusammen mit den Daten übertragen werden (*explicit typing*) oder sind beim Empfänger a priori bekannt (*implicit typing*).

Alle untersuchten Message-Passing-Systeme sowie der Standard MPI-1 sehen keine automatische Übertragung von Typinformationen vor, d.h. sie unterstützen standardmäßig nur die implizite Typisierung. Ist eine explizite Typisierung erforderlich, liegt es in der Verantwortung des Anwenders, für die Übertragung der notwendigen Typinformationen zu sorgen. Dies kann entweder in Form separater Nachrichten erfolgen oder die Typinformationen werden zusammen mit den Daten in einer Nachricht „verpackt".

3.2.4 Konvertierung der Datenrepräsentation

Diese Operation ist nur beim Transport von Daten zwischen Architekturen mit unterschiedlichen Repräsentationen gleicher Datentypen notwendig. Somit spielt sie nur eine Rolle, wenn heterogene Computercluster als MIMD-Plattformen benutzt werden sollen.

Die Konvertierung kann entweder direkt von der Repräsentation der Sender- in die der Empfängerarchitektur oder über ein architekturunabhängiges Austauschformat (*external data representation*, XDR[7]) erfolgen.

Der Vorteil einer direkten Konvertierung der Datenrepräsentation besteht im geringeren Overhead. Problematisch ist aber, daß für jede Kombination von Rechnerarchitekturen mit unterschiedlicher Datenrepräsentation verschiedene Konvertierungsroutinen erforderlich sind.

Bei der indirekten Konvertierung über das XDR-Format ist dagegen pro Rechnerarchitektur nur ein Satz von Konvertierungsroutinen notwendig, was die Einführung neuer Architekturen außerordentlich vereinfacht.

Für den Anwender ist es wünschenswert, daß ein Message-Passing-System nicht nur die Konvertierung von Datenrepräsentationen unterstützt, sondern diese auch in transparenter Weise abwickelt, d.h. automatisch erkennt, zwischen welchen Architekturen tatsächlich eine Konvertierung erfolgen muß.

Eine solche Transparenz der Konvertierungsoperation wird nur vom Standard MPI-1 vorgeschrieben. Ob eine notwendige Konvertierung direkt oder indirekt über ein Austauschformat erfolgt, bleibt der jeweiligen Implementation überlassen.

Alle anderen untersuchten Message-Passing-Systeme unterstützen eine indirekte Konvertierung über das XDR-Format, die auf der Senderseite explizit angefordert werden muß.

[7]XDR ist eine von Sun Microsystems Inc. entwickelte Spezifikation zur architekturunabhängigen Kodierung von Daten (s. [18]), die vom ARPA Network Information Center als Standard RFC1014 verabschiedet wurde.

3.3 Kollektive Operationen

Im Unterschied zu den bisher betrachteten Operationen basieren kollektive Operationen nicht unmittelbar auf dem nachrichten-orientierten Kommunikationsmodell, nach dem nur der Transport von Datenobjekten von einer Quelle zu einem oder mehreren Zielen möglich ist. Bei kollektiven Operationen können dagegen alle beteiligten Prozesse zugleich Datenquelle als auch Datensenke sein. Ein weiterer prinzipieller Unterschied gegenüber einfachen Nachrichtenübergaben besteht darin, daß Datenobjekte nicht nur transportiert, sondern auch modifiziert werden können. In diesem Sinne gehören kollektive Operationen nicht zu den Basisoperationen eines Message-Passing-Systems - können aber durch diese realisiert werden.

Die in den untersuchten Message-Passing-Systemen unterstützten, kollektiven Operationen lassen sich in die Klassen Transport-, Reduktions- und Synchronisationsoperationen unterteilen.

3.3.1 Transportoperationen

Der Standard MPI-1 definiert insgesamt fünf kollektive Transportoperationen (s. Abb. 3.3), die nach ihrer Interaktionsstruktur in drei Gruppen eingeordnet werden können:

One-to-many

Broadcast: Ein Root-Prozeß liefert ein Datenobjekt, welches an alle beteiligten Prozesse verteilt wird.

Scatter: Ein Root-Prozeß liefert ein Datenobjekt, welches in Portionen zerlegt wird. Alle beteiligten Prozesse erhalten jeweils eine Portion des Datenobjekts.

Many-to-one

Gather: Alle beteiligten Prozesse liefern jeweils ein Datenobjekt. Diese werden zu einem neuen Datenobjekt zusammengesetzt, welches als Ergebnis der Operation in einem Root-Prozeß zur Verfügung steht.

Many-to-many

Allgather: Alle beteiligten Prozesse liefern jeweils ein Datenobjekt. Diese werden zu einem neuen Datenobjekt zusammengesetzt, welches als Ergebnis der Operation allen beteiligten Prozessen zur Verfügung steht.

Alltoall: Alle beteiligten Prozesse liefern jeweils ein Datenobjekt, welches in Portionen zerlegt wird. Alle Portionen werden zu neuen Datenobjekten kombiniert, von denen als Ergebnis der Operation den beteiligten Prozessen jeweils eines zur Verfügung steht.

P4 und TCGMSG stellen nur die Broadcast-Operation zur Verfügung. PVM unterstützt darüber hinaus noch die Operationen Scatter und Gather.

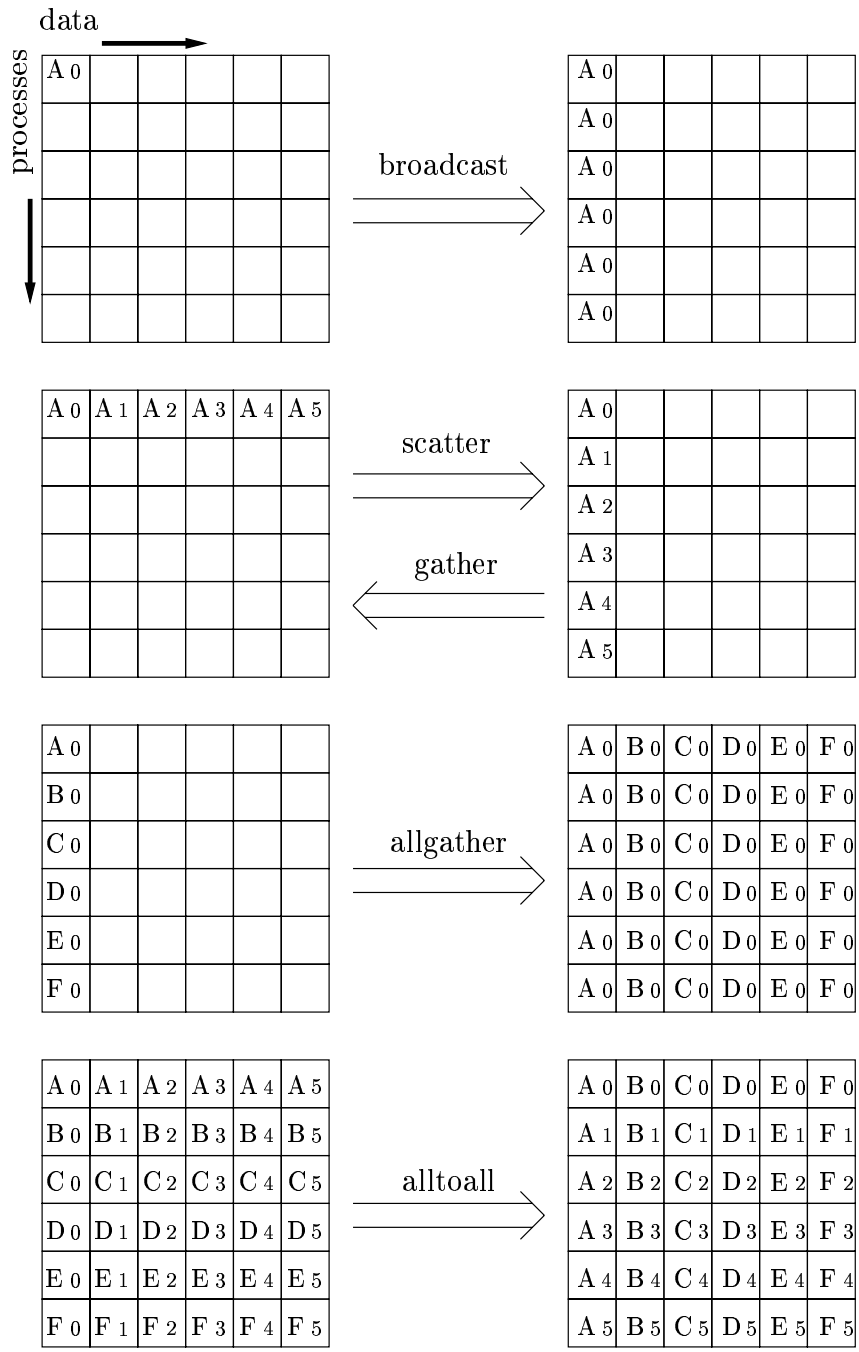


Abbildung 3.3: Kollektive Transportoperationen (nach [77])

3.3.2 Reduktionsoperationen

Kollektive Reduktionsoperationen sind arithmetische oder logische Operationen über Daten mehrerer Prozesse. Sie werden in allen untersuchten Message-Passing-Systemen zweiteilig realisiert, d.h. durch eine generische und eine spezifische Operation.

Die generische Operation bestimmt die Anzahl der Reduktionsergebnisse und in welchen der beteiligten Prozesse diese nach Abschluß der Operation zur Verfügung stehen. Die in MPI-1 definierten generischen Operationen sind in Abbildung 3.4 dargestellt.

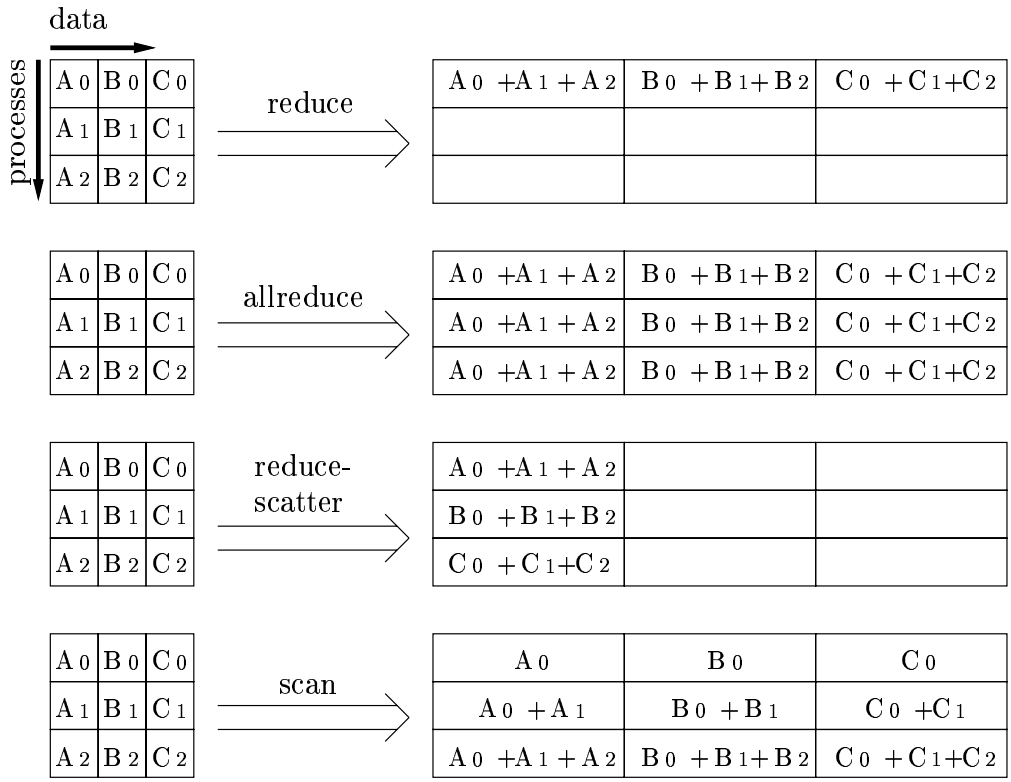


Abbildung 3.4: Generische Reduktionsoperationen (nach [77])

In den untersuchten Message-Passing-Systemen wird jeweils nur eine generische Reduktionsoperation unterstützt (Allreduce in P4 und TCGMSG, Reduce in PVM).

Die spezifische Operation ist die eigentliche arithmetische oder logische Operation, die zur Reduktion der Daten benutzt wird. Für die gebräuchlichsten Operationen, wie Summation, Multiplikation, logische UND/ODER-Verknüpfungen etc., werden bereits Funktionsimplementationen zur Verfügung gestellt. Darüber hinaus können weitere Operationen durch den Anwender definiert werden.

3.3.3 Synchronisationsoperationen

Synchronisationsoperationen dienen zur Koordinierung der primär asynchron erfolgenden Abarbeitung von Prozessen. Sie nehmen innerhalb der Klasse der kollektiven Operationen eine gewisse Sonderstellung ein, da bei diesen Operationen kein Transport von Anwenderdaten erfolgt.

Alle untersuchten Message-Passing-Systeme stellen zur kollektiven Synchronisation jeweils eine Barrier-Funktion zur Verfügung. Durch den Aufruf einer Barrier-Funktion wird die Ausführung eines Prozesses solange blockiert, bis alle anderen an der kollektiven Operation beteiligten Prozesse ebenfalls einen korrespondierenden Funktionsaufruf gestartet haben.

3.3.4 Gruppenkonzepte

Bezüglich der Teilnahme von Prozessen eines MIMD-Programms an kollektiven Operationen sind zwei unterschiedliche Verfahren üblich:

- Eine kollektive Operation bezieht sich immer auf alle Prozesse eines Programms (TCGMSG, P4).
- Der Anwender muß die teilnehmenden Prozesse in Form einer Gruppe spezifizieren (MPI-1, PVM).

Das erste Verfahren ermöglicht eine sehr einfache Handhabung, da keine Spezifizierung von Prozessen durch den Nutzer erforderlich ist. Wegen der geringen Flexibilität ist die Anwendbarkeit des Verfahrens aber im wesentlichen auf die SPMD-Programmierung beschränkt.

Das zweite Verfahren ist dagegen auch für MPMD-Programme geeignet, setzt aber die Verwaltung von Prozeßgruppen durch das Message-Passing-System voraus.

MPI-1 unterstützt *statische Prozeßgruppen*, d.h. einer existierenden Gruppe können weder neue Prozesse beitreten, noch können enthaltene Prozesse die Gruppe verlassen. Für das in PVM implementierte Konzept der *dynamischen Prozeßgruppen* gelten diese Einschränkungen nicht.

Die Überlagerung von Gruppen ist sowohl in MPI-1 als auch in PVM möglich, d.h ein Prozeß kann in mehreren Gruppen Mitglied sein.

3.4 Vergleichende Bewertung

Bezüglich der Komplexität bilden die untersuchten Systeme und der MPI-Standard zwei Gruppen:

P4 und TCGMSG verfügen über eine vergleichsweise begrenzte Funktionalität (nur statische Prozeßinstantiierung, keine Unterstützung bei der Überführung komplexer oder mehrerer Daten- in Nachrichtenobjekte und umgekehrt, wenige kollektive Operationen ohne Gruppenkonzept). Eine ganze Reihe von erfolgreich gelösten, realen Problemstellungen (s. [60, 15, 30]) belegt aber, daß diese Beschränkungen, vor allem für die Entwicklung von SPMD-Programmen, eine eher untergeordnete Rolle

spielen. Da beide Systeme das Ergebnis eines kontinuierlichen Entwicklungsprozesses sind, der bereits 1984 mit dem gemeinsamen Vorgänger m4 begann, kann man davon ausgehen, daß sie einen ausgewogenen Kompromiß zwischen Komplexität und leichter Handhabbarkeit repräsentieren.

Der Funktionsumfang von PVM und MPI-1 ist dagegen wesentlich umfangreicher. Dementsprechend größer ist der Einarbeitungsaufwand für den Anwender. So besteht allein das Interface zur nachrichten-orientierten Kommunikation bei PVM aus 39 und bei MPI-1 aus 52 Funktionen (gegenüber 12 bei P4 und 4 bei TCGMSG). Ansonsten unterscheiden sich PVM und MPI-1 in Teilbereichen erheblich voneinander.

Bei MPI-1 liegt der Schwerpunkt auf Vollständigkeit hinsichtlich aller denkbaren semantischen Varianten von nachrichten-orientierten Kommunikationsoperationen und kollektiven Operationen, einschließlich einem sehr leistungsfähigen Gruppenkonzept. Da die Prozeßverwaltung aus der Version 1 des Standards noch weitgehend ausgeklammert wurde, ist nur die statische Instantiierung von Prozessen vorgesehen, wodurch sich Einschränkungen im Bereich der MPMD-Programmierung ergeben.

PVM zeichnet sich durch eine Reihe von Eigenschaften aus, die weder im Standard MPI-1 noch in den anderen untersuchten Message-Passing-Systemen anzutreffen sind. So kann sowohl die Plattformkonfiguration als auch die Instantiierung von Prozessen dynamisch erfolgen. Damit bietet PVM die Voraussetzungen für die Erstellung von fehler- und lasttoleranten Anwendungen. Im Unterschied zu MPI-1 ist auch des Gruppenkonzept in PVM dynamisch. Ob dynamische Prozeßgruppen für die Parallelverarbeitung tatsächlich sinnvoll sind, ist nach [32] aber noch Gegenstand von Diskussionen.

Eine tabellarische Zusammenfassung der Untersuchungsergebnisse ist im Anhang A aufgeführt.

Kapitel 4

Wissenschaftlich-technische Berechnungs- und Visualisierungssysteme (sequentielle SCEs)

Bis zum Ende der 70er Jahre erfolgte die Programmierung wissenschaftlich-technischer Anwendungen hauptsächlich mit kompilierbaren Hochsprachen (Fortran, C u.a.) unter Nutzung von numerischen Programmbibliotheken (PACK-Bibliotheken, NAG, IMSL etc.). Diese Programmiertechnik war in erster Linie auf die Erzeugung von laufzeit- und speichereffizientem Kode und damit auf die optimale Nutzung der Hardwareressourcen ausgerichtet. Die Entwicklung von Anwendungen war dagegen aufgrund des mehrfach zu durchlaufenden Zyklus: Editieren → Kompilieren → Testen sehr aufwendig und erforderte grundsätzlich umfangreiche Programmierkenntnisse.

Genau dieses Problem wurde mit den, in den 80er Jahren aufkommenden, wissenschaftlich-technischen Entwicklungsumgebungen (SCEs)[1] überwunden. Im Unterschied zur klassischen Programmiertechnik ist die effektive Entwicklung von Prototypen (*rapid prototyping*) das zentrale Anliegen der SCEs, was im wesentlichen durch zwei Schlüsselkonzepte erreicht wird:

- SCEs arbeiten *interpretativ*, wodurch die interaktive Ausführung von Routinen ermöglicht wird. Dadurch können einerseits die in den SCEs enthaltenen

[1] Bislang wurden wissenschaftlich-technische Entwicklungsumgebungen, die primär auf numerischen Berechnungen beruhen, in der Literatur als *Array-Oriented & Linear Algebra Systems* oder nach ihrem bekanntesten Vertreter als *MATLAB-like Systems* bezeichnet. Diese Begriffe sind jedoch zu eng gefaßt (in Anbetracht der in modernen Systemen dieser Klasse integrierten Funktionalität) bzw. nicht aussagekräftig genug. Im Rahmen dieser Arbeit wurde deshalb auf die sinngemäße Übersetzung des Terminus *scientific and technical computing environment for numeric computation and visualization*, wie er u.a. in [82] zur Charakterisierung solcher Systeme benutzt wird, zurückgegriffen und von diesem das Kürzel *SCE* abgeleitet.

Im engeren Sinne kann die Klasse der SCEs ausschließlich mit numerischen Berechnungen assoziiert und damit als das Gegenstück zur Klasse der CAS (Computeralgebra-Systeme; symbolisches Rechnen) angesehen werden. Diese Betrachtungsweise ist historisch begründet und für viele reale Systeme noch zutreffend.

Berücksichtigt man die aktuelle Entwicklung, so gehören zur Klasse der SCEs aber auch hybride Umgebungen, in denen sowohl numerische als auch symbolische Methoden bereitgestellt werden.

In dieser Arbeit wird der Begriff SCE ausschließlich im o.g. engeren Sinne verwendet.

Routinen ohne Programmierung direkt aufgerufen und andererseits vom Nutzer definierte Routinen einfach und schnell getestet werden, weil die Erstellung eines Programmrahmens sowie die Kompilation entfallen.

- SCEs stellen zur Programmierung integrierte *array-orientierte Sprachen* zur Verfügung, in denen Datenstrukturen in der Regel weder explizit deklariert noch dimensioniert werden müssen. Die Syntax dieser Sprachen orientiert sich stark an in der Mathematik übliche Notationen. Damit können wissenschaftlich-technische Problemstellungen, die hauptsächlich auf numerischen Berechnungen mit Vektoren und Matrizen beruhen, sehr effizient kodiert werden.

Während sich die algorithmische Funktionalität der ersten SCEs, wie z.B. Classic MATLAB ([57]), noch auf die Routinen der eingebundenen Numerikbibliotheken beschränkte, stellen heutige SCEs darüber hinaus leistungsfähige Visualisierungsmethoden und umfangreiche fachspezifische Algorithmensammlungen bereit.

Trotz der Vielzahl heute verfügbarer SCEs (s. Tab. 4.1) unterscheiden sich diese konzeptionell kaum voneinander. Die Ursache dafür ist, daß die Entwicklung der meisten SCEs von einer gemeinsamen Wurzel, dem System Classic MATLAB[2], ausging. Classic MATLAB wurde von 1977 bis 1984 von Cleve Moler an der Universität von New Mexico in Fortran unter Verwendung einiger Routinen der Linpack- und Eispack-Bibliothek implementiert. Viele kommerzielle und freie Systeme, die in den nachfolgenden Jahren entwickelt wurden, basieren entweder direkt auf Classic MATLAB, wie z.B. CTRL-C, oder orientieren sich zumindest an diesem System bzw. an den späteren MATLAB-Versionen der Firma The MathWorks Inc.

Classic MATLAB	erste MATLAB-Implementierung (1977-1984) in Fortran unter Verwendung von Linpack- und Eispack-Routinen von C. Moler
MATLAB	zweite MATLAB-Implementierung in C von S. Bangert und J. Little; ab 1985 kommerzielle Versionen (PC-, Mac-, Pro-, 386-MATLAB, MATLAB 3.5, 4, 5) von The MathWorks Inc.
CTRL-C	kommerzielles System von SCT Inc. auf der Basis von Classic MATLAB
MATRIX_X	kommerzielles System von Integrated Systems Inc.; MATLAB-ähnlich
GAUSS	kommerzielles System von Aptech Systems Inc.
O-Matrix	kommerzielles System von Harmonic Software Inc.
Euler	PD-Software von R. Grothmann (Katholische Universität Eichstaett)
MATCALC	PD-Software von M. Gerberg und E. Moore (Univ. of New South Wales)
Octave	PD-Software von J. W. Eaton (University of Wisconsin); weitgehend MATLAB-kompatible Syntax
RLab	PD-Software von I. Searle; teilweise MATLAB-kompatible Syntax
Scilab	PD-Software, entwickelt am Institut National de Recherche en Automatique (INRIA); weitgehend MATLAB-kompatible Syntax

Tabelle 4.1: Auswahl einiger bekannter SCEs

[2]Die ursprüngliche Bezeichnung für dieses System war MATLAB (*matrix laboratory*). Der Begriff Classic MATLAB wurde von C. Moler erst in den 90er Jahren eingeführt, um Verwechslungen mit den später entwickelten, gleichnamigen Systemen der Firma The MathWorks Inc. zu vermeiden.

Auf eine vergleichende Analyse verschiedener SCEs soll deshalb im Rahmen dieser Arbeit verzichtet und im folgenden nur die grundsätzlichen Aspekte des Aufbaus und der Funktionsweise von SCEs betrachtet werden.

4.1 Aufbau und Komponenten einer SCE

Abbildung 4.1 zeigt die wichtigsten Komponenten und deren Zusammenspiel innerhalb einer SCE.

Die Nutzerschnittstelle einer SCE wird durch den Parser und den Interpreter sowie durch das Output-Modul gebildet. Bei interaktiver Benutzung der SCE (Kommando-Mode) erfolgt die Eingabe von Daten und Instruktionen kommandozeilenorientiert. Im Batch-Mode (s. Abschn. 4.4) ist aber auch das Parsen und Interpretieren von Dateien möglich. Über das Output-Modul können Texte, Grafiken und Daten in unterschiedlichen Formaten alternativ auf dem Bildschirm ausgegeben oder in Dateien geschrieben werden.

Umfangreiche bzw. formatierte Datensätze, die in Dateien gespeichert sind, können über das Input-Modul geladen werden. Über dieses Modul erfolgen darüber hinaus sämtliche Dateneingaben (von Datei oder Terminal) innerhalb von Anwendungen.

Die dynamische Speicherverwaltung stellt sogenannte *Workspaces* bereit, in denen sowohl die eingegebenen Daten als auch die Ergebnisdaten von Operationen abgelegt werden. Dabei wird zwischen einem Basis-Workspace, der permanent vorhanden ist, und temporären Workspaces, die während der Ausführung von Funktionen einen lokalen Variablenraum bereitstellen, unterschieden.

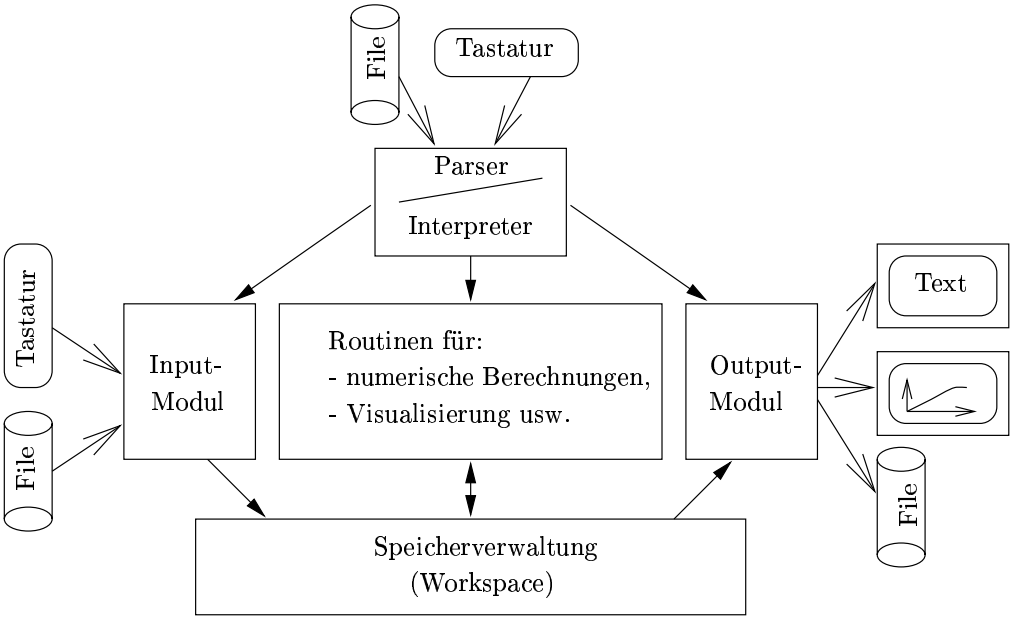


Abbildung 4.1: SCE-Architektur (Prinzipdarstellung)

Die eigentliche Funktionalität einer SCE wird durch ein oder mehrere Module, die Routinen für numerische Berechnungen, Visualisierungsoperationen etc. enthalten, bereitgestellt. Diese Routinen können in unterschiedlicher Form vorliegen. Fundamentale und laufzeitkritische Routinen werden meist in Form von Objektkode als Built-in-Funktionen fest in das System integriert. Weniger häufig benutzte oder nur für bestimmte Anwendungsgebiete relevante Routinen können dagegen in der SCE-eigenen Programmiersprache implementiert sein und werden dann wie nutzerdefinierte Routinen (s. Abschn. 4.3) ausgeführt. Einige wenige Systeme, wie z.B. MATLAB und Octave, unterstützen auch das dynamische Laden von Objektkode.

4.2 Assoziative Speicherung und Opaque-Objekte

Eine für das schnelle Prototyping besonders vorteilhafte Eigenschaft vieler SCEs ist, daß Datentypen und Dimensionen nicht explizit deklariert werden müssen. Voraussetzungen dafür sind:

- die Verwaltung der Workspaces als assoziative Speicher und
- die transparente Repräsentation von Daten zusammen mit ihren Typ- und Dimensionsinformationen in Form von Opaque-Objekten.

Da bei assoziativer Speicherung Datenobjekte nicht wie gewöhnlich über Adressen, sondern über ihren Namen referenziert werden, ist es für den Anwender völlig unerheblich *wo* Datenobjekte residieren. Wird ein Datenobjekt unter einem Namen abgelegt, der noch nicht im Workspace existiert, beschafft die Speicherverwaltung automatisch den erforderlichen Speicherplatz. Existiert bereits ein Datenobjekt unter diesem Namen, wird es, wenn möglich, mit dem aktuell zu speichernden Datenobjekt überschrieben. Ist ein Überschreiben aufgrund unterschiedlicher Datentypen oder Dimensionen nicht möglich, wird der Speicherbereich des „alten“ Datenobjektes freigegeben und das aktuelle Datenobjekt an einer anderen freien Stelle im Adreßraum gespeichert. Analog wird verfahren, wenn vorhandene Datenobjekte nicht komplett durch neue zu ersetzen, sondern nur zu modifizieren sind, d.h. erweitert oder nur teilweise ersetzt werden sollen.

Aus der Sicht des Anwenders, der mit einem Variablennamen genau ein Objekt assoziiert, lassen sich damit Typ und Dimensionen eines Datenobjektes während der Laufzeit beliebig verändern.

Da aber sowohl die Speicherverwaltung als auch alle anderen Operationen einer SCE die jeweils aktuellen Typ- und Dimensionsinformationen über ein Datenobjekt benötigen, werden diese zusammen mit den Daten in einem Opaque-Objekt gespeichert. Opaque-Objekte haben den Vorteil, daß sie die konkrete Repräsentation von Datenstrukturen verbergen und das Lesen bzw. Manipulieren der enthaltenen Daten nur über Zugriffsfunktionen gestatten.

4.3 Programmierung in einer SCE

Die Benutzung einer SCE ähnelt der Arbeit mit einem programmierbaren Taschenrechner[3], d.h. die Problemlösung erfolgt sukzessive durch Anwendung der zur Verfügung stehenden Routinen. Mit Hilfe einer integrierten Programmiersprache können einer SCE neue Routinen, unter Verwendung der bereits vorhanden, hinzugefügt werden.

Die Programmierung innerhalb einer SCE dient damit, im Gegensatz zur compilerbasierten Programmiertechnik, nicht der Erstellung abgeschlossener, selbständig ausführbarer Anwendungsprogramme, sondern der problemspezifischen Erweiterung der Funktionalität der SCE selbst. In diesem Sinne kann eine SCE auch als abstrakte Von-Neumann-Maschine mit erweiterbarem Instruktionssatz betrachtet werden (s. Abb. 4.2).

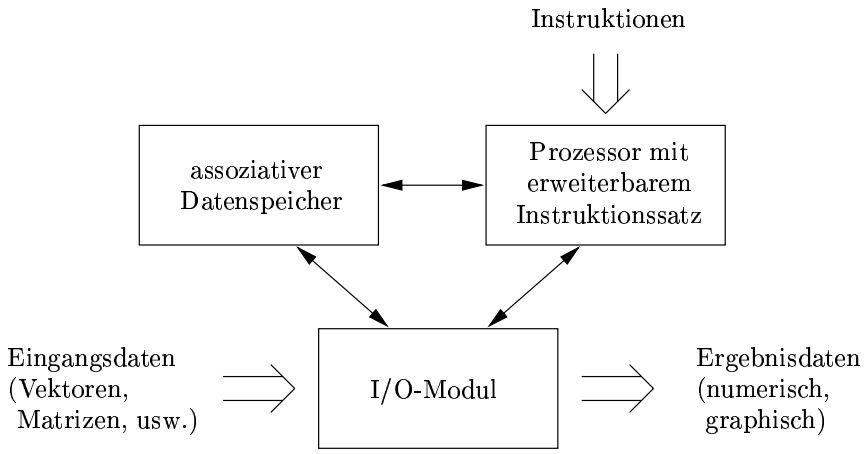


Abbildung 4.2: SCE als abstrakte Von-Neumann-Maschine mit erweiterbarem Instruktionssatz

Nutzerdefinierte Routinen können in Form von Skripten oder Funktionen implementiert werden. Die Datenübergabe erfolgt bei Skripten über den Workspace der aufrufenden Ebene, d.h an interaktiv ausgeführte Skripte können Daten über den Basis-Workspace übergeben werden. Innerhalb von Funktionen steht ein lokaler Variablenraum in Form eines temporären Workspace zur Verfügung. Die Übergabe von Eingangs- und Ergebnisdaten erfolgt über Funktionsparameter. Die in SCEs realisierten Funktionskonzepte sind im Vergleich zu C-Funktionen bzw. Fortran-Routinen wesentlich leistungsfähiger. So sind unter anderem variable Argumentsignaturen und Default-Parameter möglich. Skripte und Funktionen können in beliebiger Schachtelungstiefe miteinander kombiniert werden.

[3]Im Unterschied zu Taschenrechnern können in SCEs nicht nur Fließpunktskalare, sondern Daten-Arrays verschiedenen Typs (Vektoren und Matrizen mit reellen oder komplexen Elementen, Zeichenketten, Polynome etc.) verarbeitet werden.

4.4 Prototyping- und Produktionsphase

Wie bereits am Anfang dieses Kapitels erwähnt, sind SCEs primär für das schnelle Prototyping konzipiert. Aufgrund der interpretativen Abarbeitung weisen fertige SCE-Anwendungen in der Regel einen Laufzeit- und Speicher-Overhead gegenüber vergleichbaren kompilierten Anwendungen auf.

Bestehen für den Einsatz einer Anwendung in der Produktionsphase erhöhte Anforderungen hinsichtlich Laufzeit- und Speichereffizienz, muß der in einer SCE entwickelte Prototyp in einer kompilierbaren Programmiersprache reimplementiert werden. Dies kann entweder manuell oder mit Hilfe von Übersetzern (z.B. MATLAB- oder MATCOM-Compiler; s. [84, 51]) erfolgen.

Da der Overhead von Anwendungen in einer modernen SCE relativ gering ist, kann aber häufig auf eine Reimplementation verzichtet und die SCE als Laufzeitsystem in der Produktionsphase benutzt werden. Dies kann wie in der Prototypingphase entweder interaktiv oder batchorientiert[4] erfolgen.

[4]Bearbeitung von nicht-interaktiven Jobs im Hintergrund.

Kapitel 5

Parallelverarbeitung und SCEs

Nachdem die grundlegenden Aspekte der Parallelverarbeitung, insbesondere der Programmierung von MIMD-Systemen auf der Basis des nachrichten-orientierten Kommunikationsmodells sowie der Aufbau und die Funktionsweise konventioneller SCEs betrachtet wurden, soll in diesem Kapitel untersucht werden, welche Ansätze existieren, die beiden Konzepte:

- Parallelverarbeitung und
- SCE-basierte Anwendungsentwicklung

sinnvoll miteinander zu kombinieren.

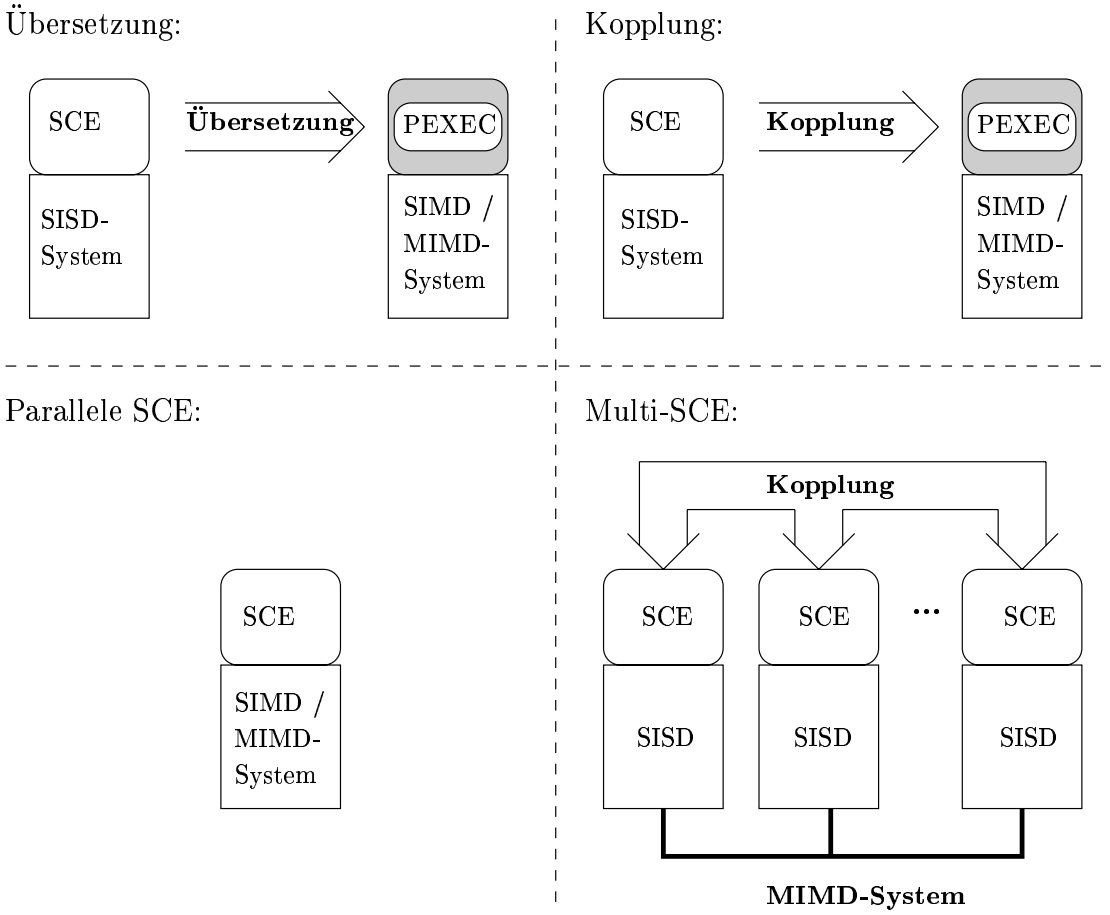
Da sich der Kreis von Anwendern, denen Parallelverarbeitungssysteme zur Verfügung stehen, in den letzten 10 Jahren sprunghaft vergrößert hat, gab und gibt es zahlreiche Bemühungen, die Vorteile der Parallelverarbeitung auch im Bereich der wissenschaftlich-technischen Entwicklungsumgebungen nutzbar zu machen. Trotz der Vielfalt möglicher Varianten lassen sich die vier in Abbildung 5.1 dargestellten, grundlegenden Ansätze unterscheiden.

Bei praktischen Realisierungen liegt der Schwerpunkt auf MIMD-Plattformen, da diese aufgrund des geringeren Preises und der flexibleren Verwendbarkeit deutlich verbreiteter sind als SIMD-Plattformen.

5.1 Übersetzung von SCE-Prototypen

Im Abschnitt 4.4 wurde bereits erwähnt, daß bei laufzeitkritische Anwendungen, die in einer SCE entwickelten Prototypen in kompilierbare Programmiersprachen übersetzt werden. Mittels Übersetzung ist es natürlich nicht nur möglich, ausführbare Programme für sequentielle, sondern auch für parallelverarbeitende Plattformen zu erzeugen. Einige dafür geeignete Werkzeuge sind in der Tabelle 5.1 aufgeführt.

Da die erste Gruppe von Compilern (MATLAB, MATCOM und FALCON; s. [84, 51, 19]) den Kode eines SCE-Prototypen lediglich in eine sequentielle Programmiersprache übersetzt, muß sich hier eine möglichst automatisch ausgeführte Vektorisierung bzw. Parallelisierung (s. Abschn. 2.3.1.3 und 2.3.2.2) anschließen.



PEXEC ... ausführbares Parallelprogramm

Abbildung 5.1: Ansätze zur Nutzung der Parallelverarbeitung in Verbindung mit SCEs

Die zweite Gruppe von Compilern (Otter und ISI; s. [71, 41]) erzeugt dagegen bereits Programmkode, der auf einem Message-Passing-System bzw. auf einer parallelen Bibliothek basiert und damit unmittelbar für parallele Plattformen geeignet ist.

Eine gewisse Sonderstellung nimmt die SCE CONLAB ([22]) mit dem zugehörigen Compiler ein. Während es bei allen anderen Umgebungen um die Übersetzung sequentieller Prototypen geht, handelt es sich bei CONLAB bereits um explizit parallel programmierte Prototypen[1]. Damit muß bei der Übersetzung in Produktionskode keine Parallelisierung mehr erfolgen.

Insgesamt hat der Übersetzungsansatz zwei entscheidende Nachteile:

- Die Erzeugung von Produktionskode basiert auf der Erkennung impliziter Parallelität im sequentiellen SCE-Prototyp (mit Ausnahme von CONLAB). Die

[1]Da CONLAB selbst aber nur auf Uniprozessorsystemen arbeitet, können diese Prototypen innerhalb der Umgebung nur quasi-parallel ausgeführt werden.

SCE	Übersetzung					Target
MATLAB	MATLAB-Compiler MATCOM-Compiler FALCON-Compiler	→ → →	C C++ Fortran 90	→ →	Vektorisierung Parallelisierung	SIMD MIMD
MATLAB	Otter-Compiler ISI-Compiler	⟶ ⟶			C/MPI C/par. Bib.	MIMD
CONLAB	CONLAB-Compiler	⟶			C/PICL	MIMD

Tabelle 5.1: Übersetzung von SCE-Prototypen für parallele Zielplattformen

Realisierbarkeit einer leistungsfähigen vollautomatischen Parallelisierung bzw. Vektorisierung wird aber grundsätzlich in Frage gestellt (s. [7]).

- Die Vorteile der Parallelverarbeitung sind innerhalb der SCE nicht nutzbar.

5.2 Kopplung von SCEs und Parallelverarbeitungssystemen

Die direkte Kopplung von SCEs mit anderen Hard- und/oder Softwaresystemen ist ein weiterer Ansatz, der bereits von sequentiellen automatisierungstechnischen Problemstellungen, wie Meßwerterfassung und Steuerung von Aktoren, bekannt ist und sich im gewissen Maße auch für die Verbindung von Parallelverarbeitung und SCEs verwenden läßt. So werden in der Literatur verschiedene Kopplungen von SCEs mit MIMD- und SIMD-Systemen beschrieben ([48, 1]). Beim Kopplungs-Ansatz stellt die SCE die Benutzerschnittstelle (*front-end*) zu einem Parallelverarbeitungssystem dar. Damit können zwar von einer SCE aus externe parallele Anwendungen genutzt werden, die Entwicklung paralleler Prototypen innerhalb der SCE ist jedoch nicht möglich.

5.3 Parallele SCEs

Im Unterschied zum Übersetzungs- und Kopplungsansatz versucht man bei parallelen SCEs nicht nur eine mehr oder weniger lose Verbindung zur Parallelverarbeitung herzustellen, sondern diese im Innern einer SCE zu nutzen. Die ersten bekannt gewordenen Versuche zur Implementation paralleler SCEs wurden von der Firma The MathWorks Inc. unternommen ([58]). Ausgangspunkt dieser Versuche war die sequentielle SCE MATLAB, deren Architektur (s. Abb. 4.1) unverändert übernommen wurde. Da sich innerhalb dieser Architektur nur das Modul mit den numerischen Basisroutinen parallelisieren läßt, wurden ausschließlich diese gegen parallele Implementierungen ersetzt.

Eine erste experimentelle Version, die um 1986 auf einem Multiprozessorsystem ohne gemeinsamen Speicher (Intel iSPC mit 128 Prozessoren) getestet wurde, erbrachte uneingeschränkt negative Ergebnisse und zeigte damit, daß die Gra-

nularität der meisten numerischen SCE-Basisroutinen für eine effiziente parallele Ausführung auf einer Hardware ohne gemeinsamen Speicher zu feinkörnig ist. Verschärft wurde diese Situation noch durch die vom sequentiellen SCE-Konzept unverändert übernommene Speicherverwaltung, die die Haltung sämtlicher Daten in einem zentralen Workspace vorsah.

Um 1989 wurden diese Experimente auf einer Ardent Titan wiederholt. Da dieser Rechner mit gemeinsamen Speicher arbeitete, verfügte er einerseits über eine wesentlich höhere Kommunikationsleistung als die im ersten Versuch eingesetzte Plattform und ermöglichte andererseits die problemlose Implementation der zentralen Speicherverwaltung[2]. Aufgrund dieser Voraussetzungen stellten sich nunmehr positive Ergebnisse ein. Wegen der geringen Prozessoranzahl des verwendeten Rechners[3] war der erzielbare Leistungszuwachs jedoch relativ begrenzt.

Daraufhin stellte MathWorks nach eigenen Angaben die Entwicklung von parallelen MATLAB-Versionen für unbestimmte Zeit ein. Jüngste Arbeiten zur Einbindung paralleler Numerikbibliotheken in OCTAVE und MATLAB auf einer SPP1600/XA-16 ([44]) bestätigten aber nochmals, daß der Ansatz paralleler SCEs für moderne Multiprozessorsysteme mit gemeinsamen Speicher grundsätzlich sinnvoll ist.

Die erwiesene Nichteignung für Plattformen ohne gemeinsamen Speicher und damit für die weit verbreiteten Computercluster schränkt die Anwendbarkeit insgesamt allerdings erheblich ein. Darüber hinaus erlaubt auch dieser Ansatz keine explizite parallele Programmierung von Prototypen[4].

Versuche, von konventionellen SCEs Implementierungen für SIMD-Plattformen abzuleiten, sind bisher nicht bekannt geworden.

5.4 Multi-SCEs

Rekapituliert man, daß das primäre Anliegen des SCE-Konzepts die Bereitstellung einer für das schnelle Prototyping geeigneten Entwicklungsumgebung ist, muß man feststellen, daß alle bisher betrachteten Ansätze ungeeignet sind, dieses Konzept so weiterzuentwickeln, daß es, wie bei der sequentiellen Problemlösung, auch für das schnelle Prototyping von parallelen Lösungsverfahren einsetzbar ist[5].

In Anbetracht dieser Situation wurde 1993 der Multi-SCE-Ansatz von einer Gruppe um Anne E. Trefethen (Cornell University, New York) und vom Autor dieser Arbeit unabhängig voneinander entwickelt. Er beruht auf einer Verkopplung von konventionellen SCEs (s. Abb. 5.1) und gestattet die interaktive Entwicklung und Ausführung von explizit parallelen Prototypen. Der Multi-SCE-Ansatz ist für das gesamte Spektrum der MIMD-Architekturen, aber nicht für SIMD-Plattformen geeignet.

Das Konzept und die Implementation sowie beispielhafte Anwendungen einer Multi-SCE werden in den nachfolgenden Kapiteln ausführlich behandelt.

---

[2] Workspaces können im gemeinsamen Speicher angelegt werden.

[3] Die realisierbare Prozessorzahl ist bei Architekturen mit gemeinsamen Speicher systembedingt geringer als bei Architekturen mit verteiltem Speicher.

[4] Die Programmierung in parallelen SCEs erfolgt wie in Fortran 90 implizit daten-parallel.

[5] Eine Ausnahme stellt allein das im Abschnitt 5.1 erwähnte System CONLAB dar, welches zumindest die quasi-parallele Ausführung von explizit parallel programmierten Prototypen gestattet.

Kapitel 6

Konzept einer Multi-SCE

Im Unterschied zu den bereits existierenden Ansätzen zur Verbindung von SCEs und Parallelverarbeitung (s. Kap. 5) stellt der Multi-SCE-Ansatz eine tatsächliche Erweiterung des konventionellen SCE-Konzepts dar. So gestatten die vorhandenen Ansätze zwar die sinnvolle Nutzung der Parallelverarbeitung in einem mehr oder weniger engen Zusammenhang mit SCEs, das schnelle Prototyping innerhalb einer SCE bleibt aber nach wie vor auf die sequentielle Programmierung beschränkt. Multi-SCEs eröffnen dagegen die Möglichkeit, diese Technik uneingeschränkt auch für die explizite parallele Programmierung zu nutzen.

In diesem Kapitel sollen zuerst mögliche Architekturen von Multi-SCEs über Analogiebeziehungen zu parallelen Rechnerarchitekturen hergeleitet werden. Für eine ausgewählte Multi-SCE-Architektur werden anschließend die wichtigsten konzeptionellen Aspekte untersucht.

Da der Multi-SCE-Ansatz keine konkurrierende Variante zu den anderen Ansätzen darstellt, sondern eine Ergänzung, werden abschließend die Beziehungen zwischen den verschiedenen Ansätzen dargestellt.

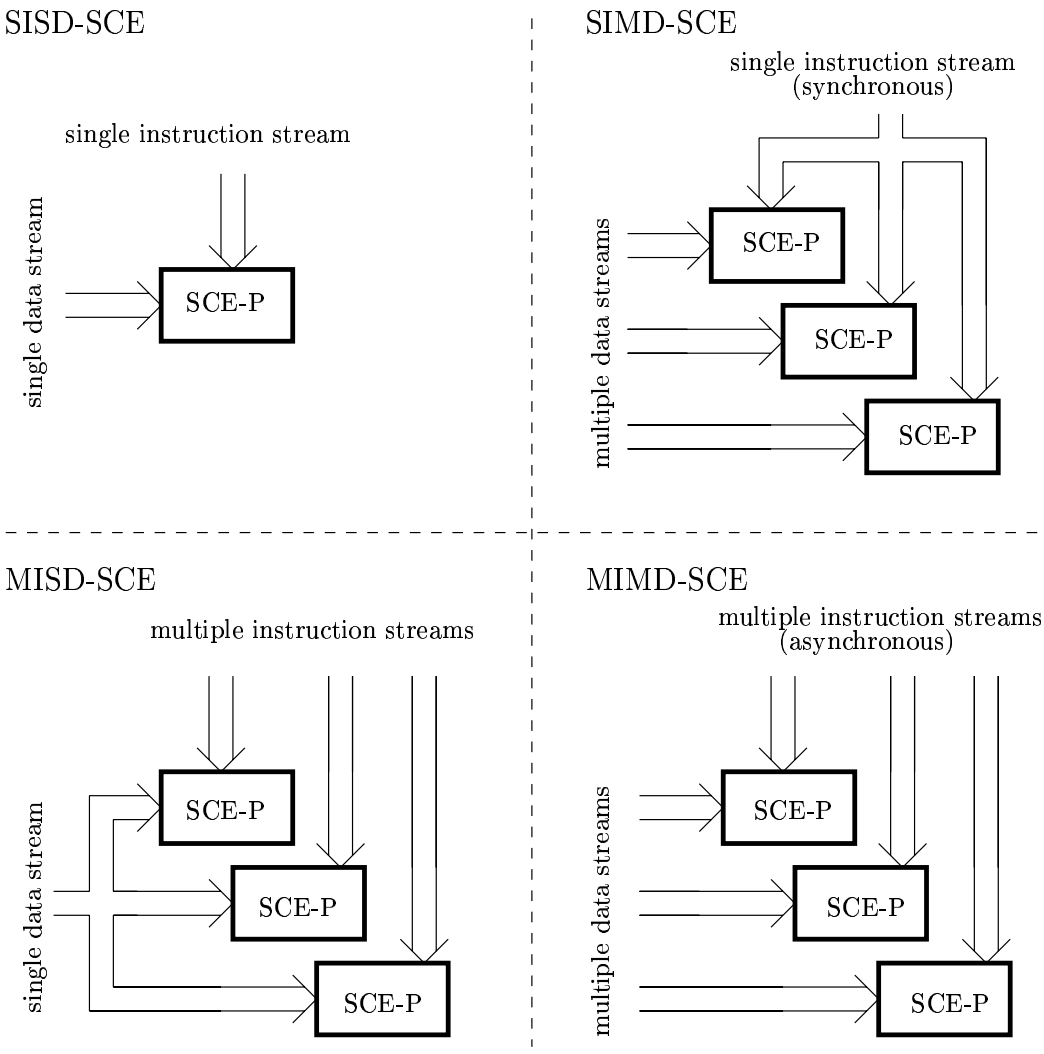
6.1 Multi-SCE-Architekturen

Im Abschnitt 4.3 wurde bereits erläutert, daß eine konventionelle SCE als abstrakte Von-Neumann-Maschine betrachtet werden kann. Damit liegt der Gedanke nahe, die im Abschnitt 2.2 für Rechnerarchitekturen betrachteten Übergänge vom sequentiellen Verarbeitungsprinzip (SISD) zu den verschiedenen Parallelverarbeitungsschemen (MISD, SIMD und MIMD) auf der Ebene von SCEs nachzuvollziehen (s. Abb. 6.1).

Danach könnte man die konventionellen SCEs als SISD- oder Uni-SCEs bezeichnen.

Eine sinnvolle Interpretation des MISD-Prinzips ist bereits im Zusammenhang mit Rechnerarchitekturen sehr umstritten (s. Abschn. 2.2) und soll deshalb auf der SCE-Ebene nicht näher betrachtet werden.

Die SIMD-Verarbeitung wird bei Rechnerarchitekturen eingesetzt, um innerhalb des Prozessors (im Sinne des Von-Neumann-Modells) Parallelität einzuführen. Eine analoge Anwendung des Verarbeitungsprinzips auf der SCE-Ebene ist durchaus



SCE-P ... abstrakter SCE-Processor (in Analogie zu PE ... processing element)

Abbildung 6.1: Anwendung des Formalismus nach Flynn auf SCEs

denkbar, als Ergebnis entsteht aber keine Multi-SCE (mehrere gekoppelte SCEs), sondern eine einzelne SCE mit interner synchroner Parallelverarbeitung.

Die abstrakten Prozessoren einer Multi-SCE im Sinne von Verarbeitungselementen eines SIMD-Prozessors zu verwenden (wie in Abb. 6.1 dargestellt) ist ebenfalls wenig sinnvoll, da ein solcher Ansatz auf einer SIMD-Architektur nicht implementiert werden kann[1]. Die Realisierung auf einer MIMD-Architektur wäre zwar möglich, entspräche aber praktisch einer MIMD-SCE, die mit mehreren identischen Instruktionsströmen synchron betrieben wird. In diesem Fall ist es sinnvoller, von einer

[1]Ein abstrakter SCE-Prozessor muß selbst in Form eines Programms implementiert werden und kann somit nicht auf ein Verarbeitungselement einer SIMD-Architektur abgebildet werden.

simulierten SIMD-Verarbeitung innerhalb einer MIMD-SCE zu sprechen.

Die MIMD-Verarbeitung wird auf der Ebene der Rechnerarchitekturen durch Verwendung mehrerer, asynchron arbeitender Prozessoren realisiert. Nach dem Von-Neumann-Modell handelt es sich dabei, im Gegensatz zu den Verarbeitungselementen einer SIMD-Architektur, um vollständige Prozessoren. Die Abbildung von abstrakten SCE-Prozessoren auf die Prozessoren einer MIMD-Architektur ist deshalb problemlos möglich und führt unmittelbar zum Multi-SCE-Ansatz. Dieser stellt somit ein Synonym für die SCE-basierte MIMD-Verarbeitung und damit für asynchrone Parallelverarbeitung dar.

Den Speichermodulen der Rechnerarchitekturen entsprechen die Workspaces auf der SCE-Ebene. Entsprechend ihrer Prozessorzuordnung kann auch hier zwischen lokalen und gemeinsamen Workspaces unterschieden werden (s. Abb. 6.2).

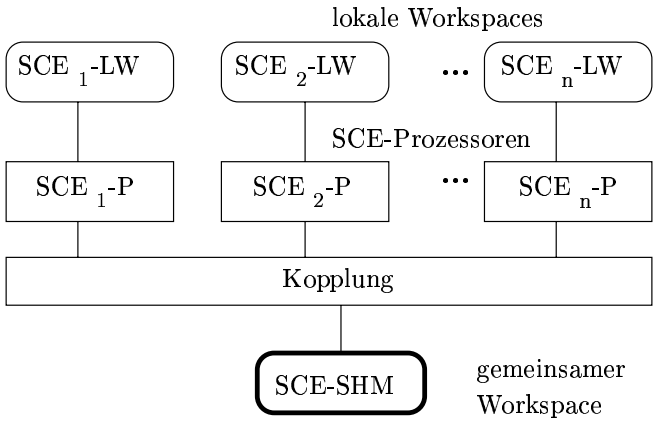


Abbildung 6.2: Multi-SCE mit lokalen und gemeinsamen Workspaces

Da die meisten MIMD-Architekturen über lokale Speichermodule verfügen, ist die Realisierung von lokalen Workspaces in der Regel ohne Schwierigkeiten möglich. Die Bereitstellung eines gemeinsamen Workspace ist dagegen bei MIMD-Architekturen ohne gemeinsamen physischen Speicher (lose gekoppelte MIMD-Systeme) problematischer. Auf solchen Architekturen muß entweder auf der Ebene der Systemsoftware oder innerhalb der Multi-SCE (s. Abb. 6.3) ein virtueller gemeinsamer Speicher emuliert werden. Damit läßt sich zwar in jedem Falle ein gemeinsamer SCE-Workspace realisieren, die für lose gekoppelte Systeme notwendige Emulation erzeugt aber zusätzlichen Overhead.

Da für die nachrichten-orientierte Kommunikation zwischen den SCEs kein gemeinsamer Workspace notwendig ist, kann diese als native Kommunikationsmethode für Multi-SCEs auf der Basis lose gekoppelter MIMD-Systeme angesehen werden. Ein gemeinsamer SCE-Workspace und damit die Möglichkeit der speicher-orientierten Kommunikation sind dagegen optionale Elemente und sollen im Rahmen dieser Arbeit nicht näher betrachtet werden.

Multi-SCEs mit ausschließlich lokalen Workspaces können aus mehreren Instanzen konventioneller SCEs aufgebaut werden. Die Architektur der konventionellen

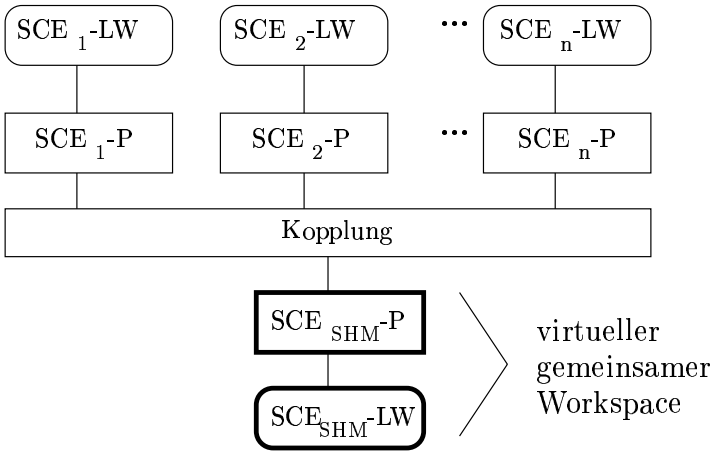


Abbildung 6.3: Emulation eines gemeinsamen Workspaces innerhalb einer Multi-SCE

SCEs muß dazu nur um ein Kommunikationsmodul erweitert werden, das den Nachrichtenaustausch zwischen den SCE-Instanzen einer Multi-SCE ermöglicht (s. Abb. 6.4).

Das Kommunikationsmodul kann entweder unmittelbar über den plattformspezifischen Interaktionsdiensten oder mit Hilfe von Message-Passing-Systemen (s. Kap. 3) realisiert werden. Der Vorteil der ersten Variante besteht im vergleichsweise geringen Kommunikationsoverhead. Aufgrund der Plattformabhängigkeit des Kommunikationsmoduls ist die Unterstützung einer größeren Zahl von Architekturen und

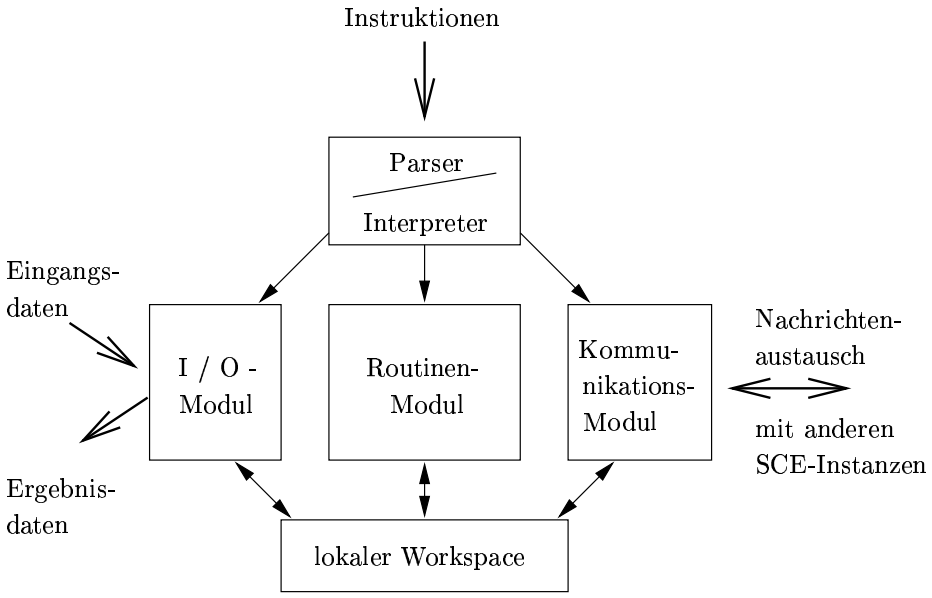


Abbildung 6.4: Architektur einer SCE-Instanz innerhalb einer Multi-SCE

Betriebssystemen aber äußerst aufwendig. Dagegen zeichnet sich die zweite Variante durch eine hohe Portabilität aus, da die gebräuchlichen Message-Passing-Systeme nahezu alle relevanten Plattformen unterstützen.

Weil im SCE-Kontext die Forderung nach hoher Portabilität deutlich stärker priorisiert ist als die Erreichbarkeit eines minimalen Overheads, wird im weiteren nur von Multi-SCEs auf der Basis von Message-Passing-Systemen ausgegangen.

6.2 Konfiguration einer Multi-SCE

Die fundamentalen Konfigurationsoptionen einer Multi-SCE sind:

- die Anzahl der SCE-Instanzen, aus der sie besteht und
- die Lokalität der SCE-Instanzen, d.h. auf welchen Prozessoren bzw. Hosts sie residieren.

Da das hier behandelte Multi-SCE-Konzept von Realisierungen auf der Basis von Message-Passing-Systemen ausgeht, ist die konkrete Art und Weise einer Multi-SCE-Konfiguration weitgehend vom verwendeten Message-Passing-System abhängig.

Unterstützt das Message-Passing-System nur eine statische Prozeßinstantiierung (s. Abschn. 3.1.1) so kann auch die Konfiguration der Anzahl und der Lokalität der SCE-Instanzen nur statisch erfolgen. Bei Message-Passing-Systemen mit dynamischer Prozeßinstantiierung kann dagegen eine existierende Multi-SCE-Konfiguration zum Beispiel durch laufende Anwendungen verändert werden. Eine unabhängige Prozeßinstantiierung erlaubt außerdem den selbständigen Beitritt einer autonomen SCE-Instanz zu einem Multi-SCE-Verband[2]. Eine Darstellung der Zusammenhänge zwischen den verschiedenen Methoden der Prozeßinstantiierung und der Multi-SCE-Konfiguration zeigt Abbildung 6.5.

Im Abschnitt 3.1.1 wurde die Möglichkeit des expliziten und transparenten Prozeß-Mappings auf der Ebene von Message-Passing-Systemen erläutert. Für die Konfiguration einer Multi-SCE bedeutet dies, daß bei explizitem Mapping sowohl die Anzahl der SCE-Instanzen als auch deren genaue Lokalität spezifiziert werden muß. Bei transparentem Mapping ist dagegen nur die Anzahl anzugeben. Transparentes Mapping auf der SCE-Ebene ist auch realisierbar, wenn das Message-Passing-System nur explizites Mapping unterstützt. Die dazu notwendigen Algorithmen müssen dann allerdings innerhalb der Multi-SCE implementiert werden.

Da die Konfiguration einer Multi-SCE die Allokation umfangreicher und begrenzter Ressourcen zur Folge hat, stellt sie eine teure Operation dar, die mit anderen Anwendungen, Nutzern oder systembedingten Limitierungen in Konflikt geraten kann.

Der hohe Preis einer Multi-SCE-Konfiguration resultiert hauptsächlich aus der erheblichen Programmgröße moderner SCEs (Imagegrößen weit über 1 MByte). Die Folge sind lange Instantiierungszeiten und ein starker Verbrauch an Arbeitsspeicher,

[2]Diese Möglichkeit der Prozeßinstantiierung/Multi-SCE-Konfiguration ist für parallele Anwendungen weniger interessant. In verteilten Anwendungen ermöglicht sie jedoch sehr flexible Lösungen für das Monitoring u.ä.

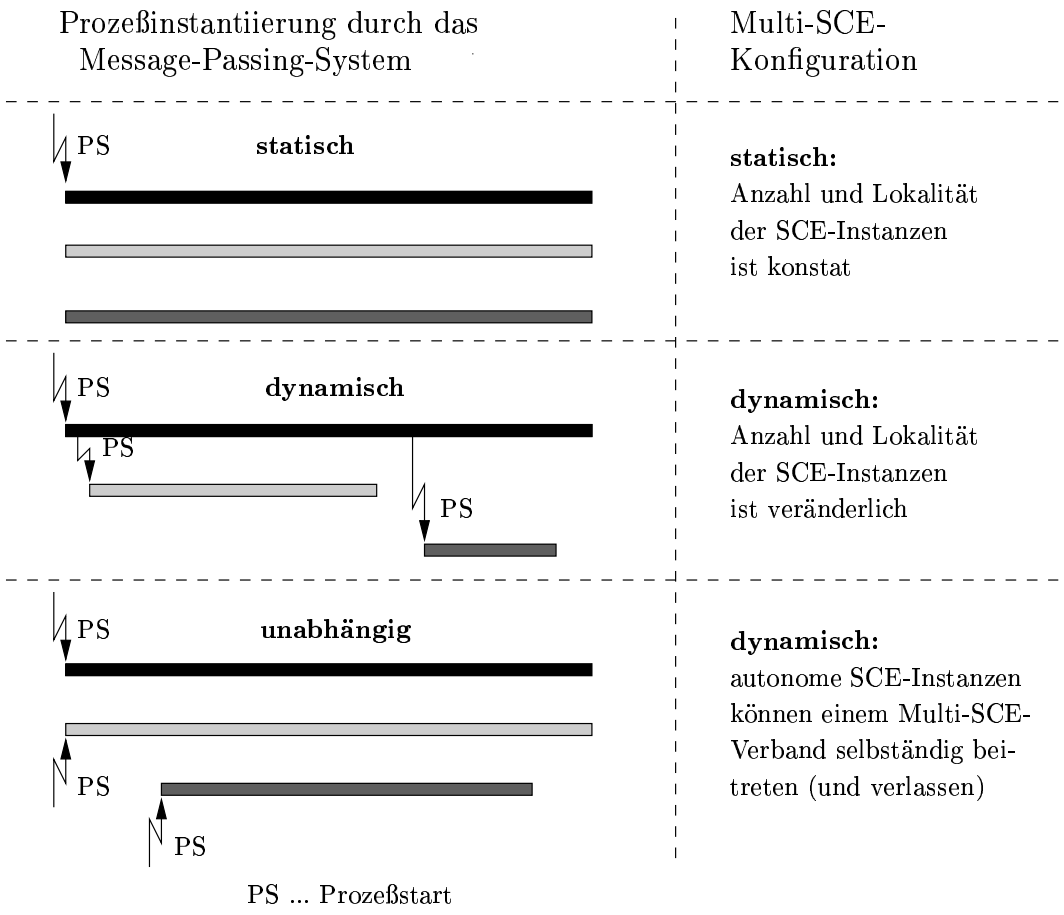


Abbildung 6.5: Zusammenhang zwischen der Prozeßinstantiierung des Message-Passing-Systems und der Konfiguration einer Multi-SCE

insbesondere wenn mehrere SCE-Instanzen auf einem Host bzw. Prozessor plaziert werden.

Konfigurationsspezifische Konflikte mit anderen Anwendungen (z.B. einer weiteren Multi-SCE des Nutzers) können auftreten, wenn diese über einer gemeinsamen virtuellen parallelen Maschine arbeiten. So kann beispielsweise der Fall eintreten, daß eine Anwendung die Deallokation eines Knotens aus der virtuellen Maschine veranlaßt, während dieser von einer anderen Anwendung noch benutzt wird. Derartige Konflikte können aber von intelligent implementierten Message-Passing-Systemen und Multi-SCEs für den Nutzer transparent gelöst werden.

Nicht transparent lösbar sind dagegen in der Regel Ressourcenerschöpfungen. Ist zum Beispiel die Anzahl der gleichzeitig benutzbaren SCE-Instanzen aus lizenzrechtlichen Gründen begrenzt (wie bei kommerziellen SCEs üblich), können Konflikte sowohl zwischen Anwendungen eines Nutzers als auch verschiedener Nutzer auftreten.

Neben den oben genannten fundamentalen Konfigurationsoptionen einer Multi-SCE existieren je nach Systemplattform und Message-Passing-System weitere Konfigurationsmöglichkeiten, die sich vor allem auf die Eigenschaften einzelner SCE-Instanzen auswirken. Einige typische instanzbezogene Optionen sind in Abbildung 6.6 dargestellt.

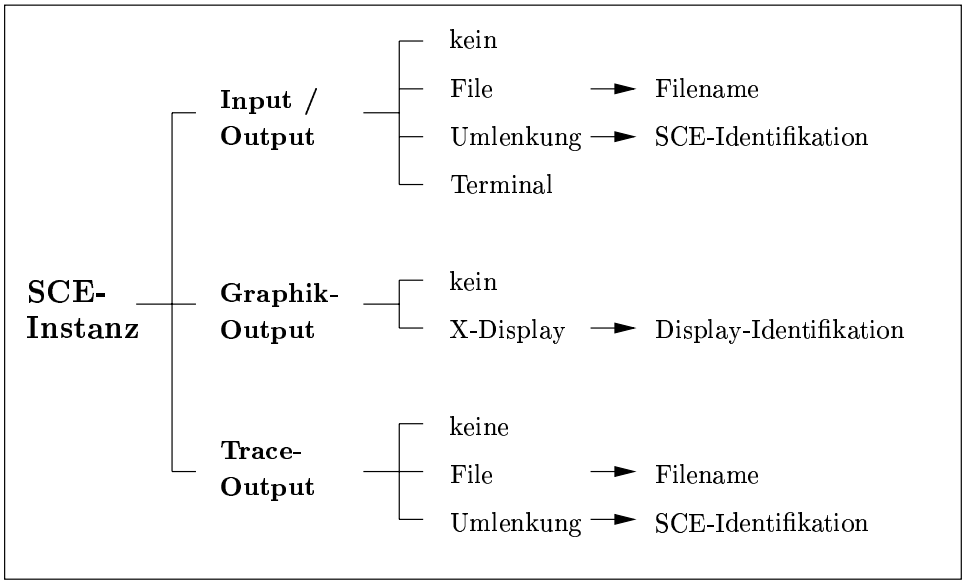


Abbildung 6.6: Auswahl instanzbezogener Konfigurationsoptionen

6.3 Ausführung von Multi-SCE-Anwendungen

Die Ausführung von Anwendungen in einer Multi-SCE ist das Pendant zur im Abschnitt 3.1 beschriebenen Prozeßverwaltung in Message-Passing-Systemen. Was auf beiden Ebenen unter einer Anwendung verstanden wird, unterscheidet sich jedoch voneinander.

Eine Anwendung auf der Basis eines Message-Passing-Systems ist ein kompiliertes MIMD-Programm nach dem MPMD- oder SPMD-Modell[3], das zur Ausführung in Form von Prozessen instantiiert werden muß. Mit der Instantiierung erfolgt eine Prozeß-Prozessor-Zuordnung (Prozeßmapping) und die Vergabe von Prozeßidentifikatoren. Letztere sind Voraussetzung für Interaktionen zwischen den Prozessen einer Anwendung.

In einer SCE werden, wie im Abschnitt 4.3 beschrieben, keine abgeschlossenen, selbständig ausführbaren Anwendungsprogramme erstellt, sondern nutzerdefinierte Routinen (Skripte und Funktionen), die die SCE-Funktionalität anwendungsspezifisch erweitern. Die eigentliche Ausführung einer Anwendung besteht im Aufruf einer oder mehrerer Routinen, ähnlich der Arbeit mit einem Taschenrechner.

[3] *multiple programs multiple data* bzw. *single program multiple data*, s. Abschn. 3.1.1

Während in konventionellen SCEs der Aufruf von Routinen nur sequentiell möglich ist, sind in einer Multi-SCE Routinen auf allen beteiligten SCE-Instanzen simultan und unabhängig voneinander ausführbar. Abhängig davon, ob auf den SCE-Instanzen unterschiedliche oder identische Routinen-Sequenzen zur Ausführung gebracht werden, handelt es sich um eine MIMD-Verarbeitung nach dem MPMD- bzw. SPMD-Modell.

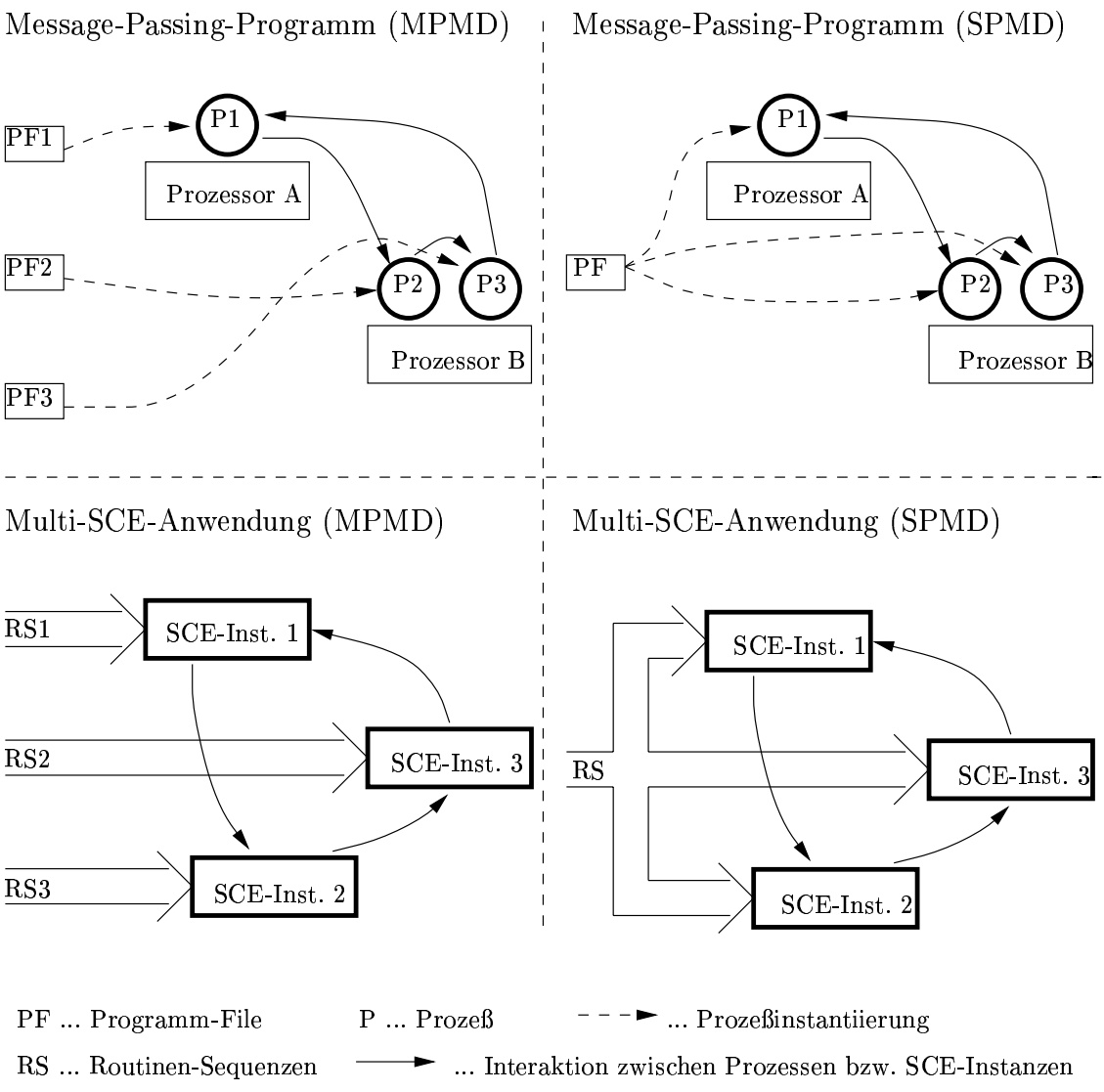


Abbildung 6.7: Ausführung von Message-Passing-Programmen und Multi-SCE-Anwendungen

Zwischen der in Abbildung 6.7 dargestellten Ausführung von Anwendungen auf der Basis von Message-Passing-Systemen und Multi-SCEs existieren drei wesentliche Unterschiede:

1. Die Ausführung von Message-Passing-Programmen beruht auf einem parallelen Prozeßkonzept, Multi-SCEs basieren dagegen unmittelbar auf der parallelen Abarbeitung mehrerer Instruktionsströme (Routinensequenzen).
2. Demzufolge finden Interaktionen (Nachrichtenübergaben) in einem Message-Passing-Programm zwischen Prozessen und in einer Multi-SCE zwischen SCE-Instanzen statt.
3. Mehrere Prozesse eines Message-Passing-Programms können auf einem Prozessor quasi-parallel ablaufen[4]. Eine SCE-Instanz kann dagegen nur einen einzelnen Instruktionsstrom verarbeiten.

Da es in einer Multi-SCE primär keine Prozesse gibt, existiert auch kein Instantiierungsproblem. Die SCE-Instanzen sind nach der Konfiguration einer Multi-SCE existent und stehen bereit, jeweils einen Instruktionsstrom zu verarbeiten. Der Instruktionsstrom einer SCE-Instanz kann aus verschiedenen Quellen gespeist werden, wobei auch Überlagerungen erlaubt sind. Mögliche Quellen sind:

1. die interaktive Eingabe durch den Nutzer,
2. eine Eingabedatei (batchorientierte Verarbeitung) und
3. andere SCE-Instanzen.

Die ersten beiden Quellen sind bereits von den konventionellen SCEs her bekannt (s. Abschn. 4.4) und sind geeignet, sequentielle Routinen auf mehreren SCE-Instanzen parallel auszuführen.

Die dritte Methode eröffnet die Möglichkeit, von einer SCE-Instanz aus Routinen auf anderen SCE-Instanzen zur Ausführung zu bringen. Dies ist wiederum im Sinne einer MPMD- oder SPMD-Verarbeitung möglich. So würde die Operation:

$$evaluate \left(\begin{array}{l} instruction_A\,,\, instance_1 \\ instruction_B\,,\, instance_2 \\ instruction_C\,,\, instance_3 \end{array} \right)$$

die Ausführung der Routinen A, B und C auf drei verschiedenen SCE-Instanzen bewirken, wobei in diesem Fall die SCE-Instanzen explizit spezifiziert worden sind. Dagegen hätte die Operation:

$$evaluate \left(\; instruction_A\,,\, 3 \; \right)$$

die dreifache Ausführung der Routine A zur Folge. Da die zu benutzenden SCE-Instanzen nicht angegeben sind, müssen diese von der Multi-SCE selbständig ausgewählt werden.

[4]Es sei hier vorausgesetzt, daß es sich bei der unterliegenden Plattform um ein Multitasking-System handelt.

Obwohl die Ausführung von Routinen auf anderen SCE-Instanzen Ähnlichkeiten mit der Prozeßinstantiierung eines Message-Passing-Programmes aufweist, so kann man auch hier zwischen statischer, dynamischer und unabhängiger Ausführung und explizitem bzw. transparentem Mapping von Routinen auf SCE-Instanzen unterscheiden, dürfen beide Konzepte nicht gleichgesetzt werden, weil Prozesse einen eigenen, abgeschlossenen Ausführungskontext besitzen, SCE-Routinen jedoch nicht. Werden beispielsweise zwei Routinen nacheinander auf einer SCE-Instanz ausgeführt, so findet die zweite Routine den SCE-Workspace so vor, wie ihn die erste Routine hinterlassen hat.

Gerade unter dem Aspekt des schnellen Prototypings ist dieser Unterschied zum Prozeßkonzept nicht als Nachteil zu bewerten, sondern eröffnet die Möglichkeit, Workspaces vor und nach der Ausführung von Routinen interaktiv zu inspizieren und zu modifizieren. Andererseits ist es mit sehr geringem Aufwand möglich, ein Prozeßkonzept auf der Basis einer Multi-SCE zu realisieren, wenn dies für eine Anwendung sinnvoll erscheint.

6.4 Kommunikation innerhalb einer Multi-SCE

Um Daten zwischen den SCE-Instanzen austauschen zu können, muß eine Multi-SCE über entsprechende Kommunikationsmöglichkeiten verfügen. Wie schon im Abschnitt 6.1 erläutert, erfolgt die Kommunikation in einer Multi-SCE ohne gemeinsamen Workspace durch Nachrichtenübergaben.

Die Grundlagen der nachrichtenbasierten Kommunikation wurden bereits im Abschnitt 3.2 ausführlich behandelt. Dort wurde ausgeführt, daß ein Datentransport aus den Teiloperationen:

- Überführung von Datenobjekten in ein Nachrichtenobjekt,
- Übergabe eines Nachrichtenobjekts und
- Überführung eines Nachrichtenobjekts in Datenobjekte

besteht. Beim Transport von Daten zwischen Plattformen mit unterschiedlicher Datenrepräsentation ist eine zusätzliche

- Konvertierungsoperation

notwendig.

Da Datenobjekte in kompilierbaren Programmiersprachen wie C und Fortran verschiedene Datentypen besitzen können und Typinformationen nur implizit bekannt sind, müssen beim Datentransport in der Regel alle Teiloperationen explizit ausgeführt werden. Dadurch ist der Transport von Daten auf der Ebene von Message-Passing-Systemen relativ aufwendig.

In einer Multi-SCE existieren dagegen günstigere Voraussetzungen, da hier grundsätzlich alle Daten zusammen mit ihren Typ- und Dimensionsinformationen in Form von Opaque-Objekten assoziativ gespeichert werden (s. Abschn. 4.2). Damit muß

der Nutzer nur noch spezifizieren, welche Daten transportiert werden sollen. Die Umwandlung von Daten in Nachrichtenobjekte und umgekehrt sowie eine eventuelle Konvertierung der Repräsentation kann von der Multi-SCE automatisch ausgeführt werden.

Die Semantik der Kommunikationsoperationen in einer Multi-SCE unterscheidet sich nicht von denen in Message-Passing-Systemen. So sind auch hier, je nach Systemvoraussetzungen, sequentielle und nebenläufige Operationen sowie Sendeoperationen mit unterschiedlichen Synchronisationseigenschaften möglich.

Eine weitere Vereinfachung gegenüber Message-Passing-Systemen kann bei der Nachrichtenselektion auf der Empfängerseite eingeführt werden. Im Abschnitt 3.2.1.3 wurde erläutert, daß diese in Message-Passing-Systemen anhand des Absenders und/oder der Markierung einer Nachricht erfolgt. Dabei werden zur Nachrichtenmarkierung Integer-Identifikatoren (*message tags*) benutzt. Da in einer Multi-SCE ein zu transportierendes Datenobjekt beim Aufruf einer Sendefunktion über seinen Namen spezifiziert wird, liegt es nahe, diesen anstelle eines Integer-Identifikators zur Nachrichtenselektion auf der Empfängerseite zu benutzen.

Die Verwendung von Namen zur Nachrichtenselektion verbessert die Lesbarkeit des Programmtextes bzw. die Handhabung bei interaktiver Benutzung. Zusammen mit der Einschränkung, daß durch eine Kommunikationsoperation jeweils nur ein einzelnes Datenobjekt übertragen werden darf, wird es schließlich möglich, auf die Einführung des Begriffs *Nachricht* in den SCE-Kontext gänzlich zu verzichten. Für den Anwender einer Multi-SCE heißt das, daß er direkt *Datenobjekte* senden und empfangen kann.

Die Einschränkung der Kommunikationsoperationen auf die Übertragung einzelner Datenobjekte führt innerhalb einer Multi-SCE zu keinem Funktionalitätsverlust, da SCEs in der Regel über eigene, sehr effiziente Methoden zur Bildung von Meta-Objekten verfügen. Damit kann der Anwender bei Bedarf mehrere Datenobjekte für einen gemeinsamen Transport zusammenfassen.

6.5 Unterstützung weiterer Interaktionsformen

Der im vorangegangenen Abschnitt diskutierte Austausch von Datenobjekten zwischen den SCE-Instanzen einer Multi-SCE ohne gemeinsamen Workspace ermöglicht eine explizite nachrichten-orientierte MIMD-Programmierung nach dem MPMD- oder SPMD-Modell. Optional können über dem nachrichten-orientierten Paradigma komplexere Interaktionsmechanismen aufgebaut und damit weitere parallele Programmiermodelle unterstützt werden.

Die vollständige Abbildung eines abstrakten Paradigmas, wie z.B. Linda, auf das nachrichten-orientierte Kommunikationsmodell einer Multi-SCE stellt jedoch eine umfangreiche und völlig eigenständige Problematik dar. Aus diesem Grunde sollen nachfolgend nur zwei relativ einfache Erweiterungen betrachtet werden, die sich aber in einer Reihe von Anwendungen als sehr nützlich erwiesen haben.

6.5.1 Scatter- und Gatheroperationen

Die Operationen Scatter (Zerlegung eines Datenobjektes und Verteilung der Portionen an mehrere Empfänger) und Gather (Zusammensetzung von Portionen mehrerer Absender zu einem Datenobjekt) wurden bereits im Zusammenhang mit den kollektiven Transportoperationen in Message-Passing-Systemen im Abschnitt 3.3.1 eingeführt.

Auf der Ebene einer Multi-SCE ermöglichen diese Operationen ein komfortables Verteilen und Einsammeln von Vektor- oder Matrixelementen. Für die Bereitstellung solcher Operationen innerhalb einer Multi-SCE existieren zwei Möglichkeiten:

1. Implementation auf Basis der kollektiven Transportoperationen des Message-Passing-Systems,

2. Implementation auf Basis der Multi-SCE-Kommunikationsfunktionen *Send* und *Receive*.

Der Vorteil der ersten Variante besteht darin, daß sich auf diese Weise auch sämtliche anderen kollektiven Operationen, die eventuell durch das Message-Passing-System unterstützt werden, innerhalb der Multi-SCE bereitstellen lassen. Nachteilig ist jedoch, daß gerade im Bereich der kollektiven Operationen noch erhebliche Inkompatibilitäten zwischen den gebräuchlichen Message-Passing-Systemen bestehen (s. Abschn. 3.3 und Anhang A.3), wodurch die Portabilität einer Multi-SCE bezüglich der Verwendung unterschiedlicher Message-Passing-Systeme stark eingeschränkt werden würde.

Eine Realisierung nach der zweiten Variante ist dagegen völlig unabhängig vom konkreten Message-Passing-System, erfordert dafür aber einen größeren Implementationsaufwand. Dieser kann jedoch in Grenzen gehalten werden, wenn die Funktionen Gather und Scatter als nicht-kollektive Operationen implementiert werden, weil dann auf die Einführung eines Gruppenkonzeptes verzichtet werden kann.

6.5.2 Remote-Procedure-Calls

Die entfernte Ausführung von Routinen (*remote procedure call,* RPC) ist eine aus der Client-Server-Programmierung bekannte Technik ([18]). Mit Hilfe des RPC-Mechanismus kann ein Client Aufträge an einen Server übergeben, welcher nach erfolgter Bearbeitung die Ergebnisse an den Client zurück gibt.

Remote-Procedure-Calls sind sehr einfach zu handhaben, da ihr Ausführungsmodell (s. Abb. 6.8) stark an das von lokalen Unterprogrammaufrufen angelehnt ist. So kann auch im ursprünglichen RPC-Modell die Ausführung im Client erst fortgesetzt werden, wenn die Abarbeitung der entfernten Routine abgeschlossen ist und die Ergebnisparameter an den Client übergeben worden sind (*synchroner RPC*).

Für die parallele Verarbeitung sind deshalb nur modifizierte RPC-Modelle interessant, in denen der Client nach dem Auslösen eines Remote-Procedure-Calls seine Arbeit sofort fortsetzen kann – also nicht unmittelbar auf die Rückgabe der Ergebnisparameter warten muß (*asynchrone RPCs*). Auf diese Weise können mehrere

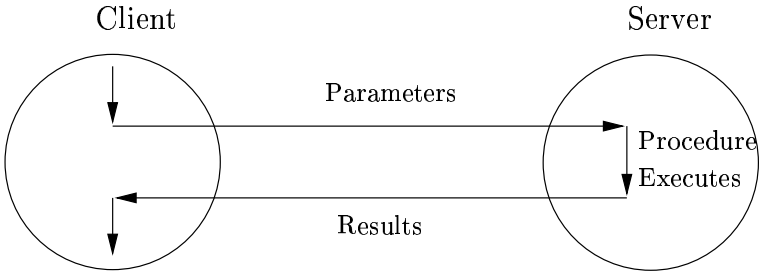


Abbildung 6.8: Remote-Procedure-Call (synchron)

Aufträge zur parallelen Bearbeitung an verschiedene Server abgesetzt werden. Dieses Verfahren hat jedoch den Nachteil, daß die Rückgabe der Ergebnisparametern nunmehr durch Aufruf einer Completion-Funktion explizit vom Client angefordert werden muß.

Innerhalb einer Multi-SCE sind jeweils die Vorteile des synchronen und des asynchronen RPC-Modells (einfache Handhabung und parallele Bearbeitung von Aufträgen) in einem Mechanismus, der hier als *vektorieller Remote-Procedure-Call* bezeichnet werden soll, kombinierbar. Die Anwendung von vektoriellen RPCs kann im Sinne einer MPMD- oder SPMD-Verarbeitung erfolgen. So würden durch die Operation

$$\begin{pmatrix} results_A \\ results_B \\ results_C \end{pmatrix} = evaluate \begin{pmatrix} instruction_A\,, parameters_A\,, instance_1 \\ instruction_B\,, parameters_B\,, instance_2 \\ instruction_C\,, parameters_C\,, instance_3 \end{pmatrix}$$

die Routinen A, B und C auf drei verschiedenen SCE-Instanzen parallel ausgeführt werden. Dagegen hätte die Operation

$$\begin{pmatrix} results_1 \\ results_2 \\ results_3 \end{pmatrix} = evaluate \left(instruction_A\,, \begin{bmatrix} parameters_1\,, instance_1 \\ parameters_2\,, instance_2 \\ parameters_3\,, instance_3 \end{bmatrix} \right)$$

die Ausführung der Routine A auf verschiedenen SCE-Instanzen mit unterschiedlichen Parametern zur Folge.

Es ist leicht zu erkennen, daß es sich bereits bei den im Abschnitt 6.3 besprochenen Operationen zur Ausführung von Multi-SCE-Anwendungen um vereinfachte Formen vektorieller RPCs (ohne Parameterübergaben) handelte.

6.6 Benutzung einer Multi-SCE

Der auffälligste Unterschied zwischen der Arbeit mit einer konventionellen und einer Multi-SCE besteht darin, daß letztere aus mehreren interaktiven SCE-Instanzen bestehen kann, wobei jede Instanz vom Nutzer Eingaben entgegennehmen und eigene Ausgaben produzieren kann. In einer Mehrfach-Fenster-Umgebung, wie z.B. dem

X-Window-System, wird dazu jede interaktive SCE-Instanz mit einem eigenen virtuellen Terminal verbunden. Wenn nur ein einzelnes Terminal zur Verfügung steht, kann dieses vom Nutzer zwischen verschiedenen SCE-Instanzen umgeschaltet werden. Auf diese Art und Weise können parallele Prototypen auf der Basis einer relativ kleinen Zahl von SCE-Instanzen interaktiv entwickelt und getestet werden.

Die Ausführung von parallelen Anwendungen in der Produktionsphase, mit einem unter Umständen sehr viel größerem Parallelitätsgrad als in der Prototypingphase, erfolgt dagegen in der Regel auf Multi-SCE-Konfigurationen, die aus einer größeren Zahl von Hintergrundinstanzen (SCE-Instanzen ohne Verbindung zu einem Terminal) und nur einer interaktiven SCE-Instanz als Benutzerschnittstelle bestehen. Für eine batchorientierte Verarbeitung können aber auch sämtliche SCE-Instanzen einer Multi-SCE als Hintergrundprozesse konfiguriert werden.

6.7 Kombinationen mit anderen Ansätzen der SCE-basierten Parallelverarbeitung

Im Kapitel 5 wurde erläutert, daß neben dem Multi-SCE-Ansatz drei weitere allgemeine Ansätze zur Verbindung von SCE-basierter Arbeit und Parallelverarbeitung existieren. Jeder dieser Ansätze kann sinnvoll mit dem Multi-SCE-Ansatz kombiniert werden:

Multi-SCEs und Übersetzung von SCE-Prototypen: Entsprechend dem ursprünglichen SCE-Konzept kann sich die Nutzung einer Multi-SCE ausschließlich auf die Erstellung von Prototypen beschränken. Für die Produktionsphase erfolgt eine Übersetzung in eine kompilierbare Programmiersprache. Da die Prototypen bereits explizit parallel entwickelt wurden, ist keine Parallelisierung mehr erforderlich.

Multi-SCEs und Kopplung mit Parallelverarbeitungssystemen: Im Gegensatz zu konventionellen SCEs bieten Multi-SCEs die Möglichkeit, interaktive Front-Ends zu einem oder mehreren Parallelverarbeitungssystemen modular und hierarchisch aufzubauen. Dies ist vor allem für komplexe verteilte Probleme im Bereich der Automatisierungstechnik von Interesse.

Multi-SCEs und parallele SCEs: Multi-SCEs ermöglichen die Ausnutzung der grob- bis mittelgranulären Parallelität auf der Ebene von SCE-Anwendungen – parallele SCEs dagegen die fein- bis mittelgranuläre Parallelität von vektoriellen SCE-Operationen. Durch Kombination beider Ansätze kann das gesamte Granularitätsspektrum genutzt werden.

Kapitel 7

Realisierung einer Multi-SCE

In diesem Kapitel soll eine beispielhafte Realisierung des Multi-SCE-Konzeptes auf der Basis der kommerziellen SCE MATLAB Version 4 von The MathWorks Inc. ([82]) und des frei verfügbaren Message-Passing-Systems PVM Version 3.3 ([31]) beschrieben werden.

Das System MATLAB wurde ausgewählt, weil es im automatisierungstechnischen Bereich zu den am stärksten verbreiteten SCEs gehört und für eine große Zahl von Plattformen (Betriebssysteme und Hardwarearchitekturen) erhältlich ist. Technische Aspekte spielten bei der Auswahl dagegen eine untergeordnete Rolle, da es sich beim Multi-SCE-Konzept um einen allgemeinen Ansatz handelt, der grundsätzlich mit jeder beliebigen SCE realisiert werden kann.

In gleicher Weise ist die Umsetzung des Multi-SCE-Konzeptes nicht an ein bestimmtes Message-Passing-System gekoppelt. Da es aber, wie die im Kapitel 3 vorgestellte Analyse gezeigt hat, bezüglich der unterstützten Methodenvielfalt zwischen den Systemen teilweise erhebliche Unterschiede gibt, ist die in einer Multi-SCE realisierbare Funktionalität im hohen Maße vom verwendeten Message-Passing-System abhängig. In diesem Sinne stellt die Auswahl des Systems PVM für die hier beschriebene Beispielrealisierung einen Kompromiß dar. Von den untersuchten Message-Passing-Systemen ermöglicht es als einziges die dynamische Instantiierung von Prozessen, was die Voraussetzung für die dynamische Konfiguration einer Multi-SCE ist (s. Abschn. 6.2). Nebenläufige Kommunikationsoperationen, wie sie von TCGMSG oder von MPI-Implementierungen her bekannt sind, unterstützt PVM dagegen nicht.

Weitere Multi-SCE-Realisierungen auf der Basis anderer Message-Passing-Systeme werden in [62, 66, 43] beschrieben.

7.1 Erweiterung der MATLAB-Architektur

Im Abschnitt 6.1 wurde erläutert, daß die Architektur einer konventionellen SCE für den Einsatz im Rahmen einer Multi-SCE um eine Kommunikationskomponente erweitert werden muß.

Steht der Quellkode der SCE zur Verfügung, kann ein Kommunikationsmodul direkt und permanent in die bestehende Architektur integriert werden. Da dies bei

kommerziellen SCEs wie MATLAB in der Regel nicht der Fall ist, kann eine solche interne Erweiterung nur vom Hersteller bzw. von Inhabern einer Quellkode-Lizenz vorgenommen werden.

Als Alternative zur internen Erweiterung unterstützt MATLAB das dynamische Linken von Routinen (MEX-Interface, s. [83]). Auf der Basis dieser Technik kann, wie in Abbildung 7.1 dargestellt, die Kommunikationskomponente als externes Modul realisiert werden, welches erst zur Laufzeit von einer MATLAB-Instanz geladen wird.

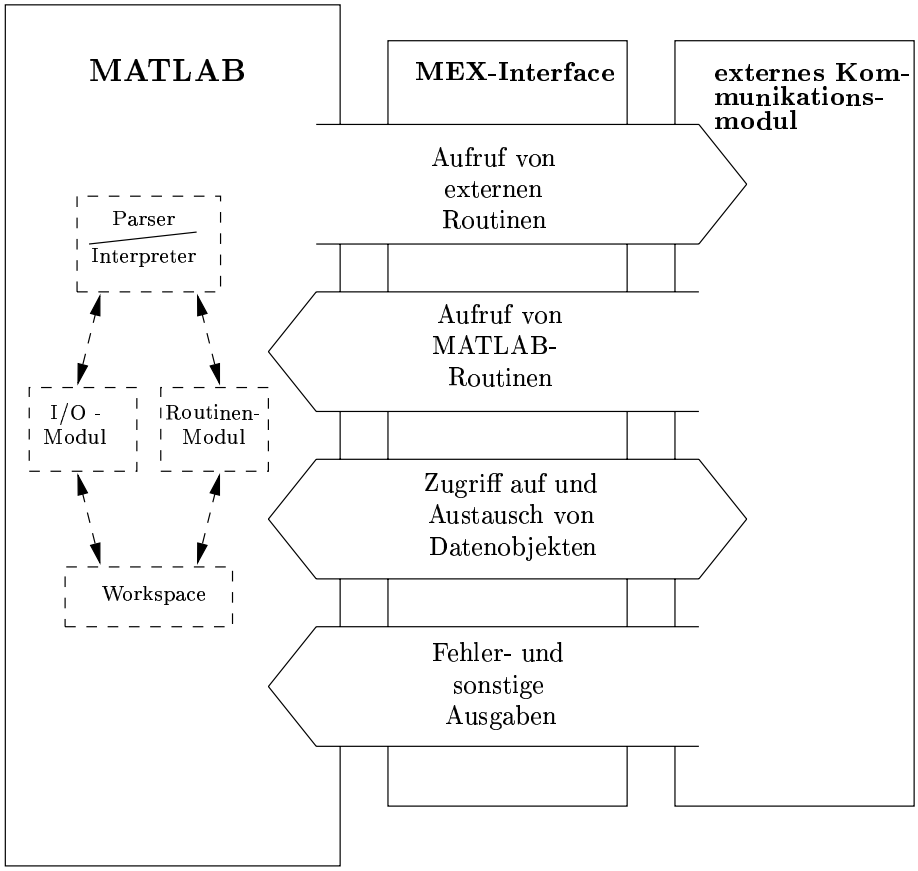


Abbildung 7.1: Erweiterung der MATLAB-Architektur um ein externes Kommunikationsmodul

Das MEX-Interface, das neben der Modulanbindung auch die Manipulation von MATLAB-Matrizen[1] unterstützt, besteht aus einer Reihe von Routinen, die im Objektkode in Form einer Bibliothek zur Verfügung stehen. Eine detaillierte Beschreibung der Routinen des MEX-Interface ist in [83] enthalten.

Externe MATLAB-Routinen, deren Programmierung in C oder Fortran erfolgen kann, müssen nach der Übersetzung, zusammen mit den Routinen des MEX-Interface, zu dynamisch ladbaren Objekten gebunden werden. In der MATLAB-Terminologie werden diese als MEX-Files oder MEX-Funktionen bezeichnet.

[1]MATLAB in der Version 4 kennt nur den Datentyp Matrix (s. Anh. B).

MEX-Funktionen können wie interne Routinen (Built-in-Funktionen) bzw. Routinen, die in der MATLAB-eigenen Programmiersprache kodiert sind (M-Funktionen oder -Skripte, zusammenfassend auch als M-Files bezeichnet) benutzt und mit diesen kombiniert werden. Zur besseren Strukturierung werden funktional zusammengehörige MEX- und M-Files zu Toolboxen zusammengefaßt und in separaten Verzeichnissen verwaltet.

Ein so realisiertes, externes Kommunikationsmodul präsentiert sich damit dem Anwender als gewöhnliche MATLAB-Toolbox.

7.2 Anbindung an das PVM-System

Einfache Message-Passing-Systeme, wie z.B. P4 oder TCGMSG, bestehen nur aus einer Kommunikationsbibliothek, die in die jeweilige Anwendung einzubinden ist. PVM stellt dagegen ein eigenständiges System dar, das aus mehreren Komponenten besteht und gleichzeitig mehreren parallelen bzw. verteilten Anwendungen eines Nutzers Kommunikations- und andere Dienste (z.B. Signale, Detektion von ausgefallenen Rechnern etc.) bereitstellen kann. Die wichtigsten Komponenten im PVM-System sind der PVM-Dämon und die PVM-Bibliothek.

Der PVM-Dämon ist ein ausführbares Programm, das auf jedem Rechner (Host), der in das PVM-System einbezogen werden soll, als Hintergrundprozeß laufen muß. In der PVM-Terminologie kann sich der Nutzer auf diese Weise aus mehreren physischen Hosts eine *parallele virtuelle Maschine* zusammenstellen.

Über die in der PVM-Bibliothek enthaltenen Interface-Routinen können Anwendungsprozesse, die als PVM-Tasks bezeichnet werden, auf die Dienste der virtuellen Maschine zugreifen. Der unmittelbare Diensterbringer für eine PVM-Task ist immer der lokale PVM-Dämon, d.h. der Dämon der auf dem selben Host wie die Task läuft.

Um die Dienste des PVM-Systems für eine MATLAB-Instanz bereitzustellen, ist also eine Einbindung der PVM-Bibliothek in das externe Kommunikationsmodul notwendig (s. Abb. 7.2).

Die PVM-Bibliothek steht in zwei Versionen zur Einbindung in C- bzw. Fortran-Programme zur Verfügung. Eine ausführliche Dokumentation aller Interface-Routinen ist in [31] enthalten.

7.3 Das DP-Toolbox-Set

Die DP-Toolbox (*Distributed and Parallel Application Toolbox*) ist eine prototypische Realisierung der in den beiden vorangegangenen Abschnitten beschriebenen Erweiterung der MATLAB-Architektur um eine PVM-basierte Kommunikationskomponente.

Die gegenwärtige Implementierung der DP-Toolbox besteht aus zwei Schichten (s. Abb. 7.3). In der unteren Schicht wurde ein Gateway zu den Interface-Routinen der PVM-Bibliothek realisiert. Da ein solches MATLAB-PVM-Interface auch außerhalb des Multi-SCE-Konzeptes sinnvoll verwendet werden kann, wurde diese Schicht

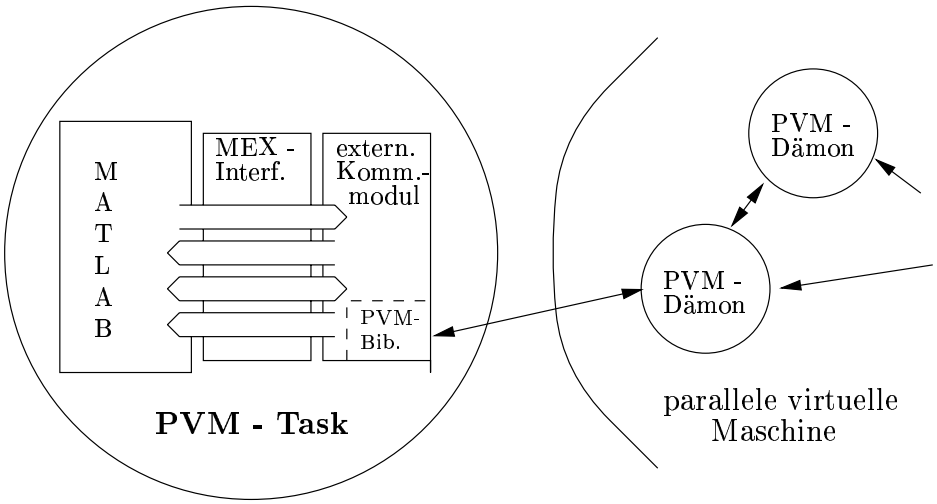


Abbildung 7.2: Anbindung einer MATLAB-Instanz an das PVM-System

als eigenständige Toolbox (*DPLOW-Toolbox*) ausgeführt. Die nach dem Multi-SCE-Konzept erforderlichen höheren Kommunikations- und Verwaltungsfunktionen wurden in der darüberliegenden zweiten Schicht (*DP(HIGH)-Toolbox*) realisiert.

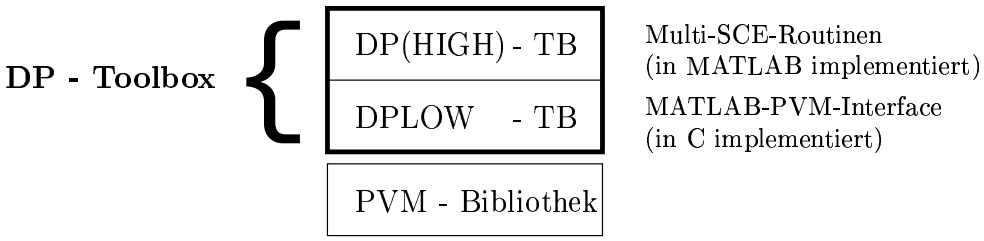


Abbildung 7.3: Das DP-Toolbox-Set

Tatsächlich handelt es sich beim Prototypen also um ein Toolbox-Set. Da dies für den Anwender aber ohne Bedeutung ist, wird die gesamte Realisierung weiterhin als DP-Toolbox bezeichnet.

Neben der Möglichkeit, einen Teil der DP-Toolbox auch für andere Zwecke zu benutzen, besteht der Vorteil der gewählten Struktur darin, daß große Teile der Implementation (fast die gesamte DP(HIGH)-Toolbox) bereits in MATLAB, d.h. mit der in MATLAB integrierten Programmiersprache, erfolgen konnte.

7.3.1 DPLOW-Toolbox

7.3.1.1 Das Gateway m2pvm

Das Standardverfahren zur Integration von Programmbibliotheken in MATLAB besteht darin, jede Bibliotheksroutine über eine separate MEX-Funktion anzubinden. Dadurch wird beim erstmaligen Aufruf einer Routine nicht der gesamte Objektkode der Bibliothek vom MATLAB-Kernel geladen, sondern nur der Kode der gerufenen Routine.

Für die Anbindung der PVM-Bibliothek ist dieses Verfahren ungeeignet, weil die enthaltenen Routinen untereinander Daten über globale Variablen austauschen und damit der gesamte Bibliothekskode als zusammenhängendes Modul geladen werden muß.

Dies wird innerhalb der DPLOW-Toolbox dadurch erreicht, daß sämtliche PVM-Routinen über eine einzige, als Gateway fungierende MEX-Funktion (`m2pvm`, s. Abb. 7.4) eingebunden werden. Die Steuerung des Gateways, d.h. die Auswahl der aufzurufenden Bibliotheksroutine, erfolgt über einen numerischen Funktionskode.

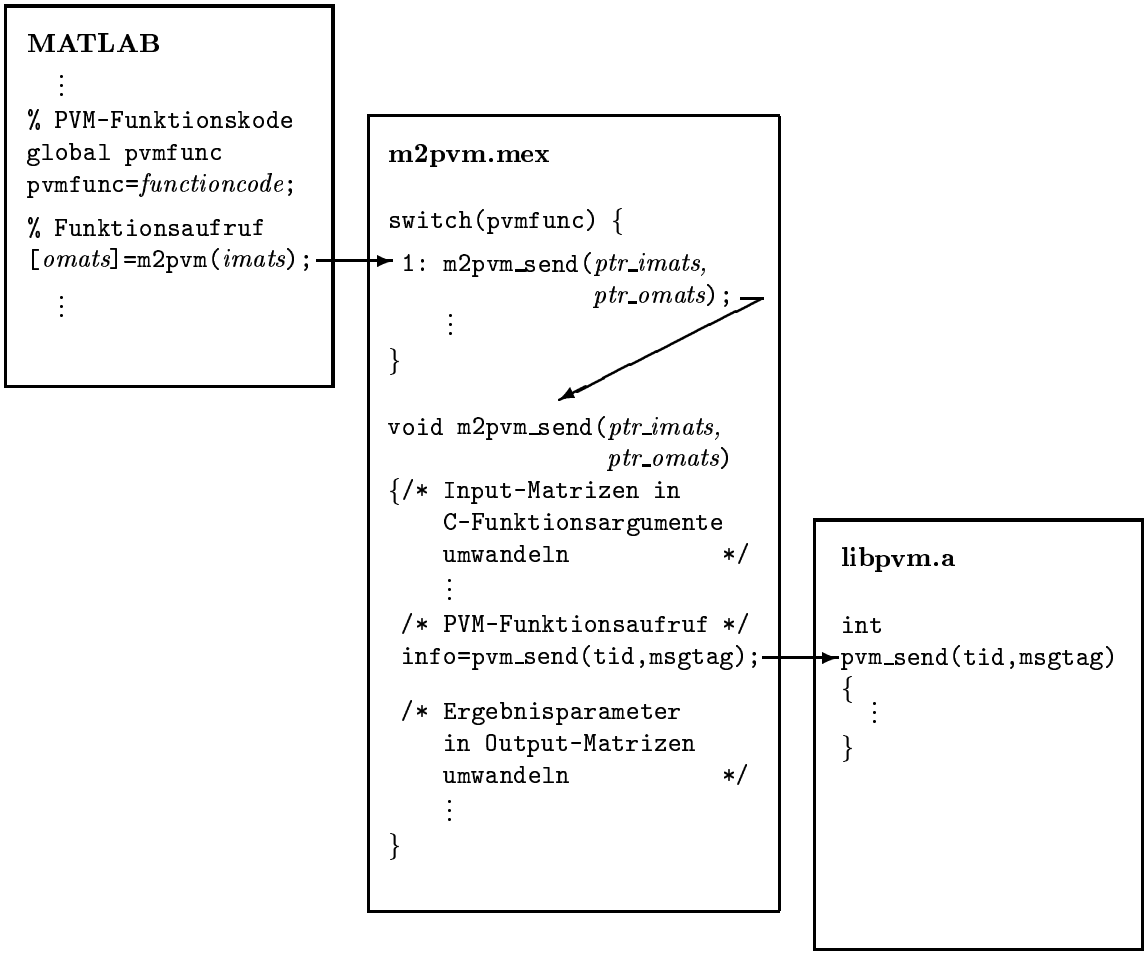


Abbildung 7.4: Aufruf einer PVM-Routine über das Gateway `m2pvm`

Bevor eine PVM-Routine gerufen werden kann, müssen die als MATLAB-Matrizen (s. Anhang B) übergebenen Parameter gemäß Tabelle 7.1 in C-Funktionsargumente umgewandelt werden. Dazu enthält das Gateway für jede Bibliotheksroutine eine separate Wrapper-Funktion, die neben der Konvertierung der Input-Parameter auch den Aufruf der PVM-Routine sowie die Rückkonvertierung der Ergebnisparameter vornimmt.

C-Funktionsargumente		MATLAB-Repräsentation
(int i) (int *ptr_i)	Integer-Wert bzw. -Referenz	ganzzahliges Skalar
(double d) (double *ptr_d)	Double-Wert bzw. -Referenz	reelles Skalar
(int *ptr_ia, int n)	Integer-Array mit n Elementen	n-dimensionaler Vektor mit ganzzahligen Elementen
(double *ptr_da, int n)	Double-Array mit n Elementen	n-dimensionaler Vektor mit reellen Elementen
(char *s)	C-String	Zeilenvektor (Elemente repräsentieren druckbare Zeichen)
(char **sa, int n)	n C-Strings	n-zeilige Matrix (jede Zeile repräsentiert eine Zeichenkette)
(struct type *ptr_strc)	Referenz auf eine Struktur	die Komponenten der Struktur werden durch separate Skalare bzw. Vektoren repräsentiert
(struct type **ptrs_strc, int n)	Referenz auf n Strukturen	korrespondierende Komponenten der Strukturen werden zu Vektoren bzw. Matrizen zusammengefaßt

Tabelle 7.1: Repräsentation von C-Funktionsargumenten in MATLAB

7.3.1.2 Das DPLOW-Anwenderinterface

Für die Erstellung von Multi-SCE-Anwendungen ist ein direkter Zugriff auf die DPLOW-Toolbox durch den Nutzer nicht notwendig. Wie bereits erwähnt, sollte aber ein separater Einsatz der Toolbox als MATLAB-PVM-Interface für andere Anwendungsbereiche möglich sein.

Da der Aufruf von PVM-Routinen über das Gateway `m2pvm` für eine interaktive Nutzung zu umständlich ist und nicht den in MATLAB üblichen Regeln entspricht, wurde die DPLOW-Toolbox mit einem Anwenderinterface ausgestattet, das folgende Eigenschaften aufweist:

- Dem Nutzer steht zum Aufruf einer PVM-Routine eine jeweils gleichnamige MATLAB-Routine zur Verfügung.
- Die Typen, Dimensionen und Werte der Input-Parameter werden vor dem Aufruf der PVM-Routine auf Gültigkeit überprüft.

- Semantisch optionale Parameter können beim Aufruf weggelassen werden.
- Liefert der Aufruf einer PVM-Routine einen Fehlerstatus, erfolgt die in MATLAB übliche Fehlerbehandlung, d.h. sofortiger Abbruch der Bearbeitung und Ausgabe der Fehlerursache im Klartext.[2]

Eine ausführliche Dokumentation der Routinen des DPLOW-Anwenderinterface ist in [68] enthalten.

7.3.2 DP(HIGH)-Toolbox

Durch die im vorangegangenen Abschnitt besprochene DPLOW-Toolbox wird MATLAB – vereinfacht ausgedrückt – zu einer netzwerkfähigen SCE, womit die Voraussetzung geschaffen worden ist, mit diesem System das Multi-SCE-Konzept umzusetzen. Die dafür notwendige Funktionalität wird durch die DP(HIGH)-Toolbox bereitgestellt.

Die gegenwärtige Implementation unterstützt den Betrieb mehrerer voneinander unabhängiger Multi-SCEs mit lokalen Workspaces. Eine Multi-SCE wird im Kontext der DP(HIGH)-Toolbox als *Distributed and Parallel MATLAB Machine* (DPMM) bezeichnet. Die Toolbox stellt entsprechend dem Multi-SCE-Konzept Funktionen

- zur Konfiguration von DPMMs,
- zur Ausführung von Routinen auf einer DPMM sowie
- zur Realisierung nachrichten-orientierter Interaktionen zwischen den in einer DPMM enthaltenen MATLAB-Instanzen (DP-Instanzen)

bereit.

7.3.2.1 DPMM-Konfiguration

Die zentrale Datenbasis zur Verwaltung einer oder mehrerer DPMMs eines Nutzers ist die DPMM-Tabelle (`dpmm_tbl`), in der sämtliche DP-Instanzen mit ihren Konfigurationsparametern aufgeführt sind (s. Tabelle 7.2). Um einen unkomplizierten Zugriff von allen DP-Instanzen aus zu ermöglichen, wurde die DPMM-Tabelle als Master-PVMD-Datenbasis realisiert. Dabei handelt es sich um eine Datenbasis, die im sogenannten Master-Dämon des PVM-Systems angelegt wird und auf die über die PVM-Bibliotheksroutinen `pvm_insert`, `pvm_delete` und `pvm_lookup` global zugegriffen werden kann (s. [31]).

Die Konfiguration einer DPMM kann statisch oder dynamisch erfolgen, d.h. während des Starts oder bei laufendem Betrieb. Eine kombinierte Anwendung beider Methoden ist möglich.

[2] Zur Erstellung fehlertoleranter Anwendungen kann diese Art der Fehlerbehandlung abgeschaltet werden. Der Fehlerstatus eines PVM-Aufrufs steht dann in MATLAB zur Verfügung und kann vom Nutzer getestet werden.

Feld	Erklärung
`dpid`	DP-Instanz-Identifikator; entspricht dem PVM-Task-Identifikator (`tid`)
`dpmmid`	Identifikator der DPMM, in der die DP-Instanz Mitglied ist
`dtid`	Identifikator des virtuellen Hosts, auf dem die DP-Instanz läuft (PVM-Dämon-Task-Identifikator)
`dpidp`	Identifikator der Vater-Instanz (wenn die DP-Instanz von einer anderen DP-Instanz aus gestartet wurde)
`run`	Run-Mode; DP-Instanz läuft interaktiv in einem eigenem Xterm oder als Hintergrundprozeß
`trace`	Trace-Mode; Option zum Führen einer Trace-Datei

Tabelle 7.2: Die Felder der `dpmm_tbl`

Der Start einer DPMM erfolgt aus einer MATLAB-Instanz heraus durch Aufruf der Funktion:

`dpon(`*`dpmmid`*`)` ,

welche die ausführende Instanz in die DPMM-Tabelle einträgt und anschließend entlang des MATLAB-Suchpfades nach der Konfigurationsdatei `dpmm{`*`dpmmid`*`}.m` sucht. Wird diese gefunden, werden entsprechend der enthaltenen Spezifikation weitere MATLAB-Instanzen gestartet und als DP-Instanzen in die DPMM-Tabelle aufgenommen. Da der Nutzer mehrere DPMMs betreiben kann, muß beim Start ein DPMM-Identifikator *dpmmid* angegeben werden. Erlaubte Werte für *dpmmid* sind ganze Zahlen ≥ 1. Erfolgt der Aufruf von `dpon` ohne Parameter, wird der Default-Wert *dpmmid*=1 benutzt (s. Abb. 7.5).

```
>>                    % Standard-Prompt einer
>>                    % MATLAB-Instanz
>>
>> dpon(3)            % DPMM Nr. 3 starten
3:45>>
3:45>>                % DP-Prompt: dpmmid:dpid>>
3:45>>
3:45>> dpoff          % laufende DPMM terminieren
>>
>> dpon               % DPMM Nr. 1 starten
1:45>>
1:45>>
```

Abbildung 7.5: Starten und Terminieren von DPMMs

Um eine Überschneidung mehrerer DPMMs zu verhindern, kann `dpon` nicht in DP-Instanzen, d.h. in MATLAB-Instanzen, die bereits Mitglied in einer DPMM sind, ausgeführt werden. Dagegen ist der Aufruf von `dpon` in einer autonomen MATLAB-Instanz mit dem DPMM-Identifikator einer bereits existierenden DPMM möglich und führt zur Aufnahme dieser Instanz in die spezifizierte DPMM.

Die Terminierung einer laufenden DPMM bzw. die Abmeldung einer selbständig beigetretenen DP-Instanz erfolgt durch Aufruf der Funktion `dpoff`.

Für die dynamische Konfiguration einer DPMM stellt die DP(HIGH)-Toolbox Funktionen zum Hinzufügen und Entfernen von DP-Instanzen (`dpadd` und `dpdel`) sowie zur Änderung instanzbezogener Konfigurationsparameter (`dpchg`) bereit. Konfigurationsänderungen, die nicht an einer laufenden DP-Instanz vollzogen werden können, wie z.B. die Änderung des Ausführungsortes (Host), realisiert `dpchg` über Instanz-Ersetzung, d.h. die vorhandene DP-Instanz wird aus der DPMM entfernt und durch eine neue DP-Instanz mit den gewünschten Parametern ersetzt. Um einen Zustandsverlust zu vermeiden, wird der Workspace-Inhalt der zu ersetzenden Instanz gerettet und von der neuen Instanz übernommen.

Neben den genannten Funktionen stellt die DP(HIGH)-Toolbox mit der Funktion `dpmm` ein alternatives Konfigurationsinterface bereit, bei dem nicht einzelne Änderungen an einer vorhandenen Konfiguration, sondern eine gewünschte Konfiguration spezifiziert wird. Die notwendigen Änderungen an der aktuellen Konfiguration werden von der Funktion `dpmm` automatisch ermittelt und vorgenommen.

Eine detailierte Beschreibung der Konfigurationsfunktionen in Form eines Glossars befindet sich im Anhang C.1.

Prozeß-Mapping

Die Konfigurationsoperationen der DP(HIGH)-Toolbox, die zwangsläufig oder unter bestimmten Umständen den Start neuer DP-Instanzen zur Folge haben (`dpon`, `dpadd`, `dpchg` und `dpmm`), unterstützen sowohl explizites als auch transparentes Prozeß-Mapping.

Zur expliziten Spezifikation des Ausführungsortes einer DP-Instanz kann ein Hostname oder ein vom PVM-System vergebener Host-Identifikator (*dtid*) benutzt werden. Unterbleibt die Angabe eines Ausführungsortes, wird die Auswahl dem PVM-System überlassen, da in der DP(HIGH)-Toolbox keine eigenen Methoden zum transparenten Prozeß-Mapping implementiert wurden.

Konfigurationskosten und -konflikte

Im Abschnitt 6.2 wurde erwähnt, daß Operationen zur Konfiguration von Multi-SCEs relativ aufwendig sind und mit anderen Anwendungen, Nutzern oder systembedingten Limitierungen in Konflikt geraten können.

Besonders kritisch hinsichtlich des Laufzeitverhaltens sind die bereits genannten Konfigurationsoperationen der DP(HIGH)-Toolbox, die den Start neuer DP-Instanzen auslösen. Da hierfür Ausführungszeiten im Sekundenbereich benötigt werden, können sie bei Anwendungen, die die Konfiguration einer DPMM häufig ändern,

einen signifikanten Anteil an der Gesamtlaufzeit verursachen. Um das Laufzeitverhalten solcher Anwendungen zu verbessern, kann die DPMM-Verwaltung mit der Option:

```
dpmmopt('reconfig','no')
```

in eine Betriebsart geschaltet werden, in der die Konfigurationsfunktionen `dpdel`, `dpchg`, `dpmm` und `dpoff` die aus einer DPMM zu entfernenden DP-Instanzen nicht terminieren, sondern einer Pseudo-DPMM mit der *dpmmid*=0 zuordnen. Dort stehen sie nachfolgenden Konfigurationsoperationen zur "Wiederverwertung" zur Verfügung, wodurch die Gesamtzahl der erforderlichen Starts von DP-Instanzen unter Umständen erheblich verringert werden kann.

Konfigurationskonflikte können überall dort auftreten, wo Ressourcen geteilt bzw. gemeinsam benutzt werden müssen. Im Falle der DP(HIGH)-Toolbox stellen in diesem Sinne die Zahl der möglichen MATLAB-Instantiierungen und das PVM-System kritische Ressourcen dar.

Wie bei kommerziellen Systemen üblich, ist die Zahl der gleichzeitig benutzbaren MATLAB-Instanzen, in Abhängigkeit vom erworbenen Lizenztyp, auf einen maximalen Wert begrenzt. Damit können durch Ressourcenerschöpfung sowohl Konflikte zwischen verschiedenen Nutzern als auch zwischen Anwendungen eines Nutzers auftreten. Da solche Konflikte innerhalb der DP(HIGH)-Toolbox nicht transparent lösbar sind, werden Konfigurationsoperationen standardmäßig mit einer Fehlermeldung abgebrochen, wenn nicht mehr genügend MATLAB-Instanzen zur Verfügung stehen. Mit der Option:

```
dpmmopt('config','block')
```

kann der Nutzer das Verhalten bei Ressourcenerschöpfung dahingehend modifizieren, daß die Ausführung von Konfigurationsoperationen solange blockiert wird, bis die erforderliche Anzahl von Instanzen verfügbar ist.

Weitere Konflikte können aus der gemeinsamen Nutzung des PVM-Systems durch mehrere PVM-Anwendungen eines Nutzers entstehen. Da das PVM-System selbst dynamisch konfigurierbar ist, besteht die Möglichkeit, daß eine Anwendung die Entfernung eines Hosts aus dem PVM-System veranlaßt, der von einer anderen Anwendung noch benutzt wird. Darüber hinaus kann auch das gesamte PVM-System ohne Rücksicht auf eventuell noch laufende Anwendungen terminiert werden. Um die Verursachung solcher Konflikte durch DPMM-Anwendungen auszuschließen, werden alle erforderlichen Konfigurationsoperationen am PVM-System durch die Funktionen der DP(HIGH)-Toolbox implizit und nach folgender Strategie ausgeführt:

- Die Konfiguration des PVM-Systems wird niemals verkleinert[3].
- Bei der Terminierung von DPMMs wird kein Versuch unternommen, auch das PVM-System zu terminieren, wenn dieses bereits beim Start der DPMMs lief.
- Wurde das PVM-Systems erst in Folge einer DPMM-Konfiguration gestartet, wird es nur dann terminiert, wenn es durch keine weiteren Anwendungen benutzt wird.

[3]Das heißt es werden keine Hosts aus dem PVM-System entfernt.

Eigenschaften von DP-Instanzen

Bei der Konfiguration einer DPMM kann festgelegt werden, ob DP-Instanzen in einem eigenen Xterm (interaktive Instanzen) oder als Hintergrundprozesse (Background-Instanzen) laufen sollen. Der Output von Background-Instanzen kann auf das Terminal von interaktiven Instanzen oder in Dateien umgeleitet werden.

Graphikausgaben sind sowohl von interaktiven als auch von Background-Instanzen aus möglich. Die Steuerung erfolgt über die Umgebungsvariable `DISPLAY`.

Zum leichteren Auffinden von Programmfehlern können DP-Instanzen optional Trace-Dateien führen, in denen sämtliche Aufrufe von Funktionen der DP(HIGH)-Toolbox protokolliert werden.

7.3.2.2 Ausführung von Routinen auf einer DPMM

Die sequentielle Ausführung von MATLAB-Routinen auf den DP-Instanzen einer DPMM kann wie in gewöhnlichen MATLAB-Instanzen entweder durch Aufruf von der Kommandozeile (bei interaktiven Instanzen) oder über eine Eingabedatei (bei Background-Instanzen) erfolgen. Darüber hinaus existieren verschiedene Möglichkeiten zum entfernten Aufruf von Routinen.

Entfernter Routinenaufruf durch Terminalumschaltung

Mit Hilfe der Funktion `dpterm(`*dpid*`)` läßt sich das Nutzerinterface einer DP-Instanz auf eine andere Instanz umschalten, so daß alle nachfolgend aufgerufenen Routinen automatisch auf der mit dem Parameter *dpid* spezifizierten Instanz ausgeführt werden. Der Hintransport der Aufrufzeile sowie der Rücktransport einer eventuellen Ergebnisausgabe oder Fehlermeldung erfolgt über PVM-Nachrichten. Ein Beispiel für den entfernten Aufruf von Routinen durch Terminalumschaltung ist in Abbildung 7.6 dargestellt.

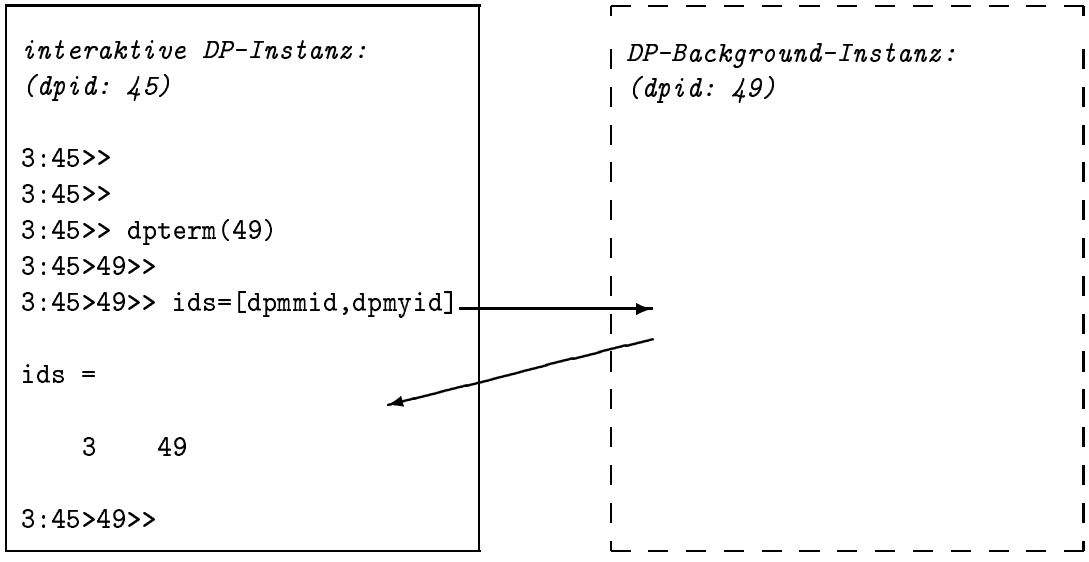


Abbildung 7.6: Entfernter Routinenaufruf durch Terminalumschaltung

Prinzipiell kann eine Terminalumschaltung sowohl auf Background- als auch auf interaktive Instanzen erfolgen. Letzteres ist aber wenig sinnvoll, da auf interaktiven Instanzen Routinen direkt von der Kommandozeile aus gestartet werden können.

Entfernter Routinenaufruf durch vektorielle RPCs

Durch direkten Routinenaufruf oder entfernten Aufruf mittels Terminalumschaltung kann der Nutzer – quasi manuell – auf mehreren DP-Instanzen einer DPMM Routinen bzw. Routinensequenzen parallel zur Ausführung bringen. Die sinnvolle Anwendbarkeit dieser Methode ist aber ausschließlich auf frühe Phasen der Prototyp-Entwicklung beschränkt. Für spätere Phasen des Prototypings und für die Produktionsphase stellt die DP(HIGH)-Toolbox zur parallelen Ausführung von Routinen den im Abschnitt 6.5.2 beschriebenen Mechanismus des vektoriellen Remote-Procedure-Calls in Form der Funktion `dpeval()` bereit.

Im Rahmen der nachrichten-orientierten Programmierung wird zum Starten von kompletten Anwendungen nur eine vereinfachte, parameterlose Form vektorieller RPCs benötigt. Dazu kann die Funktion `dpeval` in der folgenden Form benutzt werden:

`dpeval(`*`dpids,expressions,n`*`)`

Der optionale Parameter *dpids* ist ein Vektor oder eine Matrix mit den Identifikatoren der DP-Instanzen, auf denen Routinen ausgeführt werden sollen, wobei *dpids* auch den Identifikator der DP-Instanz, in der `dpeval` aufgerufen wird, enthalten darf. Unterbleibt die Angabe des Parameters *dpids* beim Aufruf, werden die DP-Instanzen zur Ausführung der Routinen von `dpeval` selbständig ausgewählt.

Der Parameter *expressions* ist eine Matrix, wobei jede Zeile einen oder mehrere MATLAB-Ausdrücke[4] enthält. Damit können mit `dpeval` neben Routinen auch Zuweisungen, arithmetische Operationen u.ä. ausgeführt werden. Enthält *expressions* nur eine Zeile, so wird diese nach dem SPMD-Modell auf allen spezifizierten bzw. automatisch ausgewählten DP-Instanzen ausgeführt. Für eine parallele Ausführung nach dem MPMD-Modell muß *expressions* für jede DP-Instanz eine Zeile enthalten.

Mit dem Parameter n kann festgelegt werden, auf wievielen DP-Instanzen `dpeval` Routinen ausführen soll. Die Angabe des Parameters n ist nur sinnvoll, wenn der Parameter *dpids* nicht angegeben wurde und *expressions* nur eine Zeile und damit einen nach dem SPMD-Modell auszuführenden Ausdruck enthält. Ansonsten ist die Anzahl der DP-Instanzen bereits durch die Anzahl der Elemente in *dpids* bzw. die Anzahl der Zeilen in *expressions* eindeutig bestimmt.

Bevor `dpeval` die auszuführenden Routinen durch entfernte Aufrufe startet, wird der vom Nutzer spezifizierte bzw. durch `dpeval` bestimmte Parameter *dpids* und der Identifikator der DP-Instanz auf der `dpeval` ausgeführt wird, an alle beteiligten DP-Instanzen übermittelt.

[4]Mehrere Ausdrücke in einer Zeile werden in MATLAB durch Komma oder Semikolon voneinander getrennt.

Innerhalb der ausgeführten Routinen können diese Informationen mit den Funktionen

dpids=dpevalids und *dpid*=dpevalparent

beschafft werden (s. Abb. 7.7).

Für die entfernte Ausführung von Routinen mit Übergabe von Aufrufparametern steht die Funktion:

dpfeval(*dpids*,*expressions*,*n*,[],*parameters*)

zur Verfügung. Eine detailierte Beschreibung der vollständigen Aufrufsyntax ist im Anhang C.1 enthalten.

```
3:45>> ids=[46,47];
3:45>> exps=
      [´dpevalparent´;
       ´dpevalids   ´];
3:45>> dpeval(ids,exps)
3:45>>
3:45>>
3:45>>
3:45>>
3:45>>
3:45>>
3:45>>
3:45>> dpeval(´dpmyid´,3)
3:45<45>> dpmyid

ans =

       45

3:45>>
```

```
3:46>>
3:46>>
3:46>>
3:46>>
3:46>>
3:46<45>> dpevalparent

ans =

       45

3:46>>
3:45>>
3:46<45>> dpmyid

ans =

       46

3:46>>
```

```
3:47>>
3:47>>
3:47>>
3:47>>
3:47>>
3:47<45>> dpevalids

ans =

       46   47

3:47>>
3:45>>
3:47<45>> dpmyid

ans =

       47

3:47>>
```

Abbildung 7.7: Parallele Ausführung von Routinen nach dem MPMD- und SPMD-Modell mit `dpeval`

Modifikation der Eingabeschleife von DP-Instanzen

Da entfernte Aufrufe von Routinen mittels Terminalumschaltung bzw. vektorieller RPCs nicht über den Standard-Eingabekanal erfolgen können, muß die Eingabeschleife von DP-Instanzen gegenüber gewöhnlichen MATLAB-Instanzen, wie in Abbildung 7.8 dargestellt, modifiziert werden. Dabei stellt sich bei interaktiven DP-Instanzen das Problem, daß Aufrufe sowohl über den Standard-Eingabekanal als auch über PVM-Nachrichten (im Folgenden vereinfacht als Netzwerkeingabe bezeichnet) erfolgen können und damit mehrere Kanäle auf eventuelle Eingaben geprüft werden

müssen. Da ein zyklisches Testen (*polling*) mehrerer Eingabekanäle eine erhebliche Prozessorbelastung zur Folge hätte, wird in der Eingabeschleife von interaktiven DP-Instanzen der Unix-Systemruf `select` benutzt. Durch `select` wird der rufende Prozeß suspendiert und verbraucht damit keine Rechenzeit mehr. Die Beobachtung der Eingabekanäle erfolgt durch das Betriebssystem, und eine erneute Aktivierung des Prozesses findet erst statt, wenn tatsächlich Eingaben auf einem der Kanäle erfolgen.

Für eine ausführliche Beschreibung des Systemrufs `select` sei auf [79] verwiesen.

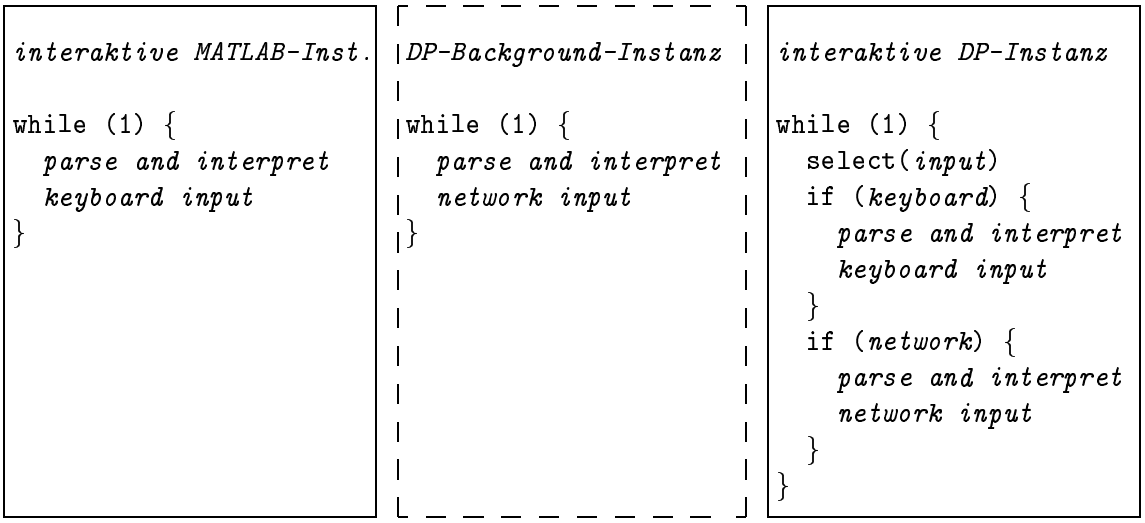


Abbildung 7.8: Eingabeschleifen bei gewöhnlichen MATLAB- und DP-Instanzen

7.3.2.3 Kommunikation zwischen DP-Instanzen

Zur Kommunikation stellt die DP(HIGH)-Toolbox die Funktionen `dpsend` und `dprecv` bereit, mit denen Matrizen zwischen DP-Instanzen transportiert werden können. Gemäß dem Multi-SCE-Konzept (s. Abschn. 6.4) müssen diese Funktionen alle für den Datentransport notwendigen Teiloperationen für den Nutzer transparent ausführen. Darüber hinaus soll die Selektion von Nachrichten nicht, wie in Message-Passing-Systemen üblich, mit Integer-Identifikatoren, sondern anhand der Namen der zu transportierenden Matrizen erfolgen. Um diese Forderungen zu erfüllen, besitzen die Kommunikationsfunktionen der DP(HIGH)-Toolbox folgende Funktionalität:

`dpsend`:

- Vergleichen der Datenrepräsentation auf der Sender- und Empfängerseite; bei Heterogenität Datenkonvertierung durch das PVM-System veranlassen
- Überführen einer MATLAB-Matrix in ein PVM-konformes Nachrichtenobjekt (vollständiger Algorithmus, siehe Anh. C.2)
- Übergeben der Nachricht an den oder die Empfänger

`dprecv`:

- Verwalten und Selektieren ankommender Nachrichten anhand von Namen
- Überführen eines PVM-Nachrichtenobjektes in eine MATLAB-Matrix (vollständiger Algorithmus, siehe Anh. C.2)

Die Semantik der Sende- und Empfangsfunktion der DP(HIGH)-Toolbox wird im wesentlichen durch die Kommunikationsoperationen des PVM-Systems bestimmt, d.h. es handelt sich um sequentielle Operationen, wobei das Senden asynchron erfolgt.

Neben der Möglichkeit, komplette Matrizen zwischen DP-Instanzen auszutauschen, unterstützt die DP(HIGH)-Toolbox auch das Verteilen und Wiedereinsammeln von Vektor- und Matrizenelementen (`dpscatter` und `dpgather`). Im Abschnitt 6.5.1 wurde erläutert, daß die Realisierung solcher Funktionen in einer Multi-SCE auf Basis der kollektiven Transportoperationen des Message-Passing-Systems oder auf Basis der Sende- und Empfangsfunktionen der Multi-SCE erfolgen kann. Wegen der besseren Portabilitätseigenschaften der zweiten Variante wurde diese zur Implementation der Toolbox-Funktionen benutzt.

7.4 Leistungsvergleich

Die Leistungsbewertung einer Multi-SCE mit Hilfe von absoluten Maßen ist im hohen Grade plattformabhängig und besitzt damit kaum allgemeinen Aussagewert. Da eine Multi-SCE in erster Linie eine komfortable Alternative zur direkten Arbeit mit einem Message-Passing-System darstellt, ist eine vergleichende Bewertung beider Ansätze mit relativen Maßen sinnvoller. Es sei aber darauf hingewiesen, daß auch eine relative Leistungsbewertung prinzipiell plattformabhängig ist.

Im folgenden sollen die Datentransport-, Instantiierungs- und Konfigurationsoperationen der DP-Toolbox mit denen des Message-Passing-Systems PVM verglichen werden. Die präsentierten Ergebnisse beruhen auf Leistungsmessungen, die auf einem Computercluster bestehend aus Sun-Classic-Workstations und 10Mbit-Ethernet (switched) durchgeführt wurden.

Bei der Interpretation der Untersuchungsergebnisse sollten zwei Punkte beachtet werden:

1. Der Leistungsvergleich für die untersuchten Operationen läßt keine unmittelbaren Schlüsse auf das Verhalten kompletter Anwendungen zu, da die Operationen je nach Anwendung unterschiedlich häufig benutzt werden.[5]

2. Bei der DP-Toolbox handelt es sich um eine Prototypimplementation, die bezüglich des Laufzeitverhaltens nicht optimiert wurde.

[5]Ein Leistungsvergleich von Anwendungen erfolgt im Kapitel 8.

7.4.1 Datentransport

Die Leistungsfähigkeit der Kommunikationsfunktionen wurde mit Hilfe der RTT-Methode bestimmt. Bei dieser Methode werden Daten zwischen zwei Kommunikationspartnern hin und her transportiert. Die Zeit für einen Datenumlauf ist die *RTT* (*round trip time*).

Als Testdaten wurden Zufallsvektoren mit unterschiedlichen Dimensionen benutzt. Für jede Dimension wurde eine mittlere $\overline{RTT}_{DP}$ und $\overline{RTT}_{PVM}$ über 1000 Datenumläufe ermittelt. Bezieht man $\overline{RTT}_{DP}$ auf $\overline{RTT}_{PVM}$, so erhält man mit $\overline{RTT}_{DP}/\overline{RTT}_{PVM}$ ein relatives Maß für den Overhead der DP-Toolbox gegenüber PVM beim Datentransport.

Der obere Teil der Abbildung 7.9 zeigt die graphische Darstellung der Untersuchungsergebnisse für die Sende-Empfangs-Operationen. Darin ist zu erkennen, daß der Transport von kleinen Datenmengen (weniger als 10 Fließpunktzahlen in doppeltgenauer Darstellung bzw. 80 Bytes) mit Hilfe der DP-Toolbox gegenüber PVM etwa zweimal soviel Zeit in Anspruch nimmt. Mit zunehmender Dimension der transportierten Daten wird der Overhead der DP-Toolbox immer geringer und ab Datenmengen von etwa 1000 Fließpunktzahlen (8 Kbytes) schließlich vernachlässigbar klein.

Diese Verhalten ist damit zu erklären, daß der Zeitverbrauch der Operationen, die mit der DP-Toolbox gegenüber PVM zusätzlich auszuführen sind, mit wachsender Datendimension weniger steigt als die für den eigentlichen Transport der Daten benötigte Zeit.

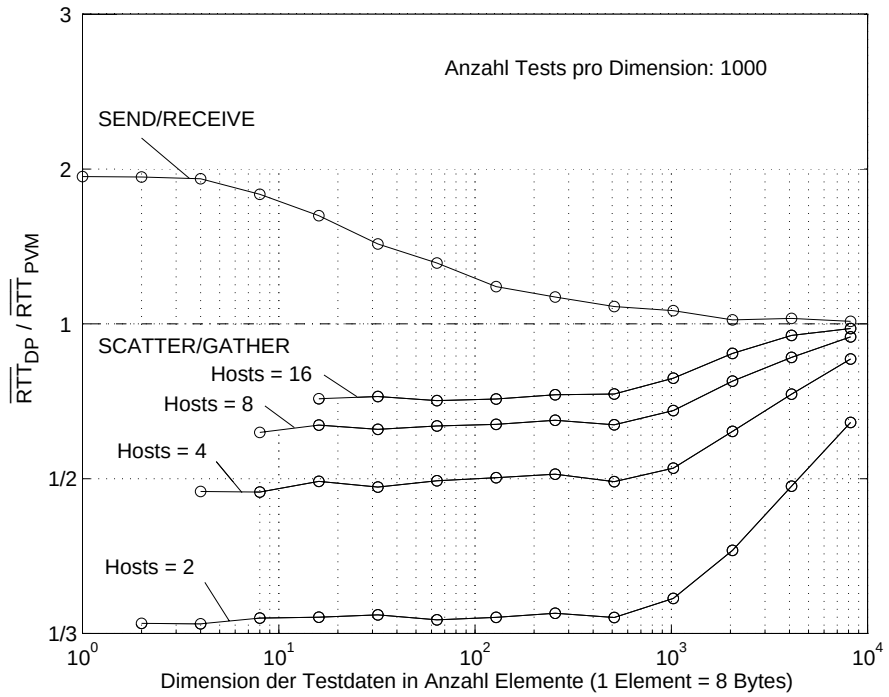


Abbildung 7.9: DP/PVM-Leistungsvergleich – Datentransport

Im unteren Teil der Abbildung 7.9 sind die Ergebnisse der Scatter/Gather-Operationen dargestellt. Im Unterschied zur Punkt-zu-Punkt-Kommunikation wur-

den hier die Elemente der Testvektoren an mehrere Kommunikationspartner verteilt und wieder eingesammelt. Dabei zeigte sich, daß bei diesen Operationen PVM gegenüber der DP-Toolbox einen deutlichen Overhead aufweist. Dieser ist bei nur zwei Kommunikationspartnern und Datenmengen von weniger als 1000 Fließpunktzahlen (8 Kbytes) am größten (etwa dreifache Ausführungszeit) und nimmt mit zunehmender Zahl von Kommunikationspartnern und wachsender Datendimension ab.

Der PVM-Overhead liegt in der unterschiedlichen internen Arbeitsweise der DP- und PVM-Funktionen begründet. Im Abschnitt 7.3.2.3 wurde erwähnt, daß die Scatter- und Gather-Funktionen der DP-Toolbox nicht auf Basis der korrespondierenden PVM-Funktionen realisiert wurden, sondern auf Basis der Punkt-zu-Punkt-Kommunikationsfunktionen Send und Receive. Dabei wurde auf die Einführung eines Gruppenkonzeptes (wie bei den kollektiven PVM-Funktionen, s. Abschn. 3.3) verzichtet, d.h. beim Aufruf der Scatter- und Gather-Funktionen der DP-Toolbox müssen die Identifikatoren aller Kommunikationspartner angegeben werden. Innerhalb der Funktionen stehen diese dann unmittelbar für die Punkt-zu-Punkt-Kommunikation zur Verfügung.

Im Gegensatz dazu werden die Kommunikationspartner bei den PVM-Funktionen Scatter und Gather mit Hilfe eines Gruppenidentifikators spezifiziert. Innerhalb der Funktionen müssen anhand des Gruppenidentifikators die Identifikatoren der Kommunikationspartner ermittelt werden. Da dies über eine Anfrage bei einem externen Prozeß (dem Gruppenserver) erfolgt, ist gegenüber den Scatter- und Gather-Funktionen der DP-Toolbox zusätzliche Kommunikation erforderlich, was zu dem in den Messungen nachgewiesenen PVM-Overhead führt.[6]

7.4.2 Instantiierungsoperationen

Die Gemeinsamkeiten und Unterschiede der Ausführung von Anwendungen in Multi-SCEs und Message-Passing-Systemen wurden ausführlich im Abschnitt 6.3 dargestellt. Danach sind PVM-Anwendungen als Prozesse zu instantiieren und Multi-SCE-Anwendungen auf Basis der DP-Toolbox als MATLAB-Routinen auszuführen.

Sowohl PVM als auch die DP-Toolbox unterstützen mehrere Instantiierungsmethoden (s. Abschn. 3.1.1 und 7.3.2.2). Da für die Produktionsphase jeweils nur eine Methode relevant ist (PVM: `pvm_spawn`; DP-Toolbox: `dpeval`), wurden nur für diese Leistungsuntersuchungen durchgeführt.

Als Maß für die Instantiierungsleistung wurde die *TST* (*task startup time*) ermittelt, d.h. die Zeit, die vom Auslösen einer Instantiierungsoperation bis zur Ausführung der ersten Anweisung in der gestarteten Task (PVM-Prozeß bzw. MATLAB-Routine) verstreicht. Als Testanwendung wurde die Implementation eines kontinuierlichen Simulationsproblems (numerische Integration eines Differentialgleichungssystems mit zwei Zustandsvariablen zur Beschreibung eines Feder-Masse-Systems, s. Abschn. 8.1) benutzt.

[6] Die bei kollektiven Operationen gegebene Möglichkeit des zeitoptimalen Datentransports über eine Baumstruktur der Operationsteilnehmer wird in der generischen PVM-Implementation nicht genutzt.

Da mit `pvm_spawn` und `dpeval` multiple Instantiierungen möglich sind, wurden die mittleren $\overline{TST}_{PVM}$ und $\overline{TST}_{DP}$ für verschiedene Taskzahlen auf Basis von jeweils 1000 Messungen bestimmt. Die Untersuchungsergebnisse sind in Abbildung 7.10 dargestellt.

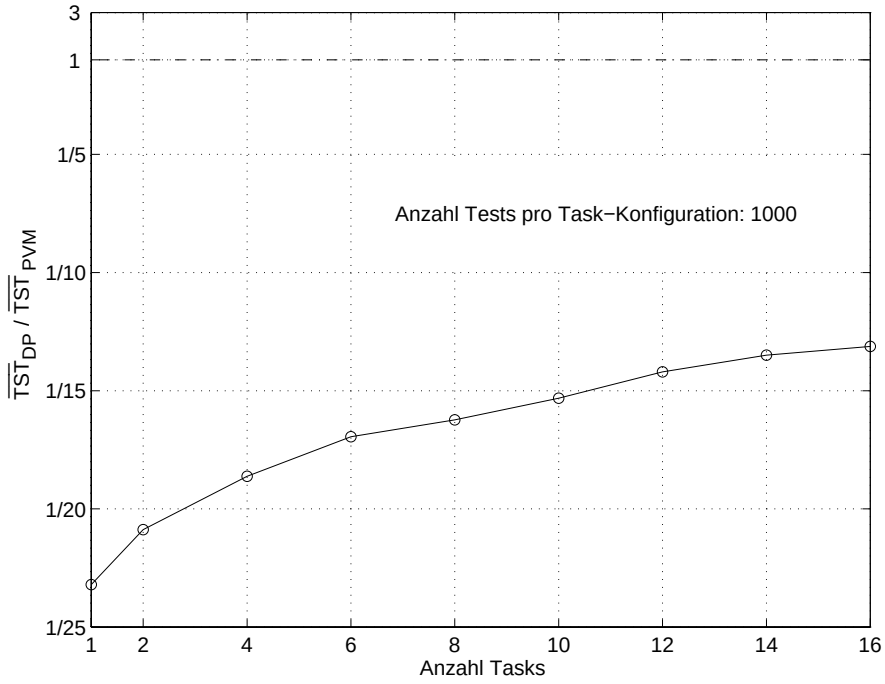


Abbildung 7.10: DP/PVM-Leistungsvergleich – Instantiierung

Erwartungsgemäß weist PVM aufgrund des großen Aufwands für die Instantiierung von Betriebssystemprozessen einen erheblichen Overhead gegenüber der entfernten Ausführung von MATLAB-Routinen mit Hilfe der DP-Toolbox auf (um eine Dekade größere Instantiierungszeiten).

7.4.3 Konfigurationsoperationen

Häufig erfolgt die PVM- bzw. DPMM-Konfiguration vor dem Start einer oder mehrerer Anwendungen. In diesen Fällen spielt die Leistungsfähigkeit der Konfigurationsoperationen für die Anwendungen keine unmittelbare Rolle. Werden innerhalb einer Anwendung Konfigurationsoperationen ausgeführt, sind für das Gesamtlaufzeitverhalten nur Konfigurationserweiterungen kritisch. (PVM: Hinzufügen von Hosts mit `pvm_addhosts`; DPMM: Hinzufügen von MATLAB-Instanzen mit `dpadd` oder `dpmm`).

Im Rahmen der Leistungsuntersuchungen wurden die mittleren Ausführungszeiten für Konfigurationserweiterungen (*configuration extension time, CET*) um ein bis maximal 16 Hosts bzw. Instanzen auf Basis von jeweils 1000 Messungen für PVM und die DP-Toolbox bestimmt. Für die DP-Toolbox wurden

drei verschiedene Meßreihen unter den folgenden Randbedingungen ermittelt:

1. *DPMM-Betriebsart: reconfig=yes; PVM nicht konfiguriert:*
   DPMM-Konfigurationserweiterungen werden ausschließlich über den Neustart von MATLAB-Instanzen realisiert. Vor dem Start der Instanzen wird die PVM-Konfiguration jeweils um die erforderliche Anzahl von Hosts erweitert.

2. *DPMM-Betriebsart: reconfig=yes; PVM konfiguriert:*
   DPMM-Konfigurationserweiterungen werden ausschließlich über den Neustart von MATLAB-Instanzen realisiert. Implizite Erweiterungen der PVM-Konfiguration sind nicht erforderlich.

3. *DPMM-Betriebsart: reconfig=no :*
   DPMM-Konfigurationserweiterungen werden ausschließlich durch „Wiederverwendung“ von MATLAB-Instanzen aus der Pseudo-DPMM realisiert (s. Abschn. 7.3.2.1).

Abbildung 7.11 zeigt die Ergebnisse der verschiedenen $\overline{CET}_{DP}$-Messungen bezogen auf die $\overline{CET}_{PVM}$-Messung.

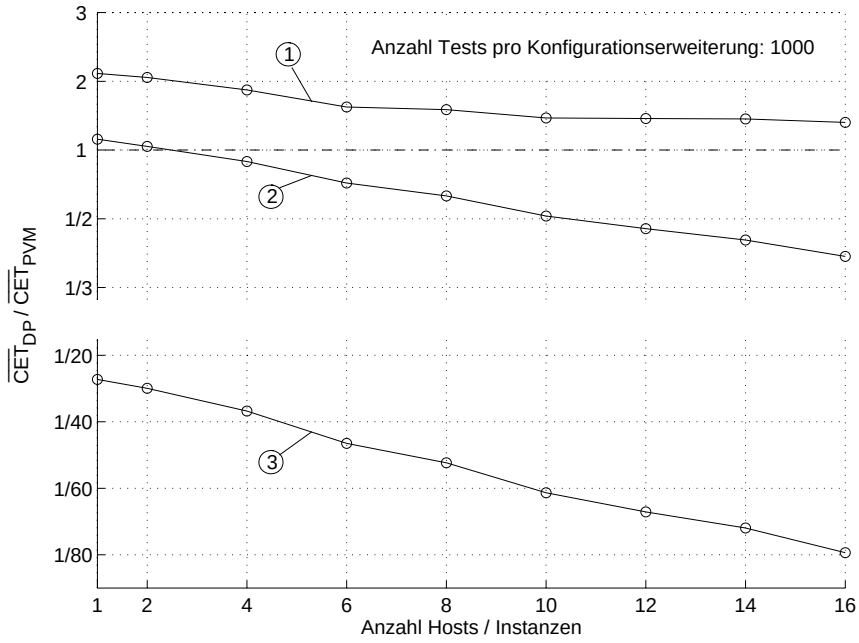


Abbildung 7.11: DP/PVM-Leistungsvergleich – Konfiguration

Bei einem nicht konfigurierten PVM-System und der Betriebsart *reconfig=yes* (Kurve 1) weisen DPMM-Konfigurationserweiterungen generell einen Overhead gegenüber PVM-Konfigurationserweiterungen auf (ca. 1,4- bis 2-fache Ausführungszeit), der auch für größere Konfigurationserweiterungen nicht vernachlässigbar klein wird. Die Ursache für dieses Overheadverhalten ist, daß bei jeder DPMM-Konfigurationserweiterung auch eine implizite Erweiterung der PVM-Konfiguration vorgenommen werden muß.

Sind keine impliziten Erweiterungen der PVM-Konfiguration notwendig (Kurve 2), weisen DPMM-Konfigurationserweiterungen um ein oder zwei MATLAB-Instanzen nur noch einen vernachlässigbaren Overhead auf. Für größere Konfigurationserweiterungen stellt sich ein linear ansteigender PVM-Overhead ein. Dieser ist dadurch erklärbar, daß die Ausführungszeit für DPMM-Konfigurationserweiterungen nahezu konstant und damit weitgehend unabhängig von der Anzahl der zu startenden MATLAB-Instanzen ist (parallele Instantiierung). Die Ausführungszeit von PVM-Konfigurationserweiterungen steigt dagegen mit der Anzahl der aufzunehmenden Hosts.

Steht in der Betriebsart *reconfig=no* eine ausreichende Anzahl von bereits laufenden MATLAB-Instanzen innerhalb der Pseudo-DPMM zur Verfügung (Kurve 3), so werden diese für Konfigurationserweiterungen benutzt, wodurch der aufwendige Neustart von MATLAB-Instanzen gänzlich entfällt. Dies führt zu einer starken Verkürzung der Ausführungszeit von DPMM-Konfigurationserweiterungen, was sich in einem erheblichen PVM-Overhead widerspiegelt.

Kapitel 8

Anwendungen

Die DP-Toolbox wurde bisher am Institut für Automatisierungstechnik der Universität Rostock für folgende Anwendungen eingesetzt:

- parallele Parameteroptimierung an einem pflanzen-physiologischen Wachstumsmodell ([80]),
- Parallelisierung des Stabilitätstests für lineare Systeme nach der Methode der konvexen Zerlegung ([23]),
- verteilte Steuerung eines Laborautomationssystems ([24]),
- diverse Kopplungen von MATLAB mit externen Programmen ([49]).

Gegenwärtige Forschungsarbeiten beschäftigen sich mit der Parallelisierung von transaktionsorientierten Simulationen auf Basis der MATLAB-GPSS-Toolbox ([69]) in Verbindung mit der DP-Toolbox.

Bei verschiedenen Nachnutzern (u.a. Universität Magdeburg, s. [6]; Daimler Benz AG, s. [70]; Universität Valladolid, Spanien; Institut Eurecom, Frankreich) wurden Public-Domain-Versionen der DP-Toolbox zur Implementation von parallelen Evolutionsalgorithmen und für verteilte Simulationen eingesetzt.

Da eine angemessene Diskussion der oben genannten Anwendungen im Rahmen dieser Arbeit nicht möglich ist, sei an dieser Stelle auf die angegebenen Publikationen verwiesen.

Für eine nähere Betrachtung in diesem Kapitel wurden drei Beispielanwendungen ausgewählt, die von F. Breitenecker als Testprobleme für parallele Simulationssysteme in [9] ausgeschrieben worden sind. Sie zeichnen sich durch folgende Eigenschaften aus:

- Aufgrund ihrer überschaubaren Komplexität können alternative Lösungen mit Hilfe von PVM und der DP-Toolbox anhand beispielhaften Programmkodes dargestellt und miteinander verglichen werden.
- Darüber hinaus sind in der Zeitschrift *EUROSIM – Simulation News Europe (SNE)* zahlreiche Lösungen der Testprobleme mit parallelen Simulationssystemen veröffentlicht worden, die ebenfalls zum Vergleich herangezogen werden können.

- Die Testprobleme überstreichen einen größeren Granularitätsbereich, wodurch die Grenzen einer sinnvollen Anwendbarkeit des Multi-SCE-Ansatzes auf Systemen ohne gemeinsamen Speicher aufgezeigt werden können.

Sämtliche in diesem Kapitel präsentierten Ergebnisse beruhen auf Leistungsuntersuchungen, die auf der bereits im Abschnitt 7.4 beschriebenen Plattform (SUN-Classic-Workstations, 10Mbit-Switched-Ethernet) durchgeführt wurden.

8.1 Monte-Carlo-Studie

8.1.1 Aufgabenstellung

Es soll ein gedämpftes Feder-Masse-System zweiter Ordnung, das durch die Gleichung

$$\begin{aligned} & m\ddot{x}(t) + kx(t) + d\dot{x}(t) = 0 \\ & \text{mit:} \quad \dot{x}(0) = 0\,, \quad x(0) = 0.1\,, \quad k = 9000\,, \quad m = 450 \end{aligned} \tag{8.1}$$

beschrieben wird, simuliert werden, wobei der Dämpfungsfaktor d aus dem Intervall $[800, 1200]$ zufällig zu wählen ist (Gleichverteilung). Insgesamt sind 1000 Simulationsläufe mit dem RK4-Verfahren[1] (feste Schrittweite: $h = 0.001$) für das Zeitintervall $[0, 2]$ durchzuführen. Aus den Ergebnistrajektorien $x(t)$ (s. Abb. 8.1) soll ein mittlerer Bewegungsverlauf $\bar{x}(t)$ berechnet werden.

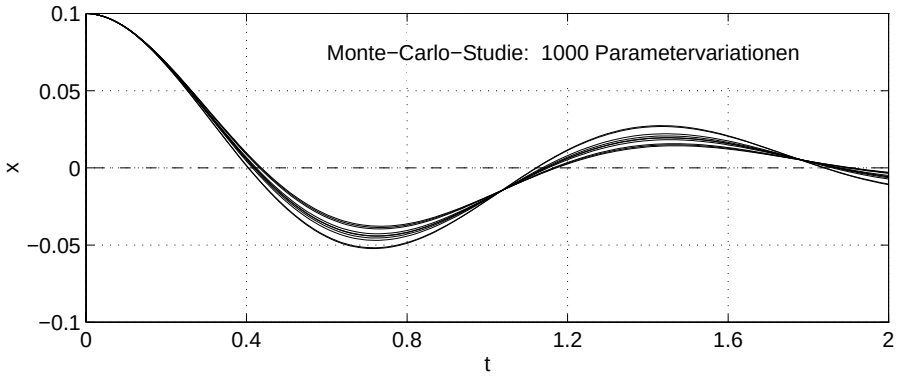


Abbildung 8.1: Simulierte Trajektorien des Feder-Masse-Systems

8.1.2 Parallele Lösung

Zur Integration mit dem RK4-Verfahren muß die Differentialgleichung zweiter Ordnung (Gl. 8.1) in ein System von Differentialgleichungen erster Ordnung überführt werden:

$$\begin{aligned} & \dot{x}_1(t) = -\tfrac{d}{m}x_1(t) - \tfrac{k}{m}x_2(t)\,, \\ & \dot{x}_2(t) = x_1(t) \\ & \text{mit:} \quad x_1(0) = \dot{x}(0) = 0\,, \quad x_2(0) = x(0) = 0.1\,. \end{aligned} \tag{8.2}$$

[1] Runge-Kutta-Integrationsverfahren 4. Ordnung (s. [8])

Bei der parallelen Lösung der Aufgabenstellung stellt sich das Problem, die Simulationsläufe der Monte-Carlo-Studie möglichst gleichmäßig auf die zur Verfügung stehenden Prozessoren bzw. Rechner[2] zu verteilen. Eine völlig gleichmäßige Auslastung der Prozessoren ergeben Prozessorzahlen p, für die gilt:

$$\frac{Anzahl\ Simulationsläufe}{p} \in \mathbf{G}^{+} \ . \tag{8.3}$$

Abbildung 8.2 zeigt die Prinzipdarstellung einer parallelen Lösung nach dem Master-Slave-Paradigma, in der die Dämpfungsfaktoren vom Master generiert und zusammen mit den Simulationsparametern an die Slaves verteilt werden. Die Slaves übernehmen jeweils einen Teil der Monte-Carlo-Studie und führen für diesen bereits die Mittelwertberechnung aus, wodurch der Datenumfang der Ergebnisrückgabe an den Master minimiert wird. Nach Erhalt sämtlicher Teilergebnisse wird der Mittelwertverlauf der gesamten Studie vom Master berechnet und abgespeichert.

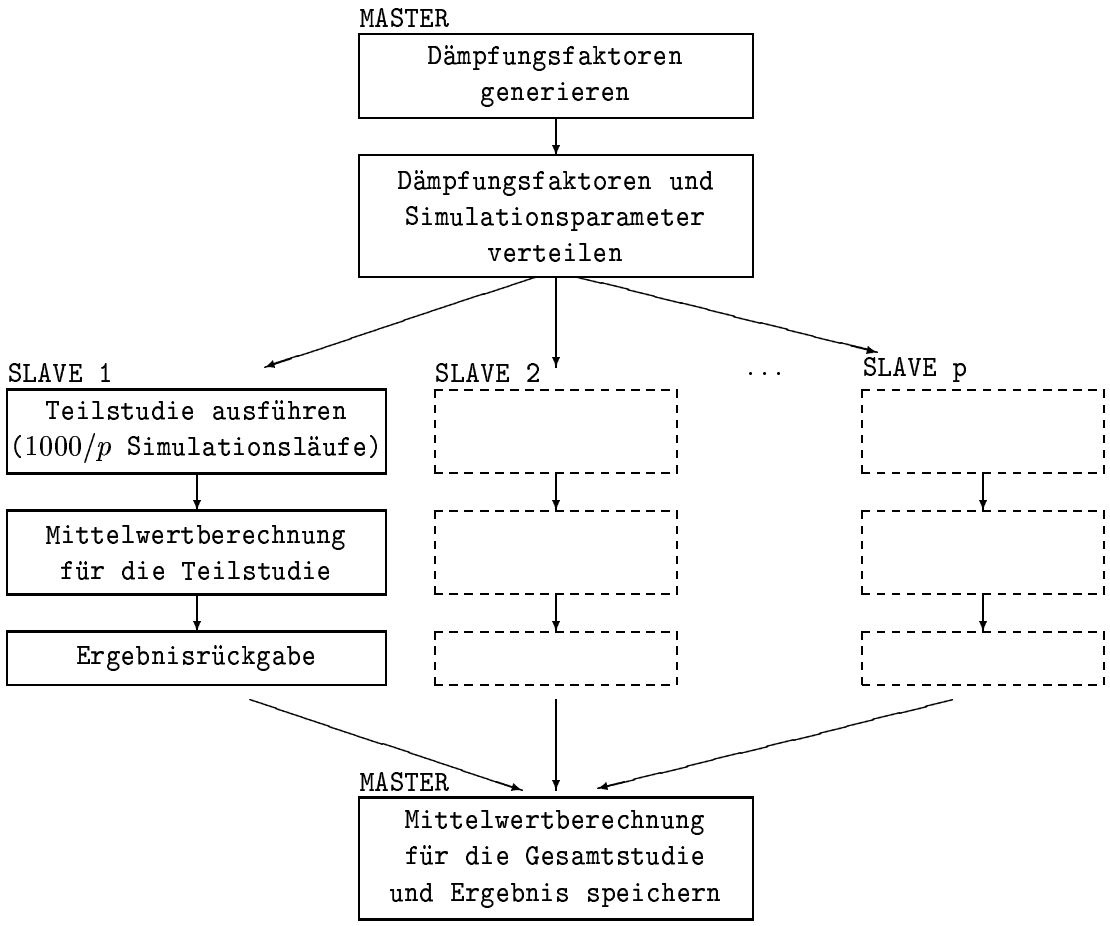


Abbildung 8.2: Prinzipdarstellung der parallelen Lösung

[2]Der Einfachheit halber wird im folgenden nur von Prozessoren gesprochen.

Beispielimplementierungen der parallelen Lösung auf Basis der DP-Toolbox und von PVM sind im Anhang D.1 enthalten. Betrachtet man den Umfang des Programmkodes beider Lösungen und bedenkt darüber hinaus, daß die Inbetriebnahme der MATLAB-Implementation interaktiv erfolgte, so ist leicht nachzuvollziehen, daß für die MATLAB-basierte Lösung ein wesentlich geringerer Entwicklungsaufwand notwendig war als für die PVM-Lösung.

8.1.3 DP/PVM-Leistungsvergleich

Da zwischen den Slaves keinerlei Kommunikation erforderlich ist, weist der in Abbildung 8.2 dargestellte parallele Lösungsansatz eine ausgesprochen grobe Granularität auf. Dementsprechend sind auch auf einem System ohne gemeinsamen Speicher positive Parallelisierungsergebnisse zu erwarten, was durch die vorgenommenen Leistungsuntersuchungen bestätigt werden konnte.

In Abbildung 8.3 sind die ermittelten Speedupwerte und die daraus ermittelte Effizienz in Abhängigkeit von der Prozessorzahl dargestellt. Sowohl die PVM- als auch die DP-Lösung weisen im Bereich der untersuchten Prozessorzahlen einen Parallelitätsgewinn in der Nähe des idealen Speedups auf. Die Effizienz sinkt mit steigender Prozessorzahl nur langsam und liegt bei 10 Prozessoren noch bei ca. 94% (PVM) bzw. 90% (DP-Toolbox).

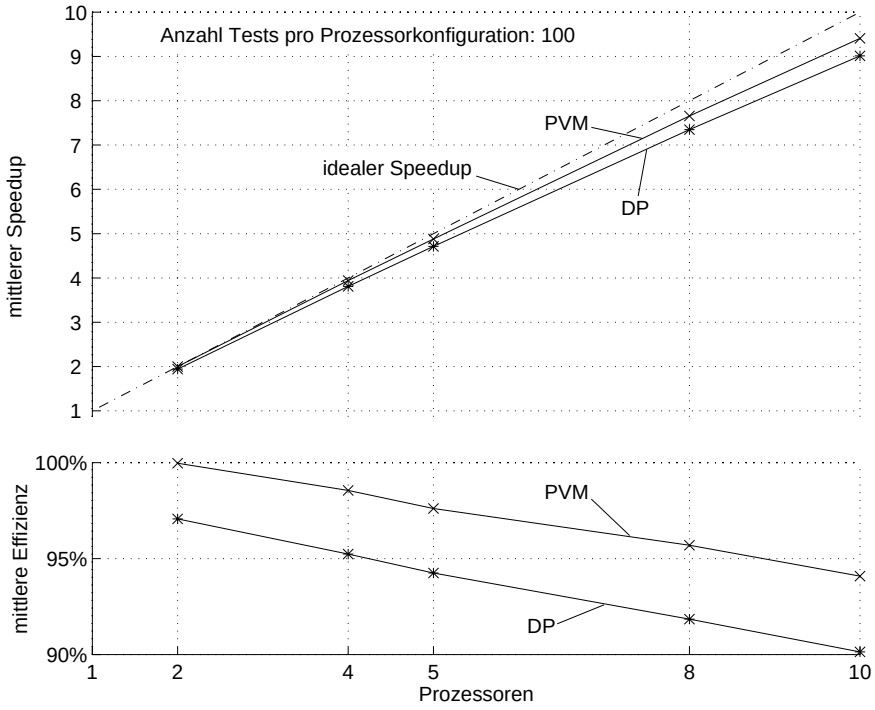


Abbildung 8.3: Speedup und Effizienz in Abhängigkeit von der Prozessorzahl

Beim direkten Vergleich der mittleren Laufzeiten $\overline{RT}_{DP}$ und $\overline{RT}_{PVM}$ ergibt sich ein geringfügiger Overhead der MATLAB-basierten Lösung (1,2- bis 1,3-fache Ausführungszeit, s. Abb. 8.4). Berücksichtigt man, daß bereits die sequentielle

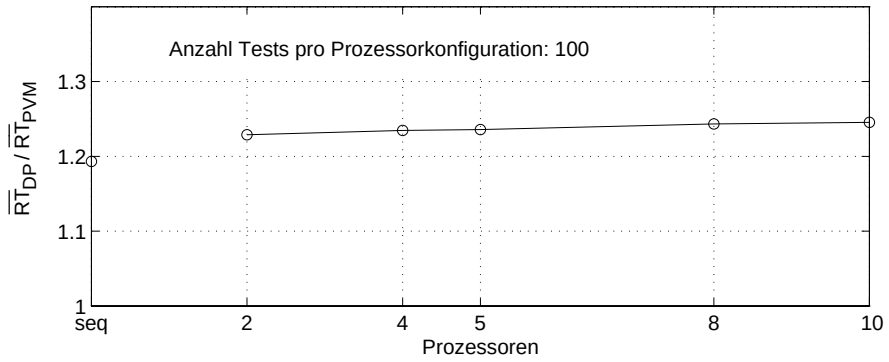


Abbildung 8.4: Laufzeitoverhead der MATLAB/DP-TB-Implementierung

MATLAB-Lösung einen Overheadfaktor von 1,2 aufweist, so wird klar, daß der durch die DP-Toolbox verursachte „zusätzliche“ Overhead nur einen Bruchteil am Gesamtoverhead ausmacht.

8.2 Simulation eines Räuber-Beute-Systems

8.2.1 Aufgabenstellung

Gegeben sei ein gekoppeltes System von 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), welches durch die folgenden Gleichungssysteme beschrieben wird[3]:

$$\begin{aligned} \dot{v}_1(t) &= a_v v_1(t) - b_v v_1(t) v_2(t) - c_v v_1(t)^2 \ , \\ \dot{v}_2(t) &= -d_v v_2(t) + e_v v_1(t) v_2(t) - f_v v_2(t)^2 + r_v \ , \\ r_v &= v_2(t)(g_v w_1(t) + h_v x_1(t) + j_v y_1(t) + k_v z_1(t)) \ ; \end{aligned} \tag{8.4}$$

$$\begin{aligned} \dot{w}_1(t) &= a_w w_1(t) - b_w w_1(t) w_2(t) - c_w w_1(t)^2 + r_w \ , \\ \dot{w}_2(t) &= -d_w w_2(t) + e_w w_1(t) w_2(t) - f_w w_2(t)^2 \ , \\ r_w &= w_1(t)(-g_w v_2(t) + h_w x_2(t)) \ ; \end{aligned} \tag{8.5}$$

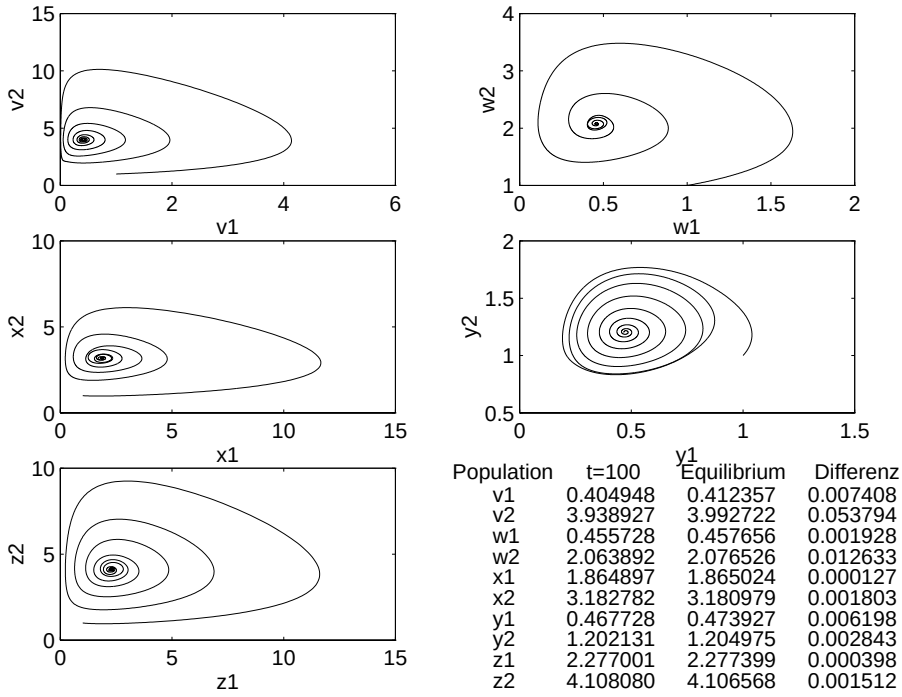
$$\begin{aligned} \dot{x}_1(t) &= a_x x_1(t) - b_x x_1(t) x_2(t) - c_x x_1(t)^2 + r_x \ , \\ \dot{x}_2(t) &= -d_x x_2(t) + e_x x_1(t) x_2(t) - f_x x_2(t)^2 + s_x \ , \\ r_x &= -g_x x_1(t) v_2(t) \ , \\ s_x &= -h_x x_2(t) w_1(t) \ ; \end{aligned} \tag{8.6}$$

$$\begin{aligned} \dot{y}_1(t) &= a_y y_1(t) - b_y y_1(t) y_2(t) - c_y y_1(t)^2 + r_y \ , \\ \dot{y}_2(t) &= -d_y y_2(t) + e_y y_1(t) y_2(t) - f_y y_2(t)^2 \ , \\ r_y &= y_1(t)(-g_y v_2(t) + h_y z_2(t)) \ ; \end{aligned} \tag{8.7}$$

[3] Anfangszustände und Modellparameter sind im Anhang D.2 aufgeführt.

$$
\begin{aligned}
\dot{z}_1(t) &= a_z z_1(t) - b_z z_1(t) z_2(t) - c_z z_1(t)^2 + r_z \ , \\
\dot{z}_2(t) &= -d_z z_2(t) + e_z z_1(t) z_2(t) - f_z z_2(t)^2 + s_z \ , \\
r_z &= -g_z z_1(t) v_2(t) \ , \\
s_z &= -h_z z_2(t) y_1(t) \ ;
\end{aligned}
\tag{8.8}
$$

Die Aufgabenstellung besteht in der Simulation des Systems im Zeitintervall $[0, 100]$ mit dem RK4-Verfahren (feste Schrittweite: $h = 0.01$) und der Bestimmung der Populationsgrößen zum Zeitpunkt $t = 100$. Die Ergebnisse einer solchen Simulation sowie die Differenzen zwischen dem Equilibrium[4] und den simulierten Populationsgrößen zum Zeitpunkt $t = 100$ sind in Abbildung 8.5 dargestellt.



Population	t=100	Equilibrium	Differenz
v1	0.404948	0.412357	0.007408
v2	3.938927	3.992722	0.053794
w1	0.455728	0.457656	0.001928
w2	2.063892	2.076526	0.012633
x1	1.864897	1.865024	0.000127
x2	3.182782	3.180979	0.001803
y1	0.467728	0.473927	0.006198
y2	1.202131	1.204975	0.002843
z1	2.277001	2.277399	0.000398
z2	4.108080	4.106568	0.001512

Abbildung 8.5: Simulationsergebnisse

8.2.2 Parallele Lösung

Aus Sicht der Modellierung ist eine Parallelisierungsstrategie, bei der die Populationspaare als Teilsysteme auf jeweils einem Prozessor simuliert werden (s. Abb. 8.6), am naheliegendsten.

Eine parallele Lösung auf Basis dieser Strategie ist jedoch nicht skalierbar, weil die Anzahl der Prozessoren durch die Problemstruktur vorgegeben ist ($p = 5$ für die gegebene Aufgabenstellung).

[4] Systemzustand bei $t \to \infty$ (Gleichgewichtszustand)

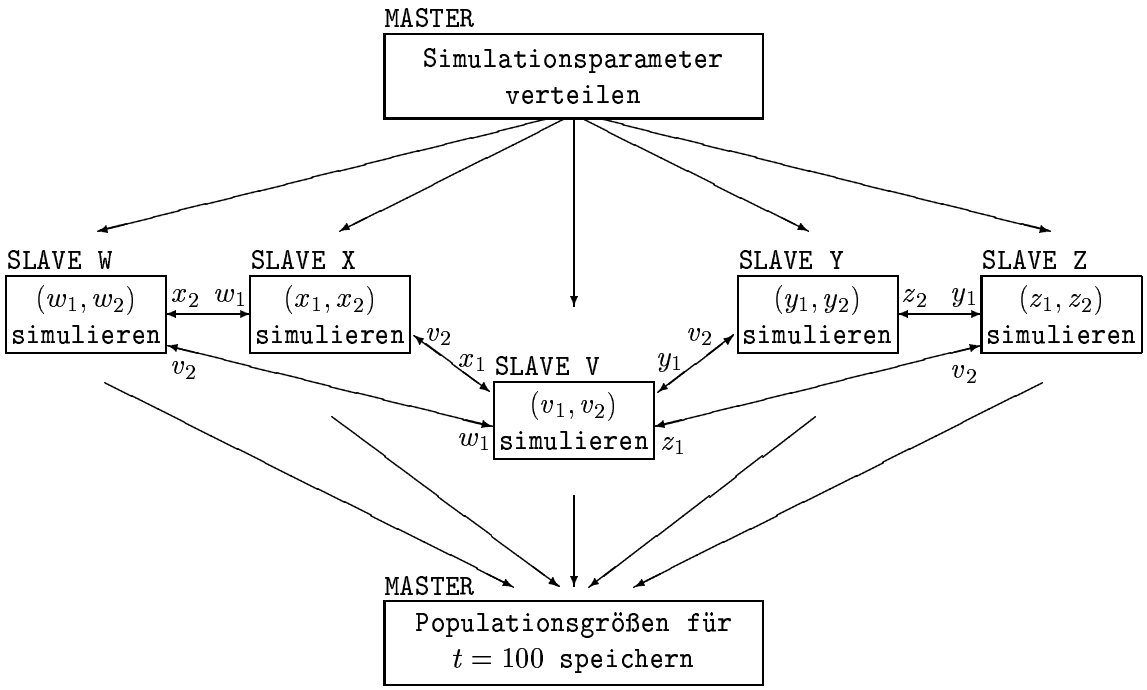


Abbildung 8.6: Prinzipdarstellung der parallelen Lösung

Darüber hinaus müssen die Slaves aufgrund der zwischen den Teilsystemen bestehenden Kopplungen untereinander kommunizieren. Soll die Genauigkeit der parallelen Simulation exakt der der sequentiellen entsprechen, so müssen beim vorgeschriebenen RK4-Verfahren die aktuellen Werte der Zustandsvariablen, die für die Kopplungen relevant sind, pro Integrationsschritt viermal ausgetauscht werden. Da der Berechnungsaufwand zur Simulation der Teilsysteme gering ist (jeweils nur zwei Zustandsvariablen), ergibt sich ein für die Abarbeitung auf parallelen Plattformen ohne gemeinsamen Speicher ungünstiges Rechen-/Kommunikationsverhältnis (feine Granularität).

Eine Möglichkeit, die Granularität der parallelen Lösung zu erhöhen, besteht darin, aktuelle Werte der Zustandsvariablen nur in Intervallen ($CINT$, *communication interval*), die ein ganzzahliges Vielfaches der Integrationsschrittweite betragen, auszutauschen. Zwischen den Kommunikationszeitpunkten werden dann zur Integration der Teilsysteme anstelle der aktuellen die am nächsten zurückliegenden Werte benutzt, wodurch sich der Simulationsfehler erhöht. Für die gegebene Aufgabenstellung kann der Fehler des Simulationsergebnisses als Summe der relativen Abweichungen der simulierten Zustände zum Zeitpunkt $t = 100$ vom Equilibrium berechnet werden. In Abbildung 8.7 ist dieser Fehler für die sequentielle Simulation und für die parallele Simulation in Abhängigkeit vom Kommunikationsintervall dargestellt.

Beispielimplementierungen der beschriebenen parallelen Lösung mit Kommunikationsintervallen auf Basis der DP-Toolbox und von PVM sind im Anhang D.2 enthalten.

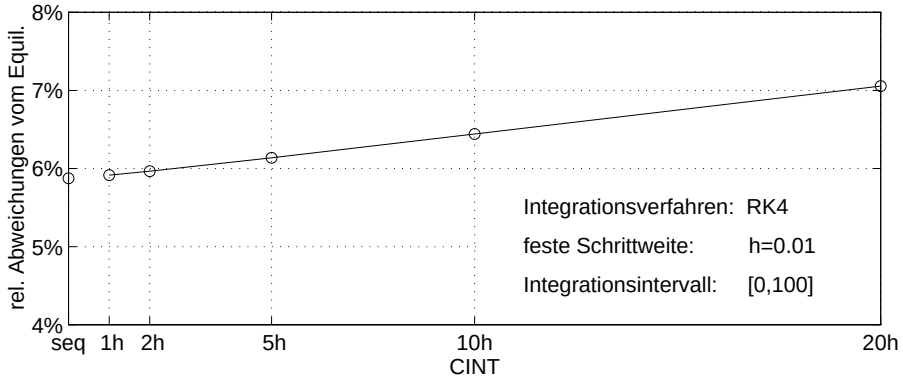


Abbildung 8.7: Simulationsfehler in Abhängigkeit vom Kommunikationsintervall

8.2.3 DP/PVM-Leistungsvergleich

Die Ergebnisse der durchgeführten Leistungsuntersuchungen (s. Abb. 8.8) bestätigen, daß die Granularität der beschriebenen Lösung durch Vergrößerung des Kommunikationsintervalls erheblich erhöht werden kann. Dennoch konnte auf der benutzten Testplattform bei einem maximalen Kommunikationsintervall vom 20-fachen der Integrationsschrittweite weder mit der DP-Toolbox noch mit PVM ein Parallelisierungsgewinn erzielt werden.

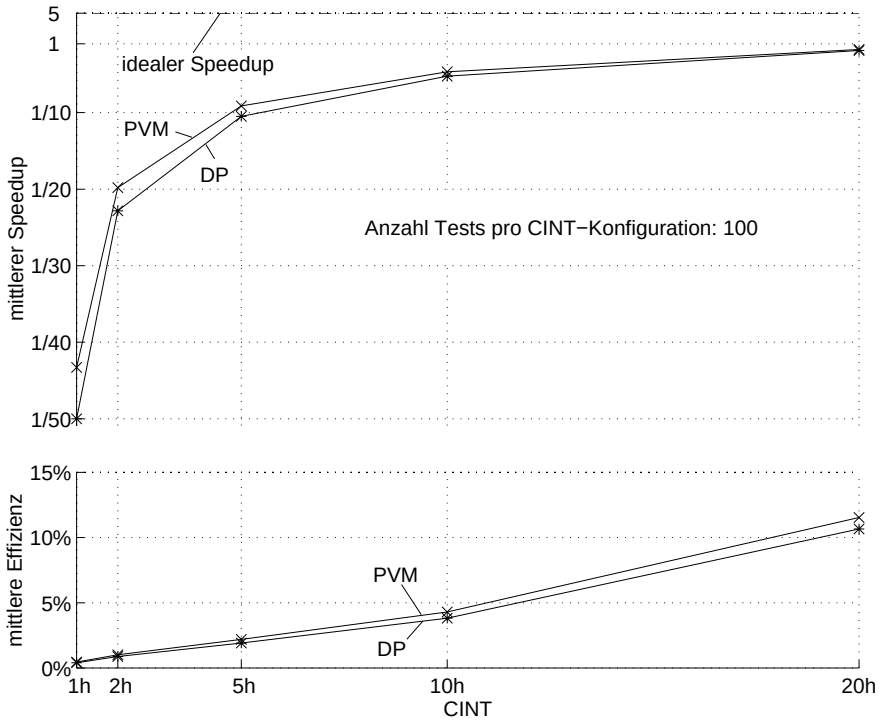


Abbildung 8.8: Speedup und Effizienz in Abhängigkeit vom Kommunikationsintervall

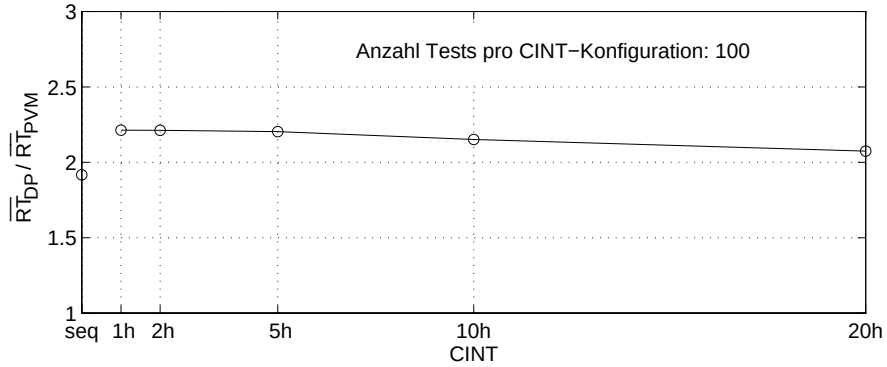


Abbildung 8.9: Laufzeitoverhead der MATLAB/DP-TB-Implementierung

Aus dem Vergleich der mittleren Laufzeiten $\overline{RT}_{DP}$ und $\overline{RT}_{PVM}$ ergibt sich ein Overheadfaktor von maximal $2,2$ für die parallele MATLAB-Lösung, der bei größer werdenden Kommunikationsintervallen langsam abnimmt und damit nur geringfügig über dem Overhead der sequentiellen MATLAB-Lösung liegt (s. Abb. 8.9).

8.3 Lösung einer partiellen Differentialgleichung

8.3.1 Aufgabenstellung

Die partielle Differentialgleichung

$$\begin{aligned} & u_{xx}(t,x) = a u_{tt}(x,t) \\ \text{mit:} \quad & u(0,t) = 0\,,\ u(L,t) = b e^{-dt} \sin \omega t\,, \\ & u(x,0) = u_x(x,0) = 0\,, \\ & L{=}10\,,\ a{=}2\,,\ b{=}1\,,\ d{=}0.2\,,\ \omega{=}1\,, \end{aligned} \tag{8.9}$$

beschreibt die Bewegung eines schwingenden Seils, das an einem Ende fixiert ist und am anderen angeregt wird (s. Abb. 8.10). Durch Ortsdiskretisierung in N äquidistante Intervalle (*method of lines*) und Ersetzung des Differentialgleichungsquotienten $u_{xx}(t,x)$ durch den zentralen Differenzenquotienten kann die partielle Differentialgleichung durch ein System von gewöhnlichen Differentialgleichungen zweiter Ordnung approximiert werden:

$$\begin{aligned} & k^2 a \ddot{u}_i(t) = u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)\,, \quad i = 1, \ldots, N-1 \\ \text{mit:} \quad & u_0(t) = u(0,t) = 0\,,\ u_N(t) = u(L,t) = b e^{-dt} \sin \omega t\,, \\ & u_i(0) = \dot{u}_i(0) = 0\,, \\ & k = L/N\,. \end{aligned} \tag{8.10}$$

Die Aufgabe besteht in der Lösung des Gleichungssystems bei einer Diskretisierung von $N{=}\,800$ im Zeitintervall $[0, 30]$ mit dem RK4-Verfahren (feste Schrittweite: $h{=}\,0.005$) und der Speicherung der Ergebnisse für u an den Orten $x{=}\,9L/10\,,\ 3L/4\,,$ $L/2\,,\ L/4$ und $L/10\,.$

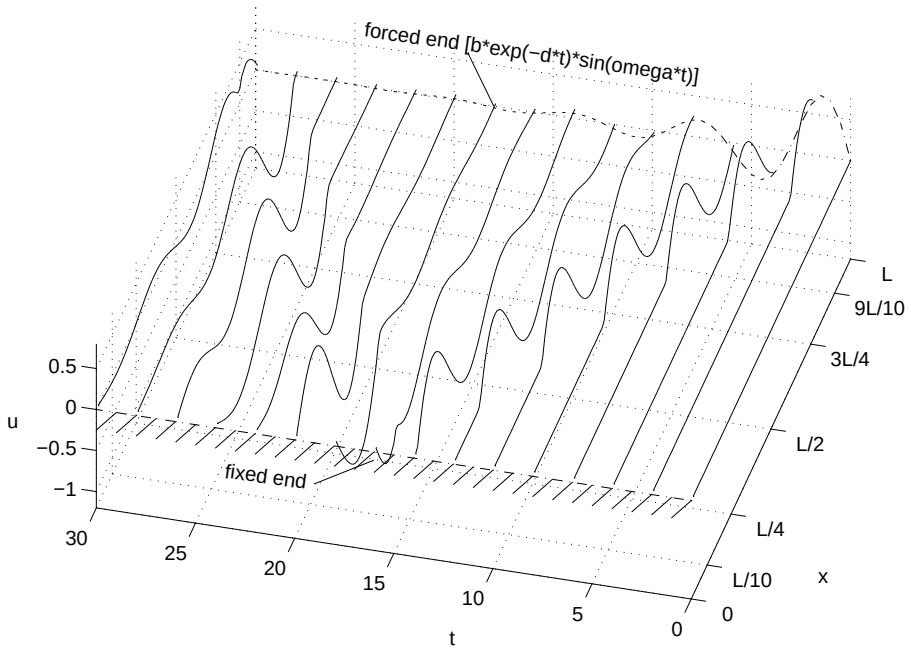


Abbildung 8.10: Bewegung eines schwingenden Seils

8.3.2 Parallele Lösung

Wie beim ersten Testproblem (Feder-Masse-System) müssen die Differentialgleichungen zweiter Ordnung (8.10) durch Substitution in Differentialgleichungen erster Ordnung überführt werden:

$$\begin{aligned}
&\dot{u}_{1_i}(t) = \tfrac{1}{k^2 a}(u_{2_{i-1}}(t) - 2u_{2_i}(t) + u_{2_{i+1}}(t)\,,\\
&\dot{u}_{2_i}(t) = u_{1_i}(t)\,, \quad i = 1,\ldots,N-1\\
&\text{mit: } u_{2_0}(t) = u_0(t) = 0\,,\ u_{2_N}(t) = u_N(t) = be^{-dt}\sin\omega t\,,\\
&\qquad u_{1_i}(0) = \dot{u}_i(0) = 0\,,\ u_{2_i}(0) = u_i(0) = 0\,.
\end{aligned} \tag{8.11}$$

Bei der vorgeschriebenen Diskretisierung entsteht so ein Gleichungssystem mit 1598 Zustandsvariablen, welches zur parallelen Berechnung möglichst gleichmäßig auf die zur Verfügung stehenden Prozessoren aufzuteilen ist.

Die Prinzipdarstellung einer parallelen Lösung nach dem Master-Slave-Paradigma zeigt Abbildung 8.11. Darin sind deutlich die Gemeinsamkeiten und Unterschiede zur Lösung des zweiten Testproblems (gekoppelte Räuber-Beute-Populationen, s. Abschn. 8.2.2) zu erkennen. So müssen auch hier die Slaves untereinander kommunizieren, der Austausch von Zustandswerten ist jedoch nur zwischen benachbarten Teilsystemen erforderlich. Darüber hinaus ist der Aufwand zur Berechnung der Teilsysteme deutlich größer als beim zweiten Testproblem. Sind Abstriche an der Genauigkeit der Berechnungsergebnisse zulässig, kann eine weitere Erhöhung der Granularität wie beim zweiten Testproblem durch Einführung von Kommunikationsintervallen erreicht werden.

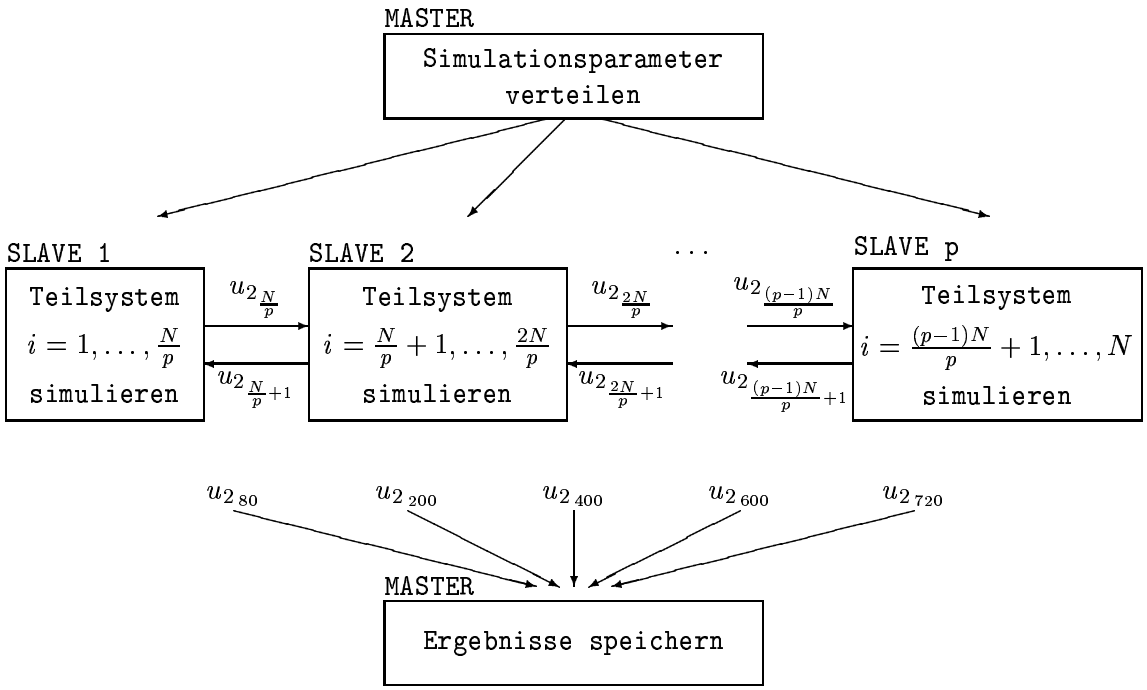


Abbildung 8.11: Prinzipdarstellung der parallelen Lösung

Eine Besonderheit bei der Lösung des Testproblems, die aus der Abbildung 8.11 nicht unmittelbar zu erkennen ist, besteht darin, daß nur jene Slaves Ergebnisse an den Master übermitteln müssen, die für die Berechnung der Seilbewegung an den in der Aufgabenstellung festgelegten Orten zuständig sind.

Beispielimplementierungen zur Lösung des Testproblems mit Hilfe der DP-Toolbox sowie mit PVM sind im Anhang D.3 enthalten.

8.3.3 DP/PVM-Leistungsvergleich

Im Rahmen der Leistungsuntersuchungen wurde das Testproblem auf unterschiedlichen Prozessorzahlen gelöst, wobei das Kommunikationsintervall jeweils der Integrationsschrittweite entsprach. Die in Abbildung 8.12 dargestellten Untersuchungsergebnisse zeigen, daß auf Systemen ohne gemeinsamen Speicher sowohl mit PVM als auch mit der DP-Toolbox ein Parallelisierungsgewinn erzielbar ist. Auf der verwendeten Testplattform liegt dieser aber weit unterhalb des idealen Speedups, was sich in sehr geringen Effizienzwerten widerspiegelt. Höhere Speedupwerte, die den Aufwand einer Parallelisierung tatsächlich rechtfertigen würden, erfordern entweder die Verwendung von Plattformen mit einer größeren Kommunikationsleistung (relativ zur Rechenleistung) oder eine Vergrößerung des Kommunikationsintervalls[5].

[5]Der Zusammenhang zwischen Speedup und Größe des Kommunikationsintervalls wurde für das Testproblem im Rahmen dieser Arbeit nicht untersucht. Der Vergrößerung des Kommunikationsintervalls sind aber prinzipiell Grenzen gesetzt (numerische Instabilität, s. [10]).

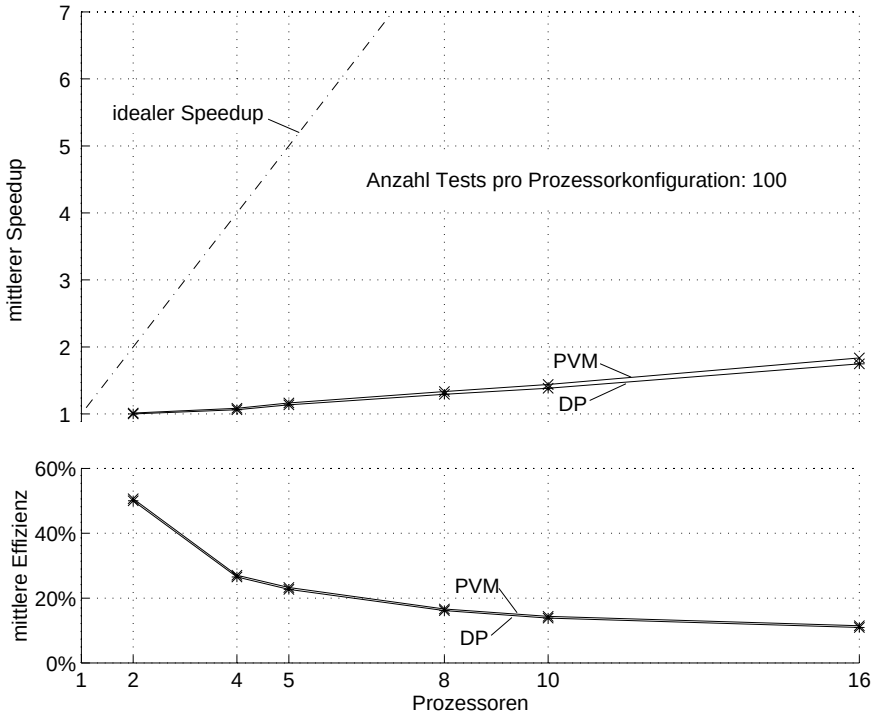


Abbildung 8.12: Speedup und Effizienz in Abhängigkeit von der Prozessorzahl

Beim direkten Vergleich der Laufzeiten der MATLAB- und der PVM-basierten Lösungen (s. Abb. 8.13) ergeben sich ähnliche Ergebnisse wie bei den beiden vorangegangenen Testproblemen, d.h. der Overhead der parallelen Lösungen auf Basis der DP-Toolbox liegt nur geringfügig über dem der sequentiellen MATLAB-Lösung.

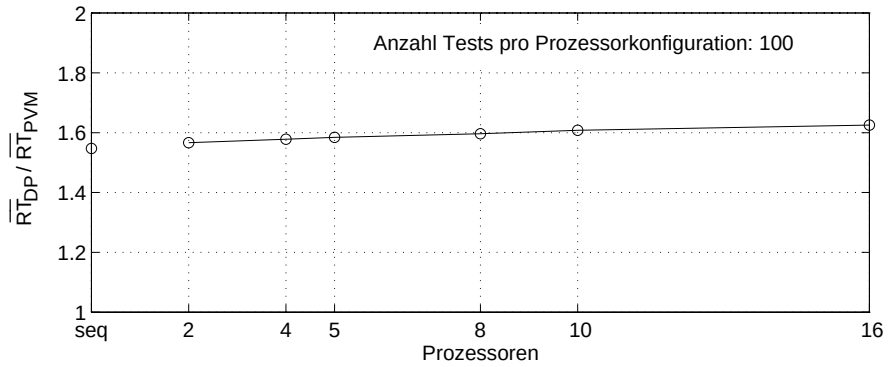


Abbildung 8.13: Laufzeitoverhead der MATLAB/DP-TB-Implementierung

Kapitel 9

Bezüge zu verwandten Arbeiten

Im Abschnitt 5.4 wurde bereits erwähnt, daß der Multi-SCE-Ansatz seit 1993 von einer Gruppe um Anne E. Trefethen (Cornell University, New York) und vom Autor dieser Arbeit unabhängig voneinander entwickelt und umgesetzt wurde. Darüber hinaus starteten bis 1996 mindestens zwei weitere Projekte, die auf einer ähnlichen Konzeption basieren. Zum erstmaligen Kontakt und Erfahrungsaustausch zwischen einzelnen Forschungsgruppen kam es am Rande der 2. Internationalen MATLAB-Konferenz in Cambridge, Massachusetts im Oktober 1995.

Nachfolgend sollen die wesentlichen Etappen der verschiedenen Projekte in Form einer Zeittafel zusammenfassend dargestellt werden. Zur Bezeichnung der Projekte werden die folgenden Kürzel benutzt:

Cornell:	Arbeiten am Cornell Theory Center und Department of Computer Science der Cornell University, New York
Rostock:	Arbeiten am Fachbereich Elektrotechnik, Universität Rostock und am Fachbereich Maschinenbau, FH Wismar (vor 1994 am Fachbereich Informatik, Universität Rostock)
Prag:	Arbeiten am Institut für Informationstheorie und Automatisierungstechnik an der Akademie der Wissenschaften der Tschechischen Republik;
Alpha Data:	seit 1995 Nachfolgeprojekt bei Alpha Data Parallel Systems, Ltd., Edinburgh
Wake Forest:	Arbeiten am Mathematics and Computer Science Department der Wake Forest University, North Carolina
Magdeburg:	Arbeiten am Institut für Automatisierungstechnik der Otto-von-Guericke Universität, Magdeburg
Daimler Benz:	Arbeiten bei der Daimler Benz AG, Abteilung Systemtechnik, Berlin

Zeittafel:

1992

Rostock: Entwicklung des plattformunabhängigen Message-Passing-Systems PSI auf Basis der Kommunikationsbibliothek LIBIPC ([55])

1993

Cornell: Experimente mit mehreren MATLAB-Instanzen auf einer IBM SP1 unter Verwendung des proprietären Message-Passing-Systems MPL (IBM)

Rostock: Einsatz des PSI-Systems zur Prototyp-Entwicklung einer verteilten und parallelen Simulationsumgebung ([61])

1994

Prag: MATLAB-Transputer-Kopplungen ([48, 46]); die Arbeiten beruhen noch ausschließlich auf dem Kopplungs-Ansatz (s. Abschn. 5.2)

Rostock: erste Version der DP-Toolbox auf Basis des PSI-Systems für MATLAB 4; Lösung der SNE-Testprobleme (s. Kap. 8) und Veröffentlichung der Ergebnisse ([64])

1995

Rostock: weitere Versionen der DP-Toolbox auf Basis des Message-Passing-Systems PVM; Veröffentlichungen zum Aufbau und zur Anwendung der Toolbox ([65, 66]); Freigabe der ersten Public-Domain-Distribution ([67])

Alpha Data: Kopplung von MATLAB (auf einem Standard-PC) mit speziellen Multiprozessorboards (Alpha-Prozessoren von DEC); auf den Alpha-Prozessoren laufen abgerüstete „MATLAB-ähnliche Interpreter“ (Kombination von Kopplungs- und Multi-SCE-Ansatz); als kommerzielle Lösung erhältlich ([1])

Oktober 1995, MATLAB-Konferenz, Cambridge, MA:
Alpha Data: Präsentation der „MATLAB-Alpha-Kopplung“ ([47])
Rostock: Präsentation der DP-Toolbox ([63])
Erfahrungsaustausch zwischen ***Cornell***, ***Alpha Data*** und ***Rostock***

Cornell: Start des MultiMATLAB-Projekts

1996

Rostock: weitere Versionen der DP-Toolbox für das Betriebssystem MS-Windows auf der Basis des Message-Passing-Systems WPVM ([43])

Wake Forest: erste Version der „Parallel Toolbox“ für MATLAB ([40]); entspricht weitgehend der unteren Schicht der DP-Toolbox (DPLOW-Toolbox); die Interaktivität ist allerdings auf eine einzige „Master-Instanz“ eingeschränkt, sämtliche „Slave-Instanzen“ laufen im Hintergrund; Freigabe der Toolbox als Public-Domain-Distribution

1997

Magdeburg: erste MATLAB-5-Portierung der DP-Toolbox ([6])

Daimler Benz: anwendungsspezifische Erweiterungen der DP-Toolbox ([70])

Cornell: erste Veröffentlichung zum MultiMATLAB-Projekt ([53]); Konzeption entspricht der DP-Toolbox, als Message-Passing-System wird eine Implementation des MPI-Standards benutzt

Kapitel 10

Zusammenfassung

Das Ziel dieser Arbeit bestand darin, einen Fortschritt hinsichtlich einer vereinfachten Nutzung der Parallelverarbeitung im Bereich des wissenschaftlich-technischen Rechnens zu erreichen. Nach den Erfahrungen des Autors, die im Verlauf mehrjähriger Forschungsarbeiten zur verteilten und parallelen Verarbeitung innerhalb der Automatisierungstechnik gesammelt wurden, findet ein solcher Fortschritt nur dann breite Akzeptanz, wenn es gleichzeitig gelingt, bisherige Investitionen (Aufwand für die Softwareerstellung und Lernaufwand beim Nutzer) zu schützen. Konsequenterweise wurde in dieser Arbeit nicht versucht, „von Grund auf neue Lösungen“ zu finden, vielmehr wurden zwei weit entwickelte Techniken — die Programmierung von Parallelverarbeitungssystemen mit Hilfe von Message-Passing-Systemen und die Erstellung wissenschaftlich-technischer Anwendungen in interaktiven Berechnungs- und Visualisierungsumgebungen — aufgegriffen und ein Ansatz aufgezeigt, wie diese nutzbringend miteinander kombiniert werden können.

Da in den potentiellen Anwendungsgebieten, wie z.B. der Automatisierungstechnik, im allgemeinen nur geringe Erfahrungen mit Parallelverarbeitungstechniken vorliegen, wurden die diesbezüglichen hardware- und softwaretechnischen Grundlagen im einführenden Teil der Arbeit ausführlich dargestellt. Dabei wurde herausgearbeitet, daß aus Gründen der Verfügbarkeit für einen größeren Anwenderkreis vorerst nur MIMD-Plattformen (Multiprozessorsysteme und Computercluster) von Bedeutung sind.

MIMD-Systeme sind asynchron arbeitende Parallelverarbeitungsplattformen, die explizit parallel programmiert werden müssen, was entweder nach dem speicher- oder dem nachrichten-orientierten Paradigma erfolgen kann. Da die effiziente Abbildung des speicher-orientierten Paradigmas auf MIMD-Systeme ohne gemeinsamen physischen Speicher und mit nur geringer Kommunikationsleistung — wie es bei den weit verbreiteten Computerclustern der Fall ist — Schwierigkeiten bereitet, wurde in der Arbeit die nachrichten-orientierte Programmierung als primäres Paradigma favorisiert.

Für die parallele Programmierung nach dem nachrichten-orientierten Paradigma steht eine Vielzahl von Message-Passing-Systemen zur Verfügung. Um den aktuellen Stand der Technik auf diesem Gebiet zu identifizieren, wurden der offizielle Message-Passing-Standard MPI und die verbreiteten Standardsysteme P4, PVM

und TCGMSG analysiert und miteinander verglichen. Dabei stellte sich heraus, daß es hinsichtlich der Komplexität erhebliche Unterschiede zwischen den Systemen gibt. Dennoch sind die Systeme prinzipiell gegeneinander austauschbar, da sich ihre grundsätzliche Funktionalität ähnelt. So werden von allen Systemen neben Funktionen für die Punkt-zu-Punkt-Kommunikation auch Funktionen zur Verwaltung von Prozessen und zur Ausführung kollektiver Operationen bereitgestellt.

Auf eine vergleichende Analyse verschiedener Berechnungs- und Visualisierungsumgebungen wurde dagegen verzichtet, weil sie sich einerseits konzeptionell kaum voneinander unterscheiden und zum anderen davon ausgegangen werden kann, daß in den Anwendungsbereichen umfangreiche Erfahrungen im Umgang mit den wichtigsten Systemen vorhanden sind. Die Ausführungen beschränken sich deshalb auf eine zusammenfassende Darstellung der wesentlichen Aspekte des Aufbaus und der Funktionsweise solcher Systeme, für die aus Gründen der sprachlichen Vereinfachung als Oberbegriff die Abkürzung SCEs (*scientific and technical computing environments*) eingeführt wurde.

Bestrebungen, die Vorteile der Parallelverarbeitung und der SCE-basierten Anwendungsentwicklung miteinander zu verbinden, gibt es, seitdem Parallelverarbeitungssysteme für einen größeren Anwenderkreis zur Verfügung stehen. Um die bisher eingeschlagenen Wege zu systematisieren, wurden die bekannt gewordenen Arbeiten analysiert. Danach muß zwischen drei grundlegenden Ansätzen unterschieden werden: der Übersetzung von sequentiellen SCE-Prototypen in ausführbare Programme für parallele Plattformen, der Kopplung von konventionellen (sequentiellen) SCEs mit Parallelverarbeitungssystemen und der SCE-internen Parallelverarbeitung (parallele SCEs). Der Übersetzungs- und der Kopplungsansatz basieren auf konventionellen SCEs und sind deshalb mit geringem Aufwand umsetzbar. Eine tatsächliche Integration von SCE-basierter Anwendungsentwicklung und Parallelverarbeitung kann mit diesen Ansätzen aber nicht erreicht werden. In parallelen SCEs erfolgt die Parallelverarbeitung dagegen völlig transparent. Bei näherer Betrachtung weist aber auch dieser Ansatz gewichtige Unzulänglichkeiten auf. Zum einen ist die Umsetzung des Ansatzes mit einem erheblichen Aufwand verbunden, da er nicht unmittelbar auf der Basis von konventionellen SCEs realisiert werden kann. Darüber hinaus ist die Granularität der meisten SCE-internen Funktionen für eine effektive Parallelisierung auf MIMD-Plattformen ohne gemeinsamen Speicher zu gering.

Diese Situation bildete die Motivation für die Entwicklung des Multi-SCE-Ansatzes, der im Unterschied zu den bereits existierenden Ansätzen das primäre SCE-Konzept, d.h. die Bereitstellung einer für das schnelle Prototyping geeigneten Entwicklungsumgebung tatsächlich auf die Parallelverarbeitung überträgt. So gestatten die vorhandenen Ansätze zwar die Nutzung der Parallelverarbeitung in einem mehr oder weniger engen Zusammenhang mit SCEs, das schnelle Prototyping innerhalb einer SCE bleibt aber auf die sequentielle Programmierung beschränkt. Der Multi-SCE-Ansatz eröffnet dagegen die Möglichkeit, diese Technik uneingeschränkt auch für die explizite parallele Programmierung zu nutzen.

Die Grundidee des Multi-SCE-Ansatzes besteht darin, mehrere konventionelle SCE-Instanzen, die auf verschiedenen Prozessoren bzw. Rechnern laufen, miteinander zu koppeln. Auf diese Weise entsteht eine asynchrone Parallelverarbeitungsplatt-

form, womit die aus der MIMD-Programmierung bekannten Konzepte unmittelbar auf der Ebene von SCEs eingesetzt werden können, d.h. eine Multi-SCE kann wie ein MIMD-System nach dem speicher- oder dem nachrichten-orientierten Paradigma programmiert werden. Der entscheidende Unterschied zur klassischen Programmierung mit kompilierbaren Sprachen ist jedoch, daß die explizite parallele Programmierung innerhalb einer Multi-SCE interaktiv erfolgen kann.

Aus den eingangs erwähnten Gründen beschränkt sich die Arbeit auf die Darstellung eines Konzepts für Multi-SCEs ohne gemeinsamen Speicher. Eine beispielhafte Realisierung erfolgte auf Basis der SCE MATLAB und des Message-Passing-Systems PVM in Form der DP-Toolbox. Bei der Auswahl beider Systeme spielten technische Aspekte nur eine untergeordnete Rolle, da der Multi-SCE-Ansatz prinzipiell mit beliebigen SCEs und Message-Passing-Systemen umgesetzt werden kann.

Die im Rahmen der Arbeit durchgeführten Leistungsuntersuchungen an drei Testapplikationen ergaben, daß der Multi-SCE-Ansatz bei den derzeit verfügbaren MIMD-Architekturen vor allem für grobgranuläre Problemstellungen geeignet ist.

Bereits zum Zeitpunkt der Fertigstellung der vorliegenden Arbeit sind Realisierungen des Multi-SCE-Ansatzes auf Basis verschiedener SCEs und Message-Passing-Systeme verfügbar und werden intensiv zur Lösung realer Problemstellungen eingesetzt.

Literaturverzeichnis

[1] Alpha Data. *MATLAB Bridge to Parallel Alpha, User's Guide.* Alpha Data Parallel Systems Ltd., 86 Causewayside, Edinburgh EH9 1PY, Scotland.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Montvale, NJ, April 1967. AFIPS Press.

[3] W. Aspray and A. Burks, editors. *Papers of John von Neumann on Computing and Computer Theory.* MIT Press, Cambridge, MA, 1987.

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[5] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley, Wokingham, 3rd edition, 1989.

[6] T. T. Binh and P. Zimmermann. Parallel-MATLAB: Eine Implementierung zur Entwicklung paralleler Matlab-Anwendungen. Technischer Bericht, Institut für Automatisierungstechnik, Otto-von-Guericke Universität, Magdeburg, September 1997.

[7] T. Bräunl. *Parallele Programmierung: Eine Einführung.* Vieweg, Braunschweig/Wiesbaden, 1993. ISBN 3-528-05142-6.

[8] F. Breitenecker, H. Ecker, and I. Bausch-Gall. *Simulieren mit ACSL.* Fortschritte in der Simulationstechnik. Vieweg, Braunschweig/Wiesbaden, 1993.

[9] F. Breitenecker et al. Comparison of parallel simulation techniques – test problems. *EUROSIM - Simulation News Europe (SNE)*, (10):21–22, March 1994.

[10] F. Breitenecker, I. Husinsky, and G. Schuster. Comparison of parallel simulation techniques – sample solutions: Workstation Cluster under PVM. *EUROSIM - Simulation News Europe (SNE)*, (10):24–25, March 1994.

[11] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[12] C. Brown. *UNIX Distributed Programming.* Prentice Hall, Hemel Hempstead, UK, 1994. ISBN 0-13-075896-5.

[13] A. Burns. *Programming in occam 2.* Addison-Wesley, Wokingham, 1988.

[14] R. Butler and E. Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801, October 1992. Revised, April 1994.

[15] T. Canfield et al. Thermal effects on the frequency response of piezoelectric crystals. In *New Methods in Transient Analysis*, PVP-Vol. 246 and AMD-Vol. 143, pages 103–108, New York, 1992. ASME.

[16] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* The MIT Press, Cambridge, MA, 1990. ISBN 0-262-03171-X.

[17] D. Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, 1993.

[18] J. R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Sun Technical Reference Library. Springer-Verlag, New York, 1990. ISBN 0-387-97247-1.

[19] L. De Rose et al. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer-Verlag, August 1995.

[20] E. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technical University Eindhoven, 1965.

[21] N. Doss et al. A model implementation of MPI. Technical report, Argonne National Laboratory, 1993.

[22] P. Drakenberg, P. Jacobson, and B. Kågström. A CONLAB compiler for a distributed memory multicomputer. In R. F. Sincovec et al., editors, *Proc. Sixth SIAM Conf. Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.

[23] P. Dünow, S. Pawletta, and W. Drewelow. A parallel and distributed version of the method of convex decomposition – a stability test for linear uncertain systems. In Z. Vukic, editor, *Proceedings of the 41st Conference KoREMA Conference*, volume 1, Opatia, Croatia, September 1996.

[24] P. Dünow, S. Pawletta, N. Stoll, and M. Teller. A distributed and parallel approach for supervisory of a complex analytical measuring system. Technical report, Institut of Automatic Control, University of Rostock, 1996.

[25] J. W. Eaton. GNU Octave: A high-level interactive language for numerical computations. http://www.che.wisc.edu/octave/doc/, February 1997.

[26] M. J. Flynn. Very high-speed computing systems. *Proc. IEEE*, 54:1901–1909, December 1966.

[27] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[28] T. J. Fountain. *Parallel computing: principles and practice.* Cambridge University Press, Cambridge, UK, 1994. ISBN 0-521-45131-0.

[29] T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1992. ISBN 0-13-651597-5.

[30] N. Galbreath et al. Parallel solution of the three-dimensional, time-dependent Ginzburg-Landau equation. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[31] A. Geist et al. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, TN 37831, May 1994.

[32] A. Geist et al. *PVM: A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

[33] R. Grothmann. The EULER Software. http://mathsrv.ku-eichstaett.de/MFG/homes/grothmann/euler/.

[34] MuPAD Group. *MuPAD user's manual.* Wiley-Teubner, 1996. ISBN 3-519-02114-5.

[35] Harmonic Software, Inc. O-Matrix overview. http://www.omatrix.com/, April 1997.

[36] R. J. Harrison. Portable Tools and Applications for Parallel Computers. *International Journal of Quantum Chemistry*, 40:847–863, 1991.

[37] A. Heck. *Introduction to Maple.* Springer, New York, 2nd edition, 1996. ISBN 0-387-94535-0.

[38] C. A. E. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[39] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger, 1981.

[40] J. Hollingsworth, K. Liu, and P. Pauca. *Parallel Toolbox for MATLAB, Manual and Reference Pages.* Wake Forest University, North Carolina, September 1996.

[41] Integrated Sensors Inc. Real-time development environment for embedded HPC systems. http://www.sensors.com/isi/Conf97, March 1997.

[42] Inmos Limited. *occam Programming Manual.* Prentice Hall International, Englewood Cliffs, NJ, 1984.

[43] T. Jeinsch. *Parallele und verteilte MATLAB-Programmierung unter Unix und MS-Windows.* Diplomarbeit, Institut für Automatisierungstechnik, Fachbereich Elektrotechnik, Universität Rostock, September 1996.

[44] P. Junglas. Paralleles MATLAB. Informationen des Rechenzentrums der TU Hamburg-Harburg, RZTU-INFO 3/96.

[45] D. Jungmann and H. Stange. *Einführung in die Rechnerarchitektur.* Studienbücher der Informatik. Hanser Verlag, 1992.

[46] J. Kadlec. Direct software bridge Matlab - transputer boards. In *Preprints EUSIPCO Conference, Edinburgh*, September 1994.

[47] J. Kadlec and N. Nakhaee. Alpha Bridge, parallel processing under MATLAB. In *2nd International MATLAB Conference*, Cambridge, MA, October 1995.

[48] J. Kadlec and P. Nedoma. *Matlab - Transputer Bridge 1.0, User Guide.* Inst. for Information Theory and Automation, The Academy of Science of The Czech Republic, Prague, 1994.

[49] T. Krüger. *Regelung und Echtzeitsimulation unter OS-9.* Diplomarbeit, Institut für Automatisierungstechnik, Fachbereich Elektrotechnik, Universität Rostock, 1994.

[50] MasPar Computer Corporation. *MasPar VAST-2 User's Guide*, February 1992. DPN 9300-9035.

[51] MathTools Ltd. MATCOM: MATLAB to C++ compiler. http://www.mathtools.com/matcom.html, 1996.

[52] T. G. Mattson. Programming environments for parallel and distributed computing: A comparison of p4, PVM, Linda and TCGMSG. Technical report, Intel Corporation, Supercomputer Systems Division, Beaverton, OR, 1995.

[53] V. Menon and A. E. Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. In *Proc. of the ACM/IEEE Conference on Supercomputing*, San Jose, CA, November 1997.

[54] M. Metclaf and J. Reid. *Fortran 90 Explained.* Oxford University Press, Oxford, New York, Tokyo, 1990.

[55] H. Meyer. LIBIPC: A C++ class library for process interaction and communication. Technical Report DB-92-2, Dep. of Computer Science, Database Research Group, University of Rostock, 1992.

[56] D. I. Moldovan. *Parallel Processing: From Application to Systems.* Morgan Kaufmann Publishers, San Mateo, CA, 1993. ISBN 1-55860-254-2.

[57] C. Moler. Is There a "Public Domain" MATLAB? ftp://csi.jpl.nasa.gov/pub/matlab/classic/README.cleve.

[58] C. Moler. Why there isn't a parallel MATLAB. *Matlab News and Notes*, page 12, Spring 1995.

[59] R. Monge. *Kommunikation in verteilten objektbasierten Systemen.* Dissertation, Inst. für Mathematische Maschinen und Datenverarbeitung (Informatik), Universität Erlangen/Nürnberg, September 1992. Arbeitsberichte des Instituts, Band 25, Nummer 7.

[60] G. Olsen et al. Inference of phylogenetic trees using maximum likelihood. In *Proceedings of the First Intel Delta Applications Workshop*, pages 247–262, 1992.

[61] S. Pawletta. Entwurf und Implementationskonzept einer verteilten und parallelen Simulationsumgebung für komplexe Experimente. Forschungsbericht CS-03-94, Fachbereich Informatik, AG Modellierung und Simulation, Universität Rostock, 1994.

[62] S. Pawletta. Einsatz verteilter und paralleler Konzepte zur Lösung automatisierungstechnischer Problemstellungen. Beitrag zum 29. Regelungstechnischen Kolloquium, 1. bis 3. März 1995 in Boppard, Universität Rostock, 1995.

[63] S. Pawletta, W. Drewelow, P. Dünow, T. Pawletta, and M. Süße. A MATLAB toolbox for distributed and parallel processing. In *2nd International MATLAB Conference*, Cambridge, MA, October 1995.

[64] S. Pawletta, T. Pawletta, and W. Drewelow. Comparison of parallel simulation techniques – solutions: Workstation Cluster / MATLAB / PSI. *EUROSIM - Simulation News Europe (SNE)*, (13):38–39, March 1995.

[65] S. Pawletta, T. Pawletta, and W. Drewelow. Distributed and parallel computing in automatic control. In Z. Vukic, editor, *Proceedings of the 40th Anniversary Conference KoREMA*, volume 1, pages 423–426, Zagreb, Croatia, April 1995.

[66] S. Pawletta, T. Pawletta, and W. Drewelow. Distributed and parallel simulation in an interactive environment. In F. Breitenecker and I. Husinsky, editors, *Proceedings of the 1995 EUROSIM Conference, Vienna, Austria*, pages 345–350. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, September 1995.

[67] S. Pawletta, T. Pawletta, W. Drewelow, and P. Dünow. *Distributed and Parallel Application Toolbox for Use with MATLAB: User's Guide and Reference Manual Version 1.* Inst. of Automatic Control, Univ. of Rostock, September 1995.

[68] S. Pawletta, T. Pawletta, W. Drewelow, and P. Dünow. *Distributed and Parallel Application Toolbox for Use with MATLAB: User's Guide and Reference Manual Version 1.3.* Inst. of Automatic Control, Univ. of Rostock, December 1996.

[69] T. Pawletta, S. Pawletta, and W. Drewelow. Transaktionsorientierte Simulation in interaktiven SCEs. In P. Lorenz and B. Preim, editors, *Simulation and Visualization*, pages 181–194, Gent, Belgium, 1998. SCS Int. Publishing House.

[70] H. Pohlheim. Verteilung und Parallelisierung von Berechnungen unter Matlab. Interner Technischer Bericht, Daimler Benz AG, Abteilung Systemtechnik, Berlin, Februar 1998.

[71] M. J. Quinn. The 'Otter' MATLAB Compiler. Workshop on Object-Oriented Approaches to Parallel Programming, University of Southampton, March 1996.

[72] A. Schill. *DCE - Das OSF Distributed Computing Environment: Einführung und Grundlagen.* Springer-Verlag, Berlin, Heidelberg, 1993. ISBN 3-540-55335-5.

[73] W. Schreiner. *Parallel Functional Programming for Computer Algebra.* Ph.d. thesis, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, September 1994.

[74] Scilab Group. Introduction to Scilab. http://www-rocq.inria.fr/scilab/doc/, June 1997.

[75] J. E. Shore. Second thoughts on parallel processing. *Computational Electrical Engineering*, 1:95–109, 1973.

[76] M. J. Shute. Categorizing parallel computer architectures. In T. J. Fountain and M. J. Shute, editors, *Multiprocessor Computer Architectures*, pages 1–38. North Holland, 1990.

[77] M. Snir et al. *MPI: The Complete Reference.* The MIT Press, Cambridge, MA, 1996.

[78] I. Sommerville and R. Morrison. *Software Development with Ada.* Addison-Wesley, Reading, MA, 1987.

[79] W. R. Stevens. *UNIX Network Programming.* Software Series. Prentice Hall, Englewood Cliffs, NJ, 1990. ISBN 0-13-949876-1.

[80] M. Süße. *Parallelverarbeitung in der interaktiven Entwicklungsumgebung MATLAB.* Diplomarbeit, Institut für Automatisierungstechnik, Fachbereich Elektrotechnik, Universität Rostock, 1996.

[81] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1995. ISBN 0-13-219908-4.

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

[83] The MathWorks, Inc. *MATLAB: External Interface Guide.* 24 Prime Park Way, Natick, MA 01760, January 1993.

[84] The MathWorks, Inc. *MATLAB Compiler User's Guide*. 24 Prime Park Way, Natick, MA 01760, 1995.

[85] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-driven and demand-driven computer architectures. *Computer Surveys*, 14(1):93–143, 1982.

[86] L. H. Trucotte. A survey of software environments for exploting networked computing resources. Technical Report MSU-EIRS-ERC-93-2, Mississipi State University, Engineering Research Center for Computational Field Simulation, Mississipi State, MS 39762, February 1993.

[87] A. Watermann. *Parallelisierung des Finite-Element-Codes SMART-Konvektion*. Dissertation, Institut für Sicherheitsforschung und Reaktortechnik, Universität Bochum, September 1995. ISSN 0944-2952.

[88] R. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.

[89] S. Wolfram. *The Mathematica book: Mathematica version 3*. Wolfram Media, Champaign, Ill., 3rd edition, 1996. ISBN 0-9650532-0-2.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

Anhang A

Vergleich der Message-Passing-Systeme P4, TCGMSG, PVM und des Message-Passing-Standards MPI-1

A.1 Prozeßverwaltung

	P4	TCGMSG	PVM	MPI-1
Prozeß-instantiierung	statisch	statisch	statisch[1], dynamisch, unabhängig	statisch[2]
Programm-aufruf	direkter Aufruf aus einer Standard-Shell	Aufruf aus Standard-Shell über spezielles Startprogramm	direkter Aufruf aus einer Standard- oder Spezial-Shell	nicht definiert
Prozeß-mapping	explizit	explizit	explizit, transparent	nicht definiert
Plattform-konfiguration	statisch	statisch	dynamisch	nicht definiert
Prozeß-identifikation	geordnete Numerierung: 0 ... N-1‡	geordnete Numerierung: 0 ... N-1‡	Numerierung ohne Ordnungs-relation	geordnete Numerierung: 0 ... N-1‡

[1]Nur für SPMD-Programme, die von einer speziellen Shell aus (PVM-Shell) gestartet werden.
[2]MPI-1 definiert keine Prozeßverwaltung, basiert aber auf einem statischen Prozeßkonzept.
‡N = Anzahl Prozesse

A.2 Nachrichtenbasierte Kommunikation

	P4	TCGMSG	PVM	MPI-1
Ausführung von Kommunikations-operationen	nur sequentiell	sequentiell oder nebenläufig	nur sequentiell	sequentiell oder nebenläufig
Synchronisations-verhalten der Sendeoperationen	synchron oder asynchron; wählbar	synchron oder asynchron; plattform-abhängig	asynchron	vier Betriebs-arten wählbar; einschließlich synchron und asynchron
Nachrichten-selektion bei Empfangs- und Probeoperationen	nach Absender und / oder Nachrichten-markierung; Wildcards möglich	nach Absender und / oder Nachrichten-markierung; Wildcards nur für Absender	nach Absender und / oder Nachrichten-markierung; Wildcards möglich	nach Absender, Nachrichtenmar-kierung (Wild-cards möglich) und / oder Kommunika-tionsdomäne (Wildcards nicht möglich)
Überführung von Daten- in Nach-richtenobjekte	nur Objekte mit zusam-menhängender Bytefolge	nur Objekte mit zusam-menhängender Bytefolge	beliebige Objekte	beliebige Objekte
Überführung von Nachrichten- in Datenobjekte	Typinformation muß a priori bekannt sein	Typinformation muß a priori bekannt sein	Typinformation muß a priori bekannt sein	Typinformation muß a priori bekannt sein
Konvertierung der Daten-repräsentation	auf Anforde-rung indirekt über XDR	auf Anforde-rung indirekt über XDR	auf Anforde-rung indirekt über XDR	wenn erforder-lich automatisch; Verfahren nicht definiert
Anzahl der Inter-facefunktionen	12	4	39	52

A.3 Kollektive Operationen und Gruppenkonzepte

	P4	TCGMSG	PVM	MPI-1
Transport-operationen	Broadcast	Broadcast	Broadcast, Scatter, Gather	Broadcast, Scatter, Gather, Allgather, Alltoall
Reduktions-operationen	Allreduce; arithm./log. Operationen: 6 vordefiniert, nutzerdefinierte möglich	Allreduce; arithm./log. Operationen: 6 vordefiniert, nutzerdefinierte möglich	Reduce; arithm./log. Operationen: 4 vordefiniert, nutzerdefinierte möglich	Reduce, Allreduce, Reduce-Scatter, Scan; arithm./log. Operationen: 12 vordefiniert, nutzerdefinierte möglich
Synchronisations-operationen	Barrier	Barrier	Barrier	Barrier
Gruppenkonzept	wird nicht unterstützt	wird nicht unterstützt	dynamische Prozeßgruppen	statische Prozeßgruppen
Anzahl der Inter-facefunktionen	22	16	14	41

Anhang B

Struktur einer MATLAB-Matrix

Der einzige Objekttyp in MATLAB Vers. 4 ist die *Matrix*. Damit sind Skalare, Vektoren, zweidimensionale Matrizen und Zeichenketten (Strings) darstellbar. Die Spezifikation einer MATLAB-Matrix erfolgt über die nachfolgend aufgeführten Properties, die über Zugriffsfunktionen ermittelt und manipuliert werden können.

Property	Typ	Bedeutung
Precision	double	interne Darstellungsgenauigkeit
Storage	full	Matrix ist vollbesetzt
	sparse	Matrix ist schwachbesetzt
ComplexFlag	real	Matrix besitzt nur Realteil
	complex	Matrix mit Real- und Imaginärteil
DisplayMode	numeric	Matrixelemente repräsentieren numerische Werte
	text	Matrixelemente repräsentieren druckbare Zeichen
M & N	m & n	Matrix mit m Zeilen und n Spalten
	1 & n	n-dimensionaler Zeilenvektor
	m & 1	m-dimensionaler Spaltenvektor
	1 & 1	Skalar
	0 & 0	Leere Matrix
pr & pi	real	Arrays mit den Elementen des Real- und Imaginärteils der Matrix; neben reellen Zahlen sind als Elementwerte auch $NaN, +\infty$ und $-\infty$ erlaubt
	NaN	
	+Inf	
	−Inf	
Name	named	Matrixname bestehend aus maximal 19 Zeichen
	unnamed	temporäre Matrix ohne Name

Nachfolgende Properties gelten nur für schwachbesetzte Matrizen:		
Nzmax	nzmax	maximale Anzahl der besetzten Elemente
ir & ic	integer	Arrays mit den Indizes der besetzten Elemente

Anhang C

DP(HIGH)-Funktionen

C.1 Glossar

In das Glossar wurden nur die Funktionen der DP-Toolbox aufgenommen, die im Kapitel 7 besprochen und zur Implementation der Beispielanwendungen (s. Anh. D) benutzt wurden. Für eine vollständige Funktionsübersicht sei auf [68] verwiesen.

Eine thematische Gliederung der Funktionen zeigt die nachfolgende Tabelle. Innerhalb des Glossars sind die Funktionen in alphabetischer Reihenfolge aufgeführt.

DPMM-Konfiguration		Seite
`dpon`	Starten einer DPMM.	124
`dpadd`	Hinzufügen von DP-Instanzen zu einer DPMM.	118
`dpchg`	Änderung instanzbezogener Konfigurationsparameter.	118
`dpdel`	Entfernen von DP-Instanzen aus einer DPMM.	119
`dpoff`	Terminierung einer DPMM.	124
`dpmm`	Starten, Konfigurieren und Terminieren einer DPMM (alternatives Ein-Kommando-Interface).	122
`dpmmopt`	Lesen und Schreiben von Optionen der DPMM-Verwaltung.	123
`dpmmid`	Bestimmung des DPMM-Identifikators.	122
`dpmyid`	Bestimmung des DP-Identifikators.	123
Ausführung von Routinen auf einer DPMM		
`dpeval`	Vektorielle RPCs ohne Aufrufparameter.	119
`dpfeval`	Vektorielle RPCs mit Aufrufparameter.	121
`dpevalids`	Ermittlung der an einem vektoriellen RPC teilnehmenden DP-Instanzen.	120
`dpevalparent`	Ermittlung der DP-Instanz, die einen vektoriellen RPC ausgelöst hat.	120
`dpterm`	Umschaltung des Kommandointerfaces auf andere DP-Instanzen.	127
Kommunikation zwischen DP-Instanzen		
`dpsend`	Senden einer MATLAB-Matrix.	126
`dprecv`	Empfangen einer MATLAB-Matrix.	125
`dpscatter`	Verteilen der Elemente einer MATLAB-Matrix.	126
`dpgather`	Einsammeln der Elemente einer MATLAB-Matrix.	121

dpadd

Aufgabe

Hinzufügen von DP-Instanzen zu einer DPMM.

Synopsis

```
dpids = dpadd(n)
dpids = dpadd(n,run_mode,trace_mode)
dpids = dpadd(where)
dpids = dpadd(where,run_mode,trace_mode)
```

Beschreibung

`dpadd(n)` erweitert eine existierende DPMM um `n` DP-Instanzen und liefert deren Identifikatoren als Ergebnis zurück. Die Verteilung der DP-Instanzen auf die zur Verfügung stehenden Hosts erfolgt automatisch durch das System (transparentes Mapping).

Mit `dpadd(where)`, wobei `where` eine Multi-String-Matrix[1] mit Hostnamen ist, kann ein explizites Mapping der DP-Instanzen erreicht werden.

Mit Hilfe der optionalen Aufrufparameter `run_mode` und `trace_mode` kann gesteuert werden, ob die DP-Instanzen interaktiv in einem Xterm (`run_mode='visible'`) oder als Hintergrundprozesse (`run_mode='unvisible'`) laufen und ob Trace-Daten generiert werden sollen (`trace_mode='trace'`: Trace-Daten werden im File `/tmp/dp.{`*dpmmid.dpid*`}.log` gesammelt; `trace_mode='notrace'`: Trace-Daten werden nicht erzeugt). Alternativ können für `run_mode` und `trace_mode` auch numerische Werte verwendet werden (`'visible'`$\equiv$`1`, `'unvisible'`$\equiv$`0`, `'trace'`$\equiv$`1`, `'notrace'`$\equiv$`0`). Sollen die Run- und Trace-Optionen für jede DP-Instanz separat spezifiziert werden, kann dies mit Hilfe von numerischen Vektoren oder Multi-String-Matrizen erfolgen.

dpchg

Aufgabe

Änderung instanzbezogener Konfigurationsparameter.

Synopsis

```
dpids = dpchg(dpids,where)
dpids = dpchg(dpids,run_mode)
dpids = dpchg(dpids,trace_mode)
dpids = dpchg(dpids,where,run_mode,trace_mode)
```

[1] jede Zeile der Matrix repräsentiert einen String

Beschreibung

Mit Hilfe der Funktion `dpchg` können der Ausführungsort sowie die Run- und Trace-Option ein oder mehrerer DP-Instanzen geändert werden. Da sich bei Änderung des Ausführungsortes oder der Run-Option auch der Identifikator einer DP-Instanz ändert, werden die nach der Operation gültigen Identifikatoren als Ergebnis zurückgeliefert.

Für die Aufrufparameter `where`, `run_mode` und `trace_mode` gelten die selben Regeln wie bei der Funktion `dpadd`.

dpdel

Aufgabe

Entfernen von DP-Instanzen aus einer DPMM.

Synopsis

```
dpdel(dpids)
```

Beschreibung

Durch `dpdel(dpids)` werden alle im Parametervektor `dpids` spezifizierten DP-Instanzen aus einer DPMM entfernt.

dpeval

Aufgabe

Vektorielle RPCs ohne Aufrufparameter.

Synopsis

```
dpeval(s)
dpeval(dpids,s)
dpeval(s,n)
[x,y,z,...] = dpeval(...)
```

Beschreibung

Die Funktion `dpeval` dient zur Ausführung vektorieller RPCs ohne Aufrufparameter und ist das parallele Pendant zur MATLAB-Funktion `eval`.

Mit dem optionalen Parameter `dpids` können DP-Instanzen spezifiziert werden, auf denen der vektorielle RPC auszuführen ist (einschließlich der DP-Instanz, die `dpeval` ausführt). Ohne den Parameters `dpids` werden die DP-Instanzen von `dpeval` selbständig ausgewählt.

Der Parameter `s` ist eine Multi-String-Matrix, wobei jede Zeile einen oder mehrere MATLAB-Ausdrücke repräsentiert. Enthält `s` nur eine Zeile, so wird diese nach dem SPMD-Modell auf allen spezifizierten bzw.

automatisch ausgewählten DP-Instanzen ausgeführt. Für eine parallele Ausführung nach dem MPMD-Modell muß `s` für jede DP-Instanz eine Zeile enthalten.

Mit dem Parameter `n` kann festgelegt werden, auf wievielen DP-Instanzen `dpeval` Ausdrücke ausführen soll. Die Angabe des Parameters `n` ist nur erlaubt, wenn der Parameter `dpids` nicht angegeben wurde und `s` nur eine Zeile und damit einen nach dem SPMD-Modell auszuführenden Ausdruck enthält. In allen anderen Fällen ist die Anzahl der DP-Instanzen bereits durch die Anzahl der Elemente in `dpids` bzw. die Anzahl der Zeilen in `s` eindeutig bestimmt.

Sollen Ergebnisparameter zurückgegeben werden, müssen alle parallel ausgeführten Ausdrücke die gleiche Anzahl von Ergebnisparametern liefern. Darüber hinaus müssen die Dimensionen korrespondierender Ergebnisparameter den Empfang und die Zusammensetzung zu einer Matrix nach den Konventionen der Funktion `dpgather` gestatten.

Erfolgt der Aufruf ohne Rückgabeparameter, kehrt `dpeval` nach Absetzen des vektoriellen RPCs sofort zurück (asynchrone Ausführung).

dpevalids

Aufgabe

Ermittlung der an einem vektoriellen RPC teilnehmenden DP-Instanzen.

Synopsis

```
dpids = dpevalids
```

Beschreibung

Die Funktion `dpevalids` kann innerhalb eines RPCs ausgeführt werden und liefert die Identifikatoren aller beteiligten DP-Instanzen zurück.

dpevalparent

Aufgabe

Ermittlung der DP-Instanz, die einen vektoriellen RPC ausgelöst hat.

Synopsis

```
dpid = dpevalparent
```

Beschreibung

Die Funktion `dpevalparent` kann innerhalb eines RPCs ausgeführt werden und liefert den Identifikator der DP-Instanz zurück, von der der RPC abgesetzt wurde.

dpfeval

Aufgabe

Vektorielle RPCs mit Aufrufparameter.

Synopsis

```
dpfeval(s,[],a,b,c,...)
dpfeval(dpids,s,[],a,b,c,...)
dpfeval(s,n,[],a,b,c,...)
[x,y,z,...] = dpfeval(...)
```

Beschreibung

Die Funktion `dpfeval` dient zur Ausführung vektorieller RPCs mit Aufrufparameter und ist das parallele Pendant zur MATLAB-Funktion `feval`.

Für die Parameter `s`, `dpids` und `n` gelten die selben Regeln wie bei der Funktion `dpeval`. Allerdings darf der Parameter `s` nicht mehr beliebige MATLAB-Ausdrücke, sondern nur Funktionen enthalten.

Die Aufrufparameter für die entfernt auszuführenden Funktionen sind durch eine leere Matrix getrennt, nach den dpfeval-Parametern anzugeben und werden gemäß den Konventionen der Funktion `dpscatter` verteilt.

dpgather

Aufgabe

Einsammeln der Elemente einer MATLAB-Matrix.

Synopsis

```
mat = dpgather(dpids)
mat = dpgather(mat_name)
mat = dpgather(dpids,mat_name)
[mat,probe] = dpgather(...)
```

Beschreibung

Die Funktion `dpgather` empfängt Teilmatrizen von verschiedenen Empfängern und bildet daraus eine zusammengesetzte Matrix.

Die Art und Weise der Zusammensetzung von `mat` richtet sich nach der Struktur von `dpids` bzw. von `mat_name`. Erfolgt die Angabe der Identifikatoren der Absenderinstanzen bzw. der Namen der zu empfangenden Teilmatrizen als Zeilenvektor (mehrere Namen in einem Zeilenvektor müssen durch Leerzeichen oder Kommata getrennt werden), so wird `mat` spaltenweise zusammengesetzt. Ist `dpids` ein Spaltenvektor bzw. `mat_name` eine

Multi-String-Matrix erfolgt die Zusammensetzung von `mat` zeilenweise. Damit eine solche Zusammensetzung möglich ist, müssen die Zeilen- bzw. Spaltendimensionen der empfangenen Teilmatrizen übereinstimmen.

`dpgather` unterstützt wie die Funktion `dprecv` den unbedingten und bedingten Empfang sowie verschiedene Varianten der Nachrichtenselektion (nach Absender und/oder Matrizennamen, s. `dprecv`).

dpmm

Aufgabe

Starten, Konfigurieren und Terminieren einer DPMM.

Synopsis

```
dpids = dpmm(n)
dpids = dpmm(n,run_mode,trace_mode)
dpids = dpmm(where)
dpids = dpmm(where,run_mode,trace_mode)
```

Beschreibung

Die Funktion `dpmm` stellt ein alternatives DPMM-Konfigurationsinterface zu den Funktionen `dpon`, `dpadd`, `dpchg`, `dpdel` und `dpoff` dar, bei dem nicht explizite Änderungen an einer vorhandenen Konfiguration, sondern eine gewünschte Konfiguration spezifiziert wird.

Die Syntax von `dpmm` entspricht exakt der der Funktion `dpadd`. Im Unterschied zu `dpadd` spezifizieren die Parameter `n`, `where`, `run_mode` und `trace_mode` aber keine Konfigurationserweiterung, sondern eine durch `dpmm` einzurichtende Konfiguration. Dies kann entweder auf Basis einer bereits existierenden DPMM oder durch den Neustart eine DPMM erfolgen.

Da der Funktion `dpmm` kein DPMM-Identifikator übergeben werden kann (vgl. `dpon`), wird zur Registrierung einer eventuell zu startenden DPMM immer der Standardwert *dpmmid*=1 benutzt.

Mit `dpmm(0)` wird eine laufende DPMM terminiert.

dpmmid

Aufgabe

Bestimmung des DPMM-Identifikators.

Synopsis

```
id = dpmmid
```

Beschreibung

Die Funktion `dpmmid` ermittelt den Identifikator der DPMM, bei der die ausführende DP-Instanz registriert ist.

Ein Aufruf von `dpmmid` in einer autonomen MATLAB-Instanz liefert als Ergebnis eine leere Matrix.

dpmmopt

Aufgabe

Lesen und Schreiben von Optionen der DPMM-Verwaltung.

Synopsis

```
option_val = dpmmopt(option_name)
dpmmopt(option_name,option_val)
```

Beschreibung

Mit Hilfe der Funktion `dpmmopt` können die folgenden Optionen der DPMM-Verwaltung gelesen oder neu gesetzt werden:

reconfig: In der Standardbetriebsart *reconfig*=`'yes'` werden DP-Instanzen, die durch Konfigurationsoperationen aus einer DPMM entfernt werden, terminiert.
In der Betriebsart *reconfig*=`'no'` werden zu entfernenden DP-Instanzen nicht terminiert, sondern einer Pseudo-DPMM mit der *dpmmid*=0 zugeordnet, wo sie nachfolgenden Konfigurationsoperationen zur „Wiederverwendung“ zur Verfügung stehen.

config: In der Standardbetriebsart *config*=`'noblock'` werden Konfigurationsoperationen mit einer Fehlermeldung abgebrochen, wenn diese aufgrund von erschöpften Ressourcen (z.B. begrenzte Anzahl von MATLAB-Lizenzen) nicht unmittelbar ausgeführt werden können.
In der Betriebsart *config*=`'block'` wird die Ausführung von Konfigurationsoperationen solange blockiert, bis die erforderlichen Ressourcen zur Verfügung stehen.

dpmyid

Aufgabe

Bestimmung des DP-Identifikators.

Synopsis

```
dpid = dpmyid
```

Beschreibung

Die Funktion `dpmyid` ermittelt den Identifikator der ausführenden DP-Instanz.

Ein Aufruf von `dpmyid` in einer autonomen MATLAB-Instanz liefert als Ergebnis eine leere Matrix.

dpoff

Aufgabe

Terminierung einer DPMM.

Synopsis

```
dpoff
```

Beschreibung

Der Aufruf von `dpoff` führt zur Terminierung der DPMM und aller DP-Instanzen, die im Rahmen der DPMM gestartet wurden.

Instanzen, die der DPMM selbständig beigetreten sind (durch `dpon`), existieren nach Terminierung der DPMM als gewöhnliche MATLAB-Instanzen weiter.

dpon

Aufgabe

Starten einer DPMM.

Synopsis

```
dpon
dpon(id)
```

Beschreibung

Durch den Aufruf von `dpon` wird eine DPMM gestartet. Dabei wird die ausführende MATLAB-Instanz zu einer DP-Instanz dieser DPMM. Mit Hilfe des optionalen Parameters `id` kann ein DPMM-Identifikator (*dpmmid*: ganze Zahl ≥ 1) angegeben werden, unter welchem die DPMM registriert wird. Ohne Parameterangabe wird der Standardwert $dpmmid = 1$ benutzt.

Eine statische Konfiguration der zu startenden DPMM kann mit Hilfe der Funktion `dpadd` in einem M-Script mit dem Namen `dpmm`{*dpmmid*}`.m` vorgenommen werden. Dieses Script wird von `dpon` automatisch ausgeführt, wenn es entlang des MATLAB-Suchpfades gefunden wird.

Um Überschneidungen mehrerer DPMMs zu verhindern, kann `dpon` nicht in MATLAB-Instanzen ausgeführt werden, die bereits als DP-Instanz in einer DPMM registriert sind.

Der Aufruf von `dpon` in einer autonomen MATLAB-Instanz mit dem DPMM-Identifikator einer bereits existierenden DPMM führt zur Aufnahme der MATLAB-Instanz in die spezifizierte DPMM.

dprecv

Aufgabe

Empfangen einer MATLAB-Matrix.

Synopsis

```
mat = dprecv
mat = dprecv(dpids)
mat = dprecv(mat_name)
mat = dprecv(dpids,mat_name)
[mat,probe] = dprecv(...)
```

Beschreibung

`dprecv` unterstützt den Empfang von MATLAB-Matrizen, die durch `dpsend` oder `dpscatter` gesendet wurden, nach folgenden Betriebsarten:

`mat = dprev`: unbedingter Empfang einer beliebigen Matrix von einem beliebigen Absender;

`mat = dprev(dpids)`: unbedingter Empfang einer beliebigen Matrix von einer der in `dpids` spezifizierten DP-Instanzen;

`mat = dprev(mat_name)`: Empfang einer Matrix mit einem in der Multi-String-Matrix `mat_name` enthaltenen Namen von einem beliebigen Absender;

`mat = dprev(dpids,mat_name)`: Empfang einer Matrix mit einem in `mat_name` enthaltenen Namen von einer der in `dpids` spezifizierten DP-Instanzen;

`[mat,probe] = dprev(...)`: bedingter Empfang, liegt keine passende Matrix zum Empfang vor, kehrt der Aufruf von `dprecv` sofort mit `mat=[]` und `probe=0` zurück.

dpscatter

Aufgabe

Verteilen der Elemente einer MATLAB-Matrix.

Synopsis

```
dpscatter(dpids,mat)
dpscatter(dpids,mat,mat_name)
```

Beschreibung

Die Funktion `dpscatter` zerlegt eine Matrix in Teilmatrizen und verteilt diese an verschiedene Empfänger.

Die Art und Weise der Zerlegung von `mat` richtet sich nach der Struktur des Parametervektors `dpids`. Erfolgt die Angabe der Identifikatoren der Empfangsinstanzen als Zeilenvektor, so wird `mat` spaltenweise zerlegt. Ist `dpids` ein Spaltenvektor, erfolgt die Zerlegung von `mat` zeilenweise. Damit eine solche Zerlegung möglich ist, muß die Anzahl der Spalten bzw. Zeilen von `mat` gleich oder größer der Anzahl der Elemente von `dpids` sein.

Alle Teilmatrizen werden von `dpscatter` unter dem Namen von `mat` gesendet. Optional kann mit `mat_name` ein alternativer Name spezifiziert werden. Sollen die Teilmatrizen unter verschiedenen Namen verteilt werden, sind diese in Form einer Multi-String-Matrix als Parameter `mat_name` anzugeben.

dpsend

Aufgabe

Senden einer MATLAB-Matrix.

Synopsis

```
dpsend(dpids,mat)
dpsend(dpids,mat,mat_name)
```

Beschreibung

Mit `dpsend(dpids,mat)` kann eine beliebige MATLAB-Matrix (full, sparse, complex etc.) an ein oder mehrere Empfänger geschickt werden. Eventuell notwendige Konvertierungen der Datenrepräsentation zwischen heterogenen Architekturen werden automatisch ausgeführt.

Mit dem optionalen Parameter `mat_name` kann ein Name für die zu sendende Matrix `mat` angegeben werden. Dies ist sinnvoll, wenn es sich bei `mat` um eine unbenannte Matrix handelt (z.B. das Ergebnis eines MATLAB-Ausdrucks).

dpterm

Aufgabe

Umschaltung des Kommandointerfaces auf andere DP-Instanzen.

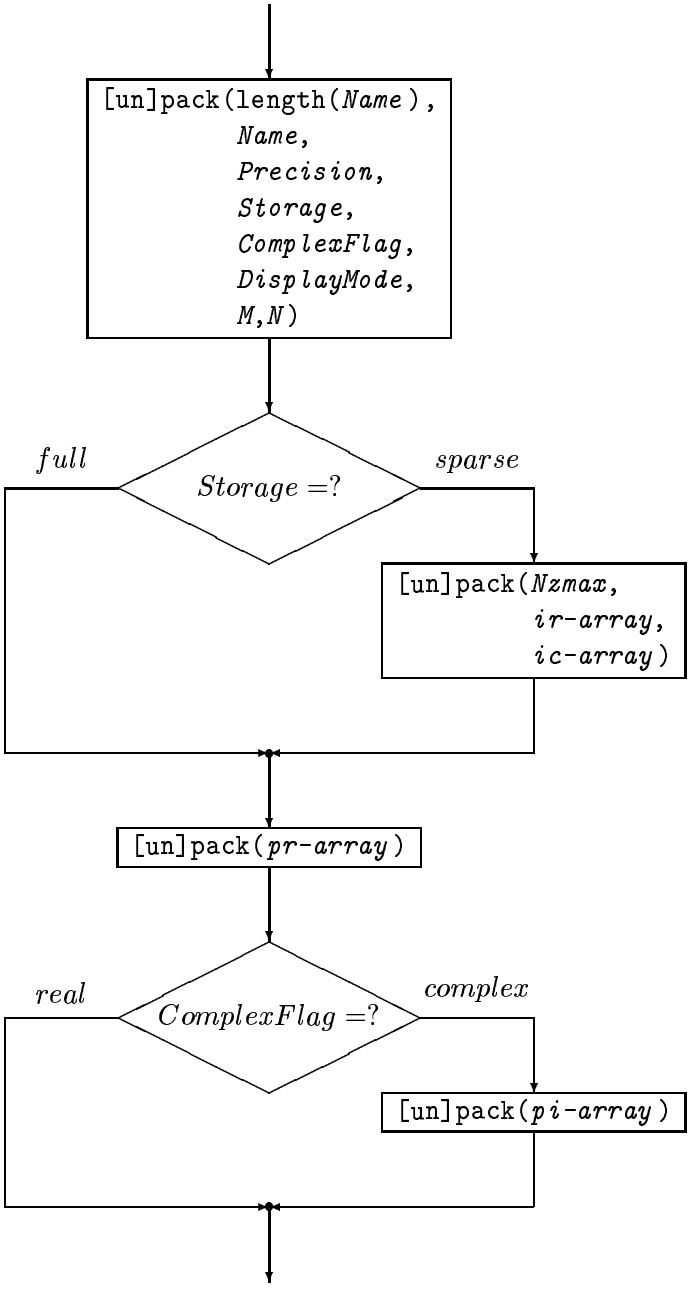
Synopsis

`dpterm(dpid)`

Beschreibung

Mit `dpterm(dpid)` kann das Kommandointerface einer DP-Instanz auf eine andere Instanz umgeschaltet werden, d.h alle nachfolgenden Eingaben werden von der mit dem Parameter `dpid` spezifizierten Instanz verarbeitet und die Ausgaben erscheinen auf der DP-Instanz, auf der `dpterm` ausgeführt wurde.

C.2 Algorithmus zum Packen und Entpacken von MATLAB-Matrizen



Anhang D

Listings der Beispielanwendungen

Die aufgeführten Programme sollen die im Kapitel 8 vorgestellten parallelen Lösungen der Beispielanwendungen illustrieren. Bei der Gestaltung der Programmkodes wurde deshalb in erster Linie auf eine gute Lesbarkeit Wert gelegt.

D.1 Monte-Carlo-Studie

MATLAB/DP-TB-Implementation

```
%------------
% dpmaster.m
%------------

   num_slaves = 10;        % number of slaves
   num_runs   = 1000;      % number of simulation runs

   t0 = 0;    tf = 2;      % simulation interval [0,2]
   h  = 0.001;             % integration stepsize

   x0 = [0 0.1]´;          % initial conditions x´(0)=0, x(0)=0.1

   % generate damping factors as random
   % quantities in the interval [800,1200]
   d = 800 + 400 * rand(num_runs,1);

   % configure a suitable dpmm
   dpids = dpmm(num_slaves+1);

   % start up slaves
   slaves = dpids(2:length(dpids));
   dpeval(slaves,´dpslave´);

   % distribute damping factors
   % and simulation parameters
   dpscatter(slaves,d)
   dpsend(slaves,[t0,tf,h,x0´])
```

```
   % gather results
   xmean = dpgather(slaves);

   % calculate final average and save it
   xmean = mean(xmean´)´
   save xmean



%-----------
% dpslave.m
%-----------

   % receive damping factors and simulation parameters
   d   = dprecv;
   par = dprecv; t0=par(1); tf=par(2); h=par(3); x0=par(4:5)´;

   % perform partial monte carlo study
   global D
   xmean = 0;
   for i = 1:length(d)
      D = d(i);
      x = rk4(´mass_spring_sys´,t0,tf,x0,h);
      xmean = xmean + x(:,2) / length(d);
   end

   % send result to master
   dpsend(dpevalparent,xmean)



%-------------------------------------
function  xdot = mass_spring_sys(t,x)
%-------------------------------------

   % model parameters
   k = 9000;  m = 450;  global D
   A = [-D/m  -k/m
          1     0 ];

   % model equations
   xdot = A * x;
```

PVM-Implementation

```
/*-------------
 * pvmmaster.c
 *-------------*/

#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

void  main(int argc, char** argv) {

   int     num_slaves = 10;        /* number of slaves          */
   int     num_runs   = 1000;      /* number of simulation runs */

   double  t0 = 0.,   tf=2.;       /* simulation interval [0,2] */
   double  h  = 0.001;             /* integration stepsize      */

   double  x0[] = {0., 0.1};       /* initial conditions
                                      x´(0)=0, x(0)=0.1         */

   int     i, j, n;                /* miscellaneous variables   */
   int     slaves[num_slaves];
   double  d[num_runs], *xmean;
   double  *xmeans[num_slaves];
   FILE    *fp;

   /* generate damping factors as random
      quantities in the interval [800,1200] */
   for (i=0; i<num_runs; i++) {
      d[i] = 800 + 400. * ( (double)rand() / (double)RAND_MAX );
   }

   /* start up slaves */
   pvm_spawn("pvmslave",(char**)0,0,"",num_slaves,slaves);

   /* distribute damping factors
      and simulation parameters */
   n = num_runs / num_slaves;
   for (i=0; i<num_slaves; i++) {
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&n,1,1);
      pvm_pkdouble(&d[i*n],n,1);
      pvm_send(slaves[i],1);
   }
   pvm_initsend(PvmDataRaw);
   pvm_pkdouble(&t0,1,1);
   pvm_pkdouble(&tf,1,1);
   pvm_pkdouble(&h,1,1);
   pvm_pkdouble(x0,2,1);
   pvm_mcast(slaves,num_slaves,1);
```

```
   /* gather results */
   for (i=0; i<num_slaves; i++) {
      pvm_recv(slaves[i],1);
      pvm_upkint(&n,1,1);
      xmeans[i] = (double*) calloc(n,sizeof(double));
      pvm_upkdouble(xmeans[i],n,1);
   }

   /* calculate final average and save it */
   xmean = (double*) calloc(n,sizeof(double));
   for (j=0; j<n; j++) {
      for (i=0; i<num_slaves; i++) {
         xmean[j] = xmean[j] + xmeans[i][j];
      }
      xmean[j] = xmean[j] / num_slaves;
   }
   fp = fopen("xmean.dat","w");
   fwrite(&n,sizeof(int),1,fp);
   fwrite(xmean,sizeof(double),n,fp);
   fclose(fp);

   pvm_exit();
   return;
}


/*-------------
 * pvmslave.c
 *-------------*/

#include <stdlib.h>
#include <pvm3.h>

void  mass_spring_sys(double, double*, double*);

void  rk4(void(*)(double, double*, double*),
            double, double, int, double*, double,
            int*, double**);
double D;

void  main(int argc, char** argv) {

   int     num_runs, x_size, i, j;
   double  t0, tf, h, x0[2], *d, **x2, *x[2], *xmean;

   /* receive damping factors
      and simulation parameters */
   pvm_recv(-1,1);
   pvm_upkint(&num_runs,1,1);
   d = (double*) calloc(num_runs,sizeof(double));
```

```
   pvm_upkdouble(d,num_runs,1);
   pvm_recv(-1,1);
   pvm_upkdouble(&t0,1,1);
   pvm_upkdouble(&tf,1,1);
   pvm_upkdouble(&h,1,1);
   pvm_upkdouble(x0,2,1);

   /* perform partial monte carlo study */
   x2 = (double**) calloc(num_runs,sizeof(double*));
   for (i=0; i<num_runs; i++) {
      D = d[i];
      rk4(mass_spring_sys,t0,tf,2,x0,h, &x_size,x);
      x2[i] = x[1];  free(x[0]);
   }

   /* calculate partial average */
   xmean = (double*) calloc(x_size,sizeof(double));
   for (j=0; j<x_size; j++) {
      for (i=0; i<num_runs; i++) {
         xmean[j] = xmean[j] + x2[i][j];
      }
      xmean[j] = xmean[j] / num_runs;
   }

   /* send results to master */
   pvm_initsend(PvmDataRaw);
   pvm_pkint(&x_size,1,1);
   pvm_pkdouble(xmean,x_size,1);
   pvm_send(pvm_parent(),1);

   pvm_exit();
   return;
}


/*-------------------------------------------------------*/
void  mass_spring_sys(double t, double* x, double* xdot) {
/*-------------------------------------------------------*/

   /* model parameter */
   double  k = 9000.,  m = 450.;
   double  a[] = {-D/m, -k/m};

   /* model equations */
   xdot[0] = a[0] * x[0]  +  a[1] * x[1];
   xdot[1] =        x[0];

   return;
}
```

D.2 Simulation eines Räuber-Beute-Systems

Anfangszustände und Modellparameter

$v_1(0) = 1$	$a_v = 2$	$b_v = 0.5$	$c_v = 0.01$	$d_v = 0.2$
$v_2(0) = 1$	$e_v = 0.4$	$f_v = 0.02$	$g_v = 0.01$	$h_v = 0.02$
	$j_v = 0.01$	$k_v = 0.03$		
$w_1(0) = 1$	$a_w = 1$	$b_w = 0.5$	$c_w = 0.02$	$d_w = 0.1$
$w_2(0) = 1$	$e_w = 0.4$	$f_w = 0.04$	$g_w = 0.02$	$h_w = 0.04$
$x_1(0) = 1$	$a_x = 3$	$b_x = 0.9$	$c_x = 0.02$	$d_x = 0.2$
$x_2(0) = 1$	$e_x = 0.2$	$f_x = 0.04$	$g_x = 0.025$	$h_x = 0.1$
$y_1(0) = 1$	$a_y = 1$	$b_y = 0.8,$	$c_y = 0.04$	$d_y = 0.2$
$y_2(0) = 1$	$e_y = 0.6$	$f_y = 0.07$	$g_y = 0.03$	$h_y = 0.025$
$z_1(0) = 1$	$a_z = 3$	$b_z = 0.7$	$c_z = 0.02$	$d_z = 0.5$
$z_2(0) = 1$	$e_z = 0.3$	$f_z = 0.04$	$g_z = 0.02$	$h_z = 0.04$

MATLAB/DP-TB-Implementation

```
%------------
% dpmaster.m
%------------

   t0 = 0;    tf=100;   % simulation interval [0,100]
   h  = 0.01;           % integration stepsize
   cint = 20;           % communication interval

   s0 = [1 1]´;         % initial conditions

   % configure a suitable dpmm
   dpids = dpmm(6);

   % start up slaves
   slaves = dpids(2:6);
   dpeval(slaves,´dpslave´)

   % distribute simulation parameters
   dpsend(slaves,[t0,tf,h,cint,s0´])

   % gather and save results
   s = dpgather(slaves);
   save s
```

```
%-----------
% dpslave.m
%-----------

   master = dpevalparent;

   % receive simulation parameters
   par = dprecv(master);
   t0=par(1); tf=par(2); h=par(3); cint=par(4); s0=par(5:6)´;

   % who is who
   global S V W X Y Z
   S = dpmyid;
   dpids = dpevalids;
   V=dpids(1); W=dpids(2); X=dpids(3); Y=dpids(4); Z=dpids(5);

   % integrate one population pair
   s = rk4_cmex(´popul_sys´,t0,tf,s0,h,´comm´,cint)

   % send results to master
   dpsend(master,s);



%--------------------------------
function  sdot = popul_sys(t, s)
%--------------------------------

   global S V W X Y Z
   global V2 W1 X1 X2 Y1 Z1 Z2

   % parameters and dependences
   if S==V
      a = 2;     b = 0.5;   c = 0.01;  d = 0.2;  e = 0.4;
      f = 0.02;  g = 0.01;  h = 0.02;  j = 0.01; k = 0.03;
      r = 0;
      u = s(2) * (g*W1 + h*X1 + j*Y1 + k*Z1);
   end
   if S==W
      a = 1;     b = 0.5;   c = 0.02;  d = 0.1;  e = 0.4;
      f = 0.04;  g = 0.02;  h = 0.04;
      r = s(1) * (-g*V2 + h*X2);
      u = 0;
   end
   if S==X
      a = 3;     b = 0.9;   c = 0.02;  d = 0.2;  e = 0.2;
      f = 0.04;  g = 0.025; h = 0.1;
      r = -g*s(1)*V2;
      u = -h*s(2)*W1;
   end
```

```
   if S==Y
      a = 1;     b = 0.8;   c = 0.04;  d = 0.2;  e = 0.6;
      f = 0.07;  g = 0.03 ; h = 0.025;
      r = s(1) * (-g*V2 + h*Z2);
      u = 0;
   end
   if S==Z
      a = 3;     b = 0.7;   c = 0.02;  d = 0.5;  e = 0.3;
      f = 0.04;  g = 0.02 ; h = 0.04;
      r = -g*s(1)*V2;
      u = -h*s(2)*Y1;
   end

   % state equations
   sdot(1) =  a*s(1) - b*s(1)*s(2) - c*s(1)^2 + r;
   sdot(2) = -d*s(2) + e*s(1)*s(2) - f*s(2)^2 + u;



%-----------------
function  comm(s)
%-----------------

   global S V W X Y Z
   global V2 W1 X1 X2 Y1 Z1 Z2

   % exchange state values between dependend systems
   if S==V
      dpsend([W X Y Z],s(2))
      W1=dprecv(W); X1=dprecv(X); Y1=dprecv(Y); Z1=dprecv(Z);
   end
   if S==W
      dpsend([V X],s(1))
      V2=dprecv(V); X2=dprecv(X);
   end
   if S==X
      dpsend(V,s(1)), dpsend(W,s(2))
      V2=dprecv(V); W1=dprecv(W);
   end
   if S==Y
      dpsend([V Z],s(1))
      V2=dprecv(V); Z2=dprecv(Z);
   end
   if S==Z
      dpsend(V,s(1)), dpsend(Y,s(2))
      V2=dprecv(V); Y1=dprecv(Y);
   end
```

PVM-Implementation

```
/*-------------
 * pvmmaster.c
 *-------------*/

#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

void  main(int argc, char** argv) {

   double  t0 = 0.,   tf = 100.;   /* simulation interval [0,100] */
   double  h  = 0.01;              /* integration stepsize        */
   int     cint = 20;              /* communication interval      */

   double  s0[] = {1., 1.};        /* initial conditions          */

   int     i, j, slaves[5];        /* miscellaneous variables     */
   int     n, sout_size;
   double  *s[10];
   FILE    *fp;

   /* start up slaves */
   pvm_spawn("pvmslave",(char**)0,0,"",5,slaves);

   /* distribute simulation parameters
      and slave ids                    */
   pvm_initsend(PvmDataRaw);
   pvm_pkdouble(&t0,1,1);
   pvm_pkdouble(&tf,1,1);
   pvm_pkdouble(&h,1,1);
   pvm_pkint(&cint,1,1);
   pvm_pkdouble(s0,2,1);
   pvm_pkint(slaves,5,1);
   pvm_mcast(slaves,5,1);

   /* gather and save results */
   j=0;
   for (i=0; i<5; i++) {
      pvm_recv(slaves[i],1);
      pvm_upkint(&sout_size,1,1);
      s[j] = (double*) calloc(sout_size,sizeof(double));
      pvm_upkdouble(s[j],sout_size,1);
      j++;
      s[j] = (double*) calloc(sout_size,sizeof(double));
      pvm_upkdouble(s[j],sout_size,1);
      j++;
   }
   fp = fopen("s.dat","w");
   n=10;
   fwrite(&n,sizeof(int),1,fp);
   fwrite(&sout_size,sizeof(int),1,fp);
   for (i=0; i<n; i++) {
      fwrite(s[i],sizeof(double),sout_size,fp);
   }

   pvm_exit();
   return;
}
```

```
/*------------
 * pvmslave.c
 *------------*/

#include <stdlib.h>
#include <math.h>
#include <pvm3.h>

void  popul_sys(double, double*, double*);

void  comm(double*);

void  rk4(void (*)(double, double*, double*),
          double, double, int, double*, double,
          void (*)(double*), int*, double**);

int     S, V, W, X, Y, Z;
double  V2, W1, X1, X2, Y1, Z1, Z2;

void  main(int argc, char** argv) {

   int     master, slaves[5];
   int     sout_size, cint, i;
   double  t0, tf, h, s0[2], *s[2];

   master = pvm_parent();

   /* receive simulation parameters
      and slave ids                 */
   pvm_recv(master,1);
   pvm_upkdouble(&t0,1,1);
   pvm_upkdouble(&tf,1,1);
   pvm_upkdouble(&h,1,1);
   pvm_upkint(&cint,1,1);
   pvm_upkdouble(s0,2,1);
   pvm_upkint(slaves,5,1);

   /* who is who */
   S=pvm_mytid();
   V=slaves[0]; W=slaves[1]; X=slaves[2]; Y=slaves[3]; Z=slaves[4];

   /* integrate one population pair */
   rk4(&popul_sys,t0,tf,2,s0,h,&comm,cint,&sout_size,s);

   /* send results to master */
   pvm_initsend(PvmDataRaw);
   pvm_pkint(&sout_size,1,1);
   pvm_pkdouble(s[0],sout_size,1);
   pvm_pkdouble(s[1],sout_size,1);
   pvm_send(master,1);

   pvm_exit();
   return;
}
```

```
/*-------------------------------------------------*/
void  popul_sys(double t, double* s, double* sdot) {
/*-------------------------------------------------*/

   double  a, b, c, d, e, f, g, h, j, k, r, u;

   /* parameters and dependences */
   if (S==V) {
      a = 2.;    b = 0.5;   c = 0.01;  d = 0.2;
      e = 0.4;   f = 0.02;  g = 0.01;  h = 0.02;
      j = 0.01;  k = 0.03;
      r = 0.;
      u = s[1] * (g*W1 + h*X1 + j*Y1 + k*Z1);
   }
   if (S==W) {
      a = 1.;    b = 0.5;   c = 0.02;  d = 0.1;
      e = 0.4;   f = 0.04;  g = 0.02;  h = 0.04;
      r = s[0] * (-g*V2 + h*X2);
      u = 0.;
   }
   if (S==X) {
      a = 3.;    b = 0.9;   c = 0.02;  d = 0.2;
      e = 0.2;   f = 0.04;  g = 0.025; h = 0.1;
      r = -g*s[0]*V2;
      u = -h*s[1]*W1;
   }
   if (S==Y) {
      a = 1.;    b = 0.8;   c = 0.04;  d = 0.2;
      e = 0.6;   f = 0.07;  g = 0.03 ; h = 0.025;
      r = s[0] * (-g*V2 + h*Z2);
      u = 0.;
   }
   if (S==Z) {
      a = 3.;    b = 0.7;   c = 0.02;  d = 0.5;
      e = 0.3;   f = 0.04;  g = 0.02 ; h = 0.04;
      r = -g*s[0]*V2;
      u = -h*s[1]*Y1;
   }

   /* state equations */
   sdot[0] =  a*s[0] - b*s[0]*s[1] - c*pow(s[0],2) + r;
   sdot[1] = -d*s[1] + e*s[0]*s[1] - f*pow(s[1],2) + u;

   return;
}
```

```
/*----------------------*/
void  comm(double* s) {
/*----------------------*/

   /* exchange state values between neighbours */
   if (S==V) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[1],1,1);
      pvm_send(W,1); pvm_send(X,1); pvm_send(Y,1); pvm_send(Z,1);
      pvm_recv(W,1); pvm_upkdouble(&W1,1,1);
      pvm_recv(X,1); pvm_upkdouble(&X1,1,1);
      pvm_recv(Y,1); pvm_upkdouble(&Y1,1,1);
      pvm_recv(Z,1); pvm_upkdouble(&Z1,1,1);
   }
   if (S==W) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[0],1,1);
      pvm_send(V,1); pvm_send(X,1);
      pvm_recv(V,1); pvm_upkdouble(&V2,1,1);
      pvm_recv(X,1); pvm_upkdouble(&X2,1,1);
   }
   if (S==X) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[0],1,1);
      pvm_send(V,1);
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[1],1,1);
      pvm_send(W,1);
      pvm_recv(V,1); pvm_upkdouble(&V2,1,1);
      pvm_recv(W,1); pvm_upkdouble(&W1,1,1);
   }
   if (S==Y) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[0],1,1);
      pvm_send(V,1); pvm_send(Z,1);
      pvm_recv(V,1); pvm_upkdouble(&V2,1,1);
      pvm_recv(Z,1); pvm_upkdouble(&Z2,1,1);
   }
   if (S==Z) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[0],1,1);
      pvm_send(V,1);
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&s[1],1,1);
      pvm_send(Y,1);
      pvm_recv(V,1); pvm_upkdouble(&V2,1,1);
      pvm_recv(Y,1); pvm_upkdouble(&Y1,1,1);
   }

   return;
}
```

D.3 Lösung einer partiellen Differentialgleichung

MATLAB/DP-TB-Implementation

```
%------------
% dpmaster.m
%------------

   num_slaves = 16;        % number of slaves

   t0 = 0;    tf = 30;     % simulation interval [0,30]
   h  = 0.005;             % integration stepsize

   N  = 800;               % discretisation intervals
   lm = [ 80 200 400 ...   % lines to monitor: L/10, L/4, L/2,
         600 720];         % 3L/4, 9L/10

   % configure a suitable dpmm
   dpids = dpmm(num_slaves+1);

   % start up slaves
   slaves = dpids(2:length(dpids));
   dpeval(slaves,´dpslave´);

   % distribute simulation parameters
   % and lines to integrate
   dpsend(slaves,[t0,tf,h,N,lm]);
   dpscatter(slaves,[1:N-1]);

   % gather and save lines to monitor
   u = dpgather(mat2str(lm));
   save u


%-----------
% dpslave.m
%-----------

   master = dpevalparent;

   % receive simulation parameters and lines to integrate
   par = dprecv(master);
   t0=par(1); tf=par(2); h=par(3); N=par(4); lm=par(5:length(par));
   li = dprecv(master);

   % determine neighbours
   dpids = [0 dpevalids 0];
   i = find(dpids==dpmyid);
   LEFT_ID  = dpids(i-1);
   RIGHT_ID = dpids(i+1);
```

```
   % integrate partial system
   global LEFT_ID  RIGHT_ID  N
   u0   = zeros(2*length(li),1);
   cint = 1;
   u = rk4('equ_sys',t0,tf,u0,h,'comm',cint);

   % send lines to monitor to master
   for i = 1:length(lm)
      j = find(li==lm(i));
      if ~isempty(j),  dpsend(master,u(:,j),int2str(lm(i)));  end
   end


%-----------------------------
function  udot = equ_sys(t,u)
%-----------------------------

   % parameters
   L = 10;  a = 2;  b = 1;  d = 0.2;  omega = 1;  global N
   k = L/N;

   % left and right border
   global LEFT_ID LEFT_VAL RIGHT_ID RIGHT_VAL
   if ~LEFT_ID,   LEFT_VAL  = 0;                          end
   if ~RIGHT_ID,  RIGHT_VAL = b*exp(-d*t)*sin(omega*t);  end

   % state equations
   n = length(u)/2;
   % u1dot
   udot(n+1)       = LEFT_VAL - 2*u(1)     + u(2);
   udot(n+2:2*n-1) = u(1:n-2) - 2*u(2:n-1) + u(3:n);
   udot(2*n)       = u(n-1)   - 2*u(n)     + RIGHT_VAL;
   udot(n+1:2*n)   = udot(n+1:2*n) / (k^2*a);
   % u2dot
   udot(1:n)       = u(n+1:2*n);


%-----------------
function  comm(u)
%-----------------

   % exchange state values between neighbours
   global LEFT_ID LEFT_VAL RIGHT_ID RIGHT_VAL
   if LEFT_ID
      dpsend(LEFT_ID,u(1));  LEFT_VAL = dprecv(LEFT_ID);
   end
   if RIGHT_ID
      dpsend(RIGHT_ID,u(length(u)/2));  RIGHT_VAL = dprecv(RIGHT_ID);
   end
```

PVM-Implementation

```
/*-------------
 * pvmmaster.c
 *-------------*/

#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

void  main(int argc, char** argv) {

   int     num_slaves = 16;        /* number of slaves             */

   double  t0 = 0.,   tf = 30.;    /* simulation interval [0,30]   */
   double  h  = 0.005;             /* integration stepsize         */

   int     N    = 800;             /* discretisation intervals     */
   int     lm[] = { 80, 200,       /* lines to monitor: L/10, L/4, */
                    400, 600, 720}; /* L/2, 3L/4, 9L/10            */
   int     lm_size = 5;

   int     i, j, li_size;          /* miscellaneous variables      */
   int     li[N-1], uout_size;
   int     slaves[num_slaves];
   double  *u[lm_size];
   FILE    *fp;

   /* start up slaves */
   pvm_spawn("pvmslave",(char**)0,0,"",num_slaves,slaves);

   /* distribute simulation parameters,
      neighbour ids and lines to integrate */
   pvm_initsend(PvmDataRaw);
   pvm_pkdouble(&t0,1,1);
   pvm_pkdouble(&tf,1,1);
   pvm_pkdouble(&h,1,1);
   pvm_pkint(&N,1,1);
   pvm_pkint(&lm_size,1,1);
   pvm_pkint(lm,lm_size,1);
   pvm_pkint(&num_slaves,1,1);
   pvm_pkint(slaves,num_slaves,1);
   pvm_mcast(slaves,num_slaves,1);
   for (i=0; i<N-1; i++) {
      li[i] = i+1;
   }
   li_size = (N-1)/num_slaves;
   for (i=0; i<num_slaves-1; i++) {
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&li_size,1,1);
      pvm_pkint(&li[i*li_size],li_size,1);
      pvm_send(slaves[i],1);
   }
```

```
   li_size = (N-1)/num_slaves + (N-1)%num_slaves;
   pvm_initsend(PvmDataRaw);
   pvm_pkint(&li_size,1,1);
   pvm_pkint(&li[i*(N-1)/num_slaves],li_size,1);
   pvm_send(slaves[i],1);

   /* gather and save lines to monitor */
   for (i=0; i<lm_size; i++) {
      pvm_recv(-1,lm[i]);
      pvm_upkint(&uout_size,1,1);
      u[i] = (double*) calloc(uout_size,sizeof(double));
      pvm_upkdouble(u[i],uout_size,1);
   }
   fp = fopen("u.dat","w");
   fwrite(&lm_size,sizeof(int),1,fp);
   fwrite(&uout_size,sizeof(int),1,fp);
   for (i=0; i<lm_size; i++) {
      fwrite(u[i],sizeof(double),uout_size,fp);
   }

   pvm_exit();
   return;
}


/*------------
 * pvmslave.c
 *------------*/

#include <stdlib.h>
#include <math.h>
#include <pvm3.h>

void  equ_sys(double, double*, double*);

void  comm(double*);

void  rk4(void (*)(double, double*, double*),
          double, double, int, double*, double,
          void (*)(double*), int*, double**);

int     N, U_SIZE;
int     LEFT_ID,  RIGHT_ID;
double  LEFT_VAL, RIGHT_VAL;

void  main(int argc, char** argv) {

   int     master, *lm, lm_size, num_slaves, *slaves;
   int     *li, li_size, uout_size, cint, i, j;
   double  t0, tf, h, *u0, **u;
```

```
   master = pvm_parent();

   /* receive simulation parameters
      neighbour ids and lines to integrate */
   pvm_recv(master,1);
   pvm_upkdouble(&t0,1,1);
   pvm_upkdouble(&tf,1,1);
   pvm_upkdouble(&h,1,1);
   pvm_upkint(&N,1,1);
   pvm_upkint(&lm_size,1,1);
   lm = (int*) calloc(lm_size,sizeof(int));
   pvm_upkint(lm,lm_size,1);
   pvm_upkint(&num_slaves,1,1);
   slaves = (int*) calloc(num_slaves+2,sizeof(int));
   pvm_upkint(&slaves[1],num_slaves,1);
   pvm_recv(master,1);
   pvm_upkint(&li_size,1,1);
   li = (int*) calloc(li_size,sizeof(int));
   pvm_upkint(li,li_size,1);

   /* determine neighbours */
   for (i=1; i<=num_slaves; i++) {
      if (slaves[i]==pvm_mytid()) {
         LEFT_ID  = slaves[i-1];
         RIGHT_ID = slaves[i+1];
      }
   }

   /* integrate partial system */
   U_SIZE = 2 * li_size;
   u0     = (double*)  calloc(U_SIZE,sizeof(double));
   u      = (double**) calloc(U_SIZE,sizeof(double*));
   cint   = 1;
   rk4(&equ_sys,t0,tf,U_SIZE,u0,h,&comm,cint,&uout_size,u);

   /* send lines to monitor to master */
   for (i=0; i<lm_size; i++) {
      for (j=0; j<li_size; j++) {
         if (lm[i]==li[j]) {
            pvm_initsend(PvmDataRaw);
            pvm_pkint(&uout_size,1,1);
            pvm_pkdouble(u[j],uout_size,1);
            pvm_send(master,lm[i]);
         }
      }
   }

   pvm_exit();
   return;
}
```

```
/*-------------------------------------------------*/
void  equ_sys(double t, double* u, double* udot) {
/*-------------------------------------------------*/

   int     n, i;

   /* parameters */
   double  L = 10.,  a = 2.,  b = 1.,  d = 0.2,  omega = 1.;
   double  k = L/(double)N;

   /* left and right border */
   if (!LEFT_ID)  { LEFT_VAL  = 0; }
   if (!RIGHT_ID) { RIGHT_VAL = b*exp(-d*t)*sin(omega*t); }

   /* state equations */
   n = U_SIZE/2;
   /* u1dot */
   udot[n]      = (LEFT_VAL - 2*u[0]   + u[1]     ) / (pow(k,2)*a);
   for (i=1; i<(n-1); i++) {
      udot[n+i] = (u[i-1]   - 2*u[i]   + u[i+1]   ) / (pow(k,2)*a);
   }
   udot[2*n-1]  = (u[n-2]   - 2*u[n-1] + RIGHT_VAL) / (pow(k,2)*a);
   /* u2dot */
   for (i=0; i<n; i++) {
      udot[i]   = u[n+i];
   }

   return;
}


/*----------------------*/
void  comm(double* u) {
/*----------------------*/

   /* exchange state values between neighbours */
   if (LEFT_ID) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&u[0],1,1);
      pvm_send(LEFT_ID,1);

      pvm_recv(LEFT_ID,1);
      pvm_upkdouble(&LEFT_VAL,1,1);
   }
   if (RIGHT_ID) {
      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&u[U_SIZE/2-1],1,1);
      pvm_send(RIGHT_ID,1);

      pvm_recv(RIGHT_ID,1);
      pvm_upkdouble(&RIGHT_VAL,1,1);
   }

   return;
}
```

Über den Autor ...

Dr.-Ing. Sven Pawletta studierte Elektrotechnik an der Universität Rostock, wo er seit 1992 in verschiedenen Forschungsgruppen für Modellbildung und Simulation der Fachbereiche Informatik und Elektrotechnik arbeitet. Die Ergebnisse seiner mehrjährigen Forschung zur parallelen und verteilten Verarbeitung auf Basis interaktiver Berechnungssysteme legte er im Juni 1998 in Form einer Dissertation vor.
Derzeit arbeitet er an der Entwicklung von Methoden zur verteilten Simulation auf Basis von Matlab in Verbindung mit der High Level Architecture (HLA).

Über diesen Band ...

Erweiterungen eines wissenschaftlich-technischen Berechnungs- und Visualisierungssystems zu einer Entwicklungsumgebung für parallele Applikationen zeigt erstmals einen praktikablen Weg auf, wie MATLAB und ähnliche Systeme für die verteilte und parallele Verarbeitung eingesetzt werden können. Zu Beginn werden einerseits die hardware- und softwaretechnischen Grundlagen der Parallelverarbeitung in leicht verständlicher Form eingeführt und andererseits die Besonderheiten der weit verbreiteten wissenschaftlich-technischen Berechnungs-umgebungen (SCEs) gegenüber der traditionellen Programmierung mit kompilierbaren Sprachen herausgearbeitet. Darauf aufbauend, wird das neue Konzept der Multi-SCEs in Analogie zu den MIMD-Rechnerarchitekturen entwickelt.
Als beispielhafte Umsetzung des Multi-SCE-Konzepts wird die DP-Toolbox (Distributed and Parallel Application Toolbox) für MATLAB beschrieben.
Abschließend werden alle Simulationsaufgaben der EUROSIM-SNE Comparison CP1 gelöst. Auf weitere publizierte Anwendungen der DP-Toolbox in der Industrie und an Universitäten wird verwiesen

Über diese Reihe ...

Die Bände dieser neuen 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.
Die **Fortschrittsberichte Simulation** konzentrieren sich auf Monographien mit speziellem Charakter, wie z. B. Dissertationen und Habilitationen, Berichte zu größeren ASIM Fachgruppen-Treffen (mit referierten Beiträgen) und Forschungsprojekten, Handbücher zu Simulationswerkzeugen (User Guides, Vergleiche, Benchmarks), und ähnliches.
ASIM, die deutschsprachige Simulationsvereinigung (**A**rbeitsgemeinschaft **SIM**ulation, zugleich Fachausschuss 4.5 der GI - Gesellschaft für Informatik) hat diese Reihe im ARGESIM/ASIM-Verlag als Ergänzung zur ASIM- Reihe ***Fortschritte in der Simulationstechnik*** (Vieweg Verlag) ins Leben gerufen, um ein rasches und kostengünstiges Publikationsmedium für neue Entwicklungen in der Simulationstechnik anbieten zu können. Die Kooperation mit den **ARGESIM Reports** der ARGE Simulation News (ARGESIM) vermittelt dabei zum europäischen Umfeld und zur internationalen Publikation.