Verifizierte globale Optimierung auf Parallelrechnern

Dissertation
von
Andreas Wiethoff

Karlsruhe 1997

Verifizierte globale Optimierung auf Parallelrechnern

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

von der Fakultät für Mathematik der
Universität Karlsruhe (TH)
genehmigte

DISSERTATION

von

Dipl.-Math. Andreas Wiethoff

aus Münster (Westf.)

Tag der mündlichen Prüfung: ..... 17. Dezember 1997
Referent: ..... Prof. Dr. U. Kulisch
Korreferent: ..... H.-Doz. Dr. W. Krämer

Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben. Mein besonderer Dank gebührt meinem Referenten, Herrn Prof. Dr. U. Kulisch, der mir die Möglichkeit gegeben hat, an seinem Institut zu arbeiten und diese Arbeit anzufertigen. Er hat die Entstehung dieser Arbeit mit großem Interesse begleitet. Ebenso möchte ich Herrn H.-Doz. Dr. W. Krämer für die freundliche Übernahme des Korreferats und für die vielen konstruktiven Anmerkungen danken.

Allen Mitarbeiterinnen und Mitarbeitern des Instituts für Angewandte Mathematik sei für die langjährige gute Zusammenarbeit gedankt. Ganz besonders möchte ich mich bei den Herren Dr. R. Klatte und Dr. D. Ratz für die vielen fruchtbaren Diskussionen und Ratschläge, die das Zustandekommen dieser Arbeit gefördert haben, bedanken.

Ein ganz besonders herzliches Dankeschön geht an meine Frau Christine für Anteilnahme, Aufheiterung und Ablenkung und an meine Eltern für andauernde Unterstützung und Rat auf meinem bisherigen Lebensweg.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Einleitung

Problemstellungen, die sich in Form eines globalen Optimierungsproblems repräsentieren lassen, treten häufig bei der mathematischen Modellierung von Anwendungen aus Wirtschaft und Wissenschaft auf. Derartige Anwendungen sind in so unterschiedlichen Bereichen wie Finanzwesen, Steuerung von industriellen Produktionsanlagen, Chip-Entwurf, Strukturoptimierung, *operations research*, Molekularbiologie uvm. zu finden. Globale Optimierung ist aus mathematischer Sicht die Bestimmung und Charakterisierung des globalen Minimums (oder Maximums) einer in der Regel nichtlinearen Funktion. Häufig ist man auch an den zugehörigen Minimalstellen interessiert, an denen das globale Minimum angenommen wird. Globale Optimierung ist wesentlich „schwieriger“ als lokale Optimierung, bei der nur ein beliebiges lokales Minimum in der Nähe eines Startpunktes gesucht wird. Viele Anwendungsprobleme sind durch eine sehr große Zahl an lokalen Minima gekennzeichnet. Es ist daher nicht praktikabel, alle lokalen Minima zu bestimmen und unter diesen dann das kleinste herauszufiltern. Stattdessen müssen globale Informationen verwendet werden, um den Suchaufwand zu beschränken.

Intervallmethoden sind gut geeignet, um mit vergleichsweise wenig Aufwand derartige globale Informationen zu erhalten. Intervalle werden auf dem Rechner durch zwei Maschinenzahlen dargestellt. Dennoch bezeichnet ein Intervall eine in der Regel unendliche Teilmenge der reellen Zahlen, nicht nur die endliche Menge der zwischen den beiden Schranken liegenden Maschinenzahlen. Intervalle sind somit Träger globaler Information. Verwendet man Intervalle in einem strikten mathematischen Kalkül, der in Abschnitt 1.4 eingeführten Intervallarithmetik, so lassen sich mathematisch gesicherte Aussagen auf dem Rechner beweisen. Beispielsweise kann durch eine einzige Maschinenintervallauswertung (sie ist nur etwa doppelt so auf-

wendig wie eine entsprechende reelle Auswertung) einer Zielfunktion bei der globalen Optimierung eine gesicherte Aussage über den Wertebereich über dem gesamten Intervall getroffen werden. Alle auf dem Rechner auftretenden Rundungsfehler werden dabei durch die Maschinenintervallarithmetik automatisch miterfaßt.

Natürlich lassen sich „klassische“ numerische Algorithmen zur globalen Optimierung nicht ohne weiteres mit der Intervallarithmetik kombinieren. Um effizient mathematisch gesicherte Ergebnisse zu bestimmen, sind andere, neuartige Algorithmen erforderlich. Für die globale Optimierung ohne Nebenbedingungen wird in dieser Arbeit ein Algorithmus eingesetzt, der in seiner ursprünglichen Form zuerst von Hansen [23] sowie in einer sehr ähnlichen Form von Moore [57] und Skelboe [74] angegeben wurde. Viele Autoren haben in den vergangenen Jahren zahlreiche effizienzsteigernde Modifikationen des ursprünglichen Algorithmus betrachtet, unter anderem verschiedene Varianten des erweiterten Intervall-Newton-Schritts. Im ersten Teil dieser Arbeit wird ein neuartiger Algorithmus angegeben, der den durch die notwendige Hessematrixauswertung aufwendigen Intervall-Newton-Schritt adaptiv gesteuert selektiv einsetzt. Dadurch wird der Intervall-Newton-Schritt nur dann ausgeführt, wenn durch ihn tatsächlich ein Fortschritt zu erwarten ist. Außerdem wird bei dem hier vorgestellten Algorithmus zur globalen Optimierung in bestimmten Phasen der Berechnung eine andere Aufteilung der aktuell zu untersuchenden Teilbox vorgenommen. Dies führt ebenfalls zu Effizienzsteigerungen im Vergleich zu bisher in der Literatur behandelten Algorithmen. Insgesamt liefert der Algorithmus gesicherte Einschließungen für das globale Minimum und alle globalen Minimalstellen. Zusätzlich wird anschließend versucht, die lokale Eindeutigkeit der Minimalstellen auf dem Rechner automatisch zu verifizieren. Der hier behandelte Algorithmus zur globalen Optimierung fällt also in die Klasse der sogenannten Verifikationsalgorithmen.

Parallelrechner gehören zur Klasse der Höchstleistungsrechner. Sie werden eingesetzt, um bislang aus Laufzeitgründen seriell unlösbare Probleme zu bewältigen. Bei einem Parallelrechner bearbeiten mehrere Prozessoren gemeinsam ein Problem. Um die durch einen Parallelrechner zur Verfügung gestellte Rechnerleistung effizient nutzen zu können, sind speziell angepaßte Programme notwendig. Auch für die Problemklasse der verifizierten globalen Optimierung sind Parallelrechner einsetzbar. Im zweiten Teil dieser Arbeit wird eine Adaption des oben skizzierten neuartigen Algorithmus zur globalen Optimierung für Parallelrechner vorgenommen. Der dabei entste-

hende parallele Algorithmus ist vollständig verteilt. Alle Prozessoren des Parallelrechners bearbeiten gleichberechtigt Teilprobleme und tauschen Informationen über den Fortschritt während des Berechnungsprozesses aus. Der Algorithmus ist somit gut skalierbar. Implementiert wurde der parallele Algorithmus in C–XSC, einer C++ Klassenbibliothek für wissenschaftliches Rechnen und MPI, der Standardbibliothek für *message passing* Programme in der Programmiersprache C. Konkret wurden in dieser Arbeit Testprobleme auf bis zu 96 Prozessoren gerechnet. Dabei konnte für viele Testprobleme eine hohe Beschleunigung erzielt werden.

Die vorliegende Arbeit ist wie folgt gegliedert: Im ersten Kapitel erfolgt eine kurze Einführung in die Grundlagen des wissenschaftlichen Rechnens, soweit sie in dieser Arbeit benötigt werden. Nach der Vorstellung der verwendeten Bezeichnungen sowie einiger grundlegender Definitionen und Sätze geben wir einen kurzen Überblick über die mathematisch fundierte Rechner- und Intervallarithmetik sowie die erweiterte Intervallarithmetik, die im Intervall-Newton-Schritt eingesetzt wird. Anschließend wird die automatische Differentiation vorgestellt, die die effiziente Berechnung von Einschließungen von Gradienten und Hessematrizen ermöglicht, ohne formal zu differenzieren oder die numerischen Schwierigkeiten herkömmlicher dividierter Differenzen in Kauf nehmen zu müssen.

Kapitel 2 gibt dann einen Überblick über die in der vorliegenden Arbeit verwendeten Grundlagen des parallelen Rechnens. Nach einer Klassifizierung von Parallelrechnerarchitekturen und der Bereitstellung der benötigten Definitionen und Begriffe zur Leistungsmessung und Beurteilung von parallelen Algorithmen wird der als Grundlage für den parallelen Algorithmus zur verifizierten globalen Optimierung dienende Lastverteilungsalgorithmus für parallele *Branch and Bound*-Algorithmen vorgestellt. Anschließend werden noch einige Details zur verwendeten Hard- und Softwareumgebung angegeben.

Aufbauend auf diesen Grundlagen wird in Kapitel 3 zunächst der neuartige serielle Algorithmus zur verifizierten globalen Optimierung hergeleitet. Dabei wird das behandelte Problem genauer spezifiziert sowie ein einfacher Grundalgorithmus zur Lösung angegeben. Anschließend werden zahlreiche Modifikationen und Erweiterungen zur Beschleunigung des Grundalgorithmus vorgestellt. Diese Erweiterungen werden in einem ersten Algorithmus, der aufgrund seiner Ähnlichkeit mit dem in [21] angegebenen Algorithmus als „*Toolbox*-ähnlich“ bezeichnet wird, zusammengeführt. Dieser *Toolbox*-

ähnliche Algorithmus dient als Vergleichsgrundlage für den neuartigen Algorithmus, der anschließend dargelegt wird. Eine große Anzahl an Testproblemen aus der Literatur zur globalen Optimierung wird dann für numerische Tests zum Vergleich verschiedener Varianten beider Algorithmen verwendet. Dabei wird in einer ausführlichen Studie die Laufzeit sowie der Aufwand für Funktions-, Gradienten- und Hessematrixauswertungen und der benötigte Speicherplatz miteinander verglichen. Hier zeigen sich deutlich die Effizienzvorteile des neuartigen Algorithmus.

Im zweiten Teil des dritten Kapitels wird dann der parallele Algorithmus zur verifizierten globalen Optimierung entwickelt. Dabei werden zunächst andere Ansätze aus der Literatur vorgestellt, danach der neue, vollständig verteilte parallele Algorithmus hergeleitet. Auch hier folgt ein Abschnitt über die Ergebnisse der numerischen Tests sowie Effizienzbetrachtungen.

Kapitel 4 gibt eine kurze Zusammenfassung der Ergebnisse dieser Arbeit sowie einen Ausblick über denkbare Erweiterungsmöglichkeiten und Ergänzungen.

Im Anhang findet sich eine detaillierte Auflistung der untersuchten Testprobleme. Zu jedem Testproblem wird die Zielfunktion, der Startbereich sowie die berechneten Ergebnisse angegeben. Außerdem enthält der Anhang eine wichtige Verbesserung der Hessematrixarithmetik aus der *Toolbox* sowie eine Auflistung aller Funktionen, die zur Verwendung von C–XSC und MPI auf Parallelrechnern notwendig sind.

Die Programme, die für die numerischen Tests verwendet wurden, finden sich im Internet unter:

`http://www.uni-karlsruhe.de/~Andreas.Wiethoff/globopt.html`

Kapitel 1

Allgemeine Grundlagen des wissenschaftlichen Rechnens

1.1 Bezeichnungen, grundlegende Definitionen und Sätze

In dieser Arbeit verwenden wir die folgenden Bezeichnungen:

$\mathbb{R}$	–	Menge der reellen Zahlen
$\mathbb{R}^n$	–	Menge der n-dimensionalen Vektoren über $\mathbb{R}$
$\mathbb{R}^{n\times m}$	–	Menge der $n \times m$-Matrizen über $\mathbb{R}$
$\mathbb{P}\mathbb{R}$	–	Potenzmenge über den reellen Zahlen
$\mathbb{P}\mathbb{R}^n$	–	Potenzmenge über den reellen Vektoren
$\mathbb{P}\mathbb{R}^{n\times m}$	–	Potenzmenge über den reellen Matrizen

Für einen Vektor $x \in \mathbb{R}^n$ bzw. eine Matrix $M \in \mathbb{R}^{n\times n}$ werden die Elemente durch *untere* Indizes und Folgen oder Aufzählungen von Vektoren bzw. Matrizen durch *obere* Indizes gekennzeichnet. Es bedeutet also:

$x_i \in \mathbb{R}$	–	i-tes Element des reellen Vektors x
$M_i \in \mathbb{R}^n$	–	i-te Zeile der reellen Matrix M
$M_{*j} \in \mathbb{R}^n$	–	j-te Spalte der reellen Matrix M
$M_{i,j} = M_{ij} \in \mathbb{R}$	–	j-tes Element der i-ten Zeile der Matrix M
$x^k \in \mathbb{R}^n$	–	k-ter Vektor in einer Aufzählung oder Folge
$M^k \in \mathbb{R}^{n \times n}$	–	k-te Matrix in einer Aufzählung oder Folge

Wir bezeichnen reelle Zahlen und reelle Vektoren mit Kleinbuchstaben, reelle Matrizen mit Großbuchstaben. Beim Übergang zu Intervallen und Intervallvektoren (siehe Abschnitt 1.4) verwenden wir Großbuchstaben auch für skalare und vektorielle Werte. Ebenso wie bei der Verwendung der 0 als reelle Null, als Nullvektor oder auch als Nullmatrix ist die Bedeutung jedoch durch den jeweiligen Zusammenhang ersichtlich. Mengen werden durch kalligraphische Symbole bezeichnet (z. B.: $\mathcal{M} \in I\!P\mathbb{R}$).

Für die inneren und äußeren Multiplikationen von reellen Zahlen, Vektoren und Matrizen verwenden wir stets das Symbol „$\cdot$“ (darf auch entfallen). Für das Skalarprodukt zweier Vektoren $a, b \in \mathbb{R}^n$ schreiben wir also

$$a \cdot b = ab = \sum_{i=1}^{n} a_i \cdot b_i.$$

Für eine reelle Funktion $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ mit $f \in C^2(D)$ bezeichnet

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \frac{\partial f}{\partial x_2}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{pmatrix}$$

den *Gradienten* von f und

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2^2}(x) & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \dots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{pmatrix}$$

die *Hessematrix* von f. Für $g : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ mit $g \in C^1(D)$ bezeichnet

$$J_g(x) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(x) & \frac{\partial g_1}{\partial x_2}(x) & \dots & \frac{\partial g_1}{\partial x_n}(x) \\ \frac{\partial g_2}{\partial x_1}(x) & \frac{\partial g_2}{\partial x_2}(x) & \dots & \frac{\partial g_2}{\partial x_n}(x) \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial g_n}{\partial x_1}(x) & \frac{\partial g_n}{\partial x_2}(x) & \dots & \frac{\partial g_n}{\partial x_n}(x) \end{pmatrix}$$

die *Jacobimatrix* von g.

Die Hessematrix ist symmetrisch (Satz von Schwarz). Es gilt ferner (mit $g(x) := \nabla f(x)$): $J_g(x) = \nabla^2 f(x)$. Für $r \in \mathbb{R}$, $x \in \mathbb{R}^n$ und $M \in \mathbb{R}^{n \times n}$ verwenden wir die folgenden Normen bzw. Bezeichnungen

$\lvert r \rvert$			– Betrag
$\lVert x \rVert$	$:=$	$\sqrt{\sum_{i=1}^{n} \lvert x_i \rvert^2}$	– euklidische Norm
$\lVert x \rVert_\infty$	$:=$	$\max_i \lvert x_i \rvert$	– Maximumnorm
$\lVert M \rVert$	$:=$	$\sqrt{\sum_{i,j=1}^{n} \lvert M_{ij} \rvert^2}$	– Schur-Norm
$\lVert M \rVert_\infty$	$:=$	$\max_i \sum_{j=1}^{n} \lvert M_{ij} \rvert$	– Zeilensummennorm
$\rho(M)$	$:=$	$\max\{\lvert\lambda\rvert \mid \lambda$ Eigenwert von $M\}$	– Spektralradius

1.2 Notation der Algorithmen

Zur Darstellung der Algorithmen wird in dieser Arbeit eine PASCAL-ähnliche Notation verwendet. Die einzelnen Schritte des Algorithmus werden durchnumeriert und jeweils durch ein Semikolon abgeschlossen. Jeder Schritt kann mehrere Anweisungen enthalten, die ebenfalls durch Semikolon getrennt sind. Folgende vier Schleifenkonstrukte werden verwendet:

for *Laufindex* := *Untergrenze* **to** *Obergrenze* **do**
Anweisung(en)

for all *Objektbeschreibung* **do**
Anweisung(en)

repeat
Anweisung(en)
until *Abbruchbedingung*

while *Durchlaufbedingung* **do**
Anweisung(en)

Innerhalb einer Schleife sind die Anweisungen

$\mathbf{exit}_{Laufindex\text{-loop}}$, $\mathbf{exit}_{while\text{-loop}}$ und $\mathbf{exit}_{repeat\text{-loop}}$

zum vorzeitigen Verlassen der jeweiligen Schleife und die Anweisung

$\mathbf{next}_{Laufindex}$

für den Abbruch des aktuellen und den Übergang zum nächsten Schleifendurchlauf zugelassen. Verzweigungen werden durch

if *Bedingung* **then**
Anweisung(en)
else
Anweisung(en)

realisiert, wobei der **else**-Teil entfallen kann. Der Aufruf von anderen Algorithmen ist ebenfalls möglich:

Algorithmusname (*Argumentliste*)

oder

Variable := Algorithmusname (*Argumentliste*)

entspricht einem Prozedur- bzw. Funktionsaufruf. Ein Algorithmus kann durch

return

an jeder Stelle verlassen werden; dabei werden die bereits berechneten Werte (auch für die aktuellen Argumente der Argumentliste des aufrufenden Algorithmus) beibehalten.

1.3 Rechnerarithmetik

Auf jedem Computer sind nur endlich viele Zahlen darstellbar. Deshalb muß die unendliche Menge der reellen Zahlen auf eine Teilmenge, die sogenannten *Gleitkommazahlen* oder auch *Maschinenzahlen*, abgebildet werden. Gleiches gilt für die Vektoren und Matrizen über den reellen Zahlen. Ferner müssen die arithmetischen Grundoperationen $+$, $-$, $\cdot$ und $/$ für die inneren und äußeren Verknüpfungen der reellen Räume auf dem Rechner durch die sogenannten *Gleitkommaoperationen* oder auch *Maschinenoperationen* approximiert werden. Kontrolliertes wissenschaftliches Rechnen auf einem Computer erfordert daher eine mathematisch fundierte Definition der Rechnerarithmetik, wie sie von Kulisch und Miranker in [44], [45] und [46] angegeben wird. In den folgenden Abschnitten wollen wir kurz die wichtigsten Definitionen und Begriffe aus der mathematisch begründeten Rechnerarithmetik erläutern.

Wir verwenden die folgenden Notationen:

R	–	Menge der Maschinenzahlen
R^n	–	Menge der n-dimensionalen Vektoren über R
$R^{n\times m}$	–	Menge der $n \times m$-Matrizen über R

Die reellen Räume werden gemäß

$$\begin{array}{lcl} \mathbb{R} & \rightarrow & R \\ \mathbb{R}^n & \rightarrow & R^n \\ \mathbb{R}^{n\times n} & \rightarrow & R^{n\times n} \end{array}$$

auf die Räume der Maschinenzahlen abgebildet. Im folgenden sei S ein Raum der linken Spalte und T der zugehörige Raum der rechten Spalte.

Definition 1.1 *Die Abbildung $\square : S \rightarrow T$ mit den Eigenschaften*

(R1) $\square a = a$ *für alle* $a \in T$, – Projektion

(R2) $a \leq b \Rightarrow \square a \leq \square b$ *für alle* $a, b \in S$, – Monotonie

heißt Rundung. *Eine Rundung heißt* antisymmetrisch, *falls gilt*

(R3) $\square(-a) = -\square a$ *für alle* $a \in S$.

Eine Rundung heißt nach unten gerichtet (nach oben gerichtet), *falls gilt*

(R4) $\square(a) \leq a$ $(\square(a) \geq a)$ *für alle* $a \in S$.

Die inneren und äußeren Verknüpfungen $+, -, \cdot$ und $/$ in S werden durch die zugehörigen Gleitkommaverknüpfungen $\boxplus$, $\boxminus$, $\boxdot$ und $\boxslash$ in T approximiert, die über das nachfolgend erläuterte Prinzip des Semimorphismus definiert sind.

Definition 1.2 *Eine antisymmetrische Rundung* $\square : S \to T$ *heißt* Semimorphismus, *wenn alle inneren und äußeren Verknüpfungen in* T *durch*

(RG) $a \boxtimes b := \square(a \circ b)$ *für alle* $a, b \in T$ *und* $\circ \in \{+, -, \cdot, /\}$

erklärt sind.

Die so definierten Verknüpfungen für Elemente aus T werden zunächst in S ausgeführt und das Ergebnis anschließend nach T gerundet. Semimorphe Verknüpfungen sind daher von *maximaler Genauigkeit*: Zwischen dem in S berechneten Verknüpfungsergebnis $a \circ b$ und seiner Approximation $a \boxtimes b$ in T liegt kein weiteres Element aus T (siehe [44] und [45]). Für die Produkträume ist dies komponentenweise zu verstehen.

Beispiel 1.1 Das Skalarprodukt zweier Gleitkommavektoren $a, b \in R^n$ wird über den Semimorphismus definiert durch

$$a \boxdot b := \square(a \cdot b) = \square\left(\sum_{i=1}^{n} a_i \cdot b_i\right).$$

Bemerkung: Im folgenden verwenden wir das Zeichen $\square$ für die antisymmetrische Rundung zur nächstgelegenen Maschinenzahl. Die nach unten

bzw. nach oben gerichteten Rundungen zur nächstkleineren bzw. nächstgrößeren Maschinenzahl bezeichnen wir mit $\bigtriangledown$ bzw. $\bigtriangleup$. Für sie gilt die Identität

$$\bigtriangledown(-x) = -\bigtriangleup x \quad \text{für alle } x \in S.$$

Die Auswertung einer aus den gerundeten Operationen $\boxplus$, $\boxminus$, $\boxdot$ und $\boxslash$ zusammengesetzten Funktion f bezeichnen wir mit $f_\square(x)$. Der Begriff „maximale Genauigkeit“ wird auch bei der Auswertung $s_\square$ von Funktionen $s : I\!R \to I\!R$ auf dem Rechner verwendet, wobei

$$s_\square(x) := \square(s(x))$$

über den Semimorphismus definiert ist. Den auf dem Rechner zur Verfügung stehenden Satz maximal genauer Funktionen (Standardfunktionen) $s : R \to R$ bezeichnen wir im folgenden mit

$$\mathcal{SF} = \{\exp, \ln, \sin, \cos, \ldots\}.$$

1.4 Intervallarithmetik

Bei der Implementierung eines numerischen Verfahrens auf einem Computer besteht das Problem, die während der Rechnung auftretenden Rundungsfehler zu kontrollieren. Rundungsfehler können daher rühren, daß die auf dem Computer verwendeten Maschinenzahlen nur einen endlichen Teilbereich der reellen Zahlen darstellen. Rechnet man jedoch mit Intervallen statt wie üblich mit reellen Zahlen, so kann man Rundungsfehler direkt bei der Rechnung berücksichtigen und einschließen. Durch die Intervallrechnung kann man daher mathematisch exakte Aussagen auf dem Rechner beweisen.

Eine ausführliche Definition und Darstellung der Intervallrechnung findet sich in [1], [2] oder (als erstes) in [57], eine gelungene Einführung z. B. in [54]. Die zugehörige Maschinenintervallarithmetik wird etwa in [44] und [45] behandelt. Im folgenden geben wir eine Zusammenstellung der wichtigsten Begriffe und Eigenschaften.

Definition 1.3 *Die Menge* $A := [\underline{A}, \overline{A}] := \{x \in I\!R \mid \underline{A} \leq x \leq \overline{A}\}$ *mit* $\underline{A}, \overline{A} \in I\!R$ *heißt* Intervall. $\underline{A} = \inf A$ *heißt* Infimum *oder* Unterschranke *von*

A, $\overline{A} = \sup A$ *heißt* Supremum *oder* Oberschranke *von* A. *Ein Intervall* A *heißt* Punktintervall, *wenn gilt* $\underline{A} = \overline{A}$.

Wir verwenden die folgenden Bezeichnungen:

$I\!I\!R$	–	Menge der Intervalle
$I\!I\!R^n$	–	Menge der n-dimensionalen Vektoren über $I\!I\!R$
$I\!I\!R^{n\times m}$	–	Menge der $n \times m$-Matrizen über $I\!I\!R$

Elemente aus $I\!I\!R^n$ werden auch als n-dimensionale *Box* oder auch als n-dimensionaler *Quader* bezeichnet.

Aus Abschnitt 1.1 übernehmen wir die Notation zur Indizierung von Intervallen, Intervallvektoren und Intervallmatrizen. Für $A \in I\!I\!R$, $X \in I\!I\!R^n$ und $M \in I\!I\!R^{n\times n}$ verwenden wir ferner die folgenden Bezeichnungen:

$$
\begin{array}{lcll}
m(A) & := & \frac{1}{2}(\underline{A} + \overline{A}) & \text{– Mittelpunkt} \\
d(A) & := & \overline{A} - \underline{A} & \text{– (absoluter) Durchmesser} \\
d(X) & := & \max\limits_{1\le i\le n} d(X_i) & \text{– (absoluter) Durchmesser} \\
|A| & := & \max\{|a| \mid a \in A\} & \text{– Betrag} \\
\langle A\rangle & := & \min\{|a| \mid a \in A\} & \text{– Betragsminimum} \\
\|X\|_\infty & := & \max\limits_i |X_i| & \text{– Maximumnorm} \\
\|M\|_\infty & := & \max\limits_i \sum\limits_{j=1}^{n} |M_{ij}| & \text{– Zeilensummennorm}
\end{array}
$$

Mittelpunkt und Betrag von Vektoren und Matrizen sind dabei komponentenweise zu verstehen. Für $A, B \in I\!I\!R$ gilt:

$$
\begin{aligned}
d(A \pm B) &= d(A) + d(B) \\
d(A \cdot B) &\le d(A)\cdot|B| + d(B)\cdot|A| \\
|A + B| &\le |A| + |B| \\
|A \cdot B| &= |A|\cdot|B|.
\end{aligned}
$$

Weitere Einzelheiten hierzu sowie die Beweise für die vorstehenden Beziehungen sind etwa in [1] zu finden.

Neben dem oben eingeführten absoluten Durchmesser eines Intervalls A bzw. eines Intervallvektors X benötigen wir noch den *relativen Durchmesser*:

$$d_{\mathrm{rel}}(A) := \begin{cases} d(A)/\langle A\rangle & ,\ 0 \notin A \\ d(A) & ,\ 0 \in A \end{cases} \quad \text{– relativer Durchmesser}$$

$$d_{\mathrm{rel}}(X) := \max_i\ d_{\mathrm{rel}}(X_i) \quad \text{– relativer Durchmesser}$$

Als weitere Schreibweisen für $A \in I\!I\!R$ und $X \in I\!I\!R^n$ verwenden wir:

$$\begin{array}{lcll} \mathring{A} & := & \{a \in A \mid \underline{A} < a < \overline{A}\} & \text{– Inneres von } A \\ \mathring{X} & := & \{x \in X \mid \underline{X_i} < x_i < \overline{X_i} \ \text{ für alle } i\} & \text{– Inneres von } X \\ \partial A & := & \{\underline{A}, \overline{A}\} & \text{– Rand von } A \\ \partial X & := & \{x \in X \mid x_i \in \partial X_i \ \text{ für ein } i\} & \text{– Rand von } X \end{array}$$

Teilmengenbeziehungen und Vergleiche sind mengentheoretisch aufzufassen und sind für Vektoren und Matrizen genau dann wahr, wenn sie für alle Komponenten wahr sind. Für $A, B \in I\!I\!R$ gilt daher:

$$\begin{array}{lcl} A = B & \Longleftrightarrow & \underline{A} = \underline{B} \ \wedge\ \overline{A} = \overline{B}, \\ A \subseteq B & \Longleftrightarrow & \underline{A} \geq \underline{B} \ \wedge\ \overline{A} \leq \overline{B}, \\ A \mathring{\subset} B & :\Longleftrightarrow & \underline{A} > \underline{B} \ \wedge\ \overline{A} < \overline{B} \ \Longleftrightarrow\ A \subseteq \mathring{B}\ . \end{array}$$

Die Verbandsoperationen $\cap$ und $\underline{\cup}$ für $A, B \in I\!I\!R$ sind wie folgt definiert:

$$\begin{array}{lcll} A \cap B & := & [\max\{\underline{A}, \underline{B}\}, \min\{\overline{A}, \overline{B}\}], & \text{– Schnitt} \\ A \mathbin{\underline{\cup}} B & := & [\min\{\underline{A}, \underline{B}\}, \max\{\overline{A}, \overline{B}\}], & \text{– Intervallhülle} \end{array}$$

wobei $A \cap B$ nur definiert ist, falls $\max\{\underline{A}, \underline{B}\} \leq \min\{\overline{A}, \overline{B}\}$ gilt. Die intervallarithmetischen Operationen werden definiert durch

$$A \circ B := \{a \circ b \mid a \in A,\, b \in B\},$$

wobei $A, B \in I\!I\!R$ und $\circ \in \{+, -, \cdot, /\}$. Die Intervalloperationen können wie

folgt explizit berechnet werden:

$$\begin{aligned}
A+B &= [\underline{A}+\underline{B},\ \overline{A}+\overline{B}],\\
A-B &= [\underline{A}-\overline{B},\ \overline{A}-\underline{B}],\\
A\cdot B &= [\min\{\underline{A}\,\underline{B},\underline{A}\,\overline{B},\overline{A}\,\underline{B},\overline{A}\,\overline{B}\},\ \max\{\underline{A}\,\underline{B},\underline{A}\,\overline{B},\overline{A}\,\underline{B},\overline{A}\,\overline{B}\}],\\
A\,/\,B &= [\underline{A},\overline{A}]\cdot[1/\overline{B},\ 1/\underline{B}]\ \text{ für } 0\notin B
\end{aligned}$$

Bei der Multiplikation kann in fast allen Fällen die explizite Berechnung des Minimums bzw. des Maximums durch vorherige Betrachtung der Intervallgrenzen von A und B vermieden werden. Für die Division ($0 \notin B$) lässt sich folgende Fallunterscheidung vornehmen (der Fall $0 \in B$ wird im Rahmen der in Abschnitt 1.5 beschriebenen erweiterten Intervallarithmetik behandelt):

$$A/B = \begin{cases}
[\overline{A}/\underline{B},\underline{A}/\overline{B}] & \text{falls } \overline{A}\le 0\ \wedge\ \overline{B}<0\\
[\underline{A}/\underline{B},\overline{A}/\overline{B}] & \text{falls } \overline{A}\le 0\ \wedge\ 0<\underline{B}\\
[\overline{A}/\overline{B},\underline{A}/\overline{B}] & \text{falls } \underline{A}<0<\overline{A}\ \wedge\ \overline{B}<0\\
[\underline{A}/\underline{B},\overline{A}/\underline{B}] & \text{falls } \underline{A}<0<\overline{A}\ \wedge\ 0<\underline{B}\\
[\overline{A}/\overline{B},\underline{A}/\underline{B}] & \text{falls } 0\le\underline{A}\ \wedge\ \overline{B}<0\\
[\underline{A}/\overline{B},\overline{A}/\underline{B}] & \text{falls } 0\le\underline{A}\ \wedge\ 0<\underline{B}
\end{cases}$$

Addition und Multiplikation sind kommutativ und assoziativ, es gilt jedoch nur die sogenannte *Subdistributivität*

$$A\cdot(B+C)\subseteq A\cdot B+A\cdot C$$

für Intervalle $A,B,C\in I\mathbb{R}$. Eine weitere wichtige Eigenschaft der Intervalloperationen ist die *Inklusionsisotonie*

$$\begin{aligned}
a\in A\ \wedge\ b\in B &\implies a\circ b\in A\circ B\\
A\subseteq C\ \wedge\ B\subseteq D &\implies A\circ B\subseteq C\circ D
\end{aligned}$$

für alle $\circ\in\{+,-,\cdot,/\}$ mit $a,b\in\mathbb{R}$ und $A,B,C,D\in I\mathbb{R}$.

Mit Hilfe der Intervallarithmetik ist es möglich, Einschließungen für den Wertebereich einer reellen Funktion zu berechnen. Dabei bezeichnet für Funktionen $f:D\to\mathbb{R}$ mit $D\subseteq\mathbb{R}$ oder $D\subseteq\mathbb{R}^n$:

$f(X)$	–	*Wertebereich* von f
$F(X)$	–	*Intervallauswertung*: Auswertung der *Intervallerweiterung* (s.u.) oder auch *Einschließungsfunktion* F von f
$\overline{F_X}$	–	$\sup F(X)$
$\underline{F_X}$	–	$\inf F(X)$

wobei für $f(X)$ und $F(X)$ stets die Beziehung $f(X) \subseteq F(X)$ gilt. Die *Intervallerweiterung* (oft auch als natürliche Intervallerweiterung bezeichnet) einer Funktion f erhält man, indem alle auftretenden Variablen durch entsprechende Intervalle und alle Operationen durch die zugehörigen Intervalloperationen ersetzt werden. In Teilausdrücken können dabei auch sogenannte Standardintervallfunktionen $s \in \mathcal{SF}$ vorkommen. Für diese gilt: $s(X) = S(X)$. Für Punktintervalle bzw. Punktintervallvektoren als Argumente von f gilt stets $f(c) = F(c)$.

Die Inklusionsisotonie gilt ebenfalls für die Intervallerweiterungen:

$$\begin{aligned} a \in A &\implies f(a) \in F(A) \\ A \subseteq B &\implies F(A) \subseteq F(B). \end{aligned}$$

Die Intervallauswertung einer Funktion f in Form der natürlichen Intervallerweiterung (auch *natürliche Intervallauswertung* genannt) führt oft zu großen Überschätzungen. Daher werden häufig auch *zentrierte Formen* oder auch *Mittelwertformen* verwendet. Details hierzu finden sich in [64]. An dieser Stelle wird nur eine kurze Übersicht gegeben:

Die Mittelwertform wird aus dem Mittelwertsatz abgeleitet und ist für $f : D \subseteq \mathbb{IR} \to \mathbb{IR}$ durch

$$F^c(X) = F(c) + F'(X) \cdot (X - c)$$

und für $f : D \subseteq \mathbb{IR}^n \to \mathbb{IR}$ durch

$$F^c(X) = F(c) + \boldsymbol{\nabla} F(X) \cdot (X - c) \tag{1.1}$$

definiert, wobei $c \in X$ ist. Häufig wird $c = m(X)$ verwendet.

Noch engere Einschließungen lassen sich mit *Taylorformen* gewinnen. Wir benötigen später die Taylorform zweiter Ordnung für $f : D \subseteq \mathbb{IR}^n \to \mathbb{IR}$:

$$F^t(X) = F(c) + \boldsymbol{\nabla} F(c)(X - c) + \frac{1}{2}(X - c)^T \boldsymbol{\nabla}^2 F(X)(X - c) \tag{1.2}$$

In jedem Fall gilt: $f(X) \subset F^c(X),\ f(X) \subset F^t(X)$.

Wir benötigen später noch eine Aussage über die Güte von Intervallauswertungen. Der folgende Satz besagt, daß der Durchmesser der Intervallauswertungen $F(X)$ gegen Null geht, sofern der Durchmesser des Argumentintervalls X ebenfalls gegen Null geht. Es ist also zu erwarten, daß für kleine Intervalle X die Intervallauswertung $F(X)$ den Wertebereich $f(X)$ nicht sehr stark überschätzt.

Satz 1.1 *Sei $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ eine reellwertige Funktion und $Y \in I\mathbb{R}^n$ mit $Y \subseteq D$. Die Intervallauswertung F genüge in jeder Variablen $x_k \in Y_k$, $1 \leq k \leq n$ einer Lipschitzbedingung für beliebige Wahl von $x_j \in Y_j$, $1 \leq j \leq n, j \neq k$. Dann existiert ein $c \geq 0$, sodaß für alle Intervalle $X \subseteq Y$ gilt:*

$$d(F(X)) \leq c \cdot d(X).$$

Beweis: Siehe [1], S. 40.

Bemerkung: Die im Satz geforderte Lipschitzbedingung ist in der Regel bei Funktionen erfüllt, die sich aus intervallmäßig ausführbaren Operationen zusammensetzen.

Beim Rechnen mit Intervallen auf Computern muß die Menge der reellen Intervalle auf die Menge der *Maschinenintervalle* abgebildet werden. Dazu werden die Intervallgrenzen durch (nach außen) gerichtete Rundungen auf Maschinenzahlen abgebildet. Man beachte jedoch, daß ein Intervall auch nach diesem Übergang auf ein Maschinenintervall

$$A := [\underline{A}, \overline{A}] := \{x \in \mathbb{R} \mid \underline{A} \leq x \leq \overline{A};\ \underline{A}, \overline{A} \in R\}$$

sämtliche *reelle* Werte zwischen den beiden Grenzen $\underline{A}$ und $\overline{A}$ repräsentiert, also nach wie vor das gesamte Kontinuum abdeckt. Gleiches gilt für die Vektoren und Matrizen über den reellen Intervallen. Um auf dem Computer auch Berechnungen durchführen zu können, müssen ferner die exakten arithmetischen Grundoperationen $+$, $-$, $\cdot$ und $/$ für die inneren und äußeren Verknüpfungen der reellen Intervallräume auf dem Rechner durch die sogenannten *Maschinenintervalloperationen* approximiert werden.

Wir verwenden die folgenden Bezeichnungen:

IR	–	Menge der Maschinenintervalle
IR^n	–	Menge der n-dimensionalen Vektoren über IR
$IR^{n\times m}$	–	Menge der $n \times m$-Matrizen über IR

Die reellen Intervallräume werden gemäß

$$\begin{array}{lcl} I\!R & \to & IR \\ I\!R^n & \to & IR^n \\ I\!R^{n\times n} & \to & IR^{n\times n} \end{array}$$

auf die zugehörigen Maschinenräume abgebildet. Im folgenden sei IS ein Raum der linken Spalte und IT der zugehörige Raum der rechten Spalte.

Definition 1.4 *Die Abbildung $\diamondsuit : IS \to IT$ mit den Eigenschaften*

(R1) $\diamondsuit A = A$ *für alle* $A \in IT$,

(R2) $A \subseteq B \;\Rightarrow\; \diamondsuit A \subseteq \diamondsuit B$ *für alle* $A, B \in IS$,

(R3) $\diamondsuit(-A) = -\diamondsuit A$ *für alle* $A \in IS$,

(R4) $A \subseteq \diamondsuit A$ *für alle* $A \in IS$,

heißt antisymmetrische, nach außen gerichtete Rundung *oder auch* Intervallrundung.

Bemerkung: Die Rundung $\diamondsuit$ ist eindeutig bestimmt (siehe [44], [45]).

Alle inneren und äußeren Verknüpfungen $+$, $-$, $\cdot$ und $/$ in IS werden durch die zugehörigen Maschinenintervallverknüpfungen $\diamondsuit\!\!+$, $\diamondsuit\!\!-$, $\diamondsuit\!\!\cdot$ und $\diamondsuit\!\!/$ in IT approximiert, die über das Prinzip des Semimorphismus durch

$$\text{(RG)} \quad A \circledcirc B := \diamondsuit(A \circ B) \quad \text{für alle } A, B \in IT \text{ und } \circ \in \{+, -, \cdot, /\}$$

definiert sind. Mit $F_\diamondsuit(X)$ bezeichnen wir die Maschinenintervallauswertung einer Intervallfunktion F unter Verwendung der gerundeten Operationen $\diamondsuit$ $+$, $\diamondsuit\!\!-$, $\diamondsuit\!\!\cdot$ und $\diamondsuit\!\!/$. Maschinenintervallfunktionen $S_\diamondsuit$ der Intervallfunktionen $S : I\!R \to I\!R$ mit $S \in \mathcal{SF}$ (Standardfunktionen) sind über den Semimorphismus definiert durch

$$S_\diamondsuit(X) := \diamondsuit(S(X)).$$

Die Inklusionsisotonie gilt ebenfalls für die Maschinenenoperationen in der Form

$$\begin{aligned} a \in A \ \wedge \ b \in B \quad &\Longrightarrow \quad a \circ b \in A \circ B \subseteq A \diamondsuit B \\ A \subseteq C \ \wedge \ B \subseteq D \quad &\Longrightarrow \quad A \diamondsuit B \subseteq C \diamondsuit D \end{aligned}$$

für alle $\circ \in \{+, -, \cdot, /\}$ mit $a, b \in R$ und $A, B, C, D \in I\,R$ und für die Maschinenintervallauswertungen in der Form

$$\begin{aligned} a \in A \quad &\Longrightarrow \quad f(a) \in F(A) \subseteq F_\diamond(A) \\ A \subseteq B \quad &\Longrightarrow \quad F_\diamond(A) \subseteq F_\diamond(B). \end{aligned}$$

Mittelpunkt und Durchmesser werden auf dem Rechner als

$$m_\square(X) = \square(m(X)) \quad \text{und} \quad d_\triangle(X) = \triangle(d(X))$$

definiert, wobei zu beachten ist, daß m und d damit zwar maximal genau sind, daß jedoch im allgemeinen gilt

$$m_\square(X) \neq m(X) \quad \text{bzw.} \quad d_\triangle(X) \neq d(X).$$

Bei Algorithmen mit automatischer Ergebnisverifikation, wie sie in späteren Kapiteln vorkommen, ist es gelegentlich sinnvoll, das zu untersuchende Intervall $X \in I\mathbb{R}$ etwas zu vergrößern, um den Ablauf des Verfahrens zu beschleunigen. Dabei wird durch die sogenannte ϵ-Aufblähung entweder eine relative (bezogen auf den Durchmesser von X) oder eine geringfügige absolute Vergrößerung vorgenommen, letzteres, um Punktintervalle in Intervalle mit einem von Null verschiedenen Durchmesser zu überführen. Sei *MinReal* die kleinste positive Gleitkommazahl, die auf dem Rechner darstellbar ist. Dann wird die ϵ-Aufblähung wie folgt definiert:

Algorithmus 1.1: EpsAufb(X, ϵ)
ϵ-Aufblähung

1. **if** $(d(X) > 0)$ **then return** $X + [-\epsilon, \epsilon] \cdot d(X)$;
   else return $X + [-MinReal, MinReal]$;

Für n-dimensionale Boxen $X \in I\mathbb{R}^n$ bzw. Matrizen $X \in I\mathbb{R}^{m \times n}$ wird die ϵ-Aufblähung komponentenweise durchgeführt.

1.5 Erweiterte Intervallarithmetik

Im Intervall-Newton-Schritt (vgl. dazu den Abschnitt 3.2.6.3) treten Intervallausdrücke der Form $r - A/B$ ($r \in \mathbb{R},\ A, B \in I\mathbb{R}$) auf, wobei möglicherweise $0 \in B$ gilt. Um diesen Fall korrekt zu behandeln, muß eine sogenannte *erweiterte Intervallarithmetik* verwendet werden. In der Literatur finden sich verschiedene Ansätze für eine solche Arithmetik (siehe etwa [24], [36] oder [65]). Wie in [70] gezeigt wird, sind jedoch einige der in der Literatur angegebenen Definitionen fehlerhaft oder unvollständig: In einigen Fällen wird die Eigenschaft der Inklusionsisotonie verletzt, in anderen Fällen die Lösungsmenge nicht richtig berechnet. Unsere Darstellung folgt daher im wesentlichen der in [70] angegebenen (vgl. auch [38]).

Im folgenden erweitern wir den Raum der reellen Intervalle. Wir lassen zu, daß die Grenze eines Intervalls auch einer der beiden *idealen Punkte* $-\infty$ und $+\infty$ sein kann. Für die Menge der *erweiterten reellen Intervalle* verwenden wir die folgenden Bezeichnungen:

$$I\mathbb{R}^\star := I\mathbb{R} \cup \{[-\infty, r] \mid r \in \mathbb{R}\} \cup \{[r, +\infty] \mid r \in \mathbb{R}\} \cup \{[-\infty, +\infty]\}$$
$$IR^\star := IR \cup \{[-\infty, r] \mid r \in R\} \cup \{[r, +\infty] \mid r \in R\} \cup \{[-\infty, +\infty]\}$$

$I\mathbb{R}^\star$ ist daher die Menge der reellen Intervalle erweitert um die unbeschränkten Intervalle. Beispielsweise ist $[-\infty, r]$ eine andere Schreibweise für die Menge $\{x \in \mathbb{R} \mid x \leq r\}$. $[-\infty, +\infty]$ bezeichnet also die gesamte reelle Achse.

Definition 1.5 *Ein Paar* $W = (W^1 \mid W^2) := W^1 \cup W^2$ *von Intervallen* $W^1, W^2 \in I\mathbb{R}^\star \cup \{\emptyset\}$ *heißt* erweitertes Intervallpaar, *wenn entweder gilt* $\sup W^1 < \inf W^2$ *oder* $W^2 = \emptyset$.

Der Durchmesser *eines erweiterten Intervallpaares* $W = (W^1 \mid W^2)$ *mit endlichen Komponenten* $W^1, W^2 \in I\mathbb{R} \cup \{\emptyset\}$ *ist definiert durch*

$$d(W) = d(W^1) + d(W^2), \quad \text{mit } d(\emptyset) = 0.$$

Seien $A, B \in I\mathbb{R}$. Dann definieren wir die *erweiterte Intervalldivision* für

den Intervallquotienten $W = A/B$ für $A, B \in I\!I\!R$ durch:

$$W := \begin{cases} A \cdot [1/\overline{B}, 1/\underline{B}] & \text{falls } 0 \notin B \\ [-\infty, +\infty] & \text{falls } 0 \in A \ \wedge \ 0 \in B \\ [\overline{A}/\underline{B}, +\infty] & \text{falls } \overline{A} < 0 \ \wedge \ \underline{B} < \overline{B} = 0 \\ [-\infty, \overline{A}/\overline{B}] \cup [\overline{A}/\underline{B}, +\infty] & \text{falls } \overline{A} < 0 \ \wedge \ \underline{B} < 0 < \overline{B} \\ [-\infty, \overline{A}/\overline{B}] & \text{falls } \overline{A} < 0 \ \wedge \ 0 = \underline{B} < \overline{B} \\ [-\infty, \underline{A}/\underline{B}] & \text{falls } 0 < \underline{A} \ \wedge \ \underline{B} < \overline{B} = 0 \\ [-\infty, \underline{A}/\underline{B}] \cup [\underline{A}/\overline{B}, +\infty] & \text{falls } 0 < \underline{A} \ \wedge \ \underline{B} < 0 < \overline{B} \\ [\underline{A}/\overline{B}, +\infty] & \text{falls } 0 < \underline{A} \ \wedge \ 0 = \underline{B} < \overline{B} \\ \emptyset & \text{falls } 0 \notin A \ \wedge \ 0 = B \end{cases} \tag{1.3}$$

Den Fall, daß bei der erweiterten Intervalldivision das Ergebnis W eine Vereinigung von zwei echten Intervallen W^1 und W^2 ist (also $W = W^1 \cup W^2$ mit $W^2 \neq \emptyset$), bezeichnen wir auch als *Splitting*.

Ferner definieren wir die *erweiterte Intervallsubtraktion* für die Differenz $U = r - W$ einer reellen Zahl r und eines erweiterten Intervallpaares W, das aus einer erweiterten Intervalldivision $W = A/B, \ A, B \in I\!I\!R$ entsteht, durch:

$$U = \begin{cases} [r - \overline{W}, r - \underline{W}] & \text{falls } W \in I\!I\!R \\ [-\infty, +\infty] & \text{falls } W = [-\infty, +\infty] \\ [-\infty, r - \underline{W}] & \text{falls } W = [\underline{W}, +\infty] \\ [r - \overline{W}, +\infty] & \text{falls } W = [-\infty, \overline{W}] \\ [-\infty, r - \underline{W^2}] \ \cup & \text{falls } W = (W^1 \mid W^2) \\ [r - \overline{W^1}, +\infty] & \quad = [-\infty, \overline{W^1}] \cup [\underline{W^2}, +\infty] \end{cases} \tag{1.4}$$

Satz 1.2 *Die gemäß (1.3) definierte erweiterte Intervalldivision und die gemäß (1.4) definierte erweiterte Intervallsubtraktion sind inklusionsisoton.*

Beweis: Siehe [70].

In den Anwendungen tritt ferner noch die Schnittbildung eines erweiterten Intervallpaares mit einem Standard-Intervall auf. Für $W = (W^1 \mid$

$W^2)$, $W^1, W^2 \in I\!I\!R^\star \cup \{\emptyset\}$ und $X \in I\!I\!R$ definieren wir:

$$W \cap X := (W^1 \cap X \mid W^2 \cap X).$$

Auch die erweiterten Intervalloperationen werden beim Übergang zu den entsprechenden Maschinenoperationen $\diamondsuit$ und $\circledcirc$ über den Semimorphismus definiert:

$$\begin{aligned} A \diamondsuit B &= \diamondsuit(A/B), \\ r \circledcirc W &= \diamondsuit(r - W), \end{aligned}$$

wobei $A, B \in IR$, $r \in R$ und $W = (W^1 \mid W^2)$ ein erweitertes Intervallpaar mit $W^1, W^2 \in I\!R^\star$ ist. Die Rundung für erweiterte Intervallpaare $W = (W^1 \mid W^2)$, $W^1, W^2 \in I\!I\!R^\star$ definieren wir durch

$$\diamondsuit(W) = \begin{cases} [-\infty, +\infty] & \text{falls } W = [-\infty, +\infty] \\ [\bigtriangledown(\underline{W}), +\infty] & \text{falls } W = [\underline{W}, +\infty] \\ [-\infty, \bigtriangleup(\overline{W})] & \text{falls } W = [-\infty, \overline{W}] \\ [-\infty, \bigtriangleup(\overline{W^1})] \cup [\bigtriangledown(\underline{W^2}), +\infty] & \text{falls } W = [-\infty, \overline{W^1}] \cup [\underline{W^2}, +\infty] \\ [\bigtriangledown(\underline{W}), \bigtriangleup(\overline{W})] & \text{falls } W = [\underline{W}, \overline{W}] \end{cases}$$

1.6 Automatische Differentiation

Im Rahmen dieser Arbeit werden an verschieden Stellen (etwa bei der Funktionsauswertung durch die Mittelwertform oder im Intervall-Newton-Verfahren) die Auswertung des Gradienten oder der Hessematrix einer Funktion $f : I\!R^n \to I\!R$ bzw. deren Einschließungen benötigt. Um Ableitungen einer Funktion zu berechnen, können die folgenden drei Methoden angewandt werden:

- Numerische Differentiation
- Symbolische Differentiation
- Automatische Differentiation

Bei der numerischen Differentiation werden die Werte der Ableitung durch Differenzenquotienten approximiert. Einschließungen von Ableitungen können jedoch nicht berechnet werden. Da außerdem die Ergebnisse bei der numerischen Differentiation häufig ungenauer als die der automatischen Differentiation sind, ist die numerische Differentiation für das vorliegende Problem nicht geeignet.

Im Gegensatz zur numerischen Differentiation kommen sowohl die symbolische als auch die weiter unten genauer erläuterte automatische Differentiation ohne Approximation aus. Die symbolische Differentiation wendet Differentiationsregeln auf den gegebenen Funktionsausdruck an und ermittelt einen neuen Funktionsausdruck für die Ableitung. Dies kann auch automatisch durchgeführt werden und ist in Computeralgebrasystemen wie Maple oder Mathematica realisiert. In den neu bestimmten Funktionsausdruck werden dann die zu untersuchenden Stellen eingesetzt und so Werte für die Ableitung an bestimmten Stellen berechnet. Anstelle eines Punktes $x \in \mathbb{R}^n$ kann auch ein Intervall $A \in I\mathbb{R}^n$ (eine n-dimensionale Box) in den Funktionsausdruck für die Ableitung eingesetzt werden. Man erhält damit die natürliche Intervallauswertung für die Ableitung. Häufig sind jedoch die durch die symbolische Differentiation ermittelten Funktionsausdrücke für die Ableitung sehr umfangreich und komplex, so daß diese nicht mehr effizient ausgewertet werden kann.

Zur Berechnung von Ableitungswerten wird im Rahmen dieser Arbeit die automatische Differentiation verwendet. Dabei setzt man die Argumentwerte während der Anwendung der Differentiationsregeln ein. Es wird so das Anwachsen der Formeln, wie es bei der symbolischen Differentiation auftritt, verhindert. Statt komplexer symbolischer Formeln müssen nur Zahlen (bzw. Zahlentupel) manipuliert werden und die Berechnung der Ableitungen erfolgt parallel mit der Berechnung des Funktionswertes.

Bei der automatischen Differentiation unterscheidet man zwischen der *Vorwärtsmethode*, die in dieser Arbeit eingesetzt und in folgenden skizziert wird, und der *Rückwärtsmethode*, die beispielsweise in [14] oder [18] dargestellt ist. Letztere ist zwar effizienter als die Vorwärtsmethode, hat jedoch wesentlich höhere Speicheranforderungen und ist aufwendiger zu implementieren. Bei der Vorwärtsmethode wird eine spezielle Differentiationsarithmetik verwendet, die mit Paaren bzw. Tripeln der Form

$$\begin{aligned} (u, u') &\quad - \quad \text{Gradientenarithmetik} \\ (u, u', u'') &\quad - \quad \text{Hessematrixarithmetik} \end{aligned}$$

mit $u \in I\!I\!R, u' \in I\!I\!R^n, u'' \in I\!I\!R^{n\times n}$ operiert. Dabei repräsentieren u, u' und u'' den Funktionswert $f(x)$, den Gradienten $\nabla f(x)$ und die Hessematrix $\nabla^2 f(x)$ einer gegebenen Funktion $f : I\!I\!R^n \to I\!I\!R$ an einer festen Stelle $x \in I\!I\!R^n$. Man erhält deshalb für die konstante Funktion $f(x) = c, c \in I\!I\!R$ und für die Funktion $f(x) = x_i, 1 \le i \le n$ (e^i bezeichne den i-ten Einheitsvektor):

$$\begin{array}{lll} f(x) = c: & (u,u') = (c,0) & (u,u',u'') = (c,0,0) \\ f(x) = x_i: & (u,u') = (x_i,e^i) & (u,u',u'') = (x_i,e^i,0) \end{array}$$

Nachstehend werden nur die Regeln für die Hessematrixarithmetik angegeben. Die Gradientenarithmetik ergibt sich einfach durch Weglassen der dritten Komponente der Tripel. Verknüpft werde dabei $W = U \circ V$, genauer

$$(w,w',w'') = (u,u',u'') \circ (v,v',v''),\ \circ \in \{+,-,\cdot,/\}.$$

Für die vier Grundrechenarten gelten die nachfolgend aufgeführten Rechenvorschriften, die sich aus den bekannten Differentiationsregeln ergeben:

$$\begin{aligned} (u,u',u'') + (v,v',v'') &= (u+v, u'+v', u''+v'') \\ (u,u',u'') - (v,v',v'') &= (u-v, u'-v', u''-v'') \\ (u,u',u'') \ \cdot\ (v,v',v'') &= (u\cdot v, u\cdot v' + u'\cdot v, \\ &\qquad u\cdot v'' + u'\cdot (v')^T + v'\cdot (u')^T + u''\cdot v) \\ (u,u',u'') \ /\ (v,v',v'') &= (u/v, (u' - u/v\cdot v')/v, \\ &\qquad (u'' - w'\cdot (v')^T - v'\cdot (w')^T - w\cdot v'')/v),\ v \neq 0 \end{aligned}$$

Für Standardfunktionen $s : I\!I\!R \to I\!I\!R$ gilt:

$$s((u,u',u'')) = (s(u), s'(u)\cdot u', s'(u)\cdot u'' + s''(u)\cdot u'\cdot (u')^T).$$

Mit Hilfe dieser Regeln kann nun für jede Funktion, die aus den vier Grundrechenarten $+, -, \cdot, /$ sowie Standardfunktionen zusammengesetzt ist, neben dem Funktionswert an einer Stelle $x \in I\!I\!R$ auch der Gradient und die Hessematrix an dieser Stelle berechnet werden.

Der Einsatz der automatische Differentiation gestaltet sich dann besonders einfach, wenn in der verwendeten Programmiersprache Operatoren und Funktionen überladen werden können, wie dies etwa in PASCAL–XSC, C++ oder Fortran 90 der Fall ist (zu PASCAL–XSC siehe [40]). Natürlicherweise ist daher auch die in dieser Arbeit verwendeten Programmierumgebung C–XSC, eine C++ Klassenbibliothek für wissenschaftliches Rechnen,

geeignet. C–XSC ist in [41] beschrieben. Die Implementierung der automatischen Differentiation lehnt sich an der in der C++ Toolbox [21] angegebenen Implementierung an. Es wurden jedoch einige wichtige Verbesserungen vorgenommen, die in Anhang B angegeben sind.

1.7 Verwaltung geordneter Listen

Im Algorithmus für die globale Optimierung müssen geordnete Listen verwaltet werden. Als Elemente treten dabei Paare $E = (X, r)$ bestehend aus einer n-dimensionalen Box $X \in I\!I\!R^n$ und einer reellen Zahl $r \in I\!R$ auf. Dabei gilt (F sei Intervallauswertung einer reellwertigen Funktion f):

$$r = \underline{F_X} = \underline{F(X)} = \inf F(X).$$

Um eine Liste manipulieren zu können, definieren wir für eine Liste L und ein Element E die folgenden Operationen:

$L := \{\}$	L wird auf die leere Liste initialisiert.
$L := E$	L wird auf eine einelementige Liste mit dem Element E initialisiert.
$L := L + E$	Das Element E wird entsprechend der gewählten Ordnung in die Liste L eingefügt.
$L := L - E$	E wird aus der Liste L entfernt.
$E :=$ Head(L)	Das vorderste Element in der Liste L wird an E zugewiesen, aber nicht aus der Liste entfernt.
Length(L)	Gibt die Länge der Liste L zurück.
forall $E \in L$ **do**	Bearbeite nacheinander alle Elemente E der Liste L.

Für die Ordnung der Listen sind verschiedene Varianten möglich: *best-first* (Sortierung nach aufsteigendem $\underline{F_X}$), *oldest-first* (Sortierung nach Alter der Boxen) usw. Je nach der gewählten Ordnung hat die Operation + eine unterschiedliche Bedeutung und ist auch unterschiedlich aufwendig.

Kapitel 2

Grundlagen des parallelen Rechnens

Die Rechenleistung von herkömmlichen (seriellen) Computern konnte in den vergangenen Jahren immer wieder erheblich gesteigert werden. Das *Moore-sche* Gesetz (nach Gordon Moore, einem der Gründer der Mikroprozessorfirma Intel), nachdem sich etwa alle 18 Monate die Geschwindigkeit der jeweils neuesten Mikroprozessoren (CPUs) verdoppelt, hat nach wie vor Bestand. Fast noch schneller als die Leistung der Mikroprozessoren wächst jedoch der Bedarf nach mehr Rechenleistung und mehr Speicherplatz. Um die gestiegenen Anforderungen bewältigen und bislang unlösbare Probleme angehen zu können, werden sogenannte Höchstleistungsrechner eingesetzt. Im allgemeinen werden zwei Arten von Höchstleistungsrechnern unterschieden: *Vektorrechner* und *Parallelrechner* (siehe z.B. [30, 61]). Auf Vektorrechnern wird eine Erhöhung der Rechenleistung durch den Einsatz von *Pipelining* im Arithmetikteil der CPU erzielt. Unter Pipelining versteht man die Zerlegung arithmetischer Grundoperationen in Teiloperationen wie Operanden holen, Exponentenvergleich, Mantissenverschiebung, Normalisierung usw. und deren parallele Ausführung auf verschiedenen Komponenten des Arithmetikteils der CPU für viele gleichartige Daten (z.B. bei der Addition der Elemente zweier Vektoren). Für viele Probleme lassen sich auf Vektorrechnern mit geeigneten Werkzeugen (vektorisierende Compiler) große Geschwindigkeitsgewinne erzielen, jedoch können bei der Vektorisierung durch die unter Umständen geänderte Abarbeitungsreihenfolge der Operanden ungenaue oder gänzlich falsche Ergebnisse berechnet werden (vgl. [19]). Die

in der vorliegenden Arbeit behandelte Problemstellung aus der Numerik (Globale Optimierung) eignet sich von ihrer Struktur her nicht so sehr für Vektorrechner, da im allgemeinen viele nichtlineare Funktionen ausgewertet werden müssen. Hingegen erzielen Vektorrechner vorwiegend dann eine hohe Effizienz, wenn viele lineare Ausdrücke (Vektor- oder Matrixoperationen) auftreten.

Im folgenden soll daher auf Parallelrechner und deren Einsatz zur effizienten Bearbeitung der o.g. Problemstellungen eingegangen werden. Dieses Kapitel enthält grundlegende Bemerkungen zu existierenden Parallelrechnerarchitekturen, zu Leistungsmessungen auf Parallelrechnern, zur Parallelisierung von seriellen Algorithmen im allgemeinen und von *Branch and Bound*-Algorithmen im besonderen sowie einige technische Details zum Parallelrechner IBM RS/6000 SP, der bei der Berechnung der numerischen Ergebnisse in dieser Arbeit zum Einsatz gekommen ist.

2.1 Klassifizierung von Parallelrechnerarchitekturen

Von einem Parallelrechner spricht man, wenn auf einem Rechner mehr als ein Prozessor (p Prozessoren) zur Bearbeitung seiner Aufgaben eingesetzt wird. Dabei ist für die Anzahl p der Prozessoren ein breites Spektrum von wenigen ($2 \leq p \leq 10$) bis zu sehr vielen (massiv parallele Systeme; $p > 1000$) möglich. Wird eine Aufgabenstellung auf einem Parallelrechner mit p Prozessoren bearbeitet, so erhofft sicher der Anwender eine Reduzierung der Gesamtrechenzeit auf $1/p$ der Rechenzeit auf einem Prozessor. Diese Beschleunigung (*Speedup*) wird jedoch nicht in allen Fällen erreicht werden können, da die Verwaltung der p Prozessoren sowie die Kommunikation zwischen den Prozessoren ebenfalls Zeit benötigt. Weitere Ausführungen hierzu folgen im nächsten Abschnitt 2.2.

Parallelrechner werden nach unterschiedlichen Kriterien klassifiziert:

2.1.1 Befehls- und Operandenstrom

Die Klassifizierung nach Befehls- und Operandenstrom wurde von Flynn [15] eingeführt. Es werden folgende Klassen von Parallelrechnern unterschieden:

SISD (*single instruction stream / single data stream*): Mit dieser Klassifizierung ist in Wirklichkeit kein Parallelrechner gemeint, sondern dies ist der herkömmliche serielle von-Neumann-Computer. Rechner in dieser Klasse verfügen über einen Instruktionsstrom (und daher in der Regel auch nur über eine CPU). Jede arithmetische Instruktion zieht eine arithmetische Operation nach sich. Dies führt zu einem einzigen Datenstrom von Eingabeargumenten und Ausgaberesultaten. Es ist im übrigen unbedeutend, ob Pipelining zur Beschleunigung der Abarbeitung arithmetischer Instruktionen eingesetzt wird (vgl. Vektorrechner).

Beispiele für SISD-Rechner sind PCs oder Workstations (natürlich nur die mit einer CPU). Der Intel Pentium Prozessor, der in den meisten heutigen PCs zum Einsatz kommt, verfügt genaugenommen jedoch über zwei Pipelines für Integerarithmetik und eine Pipeline für Gleitkommaarithmetik, die unter bestimmten Bedingungen auch gleichzeitig zum Einsatz kommen. Dies wird automatisch ohne Zutun des Anwenders durch die Befehlsdekodiereinheit der CPU veranlaßt. Streng genommen gehören also PCs mit Pentium Prozessor bereits zur Klasse der MIMD-Rechner. Es ist jedoch immer sichergestellt, daß sich das Verhalten des Programms gegenüber einer seriellen Abarbeitung nicht ändert. Ähnliches gilt auch für die RISC CPUs, die in vielen Workstations zum Einsatz kommen. „Echte“ SISD-Rechner sind daher heutzutage nur noch selten anzutreffen.

SIMD (*single instruction stream / multiple data stream*): Alle Prozessoren des Parallelrechners interpretieren dieselben Befehle eines einzigen Programms und führen sie auf unterschiedlichen Eingabedaten aus. Auf einem SIMD-Parallelrechner mit $p = m \cdot n$ Prozessoren können beispielsweise zwei $m \times n$ Matrizen mit nur einer Gleitkommaaddition pro Prozessor addiert werden. Die Gesamtrechenzeit für die Addition zweier Matrizen entspricht also der Dauer einer Gleitkommaoperation. SIMD-Parallelrechner werden aufgrund ihres einfacheren Aufbaus häufig mit sehr vielen Prozessoren ausgestattet, fallen also oft in die Kategorie der massiv parallelen Systeme. Beispiele für SIMD-Rechner sind die Connection Machine CM2 und der MasPar.

MISD (*multiple instruction stream / single data stream*): Diese Klasse von Parallelrechnern ist leer, da impliziert wird, daß mehrere verschiedene Anweisungen *gleichzeitig* auf nur einem Operanden ausgeführt werden.

MIMD (*multiple instruction stream / multiple data stream*): Alle Prozessoren des Parallelrechners führen ihn eigenes Programm aus und arbeiten daher unabhängig von einander auf verschiedenen Daten. Häufig werden Standardprozessoren (aus PCs oder Workstations) für die einzelnen Knoten des Parallelrechners eingesetzt, um die neuesten Mikroprozessoren verwenden zu können. Der Hauptentwicklungsaufwand für den Parallelrechner steckt in der Verschaltung der einzelnen Knoten untereinander. Das Verhältnis von Kommunikations- zu Rechenleistung muß ausgewogen sein, ansonsten sind keine guten Werte für die Beschleunigung durch die Parallelisierung zu erwarten. Beispiele für MIMD-Rechner sind die Intel Paragon, die Connection Machine CM5 sowie die in dieser Arbeit eingesetzte IBM RS/6000 SP.

2.1.2 Speicherkonzept

Parallelrechner werden außerdem nach dem verwendeten Speicherkonzept klassifiziert:

gemeinsamer Speicher (*shared memory*): Die Prozessoren des Parallelrechners teilen sich einen zentralen gemeinsamen Speicher. Jeder Prozessor kann auf jeden Teil des Speicher beliebig zugreifen. Zum Datenaustausch erfolgt die Kommunikation zwischen den Prozessoren über den gemeinsamen Speicher. Dabei können jedoch auch Zugriffskonflikte auftreten, insbesondere bei einer größeren Zahl von Prozessoren. So kommt es etwa zu Verzögerungen, wenn zwei verschiedene Prozessoren gleichzeitig in eine bestimmte Speicherzelle schreiben wollen. Beispiele für Parallelrechner mit gemeinsamen Speicher sind die Rechner der Firma Sequent (neuere Rechner dieser Firma verwenden verteilte Segmente mit gemeinsamen Speicher).

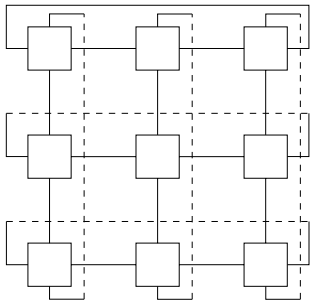
verteilter Speicher (*distributed memory*): Viele heutige Parallelrechner verwenden verteilten Speicher. Jeder Knoten des Parallelrechners verfügt über eigenen lokalen Speicher, auf den die anderen Prozessoren nicht direkt zugreifen können. Der Datenaustausch zwischen den Prozessoren erfolgt durch *message-passing* Routinen, also spezielle Kommunikationsroutinen. Zur Realisierung der Kommunikation kommen dabei Kommunikationskanäle (1:1-Verbindungen) oder Bussysteme zum Einsatz. Bei Parallelrechnern mit einer größeren Anzahl von Prozessoren ist es bei der Verwendung von Kommunikationskanälen

nicht möglich, jeden Knoten des Rechners mit jedem anderen Knoten zu verbinden. Parallelrechner auf Transputerbasis verfügen beispielsweise nur über vier Kommunikationskanäle pro Prozessor. Es werden daher verschiedene Vernetzungen der Prozessoren vorgenommen. Einige mögliche Netzwerke sind im folgenden Abschnitt über die Vernetzungsmöglichkeiten angegeben. Auch kann die Qualität der Anbindung der einzelnen Knoten untereinander (und damit die Kommunikationsgeschwindigkeit) bei verschiedenen Parallelrechnern variieren: Mit geeigneten Softwareprotokollen (etwa PVM [17] oder MPI [75]) kann ein Parallelrechner durch ein Cluster aus gewöhnlichen Workstations mit (relativ zur Prozessorgeschwindigkeit) langsamer Vernetzung durch Ethernet aufgebaut werden. Einige der heutigen Parallelrechner (so etwa die IBM RS/6000 SP, siehe Abschnitt 2.5.1) unterscheiden sich auf der Hardwareseite von einem Workstationpool prinzipiell nur durch die wesentlich schnellere Verbindung der einzelnen Knoten untereinander. Die verwendeten Kommunikationsprotokolle können weiterhin PVM oder MPI sein. Dadurch wird eine hohe Portabilität der parallelen Programme erreicht.

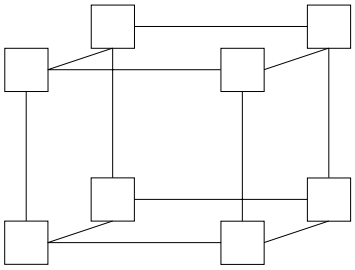
2.1.3 Vernetzung

Die einzelnen Prozessoren eines Parallelrechners können zu unterschiedlichen Netzwerken zusammengeschaltet werden. Einige Anwendungen aus der Numerik lassen sich natürlicherweise auf bestimmte Standardnetzwerke abbilden. Abbildung 2.1 zeigt einige solcher Vernetzungsmöglichkeiten.

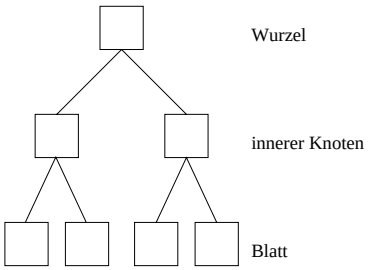
Im Abschnitt 2.4.3 benötigen wir noch spezielle *DeBruijn*-Netzwerke. Ein DeBruijn-Netzwerk der Dimension k besteht aus 2^k Knoten. Ein Knoten $x = 2^{x_1 \cdot x_2 \cdot \ldots \cdot x_k}$ ist genau dann mit einem Knoten $y = 2^{y_1 \cdot y_2 \cdot \ldots \cdot y_k}$ mit $x_i, y_i \in \{0, 1\}$ verbunden, wenn gilt: $(x_2, \ldots, x_k) = (y_1, \ldots, y_{k-1})$. DeBruijn-Netzwerke haben logarithmischen Durchmesser sowie maximal vier Verbindungen pro Prozessor, sind also gut zur Implementierung auf einem Transputersystem geeignet.



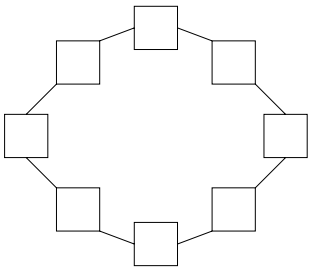
Gitter



Hypercube (hier 3d)



Baum (binaer)



Ring

Abbildung 2.1: Standardnetzwerke bei Parallelrechnern

2.2 Leistungsmessungen und Beurteilung von parallelen Algorithmen

Der Einsatz eines Parallelrechners zur Lösung eines numerischen Problems ist nur dann sinnvoll, wenn sich die Gesamtzeit zur Lösung des Problems gegenüber einer Rechnung auf einem seriellen Rechner signifikant reduzieren läßt. Ideal wäre es natürlich, wenn bei einem Parallelrechner mit p Prozessoren die Rechenzeit um den Faktor p (also auf $1/p$ der Rechenzeit auf einem seriellen Rechner) sinken würde. Ein Maß für die erzielte Beschleunigung durch den Einsatz des parallelen Algorithmus ist der sogenannte *Speedup*, der wie folgt definiert ist:

Definition 2.1 *Die* Beschleunigung (Speedup) $S(p)$ *bezeichnet das Verhältnis der Ausführungszeit eines seriellen Algorithmus* t_{ser} *zu der Ausführungszeit des zugehörigen parallelen Algorithmus* $t_{\text{par}}(p)$ *auf* p *Prozessoren:*

$$S(p) := \frac{t_{\text{ser}}}{t_{\text{par}}(p)}.$$

Wird die erzielte Beschleunigung in Relation zur eingesetzten Anzahl an Prozessoren gesetzt, so ergibt sich ein Maß für die Effizienz des Algorithmus:

Definition 2.2 *Die* Effizienz $E(p)$ *definiert die durchschnittliche Auslastung pro Prozessor:*

$$E(p) := \frac{S(p)}{p}.$$

Bemerkung:

1. Bei der Berechnung der Beschleunigung sollte t_{ser} durch den besten bekannten seriellen Algorithmus berechnet werden. Häufig ist *der* beste serielle Algorithmus jedoch nicht eindeutig bestimmt oder der beste bekannte Algorithmus ist noch nicht implementiert worden, so daß Zeitmessungen unmöglich sind. Daher werden bei den Zeitmessungen in dieser Arbeit weitgehend identische Programme für den seriellen und parallelen Algorithmus verwendet, wobei bei Zeitmessungen für den seriellen Algorithmus die parallelen Zusätze so entfernt werden, daß kein meßbarer Zusatzaufwand entsteht.

2. Im allgemeinen gilt: $1 \leq S(p) \leq p$. In Einzelfällen kann jedoch auch eine überlineare Beschleunigung (sog. *superlinearer Speedup*) mit $S(p) > p$ erzielt werden. Das Auftreten von überlinearer Beschleunigung deutet auf algorithmische Unzulänglichkeiten des seriellen Algorithmus hin, da in diesem Fall der serielle Algorithmus insgesamt mehr Teilprobleme untersucht als der parallele Algorithmus. Dies kann durch einen verbesserten seriellen Algorithmus vermieden werden.

3. Die genaue Charakterisierung eines effizienten parallelen Algorithmus ist jedoch nicht ganz leicht. Dies läßt sich durch das *Amdahlsche Gesetz* begründen: Bei konstanter Problemgröße ist jeder parallele Algorithmus asymptotisch ineffizient, da der serielle Anteil des Problems die erreichbare Beschleunigung nach oben begrenzt. Daher muß bei starker Erhöhung der Prozessorzahl auch fairerweise die Problemgröße nach oben skaliert werden.

2.3 Zur Parallelisierung von seriellen Algorithmen

Die Parallelisierung eines seriellen Algorithmus kann grundsätzlich mehrere Ziele haben: Das Hauptziel wird in der Regel sein, ein gegebenes Problem schneller zu lösen. Die Beschleunigung soll groß sein, wenn möglich sogar linear zur Anzahl an Prozessoren. Unter Umständen läßt sich durch Verteilung auf p Prozessoren jedoch auch ein anderes Ziel erreichen: Es können größere Probleme als auf einem seriellen Rechner gelöst werden, da auf dem Parallelrechner insgesamt mehr Speicher zur Verfügung stehen kann. Bei der Parallelisierung ist jedoch in jedem Fall darauf zu achten, daß das Verhältnis von Berechnung zu Kommunikation möglichst groß ist und daß möglichst alle Prozessoren ständig mit sinnvollen Berechnungen beschäftigt sind. Dazu muß das vorliegende Problem auf die p Prozessoren des Parallelrechners aufgeteilt werden (Lastverteilung).

Besonders einfach ist dies, wenn der Aufwand zur Lösung *a priori* abgeschätzt werden kann. In diesem Fall kann das Problem in p Teilprobleme, die zu ihrer Lösung den gleichen Aufwand erfordern, aufgeteilt werden. Diese Teilprobleme werden getrennt von einander gelöst und die berechneten Ergebnisse am Schluß eingesammelt. Bei einem eventuell nötigen Datenaustausch muß darauf geachtet werden, daß möglichst kein Prozessor durch die

Kommunikation in seiner Berechnung blockiert wird. In diesem Fall kann die Lastverteilung bereits vor Programmausführung vorgenommen werden, man spricht deshalb von *statischer Lastverteilung (static load balancing)*.

Häufig kann jedoch erst im Laufe der Berechnung festgestellt werden, wieviel Aufwand zur Lösung eines Problems notwendig ist. Daher müssen während der Programmausführung Teilprobleme zwischen verschiedenen Prozessoren so umverteilt werden, daß die Last der Prozessoren in etwa gleich ist, insbesondere sollten alle Prozessoren ständig an der Berechnung und keine Prozessoren „leerlaufen“ oder warten. Um dies zu erreichen, muß eine *dynamische Lastverteilung (dynamic load balancing)* vorgenommen werden.

Um von einem vorliegenden seriellen Algorithmus zu einem effizienten parallelen Algorithmus zum Einsatz auf einem Parallelrechner zu gelangen, können grundsätzlich zwei verschiedene Wege beschritten werden.

2.3.1 Implizite Parallelisierung

Die Parallelisierung wird auf der Ebene der arithmetischen Grundoperationen vorgenommen. Der serielle Algorithmus wird unverändert übernommen; die einzelnen arithmetischen Operationen werden parallel ausgeführt. Es lohnt sich auf heutigen Parallelrechnern sicher nicht, skalare arithmetische Operationen zu parallelisieren. Vielmehr werden Vektor- und Matrixoperationen parallel ausgeführt. Die Parallelisierung der Vektor- und Matrixoperationen müssen für jeden Parallelrechner neu vorgenommen werden. Für die Parallelisierung des optimalen Skalarprodukts (exaktes Skalarprodukt mit nur einer Rundung am Ende) sind einige Ideen und Implementierungshinweise in [11] und [6] aufgeführt. Eine spezielle Anpassung für Transputer findet sich etwa in [7].

Der Vorteil dieser Art der Parallelisierung liegt auf der Hand: Bestehende Anwendungsprogramme können praktisch unverändert übernommen werden. Nachteil ist jedoch, daß nur derjenige Anteil des seriellen Programms, der Vektor- und Matrixoperationen verwendet, durch die Parallelisierung beschleunigt wird. Verwendet ein serielles Programm beispielsweise 50% der Gesamtrechenzeit auf Vektor- und Matrixoperationen, so ist durch die Parallelisierung maximal eine Beschleunigung um den Faktor 2 zu erzielen. Daher eignet sich die implizite Parallelisierung hauptsächlich für Anwendungen mit einem sehr hohen Anteil an Vektor- und Matrixoperationen, beispielsweise iterative Löser für lineare Gleichungssysteme.

Eine weitere Variante der impliziten Parallelisierung ist die Verwendung einer parallelisierenden Programmiersprache bzw. Compilers, etwa HPF (High Performance Fortran). Solche Programmiersprachen erweitern bekannte Programmiersprachen um Konstrukte für die Parallelisierung (etwa FORALL Anweisung). Serielle Anwendungen müssen angepaßt werden, jedoch kann in der Regel der grundlegende Algorithmus beibehalten werden. Die effiziente Abbildung auf den konkreten Parallelrechner bleibt dem jeweiligen Compiler überlassen. Man spricht hier auch von *feinkörniger Parallelisierung (fine grain parallelism)*. Eine hohe Effizienz ist jedoch in der Regel nicht einfach erzielbar. Eine Begründung dafür ist der Ursprung vieler parallelisierender Compiler in vektorisierenden Compilern für Vektorrechner. Vektorisierende Compiler transformieren in der Regel die innerste FOR Schleife. Dies ist für die Parallelisierung nicht so gut geeignet, da zwischen zwei Synchronisationspunkten der Rechenaufwand nicht groß genug ist, um eine hohe Beschleunigung zu erzielen.

2.3.2 Explizite Parallelisierung

Heutige Parallelrechner verwenden häufig leistungsstarke Workstations als Knotenrechner, die durch spezielle Hardware sehr schnell miteinander vernetzt werden, so etwa der Parallelrechner IBM RS/6000 SP (basierend auf RS/6000 Knoten) oder Cray T3E (basierend auf DEC Alpha Knoten). Zum Einsatz kommt dabei oft eine mittlere Prozessorzahl ($10 < p < 100$). Für diese Art von Parallelrechnern ist das Programmiermodell der expliziten Parallelisierung besonders geeignet. Dabei wird durch den speziell für den Parallelrechner entwickelten Algorithmus festgelegt, auf welchen Prozessoren des Parallelrechners welche Anweisungen ablaufen und wie die Kommunikation zwischen den Prozessoren erfolgt. Zur Abwicklung der Kommunikation werden Kommunikationsbibliotheken eingesetzt. Teilweise sind diese herstellerspezifisch (so etwa das INMOS C Toolset für Transputersysteme [32]), inzwischen sind jedoch standardisierte Kommunikationsbibliotheken wie PVM [17] oder MPI [75] stark verbreitet und stehen für Parallelrechner in effizienten Implementierungen mit Schnittstellen nach C und Fortran zur Verfügung. Dadurch wird eine hohe Portabilität von parallelen Programmen erreicht.

Die Effizienz eines explizit parallelen Algorithmus muß durch den Algorithmus selbst sichergestellt werden. Für viele Problemstellungen lassen sich Algorithmen angeben, bei denen eine hohe Effizienz erzielt wird. In der Regel

folgen bei expliziten parallelen Algorithmen längeren Berechnungsphasen Datenaustausch- und damit Lastausgleichsphasen. Man spricht deshalb von *grobkörniger Parallelisierung (coarse grain parallelism)*.

Der in dieser Arbeit in Kapitel 3 betrachtete Algorithmus zur verifizierenden globalen Optimierung basiert auf einem *Branch and Bound*-Verfahren, welches sich gut explizit parallelisieren läßt. Zur Parallelisierung von *Branch and Bound*-Algorithmen und die dafür notwendigen Lastverteilungsverfahren werden im folgenden Abschnitt 2.4 einige Ausführungen gemacht.

2.4 Lastverteilung für parallele *Branch and Bound*-Algorithmen

Das in dieser Arbeit untersuchte Problem der verifizierten globalen Optimierung ist von der Struktur des Algorithmus her in die Klasse der kombinatorischen Optimierungsprobleme einzuordnen. Andere Probleme, die in diesen Bereich fallen, sind das Problem des Handlungsreisenden (*Traveling Salesman Problem*), das Rucksackproblem oder das Problem, die Knoten eines Graphen mit einer minimalen Anzahl an Farben so einzufärben, daß Knoten, die durch eine Kante miteinander verbunden sind, unterschiedlich eingefärbt sind (*Vertex Coloring Problem*). Bei dieser Klasse von Problemen geht es darum, einen Lösungsvektor zu finden, der bestimmte Nebenbedingungen erfüllt und der eine gegebene Funktion minimiert.

Der *brute force*-Ansatz, bei dem die optimale Lösung durch vollständiges Durchlaufen des Suchraums gefunden wird, scheidet für ernsthafte Problemstellungen aus, da der Lösungsraum exponentiell mit der Dimension der Eingabedaten wächst, es also rasch zu inakzeptablen Berechnungszeiten kommt. Eine Parallelisierung, die lineare Beschleunigung erreicht, ist zwar einfach, große Probleme lassen sich jedoch nach wie vor nicht lösen.

Um effiziente Lösungen für kombinatorischen Optimierungsprobleme zu finden, werden häufig *Branch and Bound*-Algorithmen eingesetzt. Im *Branching*-Schritt des Algorithmus wird der Suchraum dabei (u. U. rekursiv) in kleinere Teilprobleme zerlegt, wodurch ein Suchbaum entsteht. Durch zusätzliche Informationen über die Lösung (z. B. gesicherte Schranken), die zusammen mit den jeweiligen Teilproblemen abgespeichert werden, können diejenigen Teile des Suchbaums eliminiert werden, die die Lösung sicher nicht enthalten (*Bounding*, vgl. Abbildung 2.2). Für die Eliminierung von

Teilen des Suchbaums wird eine globale Schranke als Kriterium verwendet. Eine Übersicht über verschiedene Varianten von *Branch and Bound*-Algorithmen ist etwa in [48] zu finden.

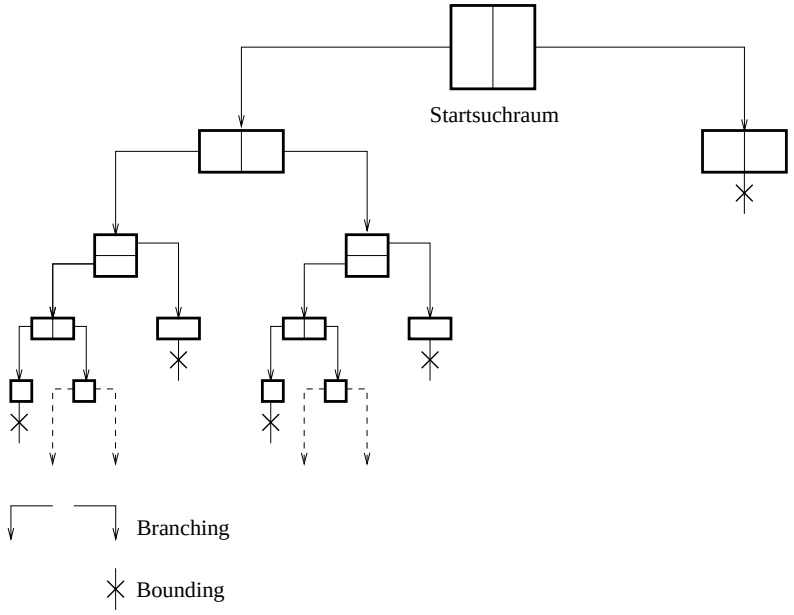


Abbildung 2.2: Suchbaum bei *Branch and Bound*-Algorithmen

Durch *Branch and Bound* kann der zu durchlaufende Suchraum häufig sehr stark reduziert werden. Am prinzipiell exponentiellen Wachstum des Suchraums (abhängig von der Dimension der Eingabedaten) ändert sich jedoch nichts.

2.4.1 Suchverfahren bei *Branch and Bound*-Algorithmen

Der Suchbaum kann bei *Branch and Bound*-Algorithmen auf verschiedene Arten durchlaufen werden:

Tiefensuche (*depth-first branch and bound*): Bei der Tiefensuche wird immer das zuletzt erzeugte Teilproblem als nächstes behandelt. Ein Zweig des Suchbaums wird abgeschnitten, sobald die Schranke der

aktuellen Teilbox schlechter als die globale beste Schranke ist. Die Bearbeitung der noch zu untersuchenden Teilprobleme erfolgt nach dem Prinzip *last-in-first-out*. Die geeignete Datenstruktur zur Verwaltung ist also ein Stapel (*stack*). Vorteil der Tiefensuche ist, daß der Speicherbedarf für den Stapel der Teilprobleme meist geringer ist als für die Besten- oder Ältestensuche. Nachteil ist jedoch, daß bei der Tiefensuche unter Umständen zu viele Teilprobleme behandelt werden, da die Verbesserung der globalen besten Schranke nicht so rasch wie bei der Bestensuche erfolgt.

Bestensuche (*best-first branch and bound*): Bei der Bestensuche wird von den vorhandenen Teilproblemen dasjenige mit der besten Schranke als nächstes bearbeitet. Damit sind immer die aussichtsreichsten Probleme in Bearbeitung, da Teilprobleme mit guter Schranke die Lösung mit höherer Wahrscheinlichkeit enthalten. Zur Verwaltung der Teilprobleme kommt eine sortierte Liste zum Einsatz. Diese Datenstruktur ist aufwendiger zu verwalten als ein Stapel oder eine Warteschlange. Ferner ist die maximale Listenlänge während der Bearbeitung in der Regel größer als bei den beiden anderen Suchstrategien, es wird also mehr Speichplatz zur Abspeicherung der Liste benötigt. Diese beiden Nachteile werden jedoch durch den folgenden Vorteil wieder wettgemacht: Bei der Bestensuche werden meist insgesamt weniger Teilprobleme bearbeitet als bei der Tiefen- oder Ältestensuche, da schneller eine gute Schranke gefunden wird und somit häufiger größere Teile des Suchbaums abgeschnitten werden können.

Ältestensuche (*oldest-first branch and bound*): Sortiert man die vorhandenen Teilprobleme in einer Warteschlange (*queue*), so werden diejenigen Teilprobleme als nächstes behandelt, die als erstes in die Warteschlange gekommen sind (Prinzip des *first-in-first-out*). Die so erhaltene Suchstrategie bezeichnet man als Ältestensuche. Aus dem *Branching*-Schritt neu erhaltene Teilboxen werden an das Ende der Warteschlange eingefügt. Dadurch erreicht man insgesamt, daß alle Boxen im Suchbaum gleichmäßig aufgeteilt werden. Auch hier können Teilprobleme überflüssigerweise bearbeitet werden.

2.4.2 Verwaltung der Teilprobleme bei der Parallelisierung

Grundsätzlich bieten sich *Branch and Bound*-Algorithmen dadurch zur Parallelisierung an, daß von den jeweils aktuell vorhandenen Teilproblemen mehrere gleichzeitig bearbeitet werden. Auf Parallelrechnern mit gemeinsamen Speicher gestaltet sich die Parallelisierung einfach, da für jeden Prozessor alle Teilprobleme jederzeit zugreifbar sind. In [51] wird die Simulation von gemeinsamen Speicher auf einem Transputersystem mit verteiltem Speicher beschrieben. Es zeigt sich jedoch, daß diese Technik nur für eine kleine Prozessorzahl sinnvoll ist und im allgemeinen zu Engpässen bei der Kommunikation führt. Nur wenn die Berechnungszeit des *Branching*-Schritts wesentlich größer als die Kommunikationszeit ist, können Engpässe vermieden werden. Dieses trifft aber nicht in jedem Fall zu.

Für Parallelrechner mit verteiltem Speicher ist die Situation komplizierter. Daher wird im allgemeinen eine andere Vorgehensweise gewählt: Die Teilprobleme werden auf die verschiedenen Prozessoren verteilt und jeder Prozessor wendet den auszuführenden Berechnungsteil des Algorithmus auf seine Teilprobleme an. *A priori* ist es bei *Branch and Bound*-Algorithmen schwierig bzw. unmöglich, Aussagen über die entstehende Last während der Berechnung zu treffen, da sich die Last der einzelnen Prozessoren im Laufe der Berechnung stark ändern kann. Für eine effiziente Parallelisierung ist daher eine dynamische Lastverteilung notwendig. Ferner muß eine neugefundene beste Schranke, die für den *Bounding*-Schritt des Algorithmus benötigt wird, möglichst rasch allen anderen Prozessoren bekannt gemacht werden, um zu verhindern, daß Teile des Suchbaums überflüssigerweise durchlaufen werden.

Für die Verwaltung der Teilboxen bei der Parallelisierung sind daher die folgenden Strategien gebräuchlich:

Zentralisierte Verwaltung: Diese Methode wird auch oft als *master-slave*-Verwaltungsmethode bezeichnet. Ein zentraler Prozessor (der *master*) speichert alle Teilprobleme sowie die beste globale Schranke. Alle anderen Prozessoren bearbeiten als Slaves die Teilprobleme, die sie vom Master zur Bearbeitung erhalten. Die Slaves senden nach der Berechnung ihre Ergebnisse (verbesserte Teilprobleme und eventuell verbesserte globale Schranke) zurück an den Master, der daraufhin die Slaves mit neuen Teilproblemen versorgt. Eine von einem

Slave verbesserte globale Schranke wird ebenfalls allen anderen Slaves zur Verfügung gestellt. Das Prinzip dieses Algorithmus ist in Abbildung 2.3 dargestellt.

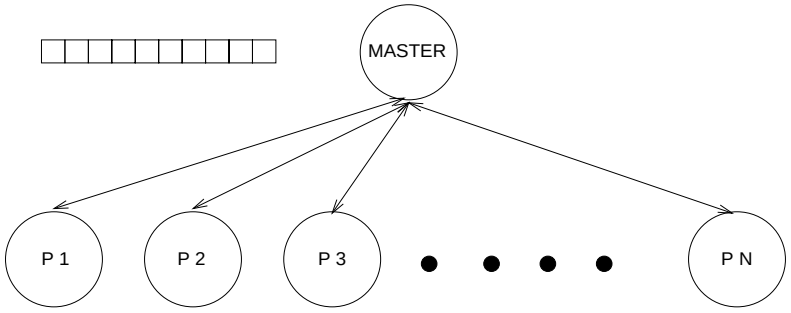


Abbildung 2.3: Prinzip des *Master-Slave*-Algorithmus

Der *Master-Slave*-Algorithmus sorgt für eine gleichmäßige Auslastung der Prozessoren. Ein großer Nachteil ist jedoch, daß die gesamte Kommunikation über den Master abgewickelt wird. Bei steigender Prozessorzahl ist bei Implementierungen dieses Algorithmus sinkende Effizienz zu beobachten, da der Master zum „Flaschenhals“ wird. Häufig tritt dies bereits bei einer Prozessorzahl $p > 16$ ein. Außerdem beschränkt der Speicherplatz des Masters die Anzahl der Teilprobleme, die während der Abarbeitung des Suchbaums entstehen kann. Für spezielle Probleme kann der *Master-Slave*-Algorithmus daher nützlich sein, im allgemeinen müssen jedoch Verbesserungen vorgenommen oder andere parallele Algorithmen ausgewählt werden.

Verteilte Verwaltung: Bei vielen parallelen *Branch and Bound*-Algorithmen werden die zu untersuchenden Teilprobleme im Netzwerk verteilt gespeichert. Dabei wurden bislang viele verschiedene Varianten verteilter Algorithmen untersucht. Verteilte Algorithmen können sich darin unterscheiden, nach welchen Strategien der Lastausgleich stattfindet und wie neugefundene beste Schranken im gesamten Netzwerk bekanntgemacht werden.

In jedem Fall sollen verteilte parallele *Branch and Bound*-Algorithmen die beiden folgenden Ziele erreichen:

1. Die Leerlaufzeit (*idletime*) jedes Prozessors soll minimal sein.

2. Der Mehraufwand an untersuchten Teilproblemen durch die parallele Suche gegenüber seriellen *Branch and Bound*-Algorithmen soll minimal sein.

Leider ist es jedoch so, daß die beiden Forderungen in einem Gegensatz zueinander stehen. Ein serieller Algorithmus untersucht in jedem Schritt dasjenige Teilproblem, das zu einer besten Lösung (Teilproblem mit bester Schranke) führen kann. Bei einem verteilten Algorithmus kann diese beste Schranke nur dann allen Prozessoren sofort bekannt sein, wenn der Vorrat an zu untersuchenden Teilproblemen in einer zentralisierten Art und Weise verwaltet wird. Wie oben ausgeführt, ist dieses im allgemeinen ineffizient. Daher wird ein verteilter Algorithmus in der Regel einen höheren Gesamtsuchaufwand gegenüber einem seriellen Algorithmus verzeichnen. Forderung 2. zielt auf die Minimierung dieses Mehraufwands ab. Um dieses Ziel zu erreichen, müssen die Lastverteilungsaktivitäten an einigen Stellen im Netzwerk gebündelt werden, können also nicht gleichmäßig verteilt werden. Auf der anderen Seite macht die erste Forderung nach einer Minimierung der Leerlaufzeiten eine Verteilung der Last über das ganze Netzwerk notwendig. Ein guter paralleler *Branch and Bound*-Algorithmus muß also durch eine geeignete Lastverteilungsstrategie einen Kompromiß zwischen diesen beiden Zielen finden.

Bevor der von uns favorisierte Lastverteilungsalgorithmus vorgestellt wird, hier noch einige Informationen und Hinweise zu anderen parallelen *Branch and Bound*-Algorithmen mit verteilter Verwaltung:

In [81] wird ein Lastverteilungsalgorithmus für das Knotenüberdeckungsproblem (*Vertex Cover Problem*) angegeben. Dieser Algorithmus basiert auf der Arbeit von Karp und Zhang [35], die randomisierte Verteilungsstrategien anwenden. Für kleinere Netzwerke ($p = 32$) liefert der Algorithmus gute Ergebnisse, für größere Netzwerke gibt es jedoch in der Endphase der Berechnung Probleme, da nur noch wenige Teilprobleme bearbeitet werden, diese aber auf einigen wenigen Prozessoren konzentriert sind.

Eine weitere Lastverteilungsmethode ist das sogenannte *Round Robin*-Verfahren, bei dem noch nicht bearbeitete Teilprobleme an Nachbarprozessoren weitergereicht werden. In [63] werden verschiedene Varianten bezüglich der Auswahlstrategien der zu versendenden Teilprobleme untersucht. Die besten Ergebnisse wurden beim Versenden des jeweils zweitbesten Teilproblems zu einem zufällig ausgewählten Prozessor erzielt. Für Paral-

lelrechner mit einer größeren Anzahl an Prozessoren wurde jedoch keine besonders große Beschleunigung erzielt (z. B. Effizienz 21.5% bei $p = 64$ Prozessoren).

Weitere Ergebnisse zu parallelen verteilten *Branch and Bound*-Algorithmen sind in [3], [39], [52], [77] und [80] zu finden.

In dieser Arbeit wird für die Parallelisierung der globalen Optimierung ein verteilter *Branch and Bound*-Algorithmus zugrunde gelegt, wie er in [53] beschrieben ist. Im folgenden Abschnitt wird der Grundalgorithmus dargestellt.

2.4.3 Ein vollständig verteilter Algorithmus zur Lastverteilung für *Branch and Bound*

Der parallele *Branch and Bound*-Algorithmus ist vollständig verteilt, die Kommunikation wird ausschließlich durch *message passing* realisiert. Jeder Prozessor hat seinen eigenen Vorrat an Teilproblemen und führt auf diesen Problemen den sequentiellen *Branch and Bound*-Algorithmus (später: den globalen Optimierungsalgorithmus) aus. Neu berechnete Lösungen werden allen anderen Prozessoren im Netzwerk durch ein *broadcast* bekannt gemacht.

Um die hier verwendete verteilte dynamische Lastverteilungsstrategie zu beschreiben, muß zunächst die Last eines Prozessors bei einem *Branch and Bound*-Algorithmus definiert werden. Bei einem solchen Algorithmus wird die Last eines Prozessors als eine Gewichtsfunktion w in Abhängigkeit der auf dem lokalen Stapel vorhandenen Teilprobleme definiert:

Definition 2.3 *Seien c die beste bisher gefundene Lösung, $\{x_1, \ldots, x_k\}$ die Schranken der Elemente in der Arbeitsliste von p_i, die zu einer besseren Lösung führen können ($x_i < c$).*

Die Last *eines Prozessors p_i ist eine Gewichtsfunktion*

$$w : \{p_1, \ldots, p_p\} \longrightarrow I\!N,$$

die entweder in konstanter Zeit berechenbar ist oder

$$w(\{x_1, \ldots, x_k\}) = w(\{x_1, \ldots, x_i\}) + w(\{x_{i+1}, \ldots, x_k\})$$

erfüllt.

Bemerkung:

1. Die Forderung, daß die Gewichtsfunktion entweder in konstanter Zeit berechenbar sein muß oder die Bedingung $w(\{x_1, \ldots, x_k\}) = w(\{x_1, \ldots, x_i\}) + w(\{x_{i+1}, \ldots, x_k\})$ erfüllt, soll sicherstellen, daß die Gewichtsfunktion effizient berechnet werden kann, da sie zur Laufzeit des Algorithmus häufig ausgewertet werden muß.

2. Die Wahl der Gewichtsfunktion hängt vom zu bearbeitenden Problem ab. Beispiele für mögliche Gewichtsfunktionen sind:

   (a) $w(p_i) = k$

   (b) $w(p_i) = \min_j \{x_j\}$

   (c) $w(p_i) = \sum_{j=1}^{k} (c - x_j)^2$

   (d) $w(p_i) = \sum_{j=1}^{k} e^{c - x_j}$

   (c) und (d) gewichten die „guten“ Teilprobleme (mit niedriger unterer Schranke) stärker. Daher ist es sinnvoll, diese Gewichtsfunktionen auszuwählen, wenn die Schranken der Teilprobleme sehr unterschiedlich sind. Haben die zu untersuchenden Teilprobleme ähnliche Schranken, so kann die Gewichtsfunktion (a) ausreichend sein.

3. Verteilte dynamische Lastverteilung ist von vielen Autoren eingehend untersucht worden. Da jedoch die meisten Arbeiten ausschließlich auf eine Minimierung der Leerlaufzeit der Prozessoren abzielen, hier jedoch der Aspekt eines Qualitätsmaßes (Gewichtsfunktion) mitbetrachtet werden soll, unterscheidet sich der nachfolgend beschriebene Algorithmus von vielen bekannten Algorithmen. Hier ist das grundlegende Ziel, ständig einen qualitativen Ausgleich der Last aller Prozessoren sicherzustellen. Eine Übersicht ähnlicher Strategien findet sich in [52].

Der Lastverteilungsalgorithmus sorgt dafür, daß die Last benachbarter Prozessoren während der Berechnung etwa auf gleichem Niveau ist. Dazu kennt jeder Prozessor die Last der benachbarten Prozessoren im Netzwerk. Falls die eigene Last eines Prozessors klein ist im Vergleich zu den Nachbarprozessoren, so werden Teilprobleme von den entsprechenden Nachbarprozessoren

angefordert. Umgekehrt, falls die eigene Last groß ist im Vergleich zu den Nachbarn, so wird Arbeit (d.h. Lastpakete bzw. Teilprobleme) abgegeben. Diese Idee des lokalen Lastausgleichs benachbarter Prozessoren führt zu annähernd gleicher Last im gesamten Netzwerk, der Algorithmus ist also durch seine Konstruktion *skalierbar*. Falls der Lastunterschied Δ benachbarter Prozessoren klein genug ist, ist der Overhead der parallelen Suche minimal, da dann nur wenig Kommunikation zum Lastausgleich notwendig ist. Falls Δ klein ist und sich die Lastsituation stark ändert, können sogenannte *Trashing*-Effekte auftreten. *Trashing*-Effekte sind aus der Betriebssystemtheorie für Speicherverwaltungsalgorithmen bekannt. Sie bedeuten, daß nur noch Speicherauslagerungsaktivitäten stattfinden, jedoch keine Berechnungen mehr, da nach dem Start einer Berechnung nach einer Speicherauslagerung sofort erneut Speicher ausgelagert wird. In unserem Zusammenhang wird unter *Trashing*-Effekt das Blockieren des parallelen *Branch and Bound*-Algorithmus durch ständigen Lastausgleich verstanden. Neue Ergebnisse werden nicht mehr berechnet. Um dies zu verhindern, wird ein zusätzlicher Kontrollprozeß ausgeführt, der die Lastverteilung überwacht.

Die grundsätzliche Idee des Lastverteilungsalgorithmus besteht also aus einer *Feedback*-Strategie mit zwei parallel laufenden Prozessen pro Prozessor:

1. Arbeitsprozeß
2. Kontrollprozeß

Das Zusammenspiel der beiden Prozesse wird in Abbildung 2.4 verdeutlicht.

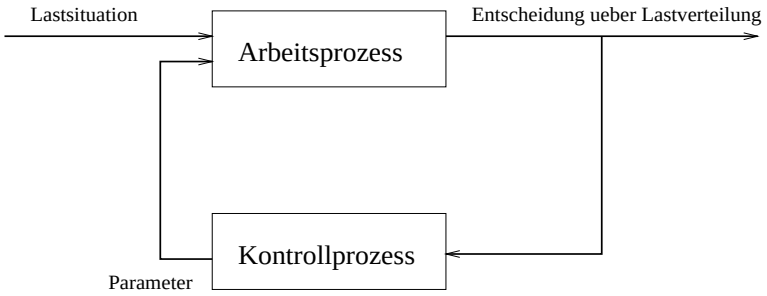


Abbildung 2.4: *Feedback*-Strategie für den Lastverteilungsalgorithmus

Der Arbeitsprozeß

Im folgenden wird der Arbeitsprozeß innerhalb des Lastverteilungsalgorithmus im Detail vorgestellt. Der Arbeitsprozeß wird durch die folgenden Eigenschaften charakterisiert:

- *Lokale* Entscheidungsbasis über Lastverteilung *(wann wird verteilt?)*
- *Lokaler* Migrationsraum für Lastverteilung *(wohin wird verteilt?)*

Um die annähernd gleiche Last eines Prozessors p und seiner Nachbarn $p_1, \ldots, p_k$ zu erreichen, sollte während der Ausführung des Algorithmus möglichst ständig gelten (Δ fest):

$$\max_{i \in \{1,\ldots,k\}} \frac{|w(p) - w(p_i)|}{|w(p)|} < \Delta$$

Bemerkung: Δ muß klein sein, da gilt ($p_i, p_j \in$ Netz mit Durchmesser d):

$$\underbrace{w(p_i) \cdot (1-\Delta)^d}_{=:\min_w} \leq w(p_j) \leq \underbrace{w(p_i) \cdot (1+\Delta)^d}_{=:\max_w} \tag{2.1}$$

Beispielsweise gilt für $d = 8, \Delta = 0.3 : \min_w = 0.05, \max_w = 8.15$. Dies würde bedeuten, daß sich die Last zweier benachbarter Prozessoren um den Faktor 20 unterscheiden darf. Dies ist sicherlich nicht wünschenswert. Δ sollte daher kleiner gewählt werden.

Im nachfolgend beschriebenen Arbeitsprozeß werden die folgenden Steuervariablen verwendet:

$\Delta_{up}, \Delta_{down}$: Aktiviere Lastverteilung, falls Last um mehr als Faktor $(1 - \Delta_{down})$ sinkt oder mehr als Faktor $(1 + \Delta_{up})$ steigt.

Δ: Prozessor p_i nimmt an Lastverteilungsaktivität von p_j teil $\Longleftrightarrow$ Last von p_i und p_j unterscheidet sich mindestens um Faktor $(1 - \Delta)$.

***min.weight*:** Lastverteilung auf Prozessor p wird nur durchgeführt, falls $w(p) >$ *min.weight* gilt.

Der Algorithmus läuft auf einem Parallelrechner mit p Prozessoren in zwei Phasen ab: In der *Startphase* wird schnell auf einem Prozessor eine Teillösung berechnet, um daraus eine grobe globale obere Schranke zu gewinnen. Anschließend wird das Ausgangsproblem in p Teilprobleme aufgeteilt, jedes der Teilprobleme wird einem der Prozessoren zugeordnet, und die eigentliche *Berechnungsphase* beginnt.

Die einzelnen Schritte des Algorithmus für den Arbeitsprozeß sind nachfolgend für einen Prozessor p mit den Nachbarprozessoren $p_1, \ldots, p_k$ aufgeführt. Zuletzt bekannte Werte für die Last von $p_1, \ldots, p_k$ werden in $w_1, \ldots, w_k$ gespeichert. Die einzelnen Teilprobleme $\{x_1, \ldots, x_k\}$ werden in einer Arbeitsliste in Form einer Warteschlange gespeichert, die nach den zugehörigen Schranken $bound(x_i)$ sortiert ist. w_{new} speichert die aktuelle Last des Prozessors p, w_{old} die vorige. Die bislang beste bekannte globale Schranke wird in der Variablen c gespeichert.

Algorithmus 2.1:
Paralleler Arbeitsprozeß für *Branch and Bound*

1. $w_1 = \ldots = w_k = 0; w_{new} = w_{old} = w(p)$;
   Initialisiere c, Arbeitsliste;
2. **if** (neue Lösung x berechnet) **then if** $(bound(x) < c)$ **then**
   a. $c := bound(x)$; Update(w_{new});
   b. Send(NEWSOLUTION,c) an alle Nachbarn;
3. **if** ((NEWSOLUTION,c') von p_j empf.) **then if** $(c' < c)$ **then**
   a. $c' := c$; Update(w_{new});
   b. Send(NEWSOLUTION,c) an alle Nachbarn außer p_j;
4. **if** $(w_{new} < w_{old} \cdot (1 - \Delta_{down}))$ **then**
   a. Send(REQUEST,w_{new}) an alle Nachbarn;
   b. $weights := weights + 1$; $w_{old} := w_{new}$;

5. **if** ((REQUEST, w_j) von p_j empfangen) **then**
   if ($w_j < w_{new} \cdot (1-\Delta)$ **and** $w_{new} > min.weight$) **then**
   a. Send(WORK,x_1) an p_j; $prob := prob + 1$;
   b. $w_{new} := w_{new} - w(x_1)$; $w_j := w_j + w(x_1)$;
6. **if** ((WORK, x) von p_j empf.) **then if** ($bound(x) < c$) **then**
   a. $w_{new} := w_{new} + w(x)$; Send(REQUEST, w_{new}) an p_j;
   b. $weights := weights + 1$; Insert(L, x);
7. **if** ($w_{new} > w_{old} \cdot (1+\Delta_{up})$ **and** $w_{new} > min.weight$) **then**
   a. Unmark(alle Nachbarn);
   b. **for** $i := 1$ **to** k **do**
   A. Wähle zufälligen nicht markierten Nachbarn p_l; Mark(p_l);
   B. **if** ($w_l < w_{new} \cdot (1-\Delta)$) **then**
   1. Send(WORK,x_1) an p_l; $prob := prob + 1$;
   2. $w_l := w_l + w(x_1)$; $w_{new} := w_{new} - w(x_1)$;
   c. Send(INFORM,w_{new}) an alle Nachbarn;
   d. $weights := weights + 1$; $w_{old} := w_{new}$;
8. **if** ((INFORM,w_j) von p_j empfangen) **then**
   Send(REQUEST, w_{new}) an p_j; $weights := weights + 1$;

Im Schritt 1. werden die oben beschriebenen Variablen initialisiert. Falls der *Branch and Bound*-Algorithmus eine neue Lösung findet, die die bislang beste bekannte globale Schranke verbessern kann, so wird diese globale Schranke den benachbarten Prozessoren bekannt gemacht (Schritt 2.). Diese verteilen, falls notwendig, in Schritt 3. die neue beste globale Schranke weiter. Falls sich die Last eines Prozessors um mehr als Δ_{down} verringert, so werden in Schritt 4. neue Teilprobleme von den Nachbarn angefordert. Diese Anfrage nach Arbeit wird in Schritt 5. beantwortet: Ist ein Prozessor um Δ Prozent mehr belastet als der anfragende Nachbar und wird außerdem noch eine Mindestlast überschritten, so wird ein Teilproblem an den Nachbarprozessor abgegeben. Falls (in Schritt 6.) ein bislang weniger belasteter Prozessor ein Teilproblem x von einem Nachbarprozessor empfängt, so wird es auf seine Brauchbarkeit überprüft. Ist es ein möglicher Kandidat

für eine Lösung ($bound(x) < c$), so wird es in die Arbeitsliste eingefügt und weitere Teilprobleme werden abgefordert. Dies dient dem sukzessiven Lastausgleich benachbarter Prozessoren. Schließlich wird in Schritt 7. der Fall behandelt, daß die lokale Lastsituation stark ansteigt. Dann werden aus der Arbeitsliste vielversprechende Teilprobleme an zufällig ausgesuchte und nicht so stark belastete Nachbarprozessoren weitergegeben. Alle Nachbarn werden anschließend über die veränderte Lastsituation informiert, die daraufhin (Schritt 8.) eventuell neue Teilprobleme anfordern.

Der Kontrollprozeß

Wenn der Arbeitsprozeß eine „gute“ neue Lösung findet und diese neue Schranke durch das Netzwerk verteilt wird, kann es zu großen Lastunterschieden benachbarter Prozessoren kommen, da lokale Arbeitslisten unterschiedlich stark reduziert werden. Ist die Steuervariable Δ klein (notwendig für größere Netzwerke, vgl. Gleichung 2.1), so wird deshalb viel Kommunikations- und Migrationsaktivität, aber nur wenig Berechnung die Folge sein. Um diesen *Trashing*-Effekt zu verhindern, wird eine *Feedback*-Strategie für den Lastverteilungsalgorithmus zum Dämpfen verwendet.

Derartige Kontrollmechanismen sind aus der Kontrolltheorie bekannt und dienen dazu, die Ergebnisse des Arbeitsprozesses in sinnvollen Schranken zu halten. Es werden untere Schranken $\Delta_{up}^{min}, \Delta_{down}^{min}$ und Δ^{min} für die entsprechenden Parameter des Arbeitsprozesses definiert. Während des Ablaufs des Lastverteilungsalgorithmus werden dessen Entscheidungen immer wieder während eines festen kurzen Zeitintervalls t (z.B. $t = 0.1$ Sekunden) gemessen. Es wird dabei protokolliert:

- Die Anzahl der verteilten Teilprobleme
- Die Anzahl der verteilten Lastinformationen

Die gemessenen Daten werden dabei in den folgenden Variablen gespeichert:

$prob^t$: Anzahl verteilter Teilprobleme (durch Send(WORK,x)) im Zeitintervall t

$weights^t$: Anzahl verteilter Lastinformationen (durch Send(INFORM,w) oder Send(REQUEST,w)) im Zeitintervall t

$av.prob^t$**:** durchschnittliche Anzahl verteilter Teilprobleme in den Zeitintervallen 0–t

$av.weights^t$**:** durchschnittliche Anzahl verteilter Lastinformatonen in den Zeitintervallen 0–t

Der Kontrollprozeß berechnet die neuen Kontrollvariablen $\Delta_{up}^{t+1}, \Delta_{down}^{t+1}$ und Δ^{t+1} nach den folgenden Regeln:

Algorithmus 2.2:
Kontrollprozeß für *Branch and Bound*

1. Bestimme $av.prob^t$ und $av.weights^t$;
2. **if** ($prob^t > A \cdot av.prob^t$ **or** $weights^t > B \cdot av.weights^t$) **then**
   a. $\Delta_{up}^{t+1} := \Delta_{up}^{t} + 0.01;$
   b. $\Delta_{down}^{t+1} := \Delta_{down}^{t} + 0.01;$
   c. $\Delta^{t+1} := \Delta^{t} + 0.01;$

   else
   a. $\Delta_{up}^{t+1} := \max\{\Delta_{up}^{t} - 0.01, \Delta_{up}^{min}\};$
   b. $\Delta_{down}^{t+1} := \max\{\Delta_{down}^{t} - 0.01, \Delta_{down}^{min}\};$
   c. $\Delta^{t+1} := \max\{\Delta^{t} - 0.01, \Delta^{min}\};$

Bemerkung: Das Verhalten des Kontrollprozesses wird durch die Parameter A und B gesteuert. Im Fall $A, B > 1$ reagiert der Kontrollprozeß nur auf ansteigende Lastverteilungaktivität, was nicht immer sinnvoll ist. Aus der Praxis (vgl. [53]) zeigt es sich, daß es günstig ist, $A = B = 0.5$ zu wählen. In diesem Fall verhält sich der Kontrollprozeß sensitiv bezüglich hoher Lastverteilungaktivität, auch wenn sie bereits abnimmt.

Für die unteren Schranken $\Delta_{up}^{min}, \Delta_{down}^{min}$ und Δ^{min} ist jeweils ein Wert von 0.01 sinnvoll.

In [53] sind einige Ergebnisse aus der Praxis für den obigen Algorithmus angegeben. Als Referenzproblem für Zeitmessungen wird das Knotenüberdeckungsproblem (*Vertex Cover Problem*) gewählt. Experimente wurden auf Ringnetzwerken mit 128 und 256 Prozessoren sowie DeBruijn-Netzwerken (vgl. Abschnitt 2.1.3) mit 256 Prozessoren durchgeführt. Es wurden dabei verschiedene Problemgrößen untersucht. Für die Effizienz des hier vorgestellten Lastverteilungsalgorithmus, der auf die qualitative Ausbalancierung

der Teilprobleme innerhalb des Netzwerks und die Minimierung von Leerlaufzeiten pro Prozessor abzielt, ist es dabei wichtig, daß hinreichend große Probleme untersucht werden. Die Hauptprobleme bei der Lastverteilung treten zu Beginn und am Ende der Berechnung auf, wenn die gesamte Last innerhalb des Netzwerks noch nicht oder nicht mehr so groß ist. Daher sollte diese Berechnungsphase einen nicht zu großen Anteil an der Gesamtberechnung ausmachen.

Für größere Probleme mit längeren Ausführungszeiten ($>$ 100 Sekunden) wurde eine Effizienz von mehr als 93% erzielt. Im Schnitt lag die erzielte Effizienz bei etwa 90%. Lüling und Monien geben an, daß der oben vorgestellte Algorithmus auch zur effizienten Parallelisierung anderer Problemstellungen, die sich durch *best-first branch and bound*-Algorithmen behandeln lassen, geeignet ist. Daß dies tatsächlich der Fall ist, wird im Kapitel 3 gezeigt.

2.5 Zur verwendeten Hardware- und Softwareumgebung

2.5.1 Der Parallelrechner IBM RS/6000 SP

Der für die Berechnung der numerischen Ergebnisse dieser Arbeit eingesetzte Parallelrechner IBM RS/6000 SP der Universität Karlsruhe gehört zur Klasse der heutzutage weit verbreiteten MIMD-Rechner mit verteiltem Speicher. Der Rechner hat eine Gesamtleistung von 110 Gigaflops (Milliarden Gleitkommaoperationen pro Sekunde) bzw. 55 GIPS (Milliarden Ganzzahloperationen pro Sekunde) und verfügt über insgesamt 130 Gigabyte Hauptspeicher: Es werden 168 Thin P2SC nodes (120 MHz) mit je 512 MB, 8 Wide nodes (77 MHz) mit je 2 GB, 8 Wide nodes (77 MHz) mit je 512 MB, 56 Wide nodes (77 MHz) mit je 256 MB und 16 Thin2 nodes (66 MHz) mit je 128 MB vollständig integriert unter dem IBM Unixderivat AIX 4.1 und dem Distributed Computing Environment (DCE) betrieben. Auf der Karlsruher SP werden die Thin2 nodes als Serverknoten im nicht-exklusiven Betrieb für mehrere Benutzer zur Programmentwicklung und die Wide nodes als Rechenknoten exklusiv für jeweils einen Benutzer für Rechnungen und Zeitmessungen genutzt. Zur Steuerung des Batch-Betriebes wird der IBM LoadLeveler eingesetzt.

Die einzelnen Knoten der SP können über eine sehr schnelle interne Verbindung, den SP-Switch (TB3), miteinander kommunizieren, wobei für Programme, die mittels MPI parallelisiert wurden, Übertragungsraten von bis zu 100 MB/sek gemessen werden konnten. Der Switch erlaubt dabei eine direkte *any to any*-Kommunikation zwischen beliebigen Knoten. Netzwerktopologisch betrachtet handelt es sich bei der SP um ein vollständig verknüpftes Netzwerk.

Auf der Softwareseite unterstützt die SP mehrere unterschiedliche Parallelisierungsmöglichkeiten: Das Programmiermodell der expliziten Parallelisierung wird durch MPI (sehr effizient implementiert) und PVM unterstützt. Für die implizite Programmmierung steht High Performance Fortran (HPF) zur Verfügung. Die Effizienz eines zwar einfacher zu entwickelnden HPF-Programms wird aber in der Regel deutlich unter der eines MPI-Programms liegen.

2.5.2 C–XSC und MPI

Als Programmierumgebung zur Implementierung der in den folgenden Kapiteln beschriebenen Algorithmen wurde C–XSC, eine C++ Klassenbibliothek für wissenschaftliches Rechnen, sowie MPI als Kommunikationsbibliothek verwendet. C–XSC ist ausführlich in [41] beschrieben und erweitert die Programmiersprache C++ um Datentypen, Operatoren und Funktionen, die für das wissenschaftliche Rechnen mit automatischer Ergebnisverifikation notwendig sind. Damit ermöglich C–XSC den einfachen Zugriff auf die in Abschnitt 1.3 kurz skizzierte, mathematisch exakt definierte Rechnerarithmetik in den Räumen des numerischen Rechnens (vgl. [44], [45]). C–XSC in Verbindung mit C++ stellt unter anderem die folgenden wichtigen Konzepte zur Verfügung:

- Reelle, komplexe, Intervall- und komplexe Intervallarithmetik mit den Datentypen *real*, *complex*, *interval* und *cinterval*
- Dynamische Vektoren und Matrizen für diese Datentypen
- Kontrollierte Rundung (auch für Ein- und Ausgabe)
- Exaktes Skalarprodukt und Dotprecisiondatentypen
- Hochgenaue Standardfunktionen für alle Standarddatentypen

- Überladen von Operatoren und Funktionen

Zur Abwicklung der Kommunikation zwischen den Knoten des Parallelrechners wird die Kommunikationsbibliothek MPI (*Message Passing Interface*) verwendet. Diese Kommunikationsbibliothek für Parallelrechner mit verteiltem Speicher ist aus der Praxis entstanden, da alle wichtigen Hersteller von Parallelrechnern, so auch IBM, bei der Standardisierung beteiligt waren. Die erste Version des Standards wurde 1994 veröffentlicht, die aktuelle Version 1.1 ist in [59] zu finden. MPI ist auf vielen Parallelrechnern sehr effizient implementiert, so auch auf der IBM SP: Kopieren innerhalb des Speichers wird nach Möglichkeit vermieden; Überlappung von Kommunikation und Berechnung (*hidden latency*) wird ebenfalls unterstützt. MPI bietet unter anderem:

- Punkt-zu-Punkt Kommunikation (z. B.: *send, receive*)
- Kollektive Kommunikation (z. B.: *barrier, broadcast*)
- Verwaltung von Prozeßgruppen und Kommunikationskontext
- Prozeßtopologien
- Unterstützung für Profiling
- Schnittstellen zu den Programmiersprachen C und Fortran

Um nun auf dem Parallelrechner bei der Kommunikation C–XSC Datentypen verwenden zu können, wurden zusätzliche Kommunikationsroutinen implementiert. Es werden im folgenden nur beispielhaft die Schnittstellen mit Erläuterungen angegeben, eine vollständige Aufstellung der neuen Funktionen ist in Anhang C zu finden. Für alle *real* und *interval* Datentypen (skalar, Vektor, Matrix) sind verfügbar:

Synchrones Senden:

```
int MPI_Send(const real& r, int dest, int tag,
             MPI_Comm comm);
```

Sendet den *real* Wert r in einer mit der Markierung *tag* versehenen Nachricht an den im Kommunikator *comm* (spezifiziert den Kommunikationskontext) enthaltenen Prozessor *dest*. Für *comm* kommt häufig

`MPI_COMM_WORLD` zum Einsatz, damit wird der vollständige Parallelrechner bezeichnet. Die Markierung *tag* ist vom Anwender frei wählbar. Durch *tag* können verschiedene Arten von Nachrichten bezeichnet werden. MPI_Send() ist blockierend, das bedeutet, daß das aufrufende Programm so lange wartet, bis die Kommunikation durchgeführt wurde, bis also auf der Gegenseite ein passender Aufruf von MPI_Recv() stattfindet.

Synchrones Empfangen:

```
int MPI_Recv(interval& i, int source, int tag,
             MPI_Comm comm, MPI_Status *status);
```

Empfängt das Intervall i vom Prozessor *source* im Kommunikator *comm*, falls eine mit *tag* markierte Nachricht anliegt. Für *source* oder *tag* können die von MPI vordefinierten Platzhalter (*wildcards*) `MPI_ANY_SOURCE` bzw. `MPI_ANY_TAG` angegeben werden. In diesem Fall enthält die Variable *status* nach dem Funktionsaufruf den tatsächlichen Sender und die tatsächliche Markierung. Außerdem wird in *status* ein eventueller Fehlerstatus gespeichert. Vektoren und Matrizen werden in ihrer Größe dem zu empfangenden Vektor bzw. der zu empfangenden Matrix angepaßt. MPI_Recv() ist ebenfalls blockierend.

Asynchrones Senden:

```
int MPI_Isend(const imatrix& im, int dest, int tag,
              MPI_Comm comm, MPI_Request *req);
```

In MPI und damit auch für die C–XSC Datentypen ist ein asynchrones Senden (nichtblockierende Kommunikation) möglich: Das Senden (hier das Senden einer *imatrix*) wird angestoßen, der Aufruf in *req* eingetragen und die Funktion MPI_Isend() kehrt in das aufrufende Programm zurück. Ob die Kommunikation erfolgreich durchgeführt wurde, kann an einer späteren Stelle im Programm unter Verwendung einer Abfragefunktion (es gibt mehrere Varianten, etwa MPI_Test() oder MPI_Wait()) und *req* abgeprüft werden. Dadurch wird eine Überlagerung von Kommunikation und Berechnung ermöglicht, da nicht auf die Beendigung der Kommunikation gewartet werden muß.

Kapitel 3

Verifizierte globale Optimierung

In den beiden vorigen Kapiteln sind die notwendigen Grundlagen und Werkzeuge zum Entwurf eines parallelen Algorithmus zur globalen Optimierung mit Ergebnisverifikation zur Verfügung gestellt worden. Das nun folgende Kapitel enthält zunächst einige Ausführung zur Problemstellung der globalen Optimierung ohne Nebenbedingungen. Anschließend wird ein neuer serieller Algorithmus, der als Grundlage für die Parallelisierung verwendet wird, hergeleitet. Dazu werden zunächst bekannte Vorarbeiten und andere Ansätze aus der Literatur vorgestellt, danach der neue serielle Algorithmus mit seinen zusätzlichen effizienzsteigernden Methoden angegeben. Numerische Beispiele und Vergleiche zwischen dem neuen und bisherigen seriellen Algorithmen schließen den Abschnitt über den seriellen Algorithmus ab.

Für die Parallelisierung wird der im vorangegangenen Kapitel hergeleitete vollständig verteilte Algorithmus zur Lastverteilung für *Branch and Bound* auf den seriellen Algorithmus zur verifizierten globalen Optimierung angewandt. Dies ist deswegen möglich, da der serielle Algorithmus von seiner Struktur her in die Klasse der *best-first branch and bound*-Algorithmen fällt.

Um zu zeigen, daß die hier vorgeschlagene Methode der Parallelisierung zur Beschleunigung der verifizierten Lösung globaler Optimierungsprobleme effizient und skalierbar ist, sowie Vergleichen mit anderen parallelen Algorithmen standhält, folgt am Ende dieses Kapitels ein Abschnitt, der die Effizienz des parallelen Algorithmus untersucht.

3.1 Einführung und Problemstellung

Viele quantitative Entscheidungsprobleme aus den Bereichen Wirtschaft, Wissenschaft und Ingenieurwesen lassen sich in Form eines globalen Optimierungsproblems modellieren. Dabei wird die gesuchte bestmögliche Entscheidung durch die Lösung des globalen Optimierungsproblems (o.B.d.A. als Minimierungsproblem formuliert) dargestellt. In jedem Fall ist man bei der Lösung des globalen Optimierungsproblems am Wert des globalen Minimums (in der Regel ein reeller Vektor) interessiert, häufig benötigt man auch noch Aussagen über die globalen Minimalstellen, an denen das globale Minimum angenommen wird.

Gegeben sei also eine stetige Funktion $f : D \subseteq \mathbb{R}^n \to \mathbb{R},\ D$ offen. Gesucht wird das globale Minimum

$$f^\star = \min_{x \in X} f(x),$$

wobei $X \subseteq D$ eine kompakte Menge ist. Ferner ist die Menge der globalen Minimalstellen

$$X^\star = \{x \in X \mid f(x) = f^\star\}$$

zu bestimmen. In späteren Abschnitten wird zusätzlich noch die zweimalige stetige Differenzierbarkeit von f benötigt.

Im allgemeinen ist die Lösung eines globalen Optimierungsproblems aufwendig zu bestimmen. Der Nachweis, daß ein (recht einfach zu findendes) lokales Minimum auch gleichzeitig ein globales Minimum ist, kann nicht direkt geführt werden. Zur Bestimmung des globalen Minimums müssen daher noch weitere Zusatzinformationen ausgewertet werden. In der Literatur existiert daher eine Vielzahl von Ansätzen zur Lösung des globalen Optimierungsproblems, siehe etwa [62] für eine Übersicht über Verfahren sowie existierende Softwarepakete zur globalen Optimierung. Prinzipiell unterscheidet man zwischen zwei Klassen von globalen Optimierungsverfahren:

1. Stochastische Verfahren
2. Deterministische Verfahren

Bei den stochastischen Verfahren (siehe [29] für eine Übersicht) werden zufällige Punkte aus X ausgewählt und entweder direkt zur Funktionsaus-

wertung verwendet oder als Startpunkt für ein lokales Minimierungsverfahren eingesetzt. Bessere Verfahren berücksichtigen dabei bisher berechnete Ergebnisse und beziehen diese in die Auswahl der neuen Startpunkte mit ein. In jedem Fall läßt sich das globale Minimum nur mit einer gewissen Wahrscheinlichkeit bestimmen. Neuere stochastische Verfahren sind das sog. *simulated annealing* Verfahren [79] oder Verfahren auf der Basis von genetischen Algorithmen [55].

Deterministische Verfahren durchsuchen den gesamten Suchraum (die Menge X) und versuchen sicherzustellen, daß am Ende das globale Minimum (oder zumindest eine Approximation davon) gefunden wird. Da der Suchaufwand mit steigender Dimension n exponentiell wächst und das globale Optimierungsproblem *NP-hard* ist, kann dies für große n nur mit gewissen Einschränkungen geschehen. Ohne Anspruch auf Vollständigkeit seien die folgenden Arten von deterministischen Verfahren angegeben:

Konkave Minimierung: Bei der konkaven Minimierung wird eine gegenüber dem allgemeinen globalen Optimierungsproblem eingeschränkte Funktionenklasse, die Klasse der konkaven Funktionen behandelt. Ferner wird vorausgesetzt, daß der zu betrachtende Suchraum X eine abgeschlossene konvexe Menge ist. Bei der Konstruktion von Algorithmen für die konkave Minimierung wird die Konkavitätseigenschaft an vielen Stellen ausgenutzt. Die resultierenden Algorithmen sind sehr effizient und erlauben die Behandlung von größeren Problemen als im allgemeinen Fall. Viele Anwendungen aus der Praxis lassen sich auf ein konkaves Minimierungsproblem zurückführen. Eine umfassende Einführung in die konkave Minimierung findet sich in [4].

Adaptive Unterteilungs- und Suchstrategien: In die Klasse der adaptiven Unterteilungs- und Suchstrategien fallen die *Branch and Bound*-Algorithmen (vgl. Abschnitt 2.4). Der im allgemeinen viel zu große Suchbaum kann durch das *branch and bound*-Prinzip stark beschnitten werden, sofern es gelingt, sichere Schranken für das globale Minimum zu finden. Insbesondere verwenden die Intervallmethoden das *branch and bound*-Prinzip. Da die Intervallmethoden die entscheidenden Grundlagen für den in dieser Arbeit untersuchten parallelen Algorithmus zur globalen Optimierung bilden, werden sie in einer eigenen Verfahrensklasse aufgeführt. Zu den adaptiven Unterteilungs- und Suchstrategien zählt man ferner den bayesianschen Zugang (vgl. [56]). Eine allgemeine Übersicht über adaptive Unterteilungs- und Suchstrategien gibt [29].

Intervallmethoden: Intervalle sind gut geeignet, um globale Optimierungsprobleme zu bearbeiten. Obwohl ein Intervall durch nur zwei Punkte dargestellt wird, repräsentiert es jedoch eine Menge, die i.a. unendlich viele Punkte enthält. Ein Intervall ist somit Träger *globaler Information*. Mit Hilfe der in Abschnitt 1.4 eingeführten Intervallauswertung ist es möglich, mit nur einer einzigen Funktionsauswertung globale Aussagen über den Wertebereich der Funktion auf einem Intervall zu treffen. In Verbindung mit einem *Branch and Bound*-Algorithmus läßt sich in vielen Fällen nicht nur effizient das globale Minimum bestimmen, sondern es können in der Regel auch noch gesicherte enge Schranken für das globale Minimum und für die globalen Minimalstellen automatisch auf dem Rechner bewiesen werden. Der Aspekt der automatischen Ergebnisverifikation ist jedoch nicht das Hauptziel der Intervallmethoden, wichtiger ist die Verwendung von Intervallen zur effizienten Repräsentierung globaler Informationen. In den folgenden Abschnitten wird der serielle und parallele Algorithmus zur globalen Optimierung ausführlich beschrieben. In [67] ist eine knappe Einführung in die globale Optimierung mit Intervallmethoden zu finden.

Enumerative Strategien: Enumerative Strategien können in einem Spezialfall der konkaven Minimierung eingesetzt werden, falls nämlich der zu betrachtende Suchraum X ein Polyeder ist. Unter diesen Voraussetzungen wird das globale Minimum in einer der Ecken (Extremalpunkte) von X angenommen. Da X nur endlich viele Ecken besitzt, können diese durch einen entsprechenden Algorithmus nacheinander untersucht werden. Die verschiedenen Varianten enumerativer Strategien (Ordnung der Extremalpunkte, Verwendung von Schnittflächen usw.) versuchen, die Extremalpunkte möglichst effizient zu untersuchen. Weitere Details finden sich etwa in [4]. Auch bei kombinatorischen Problemen werden enumerative Strategien eingesetzt.

Heuristische Verfahren: Eine Vielzahl von Algorithmen zur globalen Optimierung wird heuristisch hergeleitet. Dazu zählen etwa das Tunnelverfahren [50], die Verwendung von Füllfunktionen [16] und andere, siehe [29].

3.2 Serieller Algorithmus

Wie bereits in Abschnitt 2.2 ausgeführt, ist eine Beurteilung der Beschleunigung eines parallelen Algorithmus nur dann sinnvoll, wenn zur Messung der seriellen Laufzeit t_{ser} der beste bekannte serielle Algorithmus herangezogen wird. Daher wird in diesem Abschnitt ein neuer serieller Algorithmus zur verifizierten globalen Optimierung hergeleitet. Dieser Algorithmus verbessert die Laufzeiten bisheriger Algorithmen erheblich.

In diesem Abschnitt wird zunächst das grundlegende Arbeitsprinzip aller Algorithmen zur verifizierten globalen Optimierung vorgestellt. Anschließend werden Bausteine zur Beschleunigung des Grundalgorithmus angegeben. Als Vergleichsgrundlage für den neuen seriellen Algorithmus dient der in [21] angegebene sog. *Toolbox*-Algorithmus (bzw. verschiedene Varianten davon). Daher werden zunächst diese Varianten vorgestellt, anschließend der neue serielle Algorithmus. Angaben zu numerischen Vergleichen und Tests schließen diesen Abschnitt ab.

Das wesentliche Hilfsmittel bei der verifizierten globalen Optimierung ist die Intervallrechnung. Die ursprüngliche Problemstellung wird daher leicht modifiziert zu:

Gegeben sei eine stetige Funktion $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$. Gesucht wird das globale Minimum

$$f^\star = \min_{x \in X} f(x),$$

wobei $X \in I\mathbb{R}^n$ ein n-dimensionaler Würfel ist. Ferner ist die Menge der globalen Minimalstellen

$$X^\star = \{x \in X \mid f(x) = f^\star\}$$

zu bestimmen.

Erste Ideen zum weiter unten angegebene Algorithmus für die verifizierte globale Optimierung wurden 1974 von Skelboe [74] entwickelt, der seinerseits Vorarbeiten von Moore [57], einem der Mitbegründer der Intervallrechnung, verwendet. Dieser ursprüngliche Algorithmus, der 1979 von Fujii und Ichida [31] weiterentwickelt wurde, berechnet nur das globale Minimum $f^\star$, nicht jedoch die globalen Minimalstellen $X^\star$. Der erste Algorithmus, der

Einschließungen sowohl für das globale Minimum als auch für die Minimalstellen berechnet, stammt von Hansen und wurde 1979 für den eindimensionalen Fall in [22] veröffentlicht und 1980 in [23] auf den mehrdimensionalen Fall erweitert.

Diese Algorithmen sowie die vielen späteren Varianten (siehe unter anderem [24], [38], [65], [68]) basieren auf dem *Branch and Bound*-Prinzip. Dabei wird die Startbox X in Teilboxen (*branches*) zerlegt, beispielsweise durch Bisektion. Die zu bearbeitenden Teilboxen werden sortiert abgespeichert, wobei verschiedene Sortierungskriterien möglich sind. Einzelne Teilboxen können durch verschiedene Tests als ganzes von der weiteren Bearbeitung ausgeschlossen werden (*bounding*), da nachgewiesen werden kann, daß diese Box keine globale Minimalstelle enthält. Ansonsten erfolgt ein neuer *branching*-Schritt (Bisektion oder Multisektion) und Weiterbearbeitung, bis ein vorgegebenes Abbruchkriterium erreicht ist. Zur Beschleunigung des Verfahrens sind zusätzliche Techniken wichtig. Sie werden weiter unten erläutert.

Das *Branch and Bound*-Prinzip kann hier deshalb erfolgreich auf die Problemstellung der globalen Optimierung angewendet werden, da es die Intervallrechnung ermöglicht, durch die Intervallauswertung $F(Y)$ gesicherte Unter- und Oberschranken für den Wertebereich von f über einer Box $Y \subseteq X$ zu bestimmen. Gilt beispielsweise für zwei Boxen $Y_1, Y_2 \subseteq X$ die Beziehung $\underline{F(Y_1)} > \overline{F(Y_2)}$, so enthält Y_1 sicher keine globale Minimalstelle. Y_1 kann daher im *bounding*-Schritt von der weiteren Bearbeitung ausgeschlossen werden. Im wesentlichen wird eine effiziente Berechnung also durch das *Ausschließen* möglichst großer Teilboxen erreicht.

3.2.1 Der grundlegende Algorithmus

Für den nachfolgend beschriebenen Grundalgorithmus 3.1 sei eine Startbox $X \in I\mathbb{R}^n$ und eine Inklusionsfunktion $F : X \to \mathbb{R}$ gegeben. Der Algorithmus bestimmt Einschließungen für das globale Minimum $f^\star$ und die globalen Minimalstellen aus $X^\star$. Sowohl $f^\star$ als auch die Elemente aus $X^\star$ können im Prinzip beliebig genau (bis hinunter zur Maschinengenauigkeit) berechnet werden. Der Berechnungsaufwand steigt natürlich entsprechend. Entscheidend für die Genauigkeit der Einschließungen ist das Abbruchkriterium in Schritt 3.c.C, für das verschiedene Varianten denkbar sind. Sie werden in Abschnitt 3.2.4 im Detail diskutiert. Der Grundalgorithmus umfaßt bereits den Mittelpunktstest (oder auch *cutoff*-Test) als einfachste Maßnahme zur Beschleunigung des Verfahrens. Dabei wird ständig die bislang

beste obere Schranke $\widetilde{f}$ für das globale Minimum aktualisiert und für den *bounding*-Schritt verwendet, um Boxen aus der Arbeitsliste L zu entfernen, die in keinem Fall globale Minimalstellen enthalten können. Die Bezeichnung Mittelpunktstest stammt von der Auswertung von F am Mittelpunkt der untersuchten Box in den Schritten 1. und 3.e. Statt des Mittelpunkts könnte auch ein beliebiger anderer Punkt $c \in Y$ gewählt werden. Das Arbeitsprinzip des Mittelpunktstests wird in Abbildung 3.1 verdeutlicht. Im Beispiel sind nach dem Mittelpunktstest nur noch die Boxen Y_4 und Y_8 zu bearbeiten.

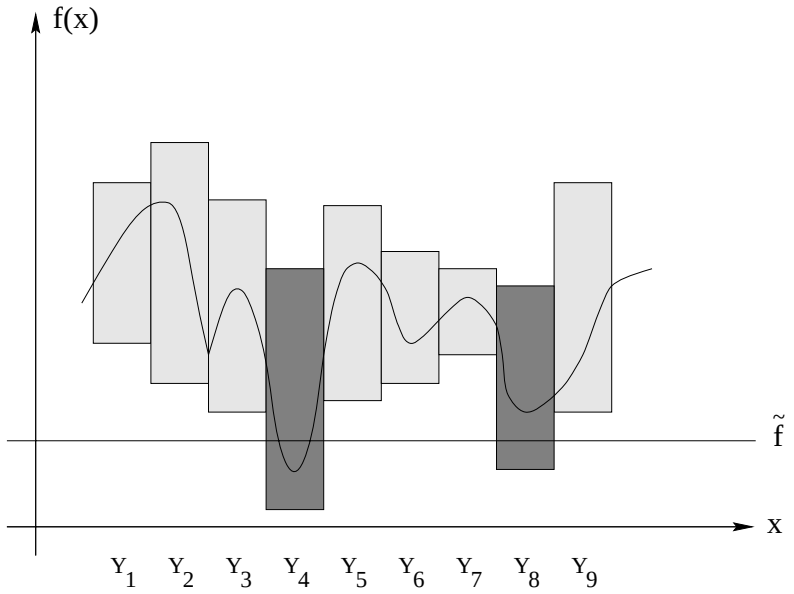


Abbildung 3.1: Der Mittelpunktstest bei der globalen Optimierung

Algorithmus 3.1:
Grundlegender Algorithmus zur globalen Optimierung

1. Initialisiere *Arbeitsliste* $L := \{(X, \underline{F(X)})\}$; $\widetilde{f} := \overline{F(m(X))}$;
2. Initialisiere *Lösungsliste* $\widehat{L} := \{\}$;
3. **while** $(L \neq \{\})$ **do**
   a. $Y :=$ Head(L); $L := L - (Y, \underline{F_Y})$;
   b. Teile Box Y in zwei Teilboxen Y_1, Y_2; **Branching**
   c. **for** $i := 1$ **to** 2 **do**
      A. $F_{Yi} := F(Y_i)$; **if** $(\underline{F_{Yi}} > \widetilde{f})$ **then next**$_i$; **Bounding**
      B. $\widetilde{f} := \min\{\widetilde{f}, \overline{F_{Yi}}\}$;
      C. **if** (Y_i erfüllt Abbruchkriterium) **then** $\widehat{L} := \widehat{L} + (Y_i, \underline{F_{Yi}})$;
         else $L := L + (Y_i, \underline{F_{Yi}})$;
   d. Wähle $(Y, \underline{F_Y}) \in L$ so, daß: $\underline{F_Y} < \underline{F_Z} \forall (Z, \underline{F_Z}) \in L$;
   e. $\widetilde{f} := \min\{\widetilde{f}, \overline{F(m(Y))}\}$;
   f. Mittelpunktstest$(L, \widetilde{f})$; **Bounding**
4. Mittelpunktstest$(\widehat{L}, \widetilde{f})$;

Als Ergebnis liefert der Algorithmus:

$$f^\star \in F^\star := [\min\{\underline{F_X} : (X, \underline{F_X}) \in \widehat{L}\}, \widetilde{f}] \quad \text{und} \quad X^\star \subseteq \bigcup_{(X, \underline{F_X}) \in \widehat{L}} X.$$

Der Mittelpunktstest wird dabei durch den folgenden Algorithmus 3.2 realisiert:

Algorithmus 3.2: Mittelpunktstest$(L, \widetilde{f})$
Der Mittelpunktstest

1. **forall** $(Y, \underline{F_Y}) \in L$ **do**
   if $(\widetilde{f} < \underline{F_Y})$ **then** $L := L - (Y, \underline{F_Y})$;
2. **return** L;

3.2.2 Beschleunigung des Grundalgorithmus

Um den im vorigen Abschnitt beschriebenen Grundalgorithmus effizienter zu gestalten, werden noch eine Reihe von Maßnahmen ergriffen. Im einzelnen behandeln diese Erweiterungen die folgenden Punkte, die in den folgenden Abschnitten genauer erläutert werden:

- Ordnung der Arbeitsliste L
- Geeignete Abbruchkriterien
- Aufteilung der aktuellen Box X
  - Bisektion oder Multisektion
  - Wahl der Bisektions- / Multisektionsrichtung
- Beschleunigungsmaßnahmen unter Verwendung von f' und f''
  - Monotonietest
  - Konkavitätstest
  - Intervall-Newton-Schritt
  - Reelle lokale Minimierungsverfahren
- Verifikationsschritt

Für die Aufteilung der aktuellen Box und die Beschleunigungsmaßnahmen wird die zweimalige stetige Differenzierbarkeit von f benötigt. Sie sei daher im folgenden vorausgesetzt.

3.2.3 Ordnung der Arbeitsliste L

Durch die Ordnung der Arbeitsliste L in Algorithmus 3.1 wird festgelegt, welche Box beim Start eines Schleifendurchlaufs als nächstes behandelt wird (vgl. Schritt 3.a.). Der Ablauf des Algorithmus wird dadurch entscheidend beeinflußt. Viele verschiedene Varianten sind bislang untersucht worden, so unter anderem:

A. Älteste Box zuerst: Diese Variante wurde zuerst von Hansen [23] untersucht. Dabei ist die Arbeitsliste L in Form einer Warteschlange organisiert. Durch die Bisektion neu entstehende Teilboxen werden an

das Ende der Liste eingefügt, so daß die Listenoperation $Y := \mathsf{Head}(L)$ immer diejenigen Boxen an Y zuweist, die sich am längsten in der Liste befinden. Das resultierende Suchverfahren ist dann ein *oldest-first branch and bound*-Verfahren (vgl. Abschnitt 2.4.1). Dieses Vorgehen sichert eine gleichmäßige Bearbeitung aller Boxen, führt jedoch dazu, daß die beste bekannte Oberschranke $\widetilde{f}$ für das globale Minimum unter Umständen nur langsam verbessert wird, mithin also zu viele Boxen bearbeitet werden, da der Suchbaum nicht rasch genug beschnitten wird.

B. Box mit größtem Durchmesser zuerst: Auch diese Variante wurde von Hansen [23] betrachtet. Hier wird die Arbeitsliste so geordnet, daß die Durchmesser $d(Y)$ aller Boxen $Y \in L$ monoton fallend sind. Diese Methode sichert ebenfalls eine gleichmäßige Unterteilung aller Boxen, hat aber aus den gleichen Gründen wie die Ältestensuche den Nachteil, daß unter Umständen zu viele Boxen bearbeitet werden. Zur Speicherung der Listenelemente wird als Datenstruktur eine sortierte Liste verwendet.

C. Neueste Boxen zuerst: Die Idee, als nächste Box die jeweils jüngsten Boxen zu behandeln, ist mit der Datenstruktur eines *stacks* (Stapel) zu realisieren. Sie wurde von verschiedenen Autoren untersucht (so etwa [27], [37] oder [66]). Im Verlaufe des Algorithmus neu entstehende Teilboxen werden an den Beginn der Arbeitsliste L eingefügt. Somit werden Boxen Y so lange untersucht, bis alle entstehenden Teilboxen das Abbruchkriterium erfüllen oder aus der Arbeitsliste gestrichen sind. Erst dann wird zur nächsten Box übergegangen. Durch dieses Vorgehen wird eine Tiefensuche (*depth-first branch and bound*-Verfahren) realisiert. Da auch bei dieser Strategie möglicherweise die beste bekannte Oberschranke $\widetilde{f}$ für das globale Minimum nur langsam verbessert wird, ist die Tiefensuche nicht in jedem Fall das beste Suchverfahren. Wichtiger Vorteil der Tiefensuche gegenüber anderen Suchverfahren ist jedoch, daß die Länge der Arbeitsliste L klein bleibt, da vorhandene Boxen vollständig bearbeitet werden. Die Tiefensuche kann also Probleme lösen, die mit anderen Suchverfahren aufgrund mangelnden Speicherplatzes nicht mehr bearbeitet werden können. Die Tiefensuche ist ferner für andere Problemklassen gut geeignet, bei der die Arbeitsliste nicht sortiert wird, z. B. verifizierte Nullstellensuche: Hier sind die einzelnen Boxen in der Arbeitsliste weitestgehend voneinander unabhängig.

D. Box mit minimalem $\underline{F_Y}$ zuerst: Während des Ablaufs des Algorithmus werden nicht nur neue Boxen Y erzeugt, sondern auch noch Einschließungen für deren Funktionswerte F_Y berechnet. Die Oberschranke $\overline{F_Y}$ wird dazu verwendet, direkt die beste bekannte Oberschranke $\widetilde{f}$ für das globale Minimum zu verbessern. Um die Wahrscheinlichkeit für eine rasche Verbesserung von $\widetilde{f}$ zu erhöhen, wird die Arbeitsliste L nach aufsteigender Unterschranke $\underline{F_Y}$ der Boxen $Y \in L$ sortiert, zuvorderst in L befindet sich also die Box Y mit minimalem $\underline{F_Y}$. Diese Sortierung von L führt auf eine Bestensuche (*best-first branch and bound*-Verfahren). Sie wurde zuerst von Skelboe [74] untersucht, ferner auch von [24], [65] und [68]. Eine geeignete Datenstruktur zur Verwaltung von L ist eine sortierte Liste.

Daß die Bestensuche anderen Suchmethoden überlegen ist, läßt sich folgendermaßen motivieren: Sei Y die vorderste Box in L. Ist der Durchmesser $d(Y)$ klein, so approximiert nach Satz 1.1 die Intervallauswertung $F(Y)$ den Wertebereich $f(Y)$ gut. Somit ist Y wahrscheinlich eine Box, die eine globale Minimalstelle enthält, in jedem Fall wird aber $\widetilde{f}$ entscheidend verbessert. Im Mittelpunktstest können anschließend viele Boxen aus L gelöscht werden. Ist hingegen der Durchmesser $d(Y)$ relativ groß, so unterschätzt $\underline{F_Y}$ die Unterschranke $\underline{f(Y)}$ des Wertebereichs wahrscheinlich deutlich. In diesem Fall wird daher als nächstes eine Box Y bearbeitet, die noch weiter geteilt werden muß, um genauere Informationen über den Wertebereich von f zu erhalten.

Vergleicht man den Aufwand zur Verwaltung der Datenstrukturen für die vier möglichen Ordnungen der Arbeitsliste, so stellt man fest, daß bei den Varianten B. und D. (sortierte Liste) das Einfügen eines neuen Elementes in die Liste aufwendiger ist als bei den Varianten A. (Warteschlange) und C. (Stapel). Dieser Nachteil wird bei der Variante D. dadurch wieder wett gemacht, daß hier der Mittelpunktstest sehr effizient durchzuführen ist, da nur das erste Element $(Y, \underline{F_Y}) \in L$ mit $\underline{F_Y} > \widetilde{f}$ ermittelt werden muß. Ab diesem Element kann der gesamte Rest der Liste gelöscht werden. Auch aus diesem Grund ist die Bestensuche anderen Suchverfahren vorzuziehen. Wir verwenden in allen untersuchten Varianten des seriellen und parallelen Algorithmus stets die Sortierung D., also die Bestensuche.

3.2.4 Abbruchkriterien

Um noch einige genauere Aussagen über die unterschiedlichen Methoden zur Auswahl der nächsten zu behandelnden Box zu treffen, gehen wir zunächst auf unterschiedliche Abbruchkriterien ein. Das Abbruchkriterium entscheidet darüber, wann eine Box in Schritt 3.c.C. des Grundalgorithmus 3.1 in die Lösungsliste $\widehat{L}$ einsortiert wird, also als Kandidat für die Lösung in Frage kommt.

Ist man vornehmlich an einem Einschluß des globalen Minimums $f^\star$ interessiert, so ist sicher das Abbruchkriterium $d(F(Y)) < \epsilon_F$ geeignet, um eine Box Y in $\widehat{L}$ aufzunehmen. Will man jedoch außerdem die globalen Minimalstellen aus $X^\star$ möglichst eng einschließen, so wird man zusätzlich $d(Y) < \epsilon_X$ forden. ϵ_X und ϵ_F legen dabei absolute Schranken für den Durchmesser fest. Haben die globalen Minimalstellen sehr unterschiedliche Größenordnungen, so ist es sinnvoller, relative Schranken als Abbruchkriterium vorzugeben:

$$d_{\mathrm{rel}}(F(Y)) < \epsilon \quad \text{oder} \quad d_{\mathrm{rel}}(Y) < \epsilon \tag{3.1}$$

ϵ darf dabei auf dem Rechner nicht kleiner als die Maschinengenauigkeit gewählt werden. Durch den Bisektionsschritt in 3.b. ist die Terminierung des Algorithmus sichergestellt, da nach hinreichend vielen Teilungen die Boxen genügend klein werden. Im vollständigen seriellen Algorithmus zur verifizierten globalen Optimierung, der zum Vergleich und als Grundlage für den parallelen Algorithmus verwendet wird, wird das Abbruchkriterium 3.1 eingesetzt.

Für das Abbruchkriterium $d(F(Y)) < \epsilon_F$ wird in [5] gezeigt, daß die Ältestensuche und die Tiefensuche im günstigsten Fall (Initialisierung $\widetilde{f} := f^\star$) im wesentlichen nicht besser als die Bestensuche im allgemeinen Fall ($\widetilde{f}$ mit $+\infty$ initialisiert) arbeiten. Dies legt nahe, daß die Bestensuche die effizienteste der vier oben angegebenen Suchmethoden für die verifizierte globale Optimierung darstellt. Der serielle und somit auch der parallele Algorithmus verwenden daher die Bestensuche.

3.2.5 Aufteilung der aktuellen Box

Während des Ablaufs des Algorithmus zur globalen Optimierung müssen Boxen aufgeteilt werden, um die Qualität der Einschließung des Wertebe-

reichs zu verbessern. Engere Einschließungen erlauben einerseits eine schnellere Verbesserung von $\widetilde{f}$ und andererseits eine frühere Entscheidung, ob durch den Mittelpunktstest eine Box gelöscht werden kann. Bei der Aufteilung der aktuellen Box sind unterschiedliche Varianten denkbar bezüglich:

- Zahl der Teilungen
- Auswahl der zu teilenden Komponente(n)
- Teilungsverhältnis

In den ersten Arbeiten zur verifizierten globalen Optimierung wurde die zu untersuchende Box Y in zwei Teile aufgeteilt, also nur eine Teilung vorgenommen (Bisektion). In [24] stellt Hansen jedoch ohne nähere Begründung fest, daß drei Teilungen, also eine Aufteilung in acht Teilboxen aus Effizienzgründen vorzuziehen sind. Eingehender untersucht wird das Problem Bisektion / Multisektion in [5]. Dort wird durch systematische Experimente festgestellt, daß sich in der Praxis zwei Teilungen (Aufteilung in vier Teilboxen) als am günstigsten erweisen. Im Abschnitt 3.2.10 finden sich eingehende Untersuchungen für den in dieser Arbeit vorgestellten neuen seriellen Algorithmus.

Die Frage, *wieviele* Teilungen der aktuellen Box vorgenommen werden sollen, steht auch in engem Zusammenhang mit der Frage, *welche* Komponenten der aktuellen Box geteilt werden. Diese Fragestellung ist in [69] ausführlich betrachtet worden. Eine optimale Teilungsrichtung k wird dabei durch die Maximierung einer Bewertungsfunktion D bestimmt, so daß gilt:

$$D(k) = \max_{i=1}^{n} D(i),$$

wobei $D(i)$ durch eine der folgenden Vorschriften berechnet wird:

A.: $D(i) := d(Y_i)$

B.: $D(i) := d(\boldsymbol{\nabla} F(Y)_i) \cdot d(Y_i)$

C.: $D(i) := d(\boldsymbol{\nabla} F(Y)_i \cdot (Y_i - m(Y_i)))$

D.: $D(i) := \begin{cases} d(Y_i) & \text{falls } 0 \in Y_i \\ d(Y_i) / \min\{|y_i| \mid y_i \in Y_i\} & \text{sonst} \end{cases}$

Umfangreiche numerische Experimente zeigen, daß Vorschrift C. knapp gefolgt von Vorschrift B. die besten Ergebnisse liefert (vgl. [5],[69]). Als Vergleichskriterien wurden dabei die benötigte Rechenzeit, Anzahl der Funktions- und Gradientenauswertungen sowie die Speicherkomplexität (max. Länge der Arbeitsliste L) herangezogen. Die der Vorschrift C. zugrundeliegende Idee ist die Minimierung des Durchmessers der Funktionsauswertung über der aktuellen Box Y (vgl. auch Mittelwertform, Gl. (1.1) sowie [68]):

$$\begin{aligned} d(F(Y)) &= d(F(Y) - f(m(Y)) \\ &\approx d(\nabla F(Y) \cdot (Y - m(Y))) \\ &= d\left(\sum_{k=1}^{n} \frac{\partial F(Y)}{\partial y_k} \cdot (Y_k - m(Y_k))\right) \\ &= \sum_{k=1}^{n} d\left(\frac{\partial F(Y)}{\partial y_k} \cdot (Y_k - m(Y_k))\right) \end{aligned}$$

Wird nun diejenige Komponente k für die Teilung ausgewählt, für die $d(\frac{\partial F(Y)}{\partial y_k} \cdot (Y_k - m(Y_k)))$ maximal ist, so ist eine möglichst starke Reduzierung von $d(F(Y))$ zu erwarten.

Sollen zwei (oder mehrere) Teilungen vorgenommen werden (Multisektion), so werden die zwei (oder mehr) besten k ausgewählt. Da *a priori* keine weiteren Informationen über die zu untersuchende Funktion vorliegen, wird bezüglich der durch die Bewertungsfunktion bestimmten Komponente(n) eine Halbierung durchgeführt; andere Teilungsverhältnisse, die für bestimmte Problemstellungen günstiger sein könnten, werden nicht betrachtet.

Wir verwenden in dieser Arbeit stets die Vorschrift C. zur Bestimmung der optimalen Teilungsrichtung(en). Der nachfolgend angegebene Algorithmus 3.3 liefert die optimale Teilungsrichtung $k = \max_{i=1}^{n} D(i)$ zurück und bestimmt gleichzeitig einen Sortierungsvektor $s = (s_1, s_2, \ldots, s_n)$ mit $1 \leq s_i \leq n$ und $s_i \neq s_j$ für $i \neq j$, für den gilt:

$$D(s_1) \geq D(s_2) \geq \ldots \geq D(s_n).$$

Dieser Sortierungsvektor s wird später an verschiedenen Stellen benötigt:

- Im Bisektions- / Multisektionsschritt (Algorithmus 3.13)

- Im sortierten Newton-Schritt (Algorithmus 3.7)
- Im Boxing-Schritt (Algorithmus 3.10)

Algorithmus 3.3: OptKompSort(Y, G, s)
Bestimmung der optimalen Teilungsrichtung(en)

1. **for** $i := 1$ **to** n **do**
   $s_i := i;\ D_i := d(G_i \cdot (Y_i - m(Y_i)))$;
2. **for** $i := 1$ **to** $n-1$ **do**
   for $j := 1$ **to** $n-1$ **do**
   if $(D_{s_{j+1}} > D_{s_j})$ **then**
   $h := s_j;\ s_j := s_{j+1};\ s_{j+1} := h$;
3. **return** s;

Der nachfolgende Algorithmus 3.4 teilt eine Box Y in Abhängigkeit von dem Parameter l in zwei oder vier Teilboxen unter Verwendung des vorher mit Algorithmus 3.3 bestimmten Sortierungsvektors s.

Algorithmus 3.4: MultiSektion(Y, c, l, s, U)
Aufteilung der aktuellen Box in l Teilboxen

1. $k := s_1;\ U_1 := Y;\ U_2 := Y;\ U_{1_k} := [\underline{Y_k}, c_k];\ U_{2_k} := [c_k, \overline{Y_k}]$;
2. **if** $(l = 4)$ **then**
   a. $k := s_2;\ U_3 := U_1;\ U_4 := U_2$;
   b. $U_{1_k} := [\underline{U_{1_k}}, c_k];\ U_{3_k} := [c_k, \overline{U_{3_k}}]$;
   c. $U_{2_k} := [\underline{U_{2_k}}, c_k];\ U_{4_k} := [c_k, \overline{U_{4_k}}]$;
3. **return** s, U;

3.2.6 Einsatz von f' und f'' zur Beschleunigung des Grundalgorithmus

Der im vorigen Abschnitt beschriebene Grundalgorithmus ist ableitungsfrei, verwendet also nicht die Ableitung von f. Damit lassen sich nicht differenzierbare Funktionen erfolgreich untersuchen. Setzt man jedoch die Differenzierbarkeit von f voraus, so läßt sich das Verfahren noch stark beschleunigen, da weitere Kriterien das Verkleinern oder gar Löschen von Teilboxen ermöglichen. Im einzelnen sind dies:

1. Monotonietest (benötigt f')
2. Konkavitätstest (benötigt f'')
3. Intervall-Newton-Schritt (benötigt f'')
4. Reelle lokale Minimierungsverfahren (benötigt f')

3.2.6.1 Monotonietest

Falls die Funktion f über der zu untersuchenden Box $Y \subseteq X$ streng monoton ist, so kann Y keinen stationären Punkt von f enthalten. Daher nimmt f das globale Minimum nicht in Y an, Y kann somit von der weiteren Bearbeitung ausgeschlossen werden. Es genügt dabei, daß f bezüglich einer Variablen $x_i,\ 1 \leq i \leq n$ streng monoton ist. Zur Überprüfung der strengen Monotonie wird eine Einschließung des Gradienten $G := \boldsymbol{\nabla} F(Y)$ berechnet. Falls

$$0 \notin G_i \text{ für ein } i \in \{1, \ldots, n\}$$

gilt, so ist f streng monoton bezüglich x_i. Y muß also nicht weiter betrachtet werden. Dieser Test auf Monotonie von f wurde zuerst in [23] und [65] beschrieben.

Diese Betrachtung vernachlässigt noch den Fall, daß der die globale Minimalstelle enthaltende Rand von Y mit dem Rand der Ausgangsbox X zusammenfällt. Hier kann Y nur auf diesen Rand reduziert werden. Aus Gründen der Übersichtlichkeit ist die Randbehandlung in der folgenden Algorithmusbeschreibung weggelassen. Details zur Randbehandlung finden sich in [68].

Algorithmus 3.5: Monotonietest(G)
Der Monotonietest

1. **for** $i := 1$ **to** n **do**
   if $(0 \notin G_i)$ **then return** *true*;
2. **return** *false*;

3.2.6.2 Konkavitätstest

Der Konkavitätstest (oder auch Nichtkonvexitätstest) prüft, ob f auf der zu untersuchenden Box $Y \subseteq X$ nicht konvex ist. Hat f an einer Stelle $y \in Y$ ein lokales (und somit möglicherweise auch ein globales) Minimum, so muß f in einer Umgebung von y konvex sein. Daraus folgt, daß die Hessematrix $H = \nabla^2 f(y)$ positiv semidefinit ist. Eine notwendige Bedingung hierfür ist die Nichtnegativität aller Diagonalelemente H_{ii}, $1 \leq i \leq n$. Kann man nun für die Intervallauswertung $H := \nabla^2 F(Y)$ über der ganzen Box Y zeigen, daß gilt:

$$\overline{H_{ii}} < 0 \text{ für ein } i \in \{1, \ldots, n\},$$

so ist $\nabla^2 f(y)$ sicher nicht positiv semidefinit für alle $y \in Y$, es liegt somit kein lokales Minimum in Y vor. Y muß also nicht weiter betrachtet werden. Der nachfolgende Algorithmus für den Konkavitätstest verzichtet wiederum auf die Beschreibung der Randbehandlung (siehe [68] für Details).

Algorithmus 3.6: Konkavitätstest(H)
Der Konkavitätstest

1. **for** $i := 1$ **to** n **do**
   if $(\overline{H_{ii}} < 0)$ **then return** *true*;
2. **return** *false*;

Für den Konkavitätstest alleine würde sich der recht hohe Aufwand der Auswertung der Hessematrix nicht lohnen. Da $H = \nabla^2 F(Y)$ aber auch noch im nachfolgend beschriebenen Intervall-Newton-Schritt verwendet wird, ist der zusätzliche Aufwand für den Konkavitätstest minimal.

3.2.6.3 Der sortierte Intervall-Newton-Schritt

Das Intervall-Newton-Verfahren dient zur Bestimmung aller Nullstellen eines nichtlinearen Gleichungssystems. Die zunächst naheliegende Idee, für die globale Optimierung das Intervall-Newton-Verfahren zur Bestimmung aller Nullstellen des Gradienten $\nabla f(x)$ einzusetzen, um die Menge aller stationären Punkte zu bestimmen und aus diesen die Menge der globalen Minimalstellen herauszufiltern, ist ineffizient. Die Menge der stationären Punkte ist meist wesentlich größer als die Menge der globalen Minimalstellen. In solchen Fällen muß also viel unnötige Arbeit geleistet werden.

Stattdessen wird auf die aktuell zu untersuchende Box Y ein einzelner Intervall-Newton-Schritt für das nichtlineare Gleichungssystem $\nabla f(y) = 0,\ y \in Y$ angewandt. Verschiedene Varianten des Intervall-Newton-Schritts sind in der Literatur ausführlich untersucht worden und etwa in [1], [60], [68] und [71] dargestellt.

Wir verwenden im nachfolgenden Algorithmus 3.7 den in [71] näher beschriebenen erweiterten sortierten Intervall-Newton-Gauß-Seidel-Schritt. Dabei wird zunächst $A \in I\!R^{n\times n}$ und $B \in I\!R^n$ wie folgt berechnet (R sei Näherungsinverse der Mittelpunktsmatrix der Hessematrix $H := \nabla^2 F(Y)$ über der ganzen Box Y: $R \approx (m(\nabla^2 F(Y)))^{-1}$):

$$A := R \cdot \nabla^2 F(Y) \text{ und } B := R \cdot \nabla F(m(Y)).$$

Ein Sortierungsvektor $s = (s_1, s_2, \ldots, s_n)$ mit $1 \le s_i \le n$ und $s_i \ne s_j$ für $i \ne j$ wird an den Intervall-Newton-Schritt als Parameter übergeben. Dieser Sortierungsvektor erfüllt $D(s_i) \ge D(s_{i+1}),\ 1 \le i \le n-1$, wobei D die Vorschrift zur Bestimmung der optimalen Teilungsrichtung der Box Y ist. Außerdem steuert der boolesche Parameter *SortINS*, ob tatsächlich der sortierte oder der herkömmliche Newtonschritt durchgeführt wird. Diese Wahlmöglichkeit wird für die numerischen Vergleiche in Abschnitt 3.2.10 benötigt.

Wir verwenden die Vorschrift C. aus Abschnitt 3.2.5:

$$D(i) := d(\nabla F(Y)_i \cdot (Y_i - m(Y_i))).$$

Das Ergebnis $N(Y)$ des eigentlichen Berechnungsschritts ergibt sich dann als

Für $i = 1, \ldots, n$:

$$Y_{s_i} := \left(m(Y)_{s_i} - \left(B_{s_i} + \sum_{\substack{j=1 \\ j \neq s_i}}^{n} A_{s_i,j} \cdot (Y_j - m(Y)_j) \right) \Big/ A_{s_i s_i} \right) \cap Y_{s_i} \quad (3.2)$$

$N(Y) := Y$

Tritt bei der erweiterten Intervalldivision durch $A_{s_i s_i}$ ein Splitting $Z = (Z^1 \mid Z^2)$ für ein $i \in \{1, \ldots, n\}$ auf, so wird der zweite entstehende Teil Z^2 abgespeichert und der erste entstehende Teil Z^1 zur Berechnung für die weiteren i verwendet. Daher entstehen maximal $n + 1$ Teilboxen aus dem Intervall-Newton-Schritt. Um die resultierenden Teilboxen möglichst klein zu halten, werden die Komponenten von $N(Y)$ nicht in aufsteigender Reihenfolge $i = 1, \ldots, n$ ermittelt, sondern zunächst diejenigen Komponenten i, für die kein Splitting auftritt ($0 \notin A_{s_i s_i}$). Dabei wird durch den Schnitt mit dem ursprünglichen Intervall Y_i das Ergebnisintervall $N(Y)_i$ erheblich verkleinert. Anschließend werden die Komponenten i ermittelt, bei denen ein Splitting auftreten kann ($0 \in A_{s_i s_i}$). Bei der Berechnung dieser Komponenten wird bereits von den übrigen verkleinerten Komponenten profitiert.

Für das Ergebnis des Intervall-Newton-Schritts gibt es daher drei Möglichkeiten:

1. Im günstigsten Fall läßt sich zeigen, daß Y keine Nullstelle von $\nabla f(y)$ enthält, also auch keinen stationären Punkt von f. Y kann daher von der weiteren Bearbeitung ausgeschlossen werden.

2. Es wird eine (meist erheblich verkleinerte) Box $Z \subseteq Y$ zurückgeliefert. Das Gebiet $Y \setminus Z$ kann keinen stationären Punkt von f enthalten. Die in den vorigen Abschnitten beschriebenen Tests (Mittelpunkts-, Monotonie- und Konkavitätstest) sind über Z aufgrund der geringeren Überschätzung bei der Intervallauswertung effizienter durchführbar.

3. Es werden p Teilboxen $V_1, \ldots, V_p$ zurückgeliefert, wobei $2 \leq p \leq n+1$ gilt. Nur in ungünstigen Fällen gilt dabei: $\bigcup_{j=1}^{k} V_i = Y$, in den meisten Fällen können auch hier größere Teile von Y entfernt werden.

Die wesentlichen Eigenschaften des Intervall-Newton-Gauß-Seidel-Schritts

bzw. des später benötigten Intervall-Newton-Verfahrens lassen sich in folgendem Satz zusammenfassen:

Satz 3.1 *Sei $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ eine stetig differenzierbare Funktion und sei $Y \in I\mathbb{R}^n$ mit $Y \subseteq D$. Dann hat die durch 3.2 definierte Box $N(Y)$ die folgenden Eigenschaften:*

1. *Jede Nullstelle $y^\star \in Y$ von ∇f liegt auch in $N(Y)$.*
2. *Falls $N(Y) = \emptyset$, so enthält Y keine Nullstelle von ∇f.*
3. *Falls $N(Y) \overset{\circ}{\subset} Y$, so existiert genau eine Nullstelle von ∇f in $N(Y)$ bzw. Y.*

Beweis: Siehe [71].

Der vollständige sortierte Intervall-Newton-Gauß-Seidel-Schritt sieht dann wie folgt aus:

Algorithmus 3.7: NewtonSchritt$(Y, H, s, SortINS, V, p)$
Sortierter Intervall-Newton-Gauß-Seidel-Schritt

1. $p := 0;\ c := m(Y);\ R := (m(H))^{-1};$
2. $A := R \cdot H;\ B := R \cdot \nabla F(c);\ Y_c := Y - c;$
3. **for** $k := 1$ **to** n **do**
   a. **if** $(SortINS)$ **then** $i := s_k$; **else** $i := k$;
   b. **if** $(0 \in A_{ii})$ **then next**$_i$;
   c. $Y_i := \left(c_i - \left(B_i + \sum_{\substack{j=1 \\ j \neq s_i}}^{n} A_{ij} \cdot Y_{c_j} \right) \Big/ A_{ii} \right) \cap Y_i;$
   d. **if** $(Y_i = \emptyset)$ **then return**;
   e. $Y_{c_i} := Y_i - c_i;$

4. **for** $k := 1$ **to** n **do**
   a. **if** $(SortINS)$ **then** $i := s_k$; **else** $i := k$;
   b. **if** $(0 \notin A_{ii})$ **then next**$_i$;
   c. $Z := \left(c_i - \left(B_i + \sum_{\substack{j=1 \\ j \neq s_i}}^{n} A_{ij} \cdot Y_{c_j}\right) \Big/ A_{ii}\right) \cap Y_i$;
   d. **if** $(Z = \emptyset)$ **then return**;
   e. $Y_i := Z_1; Y_{c_i} := Y_i - c_i$;
   f. **if** $(Z_2 \neq \emptyset)$ **then**
      $p := p + 1;\ V_p := Y;\ V_{p_i} := Z_2$;
5. $p := p + 1;\ V_p := Y$;
6. **return** V, p;

Nach dem Intervall-Newton-Schritt, der sehr aufwendig ist, werden im Algorithmus zur globalen Optimierung die anderen Tests und der Bisektionsschritt angewandt, um die neu entstandene(n) Box(en) zu reduzieren.

3.2.6.4 Reelle lokale Minimierungsverfahren

Wenn im Grundalgorithmus 3.1 durch Bisektion eine neue Box Y entsteht, so wird durch eine intervallmäßige Auswertung von f über dem Mittelpunkt der Box versucht, die beste bekannte Oberschranke $\widetilde{f}$ für das globale Minimum nach unten zu verbessern und somit beim anschließenden Mittelpunktstest mehr Boxen aus der Arbeits- und Lösungsliste zu entfernen. Der Mittelpunkt ist dabei rein zufällig gewählt, es könnte auch jeder andere Punkt in der Box zur Auswertung von f herangezogen werden. Um eine möglichst gute Verbesserung von $\widetilde{f}$ zu erreichen, können klassische reelle lokale Minimierungsverfahren wie etwa eine reelle Newtoniteration oder das cg-Verfahren eingesetzt werden. Dabei wird, ausgehend vom Mittelpunkt $m(Y)$ mit reeller Gleitkommaarithmetik ein Punkt $y \in Y$ bestimmt, an dem f einen kleineren Funktionswert annimmt. Wertet man nun $F(y)$ intervallmäßig aus, so stellt in fast allen Fällen $\overline{F(y)}$ eine bessere Oberschranke für das globale Minimum als $\overline{F(m(Y))}$ dar. Auch im Falle einer Verbesserung von $\widetilde{f}$ wird durch $\overline{F(y)}$ eine stärkere Verbesserung erreicht.

Ob sich der zusätzliche Aufwand für das reelle Minimierungsverfahren lohnt, wird von einigen Faktoren bestimmt: Durch einen guten Wert für $\widetilde{f}$ kann zum einen die Länge von L kleiner gehalten werden, da weniger Boxen in die Arbeitsliste eingefügt werden müssen. Zum anderen wird möglicherweise verhindert, daß Boxen unnötigerweise bearbeitet werden, dies hängt jedoch von der Ordnung der Arbeitsliste L ab.

Bei der Verwendung der Ordnung D. (Box mit minimalem $\underline{F(Y)}$ zuerst; Bestensuche) läßt sich zeigen (vgl. [5]), daß unabhängig von der Qualität von $\widetilde{f}$ keine Box Y im Bisektionsschritt geteilt wird, deren Unterschranke der Funktionsauswertung $\underline{F(Y)}$ größer ist als $f^\star + \epsilon$ (ϵ ist der für das Abbruchkriterium verwendete Wert). Dies bedeutet, daß eine Verbesserung von $\widetilde{f}$ nur zu einer effizienteren Listenverwaltung (schnelleres Einfügen und Mittelpunktstest) beiträgt, nicht jedoch zu einer verringerten Gesamtzahl an bearbeiteten Boxen führt.

Anders sieht es bei den drei anderen betrachteten möglichen Ordnungen der Arbeitsliste L aus. Hier wird durch die raschere Verbesserung von $\widetilde{f}$ erreicht, daß in der Regel tatsächlich weniger Boxen bearbeitet werden. Ebenfalls wird der Aufwand für die Listenverwaltung verringert.

Da in dieser Arbeit für den seriellen und parallelen Algorithmus die Bestensuche verwendet wird, wird auf den Einsatz von reellen lokalen Minimierungsverfahren verzichtet.

3.2.7 Verifikationsschritt

Am Ende des Verfahrens ist die Arbeitsliste L leer und somit vollständig abgearbeitet. Die Lösungsliste $\widehat{L}$ enthält kleine Teilboxen, die durch die verschiedenen Tests nicht gelöscht werden konnten. Vor dem Verifikationsschritt läßt sich lediglich folgendes aussagen:

$$f^\star \in F^\star := [\min\{\underline{F_X} : (X, \underline{F_X}) \in \widehat{L}\}, \widetilde{f}] \quad \text{und} \quad X^\star \subseteq \bigcup_{(X,\underline{F_X})\in\widehat{L}} X.$$

Ob die Boxen in $\widehat{L}$ tatsächlich globale Minimalstellen enthalten, wird nun untersucht.

Leider ist keine Möglichkeit bekannt, den Nachweis der Eindeutigkeit einer *globalen* Minimalstelle allgemein auf dem Rechner zu führen. Stattdessen

wird hier für alle Boxen $Y \in \widehat{L}$ versucht, die Existenz und *lokale* Eindeutigkeit eines Minimums nachzuweisen. Dazu wird als eine von zwei hinreichenden Bedingungen geprüft, ob sich

$$N(Y) \overset{\circ}{\subset} Y \tag{3.3}$$

zeigen läßt. Dies sichert wegen Satz 3.1 die Existenz und Eindeutigkeit eines stationären Punktes von f in Y, also einer Nullstelle von ∇f. Als zweite Bedingung wird getestet, ob sich

$$S \cdot Z \overset{\circ}{\subset} Z \tag{3.4}$$

mit $S := I - \kappa^{-1} \cdot H$, $H := \nabla^2 F(Y)$, $\|H\|_\infty \leq \kappa \in \mathbb{R}$ für ein $Z \in I\mathbb{R}^n$ erfüllen läßt. Unter diesen Voraussetzungen hat Ratz in [68] gezeigt, daß gilt: Alle symmetrischen Matrizen $A \in H$ sind positiv definit.

Lassen sich die beiden Bedingungen (3.3) und (3.4) erfüllen, so ist die Existenz einer in der Box Y eindeutigen Minimalstelle bewiesen. Enthält die Lösungsliste $\widehat{L}$ nur eine Box Y und läßt sich für diese Box ein lokaler Minimierer nachweisen, so handelt es sich auch um die einzige globale Minimalstelle bezogen auf die Ausgangsbox X. Falls $\widehat{L}$ mehrere Boxen enthält und sich für jede Box die lokale Eindeutigkeit der lokalen Minimalstelle zeigen läßt, so enthält jede Box in $\widehat{L}$ genau einen Kandidaten für eine globale Minimalstelle.

Falls der Nachweis der lokalen Eindeutigkeit für eine Box $Y \in \widehat{L}$ nicht gelingt, so ist dies kein Grund, Y aus der Lösungsliste zu entfernen:

- Y kann mehrere globale Minimalstellen enthalten, die entweder durch das vorgegebene Abbruchkriterium oder durch Erreichen der Maschinengenauigkeit nicht getrennt werden können.
- Y hat einen zu kleinen Durchmesser, so daß $N(Y) \overset{\circ}{\subset} Y$ nicht erfüllbar ist.
- Durch eine zu starke Überschätzung bei der Intervallauswertung von $N(Y)$ wird $N(Y) \overset{\circ}{\subset} Y$ nicht erfüllt.

In jedem Fall kann Y eine globale Minimalstelle enthalten, nur der Nachweis der lokalen Eindeutigkeit schlägt fehl.

Der nachfolgende Algorithmus versucht, die lokale Eindeutigkeit einer Minimalstelle in der Box Y nachzuweisen. Das Ergebnis wird in der Variable *unique* zurückgeliefert: Ein erfolgreicher Nachweis der lokalen Eindeutigkeit wird dabei durch Eins angezeigt, ansonsten wird Null zurückgegeben.

Algorithmus 3.8: Verifiziere(Y, *unique*)
Verifikationsschritt

1. **if** $(d(Y) = 0)$ **then** $unique := 1$; **return** Y, *unique*;
2. $k_{max} := 10$; $k := 0$; $V := Y$; $\epsilon := 0.25$; $unique := 0$;
3. **for** $i := 1$ **to** n **do** $s_i := i$;
4. **while** $((unique = 0)$ **and** $(k < k_{max}))$ **do**
   a. $Y_{old} :=$ EpsAufbl(Y, ϵ); $H := \boldsymbol{\nabla}^2 F(Y_{old})$; $k := k + 1$;
   b. NewtonSchritt$(Y_{old}, H, s, false, V, p)$;
   c. **if** $(p \neq 1)$ **then exit**$_{while}$;
   d. $Y := V_1$;
   e. **if** $(Y \overset{\circ}{\subset} Y_{old})$ **then** $unique := 1$;
   f. **if** $(Y = Y_{old})$ **then** $\epsilon := 8 \cdot \epsilon$;
5. **if** $(unique = 1)$ **then**
   a. Berechne $\kappa \geq \|H\|_\infty$; $S := I - \kappa^{-1} \cdot H$;
   b. **for** $i := 1$ **to** n **do** $Z_i := [-1, 1]$;
   c. $k_{max} := 10$; $k := 0$; $\epsilon := 0.25$;
   d. **repeat**
      $U :=$ EpsAufbl(Z, ϵ); $Z := S \cdot U$; $k := k + 1$; $\epsilon := 8 \cdot \epsilon$;
      until $((Z \overset{\circ}{\subset} U)$ **or** $(k = k_{max}))$;
   e. **if** $($**not** $(Z \overset{\circ}{\subset} U))$ **then** $unique := 0$;
6. **if** $(unique = 0)$ **then** $Y := V$;
7. **return** Y, *unique*;

3.2.8 *Toolbox*-ähnlicher Algorithmus

Der nachfolgend angegebene Algorithmus, der als Vergleichsgrundlage für den im folgenden Abschnitt 3.2.9 angegebenen neuen seriellen Algorithmus dient, basiert auf dem in der *C++ Toolbox for Verified Computing I* [21] angegebenen Algorithmus. Wir bezeichnen diesen Algorithmus daher als *Toolbox*-ähnlichen Algorithmus. Von dem ursprünglichen Algorithmus unterscheidet er sich durch:

1. Verwendung der Vorschrift C. statt der Vorschrift A. (vgl. Abschnitt 3.2.5) zur Aufteilung der aktuellen Box
2. Einführung eines Parameters l für Bisektion ($l = 2$) oder Multisektion ($l = 4$) mit Aufteilung in vier Teilboxen
3. Verwendung des sortierten Intervall-Newton-Schritts (wird durch die boolesche Variable *SortINS* aktiviert)

Mit den Steuerungsmöglichkeiten in Punkt 2. und 3. werden verschiedene Varianten des *Toolbox*-ähnlichen Algorithmus selektiert, die im Abschnitt 3.2.10 zum Vergleich mit dem neuen seriellen Algorithmus herangezogen werden.

Der *Toolbox*-ähnliche Algorithmus verwendet die in den vorangegangenen Abschnitten angegebenen Beschleunigungsmaßnahmen und den Verifikationsschritt. Er berechnet Einschließungen für alle globalen Minimalstellen $x^\star$ in der Startbox X und für das globale Minimum $f^\star$ mit einer vorgegebenen relativen Genauigkeit ϵ. Anschließend wird versucht, die lokale Eindeutigkeit der Minimalstellen in der Lösungsliste $\widehat{L}$ nachzuweisen. Das Ergebnis dieses Tests wird in der zweiten Komponente z der Elemente $(Y, z) \in \widehat{L}$ abgespeichert. Möglich ist dies, weil die zweite Komponente nach der Berechnungsphase nicht mehr benötigt wird: Das einschließende Intervall $F^\star$ für das globale Minimum $f^\star$ steht bereits fest. Ein erfolgreicher Verifikationsschritt, also der Nachweis der lokalen Eindeutigkeit der Minimalstelle, wird daher durch $z := 1$ gekennzeichnet. Andernfalls wird z auf Null gesetzt.

Algorithmus 3.9: ToolboxGlobOpt$(X, \epsilon, l, SortINS, F^\star, \widehat{L})$
Toolbox-ähnlicher serieller Algorithmus

1. $c := m(X)$; $\widetilde{f} := \overline{F(c)}$; $Y := X$; $L := \{\}$; $\widehat{L} := \{\}$;
2. **for** $i := 1$ **to** n **do** $s_i := i$;
3. **repeat**
   a. OptKompSort$(Y, \nabla F(Y), s)$; MultiSektion(Y, c, l, s, U);
   b. **for** $i := 1$ **to** l **do**
      A. $G := \nabla F(U_i)$; **if** (Monotonietest(G)) **then next**$_i$;
      B. $F_c := (F(c) + G \cdot (U_i - c)) \cap F(U_i)$;
      C. **if** $(\underline{F_c} > \widetilde{f})$ **then next**$_i$;
      D. $H := \nabla^2 F(U_i)$; **if** (Konkavitätstest(H)) **then next**$_i$;
      E. NewtonSchritt$(U_i, H, s, SortINS, V, p)$;
      F. **for** $j := 1$ **to** p **do**
         1. $G := \nabla F(V_j)$; **if** (Monotonietest(G)) **then next**$_j$;
         2. $F_c := (F(m(V_j)) + G \cdot (V_j - m(V_j))) \cap F(V_j)$;
         3. **if** $(\underline{F_c} \leq \widetilde{f})$ **then** $L := L + (V_j, \underline{F_c})$;
   c. *bisect* := *false*; **while** $((L \neq \{\})$ **and** (**not** *bisect*)) **do**
      A. $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$; $c := m(Y)$;
      B. $\widetilde{f} := \min\{\widetilde{f}, \overline{F(c)}\}$; Mittelpunktstest$(L, \widetilde{f})$; $F^\star := [\underline{F_Y}, \widetilde{f}]$;
      C. **if** $((d_{\mathrm{rel}}(F^\star) < \epsilon)$ **or** $(d_{\mathrm{rel}}(Y) < \epsilon))$ **then** $\widehat{L} := \widehat{L} + (Y, \underline{F_Y})$;
         else *bisect* := *true*;

   until (**not** *bisect*);
4. $(Y, \underline{F_Y}) :=$ Head$(\widehat{L})$; $F^\star := [\underline{F_Y}, \widetilde{f}]$;
5. **forall** $(Y, z) \in \widehat{L}$ **do** Verifiziere(Y, z);
6. **return** $F^\star, \widehat{L}$;

3.2.9 Ein neuer serieller Algorithmus zur verifizierten globalen Optimierung

In diesem Abschnitt wird ein neuer serieller Algorithmus zur verifizierten globalen Optimierung hergeleitet. Motivation für den neuen Algorithmus ist die Idee, die Zahl der Gradienten- und Hessematrixauswertungen zur Lösung eines Problems zu verringern unter Inkaufnahme einer möglichen Erhöhung der Zahl der Funktionsauswertungen. Vergleiche dazu auch die Anmerkungen in Anhang B. Erst durch die dort vorgeschlagenen Verbesserungen ist ein solches Vorgehen überhaupt sinnvoll. Insbesondere soll der Intervall-Newton-Schritt möglichst selten und dann nur bei erfolgversprechenden Boxen Y eingesetzt werden.

Eine weitere neuartige Modifikation ist das im folgenden Abschnitt vorgestellte *Boxing*-Verfahren zur modifizierten Aufteilung der aktuellen Box Y in bestimmten Situationen.

3.2.9.1 Das *Boxing*-Verfahren

Wird beim Ablauf des Algorithmus zur globalen Optimierung die Oberschranke für das globale Minimum $\widetilde{f}$ durch die Mittelpunktsauswertung $F(m(Y))$ stark verbessert (dies ist insbesondere in der Anfangsphase des Algorithmus der Fall), so kann es sinnvoll sein, noch nicht den aufwendigen Intervall-Newton-Schritt einzusetzen, sondern zunächst eine andere Aufteilungsstrategie zu verfolgen. Dazu wird die aktuelle Box Y zerlegt in eine kleine Umgebung Z des Mittelpunktes $m(Y)$ sowie die Restboxen $V_1, \ldots, V_{2n}$, die durch sukzessives komponentenweises „Herausschneiden“ von Z entstehen.

Abbildung 3.2 soll das Arbeitsprinzip des nachfolgend angegebenen Algorithmus 3.10 für das *Boxing*-Verfahren für den Fall $Y \in I\!I\!R^2$, also die Zerlegung in die Umgebungsbox Z und die Restboxen $V_1, \ldots, V_4$ verdeutlichen. Die Aufspaltung der Box erfolgt wiederum anhand des zuvor ermittelten Sortierungsvektors s, um zunächst in den günstigen Koordinatenrichtungen aufzuteilen. Ein ähnlicher Algorithmus zur Abspaltung von Singularitäten der Hessematrix findet sich in [68].

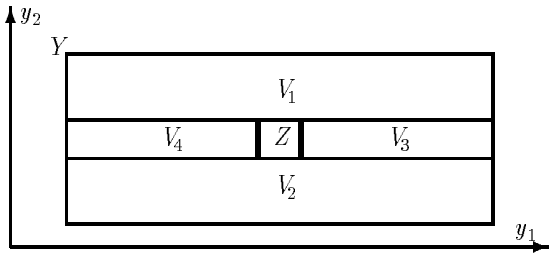


Abbildung 3.2: Aufteilung von Y durch das *Boxing*-Verfahren

Algorithmus 3.10: Boxing$(Y, c, s, \epsilon_{Box}, V, p)$
Das *Boxing*-Verfahren

1. $p := 1$;
2. **for** $i := 1$ **to** n **do**
   if $(c_i = 0)$ **then** $Z_i := (\epsilon_{Box} \cdot Y_i) \cap Y_i$;
   else $Z_i := ([1 - \epsilon_{Box}, 1 + \epsilon_{Box}] \cdot c_i) \cap Y_i$;
3. **for** $k := 1$ **to** n **do**
   a. $i := s_k$;
   b. **if** $(\underline{Y_i} \neq \underline{Z_i})$ **then**
      $r := \overline{Y_i}$; $\overline{Y_i} := \underline{Z_i}$; $V_p := Y$; $p := p + 1$; $\overline{Y_i} := r$;
   c. **if** $(\overline{Y_i} \neq \overline{Z_i})$ **then**
      $\underline{Y_i} := \overline{Z_i}$; $V_p := Y$; $p := p + 1$;
   d. $Y_i := Z_i$;
4. $V_p := Y$;
5. **return** V, p;

3.2.9.2 Selektiver Einsatz des Intervall-Newton-Schritts

Der erweiterte Intervall-Newton-Schritt bei der globalen Optimierung ist derjenige Teil des Gesamtalgorithmus, der die größten Fortschritte bei der

Reduzierung der Boxen bringen kann, da in der Regel quadratische Konvergenz in einer Umgebung der stationären Punkte vorliegt. Gleichzeitig ist er jedoch wegen der notwendigen Bestimmung der Hessematrix aufwendig durchzuführen.

Statische Entscheidung über den Einsatz des Intervall-Newton-Schritts Viele Autoren (so etwa [5] oder [67]) schlagen deshalb vor, den Intervall-Newton-Schritt nur bei genügend kleinen Boxen Y mit $d(Y) < \epsilon_{Newton}$ einzusetzen, wobei etwa $\epsilon_{Newton} = 0.01$ gewählt wird. Dieses Vorgehen steigert zwar die Anzahl der notwendigen Iterationen des Gesamtalgorithmus, soll jedoch gleichzeitig die Wahrscheinlichkeit eines erfolgreichen Intervall-Newton-Schritts erhöhen. Das Kriterium für den Einsatz des Intervall-Newton-Schritts vom *absoluten* Durchmesser $d(Y)$ abhängig zu machen, kann jedoch nicht allzu günstig sein, da eine Skalierung des zu untersuchenden Problems zu einem stark veränderten Ablauf des Algorithmus führt, somit also entweder eine Verbesserung oder eine Verschlechterung gegenüber dem Ausgangsproblems nach sich ziehen kann. Sinnvoller erscheint da zunächst ein Kriterium, das den *relativen* Durchmesser von Y verwendet. Dazu wird Schritt 3.b.C. des ursprünglichen *Toolbox*-ähnlichen Algorithmus 3.9 ersetzt durch:

3.b.C. **if** $(\underline{F_c} > \widetilde{f}$ **or** $d_{\mathrm{rel}}(U_i) \geq \epsilon_{Newton})$ **then next**$_i$;

Führt man jedoch einige Testrechnungen mit verschiedenen Werten für ϵ_{Newton} durch, so zeigt es sich, daß ein festes ϵ_{Newton} in einigen Testproblemen zu Verschlechterungen führt. In anderen Testproblemen werden Verbesserungen erzielt, jedoch ist das optimale ϵ_{Newton} für verschiedene Probleme unterschiedlich. Auch gibt es Testprobleme, bei denen es nicht sinnvoll ist, den Einsatz des Intervall-Newton-Schritts zu beschränken. Der nicht modifizierte Algorithmus 3.9 schneidet also für diese Probleme am besten ab.

Die nachfolgende Tabelle 3.1 mit den Laufzeiten (in STU[1]) einiger Testprobleme aus Anhang A soll diese Aussagen unterstreichen. Alle Testprobleme wurden mit dem *Toolbox*-ähnlichen Algorithmus 3.9 mit den Parametern $l = 4$ und *SortINS* = *true* (sortierter Newtonschritt) gerechnet. Mit anderen Parametern ergibt sich kein wesentlich verschiedenes Laufzeitverhalten. $\epsilon_{Newton} = \infty$ entspricht dem ursprünglichen Algorithmus 3.9.

[1] Standard Time Unit; 1 STU $\triangleq$ Berechnungszeit für 1000 reelle Auswertungen der Shekel-5-Funktion (S5).

Problem	$\epsilon_{Newton} = \infty$	$\epsilon_{Newton} = 1$	$\epsilon_{Newton} = 0.1$	$\epsilon_{Newton} = 0.01$
SHCB	**1.00**	2.54	2.59	2.95
G10	157.78	**22.57**	26.76	33.48
GP	**57.93**	58.61	161.72	264.07
H6	47.28	**28.30**	33.09	39.82
GEO2	45.99	95.51	31.77	**28.90**
S2.7	113.22	113.31	**84.79**	105.79
L3	549.55	340.67	**260.15**	291.68
HM3	551.13	308.75	261.59	**244.38**

Tabelle 3.1: Laufzeiten (in STU) für feste Werte von ϵ_{Newton}

In jeder Zeile ist das beste Ergebnis fettgedruckt. Man erkennt, daß keine bestimmte Wahl von ϵ_{Newton} zu bevorzugen ist.

Adaptive Entscheidung über den Einsatz des Intervall-Newton-Schritts Wie an der Ergebnistabelle im vorigen Abschnitt zu erkennen ist, bringt der selektive Einsatz des Intervall-Newton-Schritts in vielen Fällen eine signifikante Reduktion der Laufzeit, jedoch ist *a priori* nicht klar, wie ϵ_{Newton} sinnvoll zu wählen ist. Daher schlagen wir eine adaptive Strategie vor, die ϵ_{Newton} zur Laufzeit an das jeweilige Verhalten des Algorithmus anpaßt. Die grundlegende Idee ist dabei:

- Bringt der Einsatz eines Intervall-Newton-Schritts keinen Erfolg, so wird ϵ_{Newton} (bis zu einer gewissen Unterschranke) verringert. Dadurch wird der Intervall-Newton-Schritt im weiteren Programmablauf seltener eingesetzt.
- Wird hingegen die aktuelle Box Y durch Multisektion geteilt, so wird ϵ_{Newton} erhöht, bis eine Oberschranke erreicht ist. In Zukunft wird der Intervall-Newton-Schritt also häufiger eingesetzt werden.

Den Begriff „erfolgreicher“ Intervall-Newton-Schritt fassen wir genauer in der folgenden Definition:

Definition 3.1 *Seien $V_1, \ldots, V_p$ die durch den erweiterten Intervall-Newton-Schritt (Algorithmus 3.7) berechneten Teilboxen der zu untersuchenden aktuellen Box Y. Gilt $\bigcup_{i=1}^{p} V_i \neq Y$, so nennen wir den Intervall-Newton-Schritt* erfolgreich.

Bemerkung: Ein Intervall-Newton-Schritt ist also dann nicht erfolgreich, wenn entweder die Ausgangsbox Y unverändert zurückgeliefert wird ($Y = V_1$) oder bei eventuell auftretenden erweiterten Intervalldivisionen $W = (W^1 \mid W^2)$ ausschließlich der Fall $\overline{W^1} = \underline{W^2}$ auftritt, im Prinzip also nur Bisektion durchgeführt wird.

Der folgende Algorithmus 3.11 erweitert den Algorithmus 3.7 so, daß die boolesche Variable *fortschritt* anzeigt, ob der Intervall-Newton-Schritt erfolgreich war oder nicht. *fortschritt* wird dann vom aufrufenden Programm ausgewertet und zur Anpassung von ϵ_{Newton} verwendet.

Algorithmus 3.11: NewtonSchritt($Y, H, s, SortINS, V, p, fortschritt$)
Sortierter Intervall-Newton-Schritt mit Fortschrittsindikator

1. $p := 0;\ c := m(Y);\ R := (m(H))^{-1};\ fortschritt := false;$
2. + 3. wie in Algorithmus 3.7.
4. **for** $k := 1$ **to** n **do**
   a. – e. wie in Algorithmus 3.7.
   f. **if** $(Z_2 \neq \emptyset)$ **then**
      A. $p := p + 1;\ V_p := Y;\ V_{p_i} := Z_2;$
      B. **if** $(\overline{Z_1} < \underline{Z_2}$ **or** $\overline{Z_2} < \underline{Z_1})$ **then** $fortschritt := true;$
5. $p := p + 1;\ V_p := Y;$
6. **if** $(p = 1$ **and** $V_1 \neq Y)$ **then** $fortschritt := true;$
7. **return** $V, p, fortschritt;$

3.2.9.3 Komprimieren der Lösungsliste

Während des Ablaufs des Algorithmus zur verifizierten globalen Optimierung ist es möglich, daß Teilboxen Y in die Lösungsliste eingefügt werden, deren Schnitt mit anderen Boxen in der Lösungsliste nicht leer ist. Diese Boxen haben also einen gemeinsamen Rand. Häufig enthält eine der beiden Boxen mit gemeinsamem Rand gar keine globale Minimalstelle. Es kann nämlich folgender Fall vorliegen:

Sei $Y \in \widehat{L}$ eine Box, die eine globale Minimalstelle $x^\star \in X$ enthält. Sei $V \in \widehat{L}$ eine Box mit $V \cap Y \neq \emptyset$. V ist etwa dann in $\widehat{L}$ aufgenommen worden, falls $d_{\mathrm{rel}}(F(V)) < \epsilon$ gilt. Es kann aber gelten: $f(x^\star) + \epsilon \cdot \langle F(V) \rangle \geq f(v) > f(x^\star)$ für alle $v \in V$. Der Algorithmus kann also das globale Minimum nur mit einer relativen Genauigkeit von ϵ bestimmen. Verläuft f „flach“ in der Nähe von $x^\star$, so kann der oben beschriebene Fall auftreten. Es ist daher zweckmäßig, am Ende des Algorithmus diese Boxen wieder zu vereinigen, um die Zahl der Boxen in der Lösungsliste zu reduzieren. Der abschließende Verifikationsschritt ist in der Lage, durch die Kontraktion um das globale Minimum $x^\star$ die Boxen wieder stark zu verkleinern. Es kann dann für die oben beschriebene Situation meist erreicht werden, daß nach dem Verifikationsschritt der Schnitt aller Boxen in der Lösungsliste mit der Box V leer ist. Algorithmus 3.12 sorgt für die vorgeschlagene Komprimierung der Lösungsliste.

Algorithmus 3.12: Komprimiere($\widehat{L}$)
Komprimieren der Lösungsliste $\widehat{L}$

1. $L := \widehat{L}$; $\widehat{L} := \{\}$;
2. **while** $(L \neq \{\})$ **do**
   a. $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$;
   b. **do**
      A. *again* := *false*; $\tilde{L} := L$; $L := \{\}$;
      B. **while** $(\tilde{L} \neq \{\})$ **do**
         1. $(V, \underline{F_V}) :=$ Head$(\tilde{L})$;
         2. **if** $(V \cap Y = \emptyset)$ **then** $L := L + (V, \underline{F_V})$;
            else $Y := Y \cup V$; *again* := *false*;
         3. $\tilde{L} := \tilde{L} - (V, \underline{F_V})$;
      while (*again*);
   c. $\widehat{L} := \widehat{L} + (Y, \underline{F_Y})$;
3. **return** $\widehat{L}$;

3.2.9.4 Der endgültige serielle Algorithmus

Wir haben in den vorangegangenen Abschnitten alle notwendigen Bausteine für den neuen seriellen Algorithmus vorgestellt. Diese sollen nun zusammengesetzt werden. Die Darstellung des Algorithmus ist dabei in ein Hauptprogramm SerielleGlobOpt() und ein Unterprogramm BearbeiteBox() zur Bearbeitung der aktuellen Box Y zweigeteilt, da BearbeiteBox() auch im parallelen Algorithmus eingesetzt wird. Im neuen seriellen Algorithmus wird stets der sortierte Intervall-Newton-Schritt verwendet. Es werden Einschließungen für alle globalen Minimalstellen $x^\star$ in der Startbox X und für das globale Minimum $f^\star$ mit einer vorgegebenen relativen Genauigkeit ϵ berechnet. Anschließend wird versucht, die lokale Eindeutigkeit der Minimalstellen in der Lösungsliste $\widehat{L}$ nachzuweisen. Das Ergebnis dieses Tests wird in der zweiten Komponente z der Elemente $(Y,z) \in \widehat{L}$ abgespeichert (vgl. die Anmerkungen bei Algorithmus 3.9). *MaxReal* sei die größte auf der Maschine noch darstellbare Gleitkommazahl.

Algorithmus 3.13: SerielleGlobOpt$(X, \epsilon, l, F^\star, \widehat{L})$
Neuer serieller Algorithmus zur globalen Optimierung

1. $\epsilon_{Newton} := 1;\ \widetilde{f} := MaxReal;\ Y := X;\ L := \{\};\ \widehat{L} := \{\};$
2. **do**
   a. BearbeiteBox$(\epsilon, \epsilon_{Newton}, l, Y, F(Y), \nabla F(Y), L, \widetilde{f});$
   b. Mittelpunktstest$(L, \widetilde{f});\ \ weiter = false;$
   c. **while** $((L \neq \{\})$ **and** (**not** *weiter*)) **do**
      A. $(Y, \underline{F_Y}) :=$ Head$(L);\ L := L - (Y, \underline{F_Y});\ F^\star := [\underline{F_Y}, \widetilde{f}];$
      B. **if** $((d_{\mathrm{rel}}(F^\star) < \epsilon)$ **or** $(d_{\mathrm{rel}}(Y) < \epsilon))$ **then** $\widehat{L} := \widehat{L} + (Y, \underline{F_Y});$
         else $weiter := true;$
   until (**not** *weiter*);
3. $(Y, \underline{F_Y}) :=$ Head$(\widehat{L});\ F^\star := [\underline{F_Y}, \widetilde{f}];$
4. Komprimiere$(\widehat{L});$
5. **forall** $(Y,z) \in \widehat{L}$ **do** Verifiziere$(Y,z);$
6. **return** $F^\star, \widehat{L};$

Das Hauptprogramm SerielleGlobOpt() entspricht bis auf die Initialisierung von ϵ_{Newton} und $\widetilde{f}$ im wesentlichen dem ursprünglichen *Toolbox*-ähnlichen Algorithmus 3.9. Zu Beginn wird $\widetilde{f}$ auf *MaxReal* gesetzt, damit im ersten Schritt auf jeden Fall das Boxing-Verfahren ausgeführt wird. Nach der Bearbeitung einer Box Y in BearbeiteBox(), was zu einer veränderten Arbeitsliste L führt, wird der Mittelpunktstest durchgeführt. Anschließend wird eine neue Box Y aus der Arbeitsliste geholt. Falls das Abbruchkriterium erfüllt ist, wird Y in der Lösungsliste $\widehat{L}$ abgespeichert, andernfalls erfolgt die Weiterbearbeitung. Zum Schluß wird nach der Komprimierung der Lösungsliste $\widehat{L}$ für alle Boxen in der Lösungsliste der Verifikationsschritt durchgeführt.

Algorithmus 3.14: BearbeiteBox$(\epsilon, \epsilon_{Newton}, l, Y, F_Y, G, L, \widetilde{f})$
Bearbeitung der aktuellen Box Y

1. **if** (Monotonietest(G)) **then return** $L, \widetilde{f}$;
2. $c := m(Y)$; $F_c := F(c)$; $F_Y := (F_c + G \cdot (Y - c)) \cap F_Y$;
3. **if** $(\widetilde{f} < \underline{F_Y})$ **then return** $L, \widetilde{f}$;
4. OptKompSort(Y, G, s);
5. **if** $(\overline{F_c} < \widetilde{f})$ **then**
   a. $\widetilde{f}_{old} := \widetilde{f}$; $\widetilde{f} := \overline{F_c}$;
   b. **if** $(\widetilde{f}_{old} - \underline{F_Y} > 2(\widetilde{f} - \underline{F_Y})$ **and** $d_{\mathrm{rel}}(Y) > 10\epsilon)$ **then**
      Boxing$(Y, c, s, 5\epsilon, V, p)$; **goto** 8;

6. **if** $(d_{\mathrm{rel}}(Y) < \epsilon_{Newton})$ **then**
   a. $H := \nabla^2 F(Y)$;
   b. **if** (Konkavitätstest(H)) **then return** $L, \widetilde{f}$;
   c. $G_c := \nabla F(c)$;
   d. $F_Y := F_Y \cap (F_c + G_c(Y-c) + \frac{1}{2}(Y-c)^T H(Y-c))$;
   e. **if** $(\widetilde{f} < \underline{F_Y})$ **then return** $L, \widetilde{f}$;
   f. NewtonSchritt$(f, Y, H, s, true, V, p, fortschritt)$;
   g. **if** (**not** *fortschritt*) **then** $\epsilon_{Newton} := \max(0.01, \frac{1}{2}\epsilon_{Newton})$;
   h. **if** $(p = 1$ **and** $V_1 = Y)$ **then** *multisektion* := *true*;
   else *multisektion* := *false*;

   else $\epsilon_{Newton} := \min(1, \frac{3}{2}\epsilon_{Newton})$; *multisektion* := *true*;

7. **if** (*multisektion*) **then** $p := l$; MultiSektion(Y, c, l, s, V);

8. **for** $j := 1$ **to** p **do**
   $\underline{F_V} := \underline{F(V_j)}$; **if** $(\underline{F_V} \leq \widetilde{f})$ **then** $L := L + (V_j, \underline{F_V})$;

9. **return** $L, \widetilde{f}$;

Die eigentlichen Neuerungen des seriellen Verfahrens zur globalen Optimierung finden sich in der Funktion BearbeiteBox(). An BearbeiteBox() werden neben der aktuellen Box Y auch deren Funktionsauswertung $F(Y)$ sowie die Gradientenauswertung $\nabla F(Y)$ übergeben. Nach dem Monotonietest wird versucht, eine engere Einschließung von $F(Y)$ durch die Mittelwertform (vgl. Gl. 1.1) zu gewinnen. Kann durch die verbesserte Einschließung $\widetilde{f} < \underline{F(Y)}$ gezeigt werden, so enthält Y sicher keine globale Minimalstelle. Andernfalls wird ein Sortierungsvektor s bestimmt, der auf eine von drei möglichen Arten zur Aufteilung von Y verwendet wird:

Boxing-Verfahren: Kann das alte $\widetilde{f}$ durch die Auswertung $F(m(Y))$ stark nach unten verbessert werden $(\widetilde{f} - \underline{F(Y)} > 2\overline{F(m(Y))} - \underline{F(Y)})$, so wird das Boxing-Verfahren eingesetzt. Auf diese Weise entstehen zunächst relativ viele Boxen für die Arbeitsliste. In der späteren Phase des Algorithmus wird $\widetilde{f}$ nur noch langsam verbessert, daher wird nun das Boxing-Verfahren seltener zum Einsatz kommen. Aufgrund einiger numerischer Experimente wird $\epsilon_{Box} = 5\epsilon$ gesetzt.

Intervall-Newton-Schritt: Wird das Boxing-Verfahren nicht auf die aktuelle Box Y angewandt, so entscheidet das aktuelle ϵ_{Newton} darüber, ob eine Hessematrix-Auswertung und evtl. anschließend ein Intervall-Newton-Schritt durchgeführt wird. Ist $d_{\mathrm{rel}}(Y) < \epsilon_{Newton}$, so wird zunächst die Hessematrix $\boldsymbol{\nabla}^2 F(Y)$ berechnet und der Konkavitätstest ausgeführt. Gelingt es nicht, die Nichtkonvexität nachzuweisen, so wird durch eine Taylorform 2. Ordnung (vgl. Gl. 1.2) eine engere Einschließung für $F(Y)$ berechnet. Falls sich jetzt $\widetilde{f} < \underline{F(Y)}$ zeigen läßt, so kann dadurch der Intervall-Newton-Schritt eingespart werden, da Y nicht weiter betrachtet werden muß. Im anderen Fall wird der Intervall-Newton-Schritt mit Fortschrittsindikator ausgeführt. Ist der Intervall-Newton-Schritt nicht erfolgreich, so wird ϵ_{Newton} halbiert (Untergrenze ist 0.01). Falls Y unverändert zurückgegeben wird, so wird anschließend eine Multisektion durchgeführt.

Multisektion: Falls $d_{\mathrm{rel}}(Y) \geq \epsilon_{Newton}$ ist, so wird ϵ_{Newton} um 50% erhöht (Obergrenze ist 1) und ein Multisektionsschritt mit $p = l$ für Y durchgeführt.

Nach der Aufteilung von Y in p Teilboxen $V_1, \ldots, V_p$ werden diejenigen Boxen V_j mit $\underline{F(V_j)} \leq \widetilde{f}$ zur weiteren Bearbeitung in die Arbeitsliste L eingefügt. Aus dem Boxing-Schritt können maximal $2n+1$, aus dem Intervall-Newton-Schritt maximal $n+1$ und aus dem Multisektionsschritt maximal l Boxen entstehen. Die Arbeitsliste kann also höchstens linear anwachsen.

ϵ_{Newton} schwankt während des Programmablaufs immer zwischen 0.01 und 1. Diese Werte wurden experimentell ermittelt und haben sich als sinnvoll erwiesen. Bei der Adaption von ϵ_{Newton} wird ein nicht erfolgreicher Intervall-Newton-Schritt durch Halbierung stärker „bestraft“ als umgekehrt der Multisektionsschritt, der eine Erhöhung von ϵ_{Newton} um 50% nach sich zieht. Diese Strategie ist sinnvoll, da ein nicht erfolgreicher Intervall-Newton-Schritt wesentlich aufwendiger ist als eine eventuell zuviel durchgeführte Multisektion.

Abbildung 3.3 illustriert die Adaption von ϵ_{Newton} während des Programmablaufs für die zwei Testprobleme GEO2 und H6. Nach jeder Iteration, also nach jedem Aufruf von BearbeiteBox() wird der aktuelle Wert von ϵ_{Newton} ausgegeben. GEO2 benötigt 495, H6 331 Iterationsschritte. Aus der Abbildung läßt sich ersehen, daß ϵ_{Newton} für GEO2 nach unten gegen 0.01 tendiert, während sich der Wert für H6 wesentlich seltener ändert und um einen durchschnittlichen Wert von etwa 0.68 schwankt. Die durch das neue

adaptive Verfahren ermittelten Werte für ϵ_{Newton} zeigen eine gute Übereinstimmung mit den Laufzeitergebnissen für die statische Wahl von ϵ_{Newton}, welche in Tabelle 3.1 abgedruckt sind. Die dort erzielten fettgedruckten Bestwerte werden durch den neuen seriellen Algorithmus übertroffen (vgl. den folgenden Abschnitt 3.2.10).

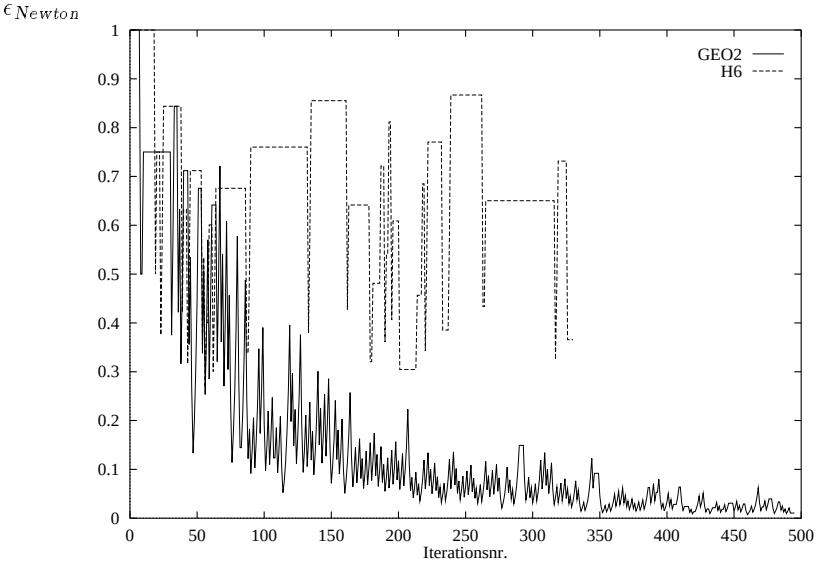


Abbildung 3.3: Adaption von ϵ_{Newton} während der Iteration

3.2.10 Numerische Ergebnisse für den seriellen Algorithmus

In diesem Abschnitt werden die genauen Ergebnisse (Laufzeiten, benötigte Funktions-, Gradienten- und Hessematrixauswertungen sowie maximale Listenlängen) für alle Testprobleme aus Anhang A für unterschiedliche Varianten des *Toolbox*-ähnlichen Algorithmus sowie des neuen seriellen Algorithmus angegeben. Es werden die Auswirkungen unterschiedlicher Teilungsstrategien (Bisektion oder Multisektion mit $l = 4$) sowie der Einsatz des sortierten Intervall-Newton-Schritts getestet. Das Testproblem KOW

wurde nicht mit in den Vergleich einbezogen, da in einigen Fällen kein Ergebnis erzielt wurde (Laufzeit zu lang).

Alle untersuchten Varianten des seriellen Algorithmus verwenden die Vorschrift C. zur Aufteilung der aktuellen Box. Alle Ergebnisse für den seriellen Algorithmus wurden auf einem Prozessor des verwendeten Parallelrechners IBM RS/6000 SP gemessen. Da die Berechnung von Gradient und Hessematrix durch automatische Differentiation erfolgt, werden die dabei anfallenden zusätzlichen Funktions- bzw. Funktions- und Gradientenauswertung nicht gesondert mitgezählt.

In allen Tabellen sind die jeweils besten Ergebnisse fettgedruckt. Ferner werden in den zweiten, dritten usw. Spalten Prozentangaben gemacht. Diese beziehen sich jeweils auf die erste Spalte. Am Ende jeder Tabelle wird die jeweilige Spalte aufsummiert. In der letzten Zeile findet sich dann der Durchschnitt aller Prozentangaben. Damit werden die verfälschenden Einflüsse einzelner „Ausreißer“ korrigiert. Diese Zahl gibt also den Erwartungswert an, um wieviel Prozent schneller oder langsamer ein einzelnes neues Problem gelöst werden kann (im Vergleich mit der ersten Spalte; unter der Annahme, daß die verwendeten Testprobleme die *real-world*-Probleme gut repräsentieren).

3.2.10.1 Varianten des *Toolbox*-ähnlichen Algorithmus

Bevor im folgenden Abschnitt die Vergleichsergebnisse des neuen seriellen Algorithmus mit dem *Toolbox*-ähnlichen Algorithmus angegeben werden, folgen hier zunächst Laufzeitmessungen (in STU) für verschiedene Varianten des *Toolbox*-ähnlichen Algorithmus 3.9. Alle Testprobleme wurden sowohl mit Bisektion ($l = 2$) als auch mit Multisektion mit $l = 4$ sowie mit dem unsortierten ($SortINS = false$) und dem sortierten ($SortINS = true$) erweiterten Intervall-Newton-Schritt (Algorithmus 3.7) gerechnet.

Prob.	u.N.-S. $l=4$	uns. Newton-Schr. $l=2$	(%S.1[2])	sort. New.-Schr. $l=4$	(%S.1)	sort. New.-Schr. $l=2$	(%S.1)
S5	1.75	2.00	(114%)	1.88	(107%)	**1.53**	(87%)
S7	3.13	2.69	(86%)	2.86	(91%)	**2.18**	(70%)
S10	3.53	3.39	(96%)	3.67	(104%)	**3.15**	(89%)
SHCB	1.00	**0.92**	(92%)	1.00	(100%)	0.93	(93%)
BR	1.77	1.04	(59%)	1.77	(100%)	**1.02**	(58%)
RO	0.24	**0.21**	(88%)	0.23	(96%)	0.29	(121%)
L8	3.30	**2.09**	(63%)	3.02	(92%)	2.20	(67%)
L9	5.65	4.27	(76%)	5.14	(91%)	**4.21**	(75%)
H3	3.83	**3.35**	(87%)	4.04	(105%)	3.41	(89%)
G5	21.02	19.85	(94%)	16.47	(78%)	**14.91**	(71%)
R4	25.14	11.59	(46%)	11.00	(44%)	**10.59**	(42%)
L12	50.92	**45.80**	(90%)	52.27	(103%)	49.18	(97%)
L18	35.22	22.63	(64%)	25.98	(74%)	**21.76**	(62%)
G7	58.78	8697.00	(14796%)	56.57	(96%)	**42.65**	(73%)
G10	162.46	24342.24	(14984%)	157.78	(97%)	**108.51**	(67%)
GP	246.17	85.43	(35%)	**57.93**	(24%)	74.34	(30%)
H6	105.27	134.44	(128%)	47.28	(45%)	**42.57**	(40%)
S2.14	**13.02**	1800.00	(13825%)	13.75	(106%)	53.91	(414%)
GEO1	30.16	54.62	(181%)	26.80	(89%)	**23.93**	(79%)
GEO2	52.62	60.10	(114%)	45.99	(87%)	**42.10**	(80%)
GEO3	49.19	89.36	(182%)	42.85	(87%)	**41.24**	(84%)
JS	64.65	40.64	(63%)	38.39	(59%)	**35.76**	(55%)
S2.7	159.62	143.10	(90%)	**113.22**	(71%)	113.79	(71%)
L3	889.10	633.22	(71%)	549.55	(62%)	**491.40**	(55%)
R8	349.24	368.23	(105%)	323.43	(93%)	**234.20**	(67%)
HM3	458.16	**395.83**	(86%)	551.13	(120%)	446.90	(98%)
HM4	2392.75	**1546.23**	(65%)	3151.41	(132%)	2565.92	(107%)
$\sum$	5187.69	38510.27	(742%)	5305.41	(102%)	**4432.58**	(85%)
Ø%			(1696%)		(87%)		(87%)

Tabelle 3.2: Laufzeiten (in STU) für verschiedene Varianten des *Toolbox*-ähnlichen Algorithmus

Aus Tabelle 3.2 können wir ablesen, daß es für den unsortierten Intervall-Newton-Schritt mit $l = 2$ (diese Variante entspricht bis auf die Verwendung

[2] %S.1 = % des entsprechenden Eintrags von Spalte 1

der Vorschrift C. zur Aufteilung der aktuellen Box dem originalen Algorithmus aus [21]) drei Testprobleme (G7, G10 und S2.14) gibt, die mehr als einhundert Mal mehr Laufzeit benötigen als alle anderen Varianten. Aus diesem Grund ist diese Variante für den praktischen Einsatz nicht geeignet. Von allen anderen Varianten schneidet die Kombination ($SortINS$ = *true*,$l = 2$) am besten ab. Für mehr als die Hälfte der Testprobleme wird die geringste Laufzeit benötigt. Daß der Einsatz des sortierten Intervall-Newton-Schritts sich auszahlt, bestätigt die Ergebnisse aus [71]. Hingegen überrascht es ein wenig, daß für den sortierten Intervall-Newton-Schritt die Bisektion ($l = 2$) besser abschneidet als die Multisektion mit $l = 4$. In [5] werden Ergebnisse angegeben, die genau den umgekehrten Sachverhalt nahelegen. In dem dort verwendeten Algorithmus wird jedoch ein statisches Entscheidungskriterium mit $\epsilon_{Newton} = 0.01$ für den Einsatz des Intervall-Newton-Schritts verwendet. Dies ist sicher die Ursache für das bessere Abschneiden der Multisektion in [5]. Auch im neuen seriellen Algorithmus, der ja auch ein (adaptives) Entscheidungskriterium ϵ_{Newton} verwendet, werden mit $l = 4$ geringere Laufzeiten gemessen als mit $l = 2$ (vgl. den folgenden Abschnitt). Unsere Ergebnisse werden auch in [9] bestätigt.

Für den Vergleich mit dem neuen seriellen Algorithmus werden aufgrund der Ergebnisse in Tabelle 3.2 und aus Gründen der Übersichtlichkeit nur die Ergebnisse für den sortierten Intervall-Newton-Schritt herangezogen.

3.2.10.2 Numerische Ergebnisse für den neuen seriellen Algorithmus

Dieser Abschnitt enthält die Vergleichsdaten des neuen seriellen Algorithmus mit dem *Toolbox*-ähnlichen Algorithmus. Alle Berechnungen wurden mit dem sortierten Intervall-Newton-Schritt durchgeführt. Die Ergebnisse für das Testproblem KOW wurden nicht mit in den Vergleich einbezogen und sind, sofern vorhanden, in Klammern aufgeführt. Sie dienen lediglich als spätere Vergleichsgrundlage für die Parallelisierung.

Ein Vergleich der Laufzeiten des neuen seriellen Algorithmus mit dem *Toolbox*-ähnlichen Algorithmus ist in Tabelle 3.3 zu finden. Man erkennt, daß für fast alle Testprobleme der neue serielle Algorithmus mit Parameter $l = 4$ die niedrigsten Laufzeiten erzielt. Gegenüber dem *Toolbox*-ähnlichen Algorithmus ist nur das Problem JS signifikant langsamer (45.71 STU gegenüber 35.76 STU). Bei dem Problem R8 ist der neue serielle Algorithmus mit Parameter $l = 2$ deutlich schneller (141.63 STU gegenüber 97.40 STU), im

Vergleich mit dem *Toolbox*-ähnlichen Algorithmus liegt jedoch eine deutliche Verbesserung vor.

Prob.	neu.s.Alg. $l=4$	neuer ser. Algor. $l=2$	(%S.1)	*Toolbox*-Algor. $l=4$	(%S.1)	*Toolbox*-Algor. $l=2$	(%S.1)
S5	**0.89**	0.93	(104%)	1.88	(211%)	1.53	(172%)
S7	1.38	**1.30**	(94%)	2.86	(207%)	2.18	(158%)
S10	1.91	**1.88**	(98%)	3.67	(192%)	3.15	(165%)
SHCB	1.84	1.38	(75%)	1.00	(54%)	**0.93**	(51%)
BR	1.38	1.62	(117%)	1.77	(128%)	**1.02**	(74%)
RO	**0.19**	0.21	(111%)	0.23	(121%)	0.29	(153%)
L8	2.28	2.65	(116%)	3.02	(132%)	**2.20**	(96%)
L9	**3.66**	4.62	(126%)	5.14	(140%)	4.21	(115%)
H3	4.50	4.91	(109%)	4.04	(90%)	**3.41**	(76%)
G5	**7.09**	8.64	(122%)	16.47	(232%)	14.91	(210%)
R4	**10.42**	10.68	(102%)	11.00	(106%)	10.59	(102%)
L12	**23.35**	29.05	(124%)	52.27	(224%)	49.18	(211%)
L18	12.90	**12.18**	(94%)	25.98	(201%)	21.76	(169%)
G7	**13.73**	16.75	(122%)	56.57	(412%)	42.65	(311%)
G10	**22.92**	26.30	(115%)	157.78	(688%)	108.51	(473%)
GP	61.49	**57.13**	(93%)	57.93	(94%)	74.34	(121%)
H6	**25.51**	34.02	(133%)	47.28	(185%)	42.57	(167%)
S2.14	**10.20**	12.00	(118%)	13.75	(135%)	53.91	(529%)
GEO1	**13.85**	17.31	(125%)	26.80	(194%)	23.93	(173%)
GEO2	**28.87**	32.42	(112%)	45.99	(159%)	42.10	(146%)
GEO3	**27.40**	34.25	(125%)	42.85	(156%)	41.24	(151%)
JS	45.71	47.80	(105%)	38.39	(84%)	**35.76**	(78%)
S2.7	**79.49**	91.92	(116%)	113.22	(142%)	113.79	(143%)
L3	**273.46**	284.50	(104%)	549.55	(201%)	491.40	(180%)
R8	141.63	**97.40**	(69%)	323.43	(228%)	234.20	(165%)
HM3	**294.13**	322.14	(110%)	551.13	(187%)	446.90	(152%)
HM4	**1724.53**	3824.83	(222%)	3151.41	(183%)	2565.92	(149%)
$\sum$	**2834.71**	4978.82	(176%)	5305.41	(187%)	4432.58	(156%)
Ø%			(113%)		(174%)		(189%)

Tabelle 3.3: Laufzeiten (in STU) für den neuen seriellen Algorithmus

Die beobachteten Laufzeiten lassen den folgenden Schluß zu:

Der neue serielle Algorithmus (mit Parameter $l = 4$) stellt eine deutliche Verbesserung gegenüber den verschiedenen Varianten des *Toolbox*-ähnlichen Algorithmus dar. Die zu erwartende Laufzeit bei einem Testproblem liegt beim *Toolbox*-ähnlichen Algorithmus mit Parameter $l = 2$ um 89% höher und beim *Toolbox*-ähnlichen Algorithmus mit Parameter $l = 4$ um 74% höher als beim neuen seriellen Algorithmus mit Parameter $l = 4$. Es wird also beinahe eine Halbierung der Laufzeit erreicht. Berechnet man einen größeren Satz an Testproblemen, der mit dem hier verwendeten vergleichbar ist, so benötigt der *Toolbox*-ähnliche Algorithmus mit Parameter $l = 4$ 87% und mit Parameter $l = 2$ 56% mehr Laufzeit als der neue serielle Algorithmus.

Um die Eigenschaften des neuen seriellen Algorithmus noch ein wenig genauer zu illustrieren, sind in den Tabellen 3.4, 3.5 und 3.6 die für die Lösung der einzelnen Testprobleme benötigten Funktions-, Gradienten- und Hessematrixauswertungen aufgeführt. Für praktische Anwendungen sind diese Zahlen fast noch wichtiger als die gemessenen Laufzeiten, da in der Praxis die auftretenden Zielfunktionen meist aufwendiger auszuwerten sind als die Zielfunktionen der in dieser Arbeit verwendeten Testprobleme.

Der Herleitung des neuen seriellen Algorithmus lag die Idee zugrunde, durch den adaptiv gesteuerten, selektiven Einsatz des Intervall-Newton-Schritts die Zahl der Gradienten- und Hessematrixauswertungen stark zu verringern. Die gemessenen Ergebnisse in den Tabellen 3.4, 3.5 und 3.6 zeigen, daß dies in der Praxis tatsächlich der Fall ist. Der neue serielle Algorithmus mit Parameter $l = 4$ benötigt im Durchschnitt beinahe doppelt so viele Funktionsauswertungen wie die beiden Varianten des *Toolbox*-ähnlichen Algorithmus, dieser Nachteil wird jedoch durch die noch stärker verringerte Zahl der Gradienten- und Hessematrixauswertungen wettgemacht: Der *Toolbox*-ähnliche Algorithmus benötigt mit Bisektion ($l = 2$) im Durchschnitt 3.5 mal und mit Multisektion ($l = 4$) 4.1 mal so viele Gradientenauswertungen wie der neue serielle Algorithmus mit $l = 4$. Bei den Hessematrixauswertungen sind die Faktoren noch deutlicher: 5.57 bzw. 6.4.

Prob.	n.s.Alg. $l=4$	neuer ser. Algor. $l=2$	(%S.1)	*Toolbox*-Algor. $l=4$	(%S.1)	*Toolbox*-Algor. $l=2$	(%S.1)
S5	120	125	(104%)	62	(52%)	**50**	(42%)
S7	120	125	(104%)	64	(53%)	**54**	(45%)
S10	125	131	(105%)	77	(62%)	**68**	(54%)
SHCB	509	465	(91%)	**91**	(18%)	128	(25%)
BR	140	153	(109%)	34	(24%)	**30**	(21%)
RO	125	125	(100%)	**47**	(38%)	65	(52%)
L8	95	107	(113%)	38	(40%)	**31**	(33%)
L9	117	141	(121%)	51	(44%)	**41**	(35%)
H3	162	182	(112%)	**41**	(25%)	48	(30%)
G5	218	244	(112%)	176	(81%)	**162**	(74%)
R4	605	634	(105%)	**206**	(34%)	250	(41%)
L12	303	353	(117%)	233	(77%)	**203**	(67%)
L18	248	226	(91%)	176	(71%)	**153**	(62%)
G7	**290**	331	(114%)	475	(164%)	365	(126%)
G10	**316**	337	(107%)	840	(266%)	573	(181%)
GP	5706	5946	(104%)	**3449**	(60%)	5234	(92%)
H6	874	1073	(123%)	**335**	(38%)	347	(40%)
S2.14	3179	3818	(120%)	**890**	(28%)	3700	(116%)
GEO1	1196	1378	(115%)	**697**	(58%)	735	(61%)
GEO2	2483	2593	(104%)	**1001**	(40%)	1190	(48%)
GEO3	2390	2627	(110%)	**1038**	(43%)	1218	(51%)
JS	532	458	(86%)	**123**	(23%)	159	(30%)
S2.7	464	495	(107%)	**192**	(41%)	262	(56%)
L3	3817	3611	(95%)	**2453**	(64%)	2466	(65%)
R8	1467	1092	(74%)	631	(43%)	**514**	(35%)
HM3	3453	3582	(104%)	2001	(58%)	**1950**	(56%)
HM4	14549	31539	(217%)	7695	(53%)	**7282**	(50%)
$\sum$	43603	61891	(142%)	**23116**	(53%)	27278	(63%)
Ø%			(110%)		(59%)		(59%)

Tabelle 3.4: Anzahl der benötigten Funktionsauswertungen

Prob.	n.s.Alg. $l=4$	neuer ser. Algor. $l=2$	(%S.1)	*Toolbox*-Algor. $l=4$	(%S.1)	*Toolbox*-Algor. $l=2$	(%S.1)
S5	**33**	41	(124%)	145	(439%)	111	(336%)
S7	**45**	54	(120%)	157	(349%)	111	(247%)
S10	**47**	56	(119%)	157	(334%)	125	(266%)
SHCB	330	303	(92%)	288	(87%)	**256**	(78%)
BR	**47**	67	(143%)	109	(232%)	62	(132%)
RO	**54**	57	(106%)	118	(219%)	138	(256%)
L8	**23**	30	(130%)	75	(326%)	54	(235%)
L9	**27**	38	(141%)	98	(363%)	81	(300%)
H3	**97**	109	(112%)	147	(152%)	118	(122%)
G5	**47**	73	(155%)	274	(583%)	243	(517%)
R4	**320**	363	(113%)	595	(186%)	539	(168%)
L12	**62**	93	(150%)	373	(602%)	336	(542%)
L18	**51**	66	(129%)	331	(649%)	270	(529%)
G7	**62**	101	(163%)	630	(1016%)	466	(752%)
G10	**71**	111	(156%)	1051	(1480%)	713	(1004%)
GP	**3978**	4391	(110%)	8616	(217%)	9648	(243%)
H6	**354**	536	(151%)	1232	(348%)	1044	(295%)
S2.14	**1080**	1605	(149%)	2602	(241%)	7843	(726%)
GEO1	**382**	583	(153%)	1823	(477%)	1496	(392%)
GEO2	**805**	1090	(135%)	3115	(387%)	2729	(339%)
GEO3	**762**	1108	(145%)	3114	(409%)	2780	(365%)
JS	**286**	305	(107%)	473	(165%)	411	(144%)
S2.7	**160**	220	(138%)	611	(382%)	564	(352%)
L3	**1187**	1511	(127%)	4209	(355%)	3749	(316%)
R8	560	**414**	(74%)	2516	(449%)	1755	(313%)
HM3	**1373**	1639	(119%)	4018	(293%)	3180	(232%)
HM4	**4682**	9422	(201%)	15336	(328%)	11876	(254%)
$\sum$	**16925**	24386	(144%)	52213	(308%)	50698	(300%)
Ø%			(132%)		(350%)		(410%)

Tabelle 3.5: Anzahl der benötigten Gradientenauswertungen

Prob.	n.s.Alg. $l=4$	neuer ser. Algor. $l=2$	(%S.1)	*Toolbox*-Algor. $l=4$	(%S.1)	*Toolbox*-Algor. $l=2$	(%S.1)
S5	**8**	9	(112%)	20	(250%)	18	(225%)
S7	**8**	9	(112%)	22	(275%)	18	(225%)
S10	**8**	9	(112%)	22	(275%)	20	(250%)
SHCB	**40**	56	(140%)	70	(175%)	60	(150%)
BR	**12**	15	(125%)	35	(292%)	18	(150%)
RO	**20**	**20**	(100%)	25	(125%)	32	(160%)
L8	**2**	**2**	(100%)	12	(600%)	10	(500%)
L9	**2**	3	(150%)	14	(700%)	12	(600%)
H3	**16**	18	(112%)	29	(181%)	22	(138%)
G5	**2**	**2**	(100%)	34	(1700%)	32	(1600%)
R4	**60**	67	(112%)	110	(183%)	114	(190%)
L12	**4**	**4**	(100%)	32	(800%)	33	(825%)
L18	**6**	7	(117%)	34	(567%)	29	(483%)
G7	**2**	**2**	(100%)	64	(3200%)	50	(2500%)
G10	**3**	**3**	(100%)	83	(2767%)	61	(2033%)
GP	**1197**	1483	(124%)	1453	(121%)	1886	(158%)
H6	**24**	37	(154%)	138	(575%)	123	(512%)
S2.14	226	**222**	(98%)	446	(197%)	1826	(808%)
GEO1	**112**	159	(142%)	335	(299%)	291	(260%)
GEO2	**234**	299	(128%)	632	(270%)	568	(243%)
GEO3	**218**	302	(139%)	555	(255%)	537	(246%)
JS	**63**	74	(117%)	99	(157%)	92	(146%)
S2.7	**50**	68	(136%)	130	(260%)	116	(232%)
L3	**281**	346	(123%)	750	(267%)	704	(251%)
R8	7	**3**	(43%)	177	(2529%)	138	(1971%)
HM3	**533**	582	(109%)	680	(128%)	583	(109%)
HM4	**1633**	3246	(199%)	2227	(136%)	1831	(112%)
$\sum$	**4771**	7047	(148%)	8228	(172%)	9224	(193%)
Ø%			(119%)		(558%)		(640%)

Tabelle 3.6: Anzahl der benötigten Hessematrixauswertungen

Der Auswertungsaufwand für Funktions-, Gradienten und Hessematrixauswertungen wird in der Literatur auch wie folgt gemessen:

Definition 3.2 *Seien FE, GE und HE die zur Lösung eines Testproblems der Dimension n benötigten Funktions-, Gradienten und Hessematrixauswertungen. Dann sind*

$$\begin{aligned} Eeff_V &:= FE + n \cdot GE + \frac{n(n+1)}{2} \cdot HE \\ Eeff_R &:= FE + \min\{4, n\} \cdot GE + n \cdot HE \end{aligned}$$

Maße für den Auswertungsaufwand der Vorwärts- bzw. Rückwärtsmethode der automatischen Differentiation.

Um den Effizienzunterschied zwischen dem neuen seriellen Algorithmus und dem *Toolbox*-ähnlichen Algorithmus zu verdeutlichen, geben wir nachfolgend anstelle zweier vollständiger Tabellen für Eeff_V und Eeff_R, die sich leicht aus den Tabellen 3.4, 3.5 und 3.6 berechnen lassen, nur die Ergebnisse für die Aufsummierung der einzelnen Zeilen an:

	n.s.Alg. $l=4$	neuer ser. Algor. $l=2$ (%S.1)	*Toolbox*-Algor. $l=4$ (%S.1)	*Toolbox*-Algor. $l=2$ (%S.1)
Eeff_V : $\sum$	118120	171133 (145%)	260407 (220%)	322688 (273%)
Ø%		(122%)	(343%)	(257%)
Eeff_R : $\sum$	101800	148569 (146%)	202799 (199%)	252502 (248%)
Ø%		(121%)	(294%)	(218%)

Tabelle 3.7: Auswertungsaufwand Eeff_V und Eeff_R

Wir erkennen, daß der Auswertungsaufwand für den neuen seriellen Algorithmus höchstens halb so groß ist wie für beide Varianten des *Toolbox*-ähnlichen Algorithmus. Dies gilt sowohl für die Vorwärts- als auch für die Rückwärtsmethode bei der automatischen Differentiation.

Auch bezüglich der maximalen Länge der Arbeitsliste L schneidet der neue serielle Algorithmus zumindest vergleichbar zum *Toolbox*-ähnlichen Algorithmus ab, wie aus Tabelle 3.8 zu erkennen ist. Insgesamt läßt sich feststellen, daß der Effizienzvorteil des neuen seriellen Algorithmus nicht durch einen wesentlich erhöhten Speicherplatzbedarf erkauft wird.

Prob.	n.s.Alg. $l=4$	neuer ser. Algor. $l=2$	(%S.1)	*Toolbox*-Algor. $l=4$	(%S.1)	*Toolbox*-Algor. $l=2$	(%S.1)
S5	23	20	(87%)	21	(91%)	**17**	(74%)
S7	25	22	(88%)	21	(84%)	**15**	(60%)
S10	27	**23**	(85%)	25	(93%)	24	(89%)
SHCB	105	50	(48%)	29	(28%)	**24**	(23%)
BR	23	18	(78%)	7	(30%)	**5**	(22%)
RO	24	23	(96%)	16	(67%)	**15**	(62%)
L8	24	21	(88%)	16	(67%)	**13**	(54%)
L9	35	28	(80%)	25	(71%)	**19**	(54%)
H3	22	16	(73%)	15	(68%)	**11**	(50%)
G5	42	**32**	(76%)	76	(181%)	70	(167%)
R4	106	80	(75%)	**50**	(47%)	53	(50%)
L12	87	**51**	(59%)	174	(200%)	152	(175%)
L18	59	**46**	(78%)	101	(171%)	90	(153%)
G7	76	**47**	(62%)	217	(286%)	173	(228%)
G10	101	**71**	(70%)	664	(657%)	420	(416%)
GP	612	**482**	(79%)	492	(80%)	563	(92%)
H6	166	123	(74%)	79	(48%)	**78**	(47%)
S2.14	157	**136**	(87%)	140	(89%)	207	(132%)
GEO1	152	**100**	(66%)	211	(139%)	163	(107%)
GEO2	206	**177**	(86%)	196	(95%)	198	(96%)
GEO3	194	170	(88%)	287	(148%)	**157**	(81%)
JS	86	71	(83%)	35	(41%)	**29**	(34%)
S2.7	68	55	(81%)	53	(78%)	**46**	(68%)
L3	672	709	(106%)	1214	(181%)	**651**	(97%)
R8	331	**168**	(51%)	226	(68%)	215	(65%)
HM3	615	626	(102%)	520	(85%)	**447**	(73%)
HM4	**1847**	3702	(200%)	1993	(108%)	2428	(131%)
$\sum$	**5885**	7067	(120%)	6903	(117%)	6283	(107%)
Ø%			(83%)		(100%)		(122%)

Tabelle 3.8: Maximale Länge der Arbeitsliste L

3.3 Paralleler Algorithmus

In diesem Abschnitt wird nun das im vorigen Abschnitt vorgestellte serielle Verfahren zur globalen Optimierung zum Einsatz auf Parallelrechnern erweitert. Dazu setzen wir den in Abschnitt 2.4.3 vorgestellten vollständig verteilten Algorithmus zur Lastverteilung für *Branch and Bound*-Algorithmen ein. Vorher werden jedoch noch im nächsten Abschnitt andere Ansätze zur Parallelisierung bei der verifizierten globalen Optimierung vorgestellt.

3.3.1 Andere Ansätze zur Parallelisierung bei der globalen Optimierung

Der serielle Algorithmus zur globalen Optimierung gehört zu der Klasse der *Branch and Bound*-Algorithmen. Die verschiedenen denkbaren Vorgehensweisen bei der Parallelisierung von *Branch and Bound*-Algorithmen wurden in Abschnitt 2.4.2 vorgestellt und mit einigen Bemerkungen zu Vor- und Nachteilen diskutiert. Denkbar sind zentralisierte Ansätze wie die *master-slave*-Verwaltung der Teilprobleme oder aber eine verteilte Verwaltung der Teilprobleme. Jeder parallele Algorithmus zur verifizierten globalen Optimierung sollte die Forderungen nach minimaler Leerlaufzeit der Prozessoren und minimalem Mehraufwand durch die parallele Suche auf einem Parallelrechner mit p Prozessoren möglichst gut erfüllen, um eine hohe Beschleunigung zu erzielen. Zur Minimierung des Mehraufwands ist es wichtig, daß die auf einem Prozessor p_i neu gefundene beste obere Schranke $\widetilde{f}$ für das globale Minimum $f^\star$ rasch an alle anderen Prozessoren verteilt wird, um einen möglichst großen Effekt beim Mittelpunktstest (dem *Bounding*-Schritt) zu erzielen.

Nachfolgend wird eine Übersicht über verschiedene in der Literatur untersuchte parallele Algorithmen gegeben. Die Einteilung erfolgt dabei nach der Verwaltung der Teilboxen durch den parallelen Algorithmus.

master-slave-**Verwaltung:** Ein paralleler Algorithmus zur verifizierten globalen Optimierung mit einer zentralisierten *master-slave*-Verwaltung (vgl. auch Abbildung 2.3) wurde von Henriksen und Madsen in [27] untersucht.

Dabei wird nur eine Arbeitsliste L mit allen zu untersuchenden Teilboxen auf dem Master zentral gespeichert. $p-1$ Slaves fordern Arbeit

(Teilboxen mit zugehöriger unterer Schranke) vom Master an, bearbeiten diese, und senden Ergebnisse (wiederum Paare von Teilboxen mit unterer Schranke sowie möglicherweise ein besseres $\widetilde{f}$) an den Master zurück. Der Algorithmus zur globalen Optimierung verwendet den Mittelpunktstest, den Monotonietest und das Intervall-Newton-Verfahren.

Praktische Tests des parallelen Algorithmus wurden auf einem Transputersystem durchgeführt. Bei den meisten untersuchten Testproblemen sank die Effizienz ab einer Prozessorzahl von $p > 16$ stark ab, bei manchen Problemen verringerte sich sogar der Speedup. Durch die zentrale Speicherung der Teilprobleme wird die Kommunikation mit dem Master zum Flaschenhals, da zu viele Prozessoren gleichzeitig mit dem Master Daten austauschen wollen. Außerdem ergibt sich eine weitere Beschränkung durch den Speicher des Masters: Da die Arbeitsliste L zentral gespeichert ist, dürfen während der Berechnung nicht mehr Teilprobleme entstehen, als in den Speicher des Masters passen.

Für große Parallelrechnersysteme ist die *master-slave*-Verwaltung also nicht so sehr geeignet, die nachfolgend vorgestellten Verfahren sind alle effizienter.

Verteilte Verwaltung in einem Ring: Von Eriksson wurde in [12] die verteilte Verwaltung der Teilboxen in einem Ring (vgl. Abbildung 2.1) untersucht. Jeder Prozessor hat in dem Ring einen Vorgänger und einen Nachfolger. Zu Beginn wird auf jedem Prozessor ein *worker*-Prozeß und ein *scheduler*-Prozeß gestartet. Die lokal vorhandenen Teilboxen werden durch den *worker*-Prozeß bearbeitet und durch den *scheduler* in einer eigene Arbeitsliste auf jedem Prozessor verwaltet. Zur Beschleunigung des Verfahrens zur globalen Optimierung wird das Krawczyk-Verfahren (siehe z. B. [60], es ähnelt dem Intervall-Newton-Schritt) eingesetzt. Der *worker*-Prozeß steuert außerdem die Kommunikation der besten oberen Schranke $\widetilde{f}$. Eine neu gefundenes besseres $\widetilde{f}$ wird an alle anderen *worker* geschickt. Durch ein asynchrones *receive* Kommando wird es sofort empfangen.

Der Lastausgleich wird durch den *scheduler* durchgeführt, er versucht, die noch vorhandenen Teilboxen möglichst gleichmäßig zu verteilen. Dabei erfolgt sowohl ein Ausgleich nach der Quantität als auch nach der Qualität der Boxen. Letztere ist an der zugehörigen unteren Schranke abzulesen. Der Ausgleich nach der Quantität wird vom

scheduler des Empfängers der Boxen angestoßen: Falls die Arbeitsliste eines Prozessors p_i leer wird, so fordert er von seinem Nachfolger p_{i+1} im Ring neue Arbeit (Teilboxen) an. Kann der Nachfolger keine Teilboxen abgeben, so wird die Anfrage wiederum an den nächsten Nachfolger weitergeleitet usw. Sobald ein Prozessor mit einer nichtleeren Arbeitsliste gefunden ist, wird eine Box direkt an den Initiator der Anfrage (Prozessor p_i) geschickt.

Bei einer Parallelisierung mit verteilter Verwaltung der Teilboxen in lokalen Arbeitslisten, die jeweils nach dem *best-first*-Prinzip (hier nach den zugehörigen unteren Schranken) sortiert sind, ist es nicht ohne weiteres sicher gestellt, daß aus globaler Sicht zu jedem Zeitpunkt die p besten Boxen bearbeitet werden. Nicht alle Prozessoren verfügen jederzeit über das optimale $\widetilde{f}$, sodaß die zweitbeste Teilbox eines Prozessors p_i besser sein kann als die beste Teilbox eines Prozessors p_j. Aus diesem Grund ist ein Ausgleich auch nach der Qualität der Boxen sinnvoll. Dieser Ausgleich wird jeweils von einem Prozessor mit vielen Boxen in der Arbeitsliste angestoßen. Überschreitet die Länge der Arbeitsliste einen bestimmten (u.U. auch dynamisch festgelegten) Schwellenwert, so wird die beste Teilbox in der Arbeitsliste an einen zufälligen Prozessor geschickt.

In [12] werden auch einige konkrete Ergebnisse angegeben. Dabei beträgt die Effizienz für drei behandelte Beispiele zwischen 60% und 122% bei der Verwendung von 16 Prozessoren und zwischen 47% und 97% bei der Verwendung von 32 Prozessoren.

Zentraler Vermittler: Ein Algorithmus, der eine Kombination von verteilter und zentralisierter Verwaltung verwendet, wurde von Berner in [5] angegeben. Dabei wird ein Prozessor des Parallelrechners als *zentraler Vermittler* ausgewählt. Dieser zentrale Vermittler speichert nicht die noch zu untersuchenden Teilboxen, sondern übernimmt lediglich den Austausch von Teilboxen zwischen den restlichen Prozessoren, die als *worker* fungieren und Teilboxen bearbeiten. Eine Übersicht über das Arbeitsprinzip des zentralen Vermittlers gibt die Abbildung 3.4.

Zu Beginn des Algorithmus wird in der Startphase die zu untersuchende Box X in $p-1$ Teilboxen aufgeteilt und auf die *worker* verteilt. Anschließend wird auf den $p-1$ *workern* eine reelle lokale Optimierung ausgehend vom Mittelpunkt der jeweiligen Teilbox gestartet, um zunächst lokal auf jedem Prozessor eine gute obere Schranke $\widetilde{f}$

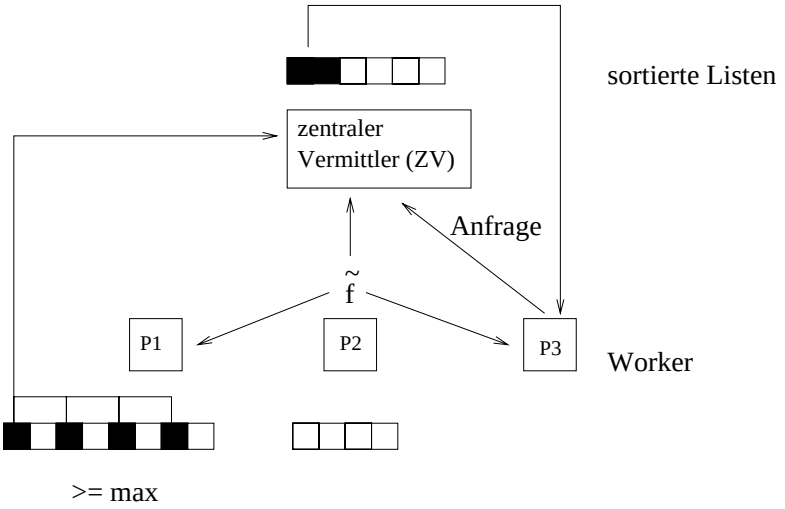


Abbildung 3.4: Arbeitsprinzip des zentralen Vermittlers

für das globale Minimum zu finden. Die beste dieser $p-1$ oberen Schranken wird dann an alle *worker* verteilt. In der nachfolgenden Berechnungsphase bearbeitet jeder *worker* seine lokale Arbeitsliste L unter Verwendung der Bestensuche. *worker*, bei denen die Arbeitsliste stark angewachsen ist, versenden überschüssige Boxen an den zentralen Vermittler. Dazu verwaltet der zentrale Vermittler eine Steuervariable *max*, die im Laufe der Berechnung geändert werden kann. Hat ein *worker* mehr als *max* Teilboxen in seiner Arbeitsliste L, so verschickt er die Hälfte seiner Teilboxen (jedoch nicht mehr als *max-send*) an den zentralen Vermittler. Dieser fügt die erhaltenen Boxen in seine Vorratsliste ein. Umgekehrt fordern *worker*, deren Arbeitsliste leer geworden ist, neue zu bearbeitende Boxen beim zentralen Vermittler an. Auch bei diesem Verfahren mit verteilten Arbeitslisten besteht wiederum das Problem, möglichst rasch auf allen Prozessoren die beste obere Schranke für das globale Minimum $\tilde{f}$ zur Verfügung zu stellen. Dies geschieht hier mit Hilfe eines asynchronen Broadcast Kommandos.

In der Arbeit von Berner sind eine große Anzahl von Testproblemen auf einer Connection Machine CM5 mit bis zu 32 Prozessoren untersucht worden. Für Probleme mit großer Laufzeit (serielle Laufzeit

zwischen 2000 und 19000 Sek.) wurde bis auf eine Ausnahme ein zufriedenstellender, leicht überlinearer Speedup erreicht. Bei Problemen mit mittlerer Laufzeit (serielle Laufzeit zwischen 100 und 1000 Sek.) wurde schon auf 16 Prozessoren für zwei von sieben untersuchten Problemen kein linearer Speedup mehr erreicht. Bei 32 Prozessoren lag die erzielte Effizienz zwischen 43% und 115%, in vier von sieben Problemen wurde überlinearer Speedup erzielt. Für kleinere Probleme mit einer seriellen Laufzeit von unter 100 Sek. wurde meist kein guter Speedup erzielt, im ungünstigsten Fall lag die Effizienz bei 32 Prozessoren nur bei etwa 18%. Die serielle Laufzeit dieses Problems lag allerdings auch nur bei 4.7 Sek., der zusätzliche Verwaltungsaufwand für den parallelen Algorithmus macht sich hier natürlicherweise stark bemerkbar.

Da für die Arbeit ein Parallelrechner mit nur 32 Prozessoren zur Verfügung stand, läßt sich leider keine unmittelbare Aussage über die Skalierbarkeit des Algorithmus des zentralen Vermittlers treffen. Anzunehmen ist jedoch, daß ab einer gewissen Prozessorzahl (diese wird sicher größer sein als die für die *master-slave*-Verwaltung) auch hier ein Blockadeeffekt beim zentralen Vermittler eintreten wird, da zu viele Anfragen gleichzeitig an den zentralen Vermittler gerichtet werden. Berner schlägt zur Lösung dieses Problems eine Aufteilung des Parallelrechners in Cluster vor, die jeweils über einen eigenen zentralen Vermittler verfügen. Die Cluster wiederum könnten durch einen übergeordneten zentralen Vermittler miteinander kommunizieren.

3.3.2 Ein neuer, vollständig verteilter paralleler Algorithmus

In diesem Abschnitt wird ein neuer, vollständig verteilter paralleler Algorithmus zur globalen Optimierung mit Ergebnisverifikation vorgestellt. Dieser Algorithmus basiert einerseits auf dem in Abschnitt 3.2.9 vorgestellten neuen seriellen Algorithmus und andererseits auf dem in Abschnitt 2.4.3 dargestellten parallelen Algorithmus zur Lastverteilung für *Branch and Bound*.

Der parallele Algorithmus läuft dabei in 4 Phasen ab:

1. **Startphase:** Aufteilung der Startbox X auf die Prozessoren

2. **Berechnungsphase:** Bearbeitung von Teilboxen und Lastausgleich

3. Sammlungsphase: Bei abgearbeiteter Arbeitsliste werden entweder Teilboxen von den Nachbarprozessoren empfangen oder die Arbeit beendet

4. Endphase: Abgleich der $\widetilde{f}$, Zusammenfassen der Lösungslisten

Die einzelnen Phasen des Algorithmus werden in den nachfolgenden Abschnitten beschrieben. Zunächst müssen jedoch noch einige Bezeichnungen definiert werden.

3.3.2.1 Voraussetzungen und Bezeichnungen für den parallelen Algorithmus, Kommunikationsroutinen

Bei der Beschreibung und Implementierung des parallelen Verfahrens wird von den folgenden Voraussetzungen ausgegangen und es werden die folgenden Bezeichnungen verwendet:

Das Verfahren wird implementiert auf einem Parallelrechner mit p Prozessoren $p_1, \ldots, p_p$. Alle Prozessoren des Parallelrechners können etwa gleich schnell mit einander kommunizieren. Auf dem in dieser Arbeit verwendeten Parallelrechner IBM RS/6000 SP wird dies durch den *SP-Switch* ermöglicht. Der SP-Switch ist ein mehrstufiges Netzwerk, das aus 4×4 bidirektionalen Crossbar-Switches aufgebaut ist. Die meisten der modernen Parallelrechner verfügen über eine ähnliche Hardwareunterstützung. Ist dies nicht der Fall (z.B. Transputersysteme), ist der Algorithmus aber leicht adaptierbar, da jeder Prozessor während der Berechnungsphase nur mit seinen Nachbarn kommuniziert. Lediglich in der Startphase und in der Endphase des Algorithmus ist die Kommunikation mit einem besonders ausgezeichneten I/O-Prozessor notwendig.

Wie unter der in dieser Arbeit verwendeten Kommunikationsbibliothek MPI (vgl. Abschnitt 2.5.2) üblich, läuft auf jedem Prozessor das gleiche Programm ab. Zur Unterscheidung zwischen den einzelnen Prozessoren ist die Nummer des eigenen Prozessors durch die globale Variable *myid* ($1 \leq myid \leq p$) bezeichnet. Für die Ein- und Ausgabe von Daten wird Prozessor p_1 verwendet. Die Startbox X liege zu Beginn auf jedem Prozessor vor. Zum Schluß der Berechnung werden die Ergebnisse auf p_1 zusammengefaßt und ausgegeben.

Der Lastverteilungsalgorithmus 3.25 verwendet zum Lastausgleich k Nachbarn $\tilde{p}_1, \ldots, \tilde{p}_k$, deren jeweils zuletzt bekannte Last in den Variablen

$w_1, \ldots, w_k$ gespeichert wird. Die eigene aktuelle Last von Prozessor *myid* wird in der Variablen w_{own} gespeichert, die eigene Last aus dem vorigen Lastverteilungsschritt in w_{old}. Die Last $w(p)$ eines Prozessors p wird definiert als die Länge seiner Arbeitsliste L:

$$w(p) := \mathsf{Length}(L).$$

Einige Anweisungen müssen nur auf dem I/O-Prozessor p_1 ausgeführt werden. Diese Anweisungen werden durch ein vorangestelltes **M:** gekennzeichnet. Im konkreten Programm sind sie durch eine Anweisung der Form

$$\textbf{if } (myid = 1) \textbf{ then } \ldots$$

zu realisieren.

In der Beschreibung des Algorithmus verwenden wir noch die aus der Programmiersprache C entlehnte abkürzende Schreibweise für die Inkrementierung bzw. Dekrementierung einer ganzzahligen Variablen i:

$$\begin{aligned} i\texttt{++} &\equiv i := i + 1 \\ i\texttt{--} &\equiv i := i - 1 \end{aligned}$$

Der parallele Algorithmus verwendet die folgenden globalen Variablen:

$w_{own}, w_{old}, w, \Delta, \Delta_{up}, \Delta_{down}$, *min.weight*: Diese Variablen sind weiter oben beschrieben.

fertig: Boolesche Variable; zeigt an, ob alle Prozessoren ihre Arbeit beendet haben.

abgemeldet: Boolesches Feld (nur auf dem I/O-Prozessor); zeigt an, welche Prozessoren bereits fertig sind.

Parallel zum Algorithmus für die globale Optimierung läuft auf jedem Prozessor der Kontrollprozeß (Algorithmus 2.2). Alle 0.1 Sekunden werden neue Werte für $\Delta_{up}, \Delta_{down}$ und Δ ermittelt. Die dafür notwendige Berechnung von *av.prob* und *av.weights* sowie die Anpassung von *prob* und *weights* ist nachfolgend aus Gründen der Übersichtlichkeit weggelassen. Nach der

Beschreibung in Abschnitt 2.4.3 sollte aber eindeutig sein, wo die entsprechenden Anweisungen einzufügen sind.

Zur Unterscheidung werden Nachrichten beim Versenden (durch Send()) an einen Prozessor i mit einer Kennzeichnung *tag* versehen. Beim Empfangen (durch Recv()) muß das passende *tag* angegeben werden. Im parallelen Algorithmus werden die folgenden *tags* verwendet:

COLLECT: Versenden der Lösungsliste an den I/O-Prozessor p_1.

NEWSOLUTION: Versenden eines verbesserten $\widetilde{f}$ an die Nachbarprozessoren.

REQUEST: Anforderung von Arbeit (=Teilboxen) von den Nachbarprozessoren.

WORK: Versenden von Teilboxen an die Nachbarprozessoren.

INFORM: Information über die eigene Lastsituation an die Nachbarprozessoren.

INFORMACK: Bestätigung der Wiederanmeldung eines Nachbarprozessors.

FINAL: Abmelden bei den Nachbarprozessoren, falls Arbeitsliste L leer ist.

FINALACK: Bestätigung der Abmeldung eines Nachbarprozessors.

GOODBYE: Endgültige Abmeldung an den I/O-Prozessor p_1.

GOODBYEACK: Bestätigung der Abmeldung durch den I/O-Prozessor p_1.

Auf dem in dieser Arbeit verwendeten Parallelrechner IBM RS/6000 SP ist das Versenden einer Nachricht mittels MPI_Send() gepuffert. Dies bedeutet, daß MPI_Send() bereits terminiert, auch wenn auf dem Empfangsprozessor noch kein passendes MPI_Recv() aufgerufen wurde. Stattdessen wird die zu übersendende Nachricht in einen Zwischenspeicher kopiert, aus dem dann MPI_Recv() auf dem Empfangsprozessor liest.

In der Beschreibung des parallelen Algorithmus verwenden wir einige Kommunikationsroutinen, die sich mit MPI leicht realisieren lassen. Sie sind nachfolgend kurz beschrieben.

GlobalesMin(r): Bestimmt das globale Minimum der auf jedem Prozessor p_i vorhandenen Variablen r und liefert dies in r zurück. Die Funktion kehrt erst zurück, wenn alle Prozessoren GlobalesMin() aufgerufen haben. Diese Funktion ist leicht mit MPI_Allreduce() aus der MPI-Bibliothek zu realisieren.

TestMsg(i, tag): Testet, ob eine Nachricht mit der Kennzeichnung *tag* von Prozessor i darauf wartet, empfangen zu werden. Falls dies der Fall ist, liefert TestMsg() das boolesche Ergebnis *true*, andernfalls *false*. TestMsg() ist also nicht blockierend. Für den Test auf Empfangsbereitschaft einer beliebigen Nachricht von einem beliebigen Prozessor kann $tag := any$ bzw. $i := any$ eingesetzt werden. Diese Funktion ist leicht mit MPI_Iprobe() aus der MPI-Bibliothek zu realisieren.

3.3.2.2 Startphase: Aufteilung der Startbox, Initialisierung

Aufteilung der Startbox X auf p Prozessoren. Zu Beginn des Algorithmus muß jeder der p Prozessoren eine separate Teilbox $X^i,\ i = 1, \ldots, p$ der Startbox X zugeteilt bekommen. Jeder Prozessor p_i beginnt dann parallel mit der Bearbeitung von X_i. Nach einer ersten Berechnungsphase findet dann der im folgenden Abschnitt beschriebene Lastausgleich statt.

Die Aufteilung von X findet parallel auf allen Prozessoren statt. Zu Beginn versendet Prozessor p_1 per Broadcast die Startbox X an jeden Prozessor. Jeder Prozessor bestimmt dann die für ihn bestimmte Teilbox X^i nach folgendem Schema:

Um auch für Prozessorzahlen p, die keine Zweierpotenz sind, eine sinnvolle Aufteilung zu finden, wird zunächst eine Zerlegung von p durchgeführt: $p = 2^l \cdot r,\ 2 \nmid r,\ l \geq 0$. Ist $r = 1$, so wird $r = 2$ gesetzt und l um eins verringert. Auf allen Prozessoren wird dann parallel die Gradientenauswertung $\nabla F(X)$ über der Startbox X und anschließend ein Sortierungsvektor $s = (s_1, \ldots, s_n)$ für die optimalen Teilungsrichtungen (vgl. Abschnitt 3.2.5) bestimmt. Die Komponente X_{s_1} wird dann zunächst r-mal geteilt, die weiteren Komponenten X_{s_2} bis X_{s_l} werden halbiert. Ist $l > n$, so wird an entsprechender Stelle wieder bei der Komponente X_{s_1} angefangen. Abbildung 3.5 verdeutlicht die Situation für $n = 2$ und $p = 24 = 3 \cdot 2^3$. Komponente X_{s_1} wird in sechs $(3 \cdot 2)$ gleiche Teile geteilt, Komponente X_{s_2} in vier $(2 \cdot 2)$.

Der nachfolgend abgedruckte Algorithmus 3.15 bestimmt die dem Prozessor p_i zugeordnete Teilbox X^i. Die Funktion BerechneStartbox() liefert die

X^{19}	X^{20}	X^{21}	X^{22}	X^{23}	X^{24}
X^{13}	X^{14}	X^{15}	X^{16}	X^{17}	X^{18}
X^{7}	X^{8}	X^{9}	X^{10}	X^{11}	X^{12}
X^{1}	X^{2}	X^{3}	X^{4}	X^{5}	X^{6}

Abbildung 3.5: Aufteilung der Startbox X $(n = 2, p = 24)$

Teilbox X^i in der übergebenen Variablen X zurück. In Schritt 4 werden die Anzahl t_i der Teile für Komponente X_{s_i} bestimmt. Im Beispiel ist also $t_1 = 6$ und $t_2 = 4$. Anschließend wird in Schritt 6 die Position pos_i der Box X^i in jeder Komponente berechnet. Im Beispiel ist für die Box X^{17} die Position $pos_1 = 5$ und $pos_2 = 3$. Zum Schluß wird in Schritt 7 X^i bestimmt. Die relativ komplizierte iterative Berechnungsweise von X^i in Schritt 7 ist damit zu begründen, daß kein Teil der Startbox X durch Rundungsfehler verloren gehen darf. Umgekehrt muß auch $X^i \cap X^j = \emptyset$ für $i \neq j$ gelten, falls X^i und X^j keine Nachbarn sind. Andernfalls (X^i und X^j benachbart) muß $d(X^i \cap X^j) = 0$ gelten. Aus diesen Gründen scheidet eine vereinfachte Berechnung wie $W := [\underline{X_i} + h \cdot (pos_i - 1), \underline{X_i} + h \cdot pos_i]$ aus.

Algorithmus 3.15: BerechneStartbox(X)
Aufteilung der Startbox

1. **for** $i := 1$ **to** n **do** $t_i := 1$; $pos_i := 1$;
2. Bestimme l, r mit $p = 2^l \cdot r$; $k := 1$; OptKompSort($X, \nabla F(X), s$);
3. **if** $(r > 2)$ **then** $t_1 := r$; $k := 2$;
4. **while** $(l > 0)$ **do**
   a. **if** $(k > n)$ **then** $k := 1$;
   b. $t_k := 2t_k$; $k := k + 1$; $l := l - 1$;
5. $l := n$; $k := myid - 1$;

6. **for** $i := 1$ **to** n **do**
   a. **if** $(t_i = 1)$ **then exit**$_{for}$;
   b. $l := l/t_i;\ pos_i := 1 + (k/l);\ k := k \bmod l;$
7. **for** $j := 1$ **to** n **do**
   a. $i := s_j$; **if** $(t_j = 1)$ **then next**$_j$;
   b. $h := d(X_i)/t_i;\ ub := \overline{X_i};\ W := [\underline{X_i}, \underline{X_i} + h];$
   c. **for** $m := 2$ **to** pos_j **do**
      if $(m < t_j)$ **then** $W := [\overline{W}, \overline{W} + h]$; **else** $W := [\overline{W}, ub]$;
   d. $X_i := W$;
8. **return** X;

Initialisierung. Die Startphase für den parallelen Algorithmus wird durch die nachfolgend beschriebene Funktion StartGO() (Algorithmus 3.16) realisiert. Zunächst wird auf jedem Prozessor die ihm zugeordnete Teilbox der Startbox X bestimmt. Durch eine Funktionsauswertung am Mittelpunkt sowie durch den anschließenden Aufruf von GlobalesMin() steht dann auf jedem Prozessor eine brauchbare obere Schranke $\widetilde{f}$ für das globale Minimum zur Verfügung. Anschließend werden die globalen Variablen initialisiert.

Algorithmus 3.16: StartGO$(X, \widetilde{f})$
Startphase für den parallelen Algorithmus

1. BerechneStartbox(X); $c := m(X)$; $\widetilde{f} := \overline{F(c)}$; GlobalesMin$(\widetilde{f})$;
2. *fertig* := *false*; $w_{own} := 1$; **for** $i := 1$ **to** k **do** $w_k := 1$;
3. $\Delta := 0.1$; $\Delta_{up} := 0.1$; $\Delta_{down} := 0.1$; *min.weight* := 5;
4. **M: for** $i := 1$ **to** p **do** *abgemeldet*$_i$:= *false*;
5. **return** $X, \widetilde{f}$;

3.3.2.3 Berechnungsphase: Bearbeitung von Teilboxen, Lastausgleich

Verteilen von $\widetilde{f}$. Wird auf einem Prozessor p_i eine verbesserte obere Schranke $\widetilde{f}$ für das globale Minimum gefunden, so muß diese so schnell wie möglich allen Prozessoren bekannt gemacht werden, damit auf den anderen Prozessoren keine Teilboxen unnötig bearbeitet werden oder die Arbeitslisten L zu stark anwachsen. Dazu werden im parallelen Algorithmus die beiden nachfolgend beschriebenen Routinen EmpfangeFmax() und VerteileFmax() verwendet.

Algorithmus 3.17: EmpfangeFmax($\widetilde{f}, i$)
Empfangen eines besseren $\widetilde{f}$ von einem Nachbarprozessor p_i

1. **while** (TestMsg($i := any$, NEWSOLUTION)) **do**
   a. Recv(p_i, NEWSOLUTION, r);
   b. **if** ($r < \widetilde{f}$) **then** $\widetilde{f} := r$;
2. **return** $\widetilde{f}, i$;

Die nachfolgend beschriebene Funktion VerteileFmax() (Algorithmus 3.18) dient sowohl zum Verteilen einer selbst gefundenen besseren oberen Schranke $\widetilde{f}$ für das globale Minimum an alle Nachbarprozessoren als auch zur Weiterleitung an vom Ursprungsprozessor noch nicht informierte Nachbarprozessoren.

Algorithmus 3.18: VerteileFmax($\widetilde{f}, i$)
Verteilen eines besseren $\widetilde{f}$ an Nachbarprozessoren

if ($i = myid$) **then for** $j := 1$ **to** k **do**
  Send($\tilde{p}_j$, NEWSOLUTION, $\widetilde{f}$);
else for $j := 1$ **to** k **do**
  if ($\tilde{p}_j$ ist nicht Nachbar von p_i) **then**
    Send($\tilde{p}_j$, NEWSOLUTION, $\widetilde{f}$);

Lastausgleich zwischen den Prozessoren in der Berechnungsphase. In der Berechnungsphase des parallelen Algorithmus, solange also die eige-

ne Arbeitsliste L nicht leer ist, findet nach jeder Behandlung einer Teilbox Y durch BearbeiteBox() (Algorithmus 3.14) ein Lastausgleich zwischen den Nachbarprozessoren statt. Die Beschreibung des Lastverteilungsalgorithmus gliedert sich in zwei Teile: Die Funktion BearbeiteNachrichtLV() behandelt eintreffende Nachrichten von einem Nachbarprozessor p_i. Das Arbeitsprinzip dieser Funktion entspricht im wesentlichen dem in Kapitel 2 beschriebenen Algorithmus 2.1. Dort sind auch weitere Erläuterungen zu finden. Zum ursprünglichen Algorithmus hinzugekommen sind hier noch weitere Kennzeichnungen (*tags*) von Nachrichten zur korrekten Abmeldung der Prozessoren untereinander. Diese *tags* müssen natürlich entsprechend behandelt werden. Die verwendete Methode zur Abmeldung aller Prozessoren untereinander ist genauer im nächsten Abschnitt erläutert.

Algorithmus 3.19: BearbeiteNachrichtLV($L, \widehat{L}, \widetilde{f}, i, tag, j$)
Bearbeitung einer Nachricht im Lastverteilungsalgorithmus

1. **if** (tag = REQUEST) **then**
   a. Recv(p_i, REQUEST, m); $w_j := m$;
   b. **if** ($m < w_{own} \cdot (1 - \Delta)$ **and** $w_{own} > min.weight$) **then**
      A. $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$;
      B. Send(p_i, WORK, $Y, \underline{F_Y}$); w_{own}--; w_j++;
2. **if** (tag = WORK) **then**
   a. Recv(p_i, WORK, $V, \underline{F_V}$); **if** ($w_j > 0$) **then** w_j--;
   b. **if** ($\underline{F_V} < \widetilde{f}$) **then**
      w_{own}++; Send(p_i, REQUEST, w_{own}); $L := L + (V, \underline{F_V})$;
3. **if** (tag = INFORM) **then**
   a. Recv(p_i, INFORM, m);
   b. **if** ($m = -1$) **then** $m := 1$; Send(p_i, INFORMACK, w_{own});
   c. $w_j := m$; Send(p_i, REQUEST, w_{own});

4. **if** (tag = FINAL) **then**
   a. Recv(p_i, FINAL);
   b. **if** ($w_{own} > 1$) **then**
      A. $(Y, \underline{F_Y})$:= Head(L); $L := L - (Y, \underline{F_Y})$;
      B. Send(p_i, WORK, $Y, \underline{F_Y}$); $w_{own} := w_{own} - 1$; $w_j := 1$;
   else $w_j = -1$; Send(p_i, FINALACK, w_{own});
5. **if** (tag = FINALACK) **then** Recv(p_i, FINALACK, m); $w_j := m$;
6. **M: if** (tag = GOODBYE) **then**
   a. Recv(p_i, GOODBYE, m); $abgemeldet_i := m$;
   b. **if** ($\forall j : abgemeldet_j = true$) **then for** $j := 1$ **to** k **do** $w_j := -1$;
7. **if** (tag = COLLECT) **then**
   a. EmpfangeLösung($i, \widehat{L}, \widetilde{f}$); $abgemeldet_i := true$;
   b. **if** (p_i ist Nachbar von p_1) **then** $w_j := -1$;
   c. Mittelpunktstest($L, \widetilde{f}$); w_{own} := Length(L);
8. **if** (tag = NEWSOLUTION) **then**
   a. Recv(p_i, NEWSOLUTION, r);
   b. **if** ($r < \widetilde{f}$) **then**
      A. $\widetilde{f} := r$; VerteileFmax($\widetilde{f}, i$);
      B. Mittelpunktstest($L, \widetilde{f}$); w_{own} := Length(L);
9. **return** $(L, \widehat{L}, \widetilde{f})$;

Der zweite Teil des Lastverteilungsalgorithmus ist die Funktion LastVerteilung(). Zu Beginn wird die Anfrage nach Teilboxen an die Nachbarn gerichtet, falls entweder die eigene Last stark abgenommen hat oder die Arbeitsliste L leer ist. Anschließend wird getestet, ob eine Nachricht von einem Nachbarprozessor vorliegt. Ist dies der Fall, wird die Funktion BearbeiteNachrichtLV() aufgerufen, die Nachricht entsprechend abgearbeitet und das Vorliegen einer weiteren Nachricht überprüft. Falls die eigene Last stark angestiegen ist und genügend Teilboxen in der Arbeitsliste L vorhanden sind, werden Teilboxen an zufällig ausgesuchte Nachbarprozessoren versandt, deren Last geringer ist.

Algorithmus 3.20: LastVerteilung($L, \widehat{L}, \widetilde{f}$)
Lastverteilung und -ausgleich zwischen den Nachbarprozessoren

1. **if** ($w_{own} < w_{old} \cdot (1 - \Delta_{down})$ **or** $w_{own} = 0$) **then**
   for $j := 1$ **to** k **do** Send($\tilde{p}_j$, REQUEST, w_{own});
2. **while** (TestMsg(i:=*any*, *tag* := *any*)) **do**
   a. Bestimme j mit $\tilde{p}_j = p_i$;
   b. BearbeiteNachrichtLV($L, \widehat{L}, \widetilde{f}, i, tag, j$);
3. **if** ($w_{own} > w_{old} \cdot (1 + \Delta_{up})$ **and** $w_{own} > min.weight$) **then**
   a. Unmark(alle Nachbarn);
   b. **for** $j := 1$ **to** k **do**
      A. Wähle zufälligen nicht markierten Nachbarn $\tilde{p}_j$; Mark($\tilde{p}_j$);
      B. **if** ($w_j < w_{own} \cdot (1 - \Delta)$) **then**
         1. $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$;
         2. Send($\tilde{p}_j$, WORK, $Y, \underline{F_Y}$); w_j++; w_{own}--;
   c. **for** $j := 1$ **to** k **do** Send($\tilde{p}_j$, INFORM, w_{own});
   d. $w_{old} := w_{own}$;
4. **return** $(L, \widehat{L}, \widetilde{f})$;

3.3.2.4 Sammlungsphase: Empfangen neuer Teilboxen oder Beenden der Arbeit

Da jeder einzelne Prozessor p_i nur über die Lastsituation der Nachbarprozessoren informiert ist, nicht jedoch über die Auslastung des gesamten parallelen Netzwerks, erfolgt die Beendigung der Arbeit in mehreren Phasen. Dadurch wird verhindert, daß einzelne Prozessoren zu früh aus der Mitarbeit an der Berechnung ausscheiden, obwohl eventuell zu einem späteren Zeitpunkt wieder mehr zu untersuchende Teilboxen vorliegen. Bis zur endgültigen Abmeldung beim I/O-Prozessor p_1 in der letzten Teilphase der Abmeldung nimmt jeder Prozessor noch am Lastverteilungsalgorithmus teil, wird also über die aktuelle Last der Nachbarprozessoren sowie über das beste bekannte $\widetilde{f}$ informiert. Wird von einem Nachbarprozessor eine Teilbox Y zur Bearbeitung empfangen, so wird zurück in die Berechnungsphase (vgl.

den vorigen Abschnitt 3.3.2.3) gesprungen und die reguläre Bearbeitung von Teilboxen wieder aufgenommen.

Warten auf Leerlaufen der Nachbarprozessoren. In der ersten Phase der Beendigung der Arbeit wird geprüft, ob die Nachbarprozessoren $\tilde{p}_1, \ldots, \tilde{p}_k$ noch Teilboxen zur Bearbeitung vorliegen haben. Ist dies der Fall (d.h.: $\exists j : w_j > 0$), so nimmt p_i weiterhin am Lastverteilungsalgorithmus teil. Dies wird in Schritt 4 des in Abschnitt 3.3.2.6 beschriebenen Hauptalgorithmus 3.25 realisiert.

Abmelden bei den Nachbarprozessoren. In der zweiten Phase der Beendigung der Arbeit erfolgt dann ein aktives Abmelden bei den Nachbarprozessoren. In Abmelden() (Algorithmus 3.22) wird an jeden Nachbarprozessor eine Nachricht mit der Kennzeichnung FINAL übersandt. Diese wird durch FINALACK von den Nachbarprozessoren bestätigt, falls diese ebenfalls eine leere Arbeitsliste L haben. Ferner setzt der Nachbarprozessor das entsprechende $(\tilde{p}_j = p_i)$ $w_j = -1$. Nachbarprozessoren mit Length$(L) > 1$ übersenden eine Teilbox zur Bearbeitung (vgl. Algorithmus 3.19). In diesem Fall wird in die Berechnungsphase zurückgekehrt.

Die Funktion BearbeiteNachrichtAbm() (Algorithmus 3.21) übernimmt die Bearbeitung von übersandten Nachrichten während der Abmeldung. Im Vergleich mit der Funktion BearbeiteNachrichtLV(), die übersandte Nachrichten während der Berechnungsphase bearbeitet, entfallen hier einige Anweisungen, die Teilboxen an Nachbarprozessoren versenden (Arbeitsliste L ist leer). Dafür muß hier zusätzlich noch das Abmelden und das eventuelle Wiederanmelden, falls doch noch von einem Nachbarprozessor eine Teilbox zur Bearbeitung übersandt werden sollte, behandelt werden. Die endgültige Abmeldung darf erst dann erfolgen, wenn dies von allen Nachbarprozessoren durch übersenden einer Nachricht mit der Kennzeichnung FINALACK bestätigt ist. Dazu wird ein boolescher Vektor *abm_ack* verwendet. Falls doch eine Wiederanmeldung erfolgt, wird eine entsprechende Nachricht (tag = INFORM mit Wert -1) an alle Nachbarprozessoren übersandt. Dieses Wiederanmelden wird ebenfalls durch INFORMACK bestätigt und in der entsprechenden Komponente des booleschen Vektors *anm_ack* abgespeichert. Außerdem wird eine Wiederanmeldung dem I/O-Prozessor p_1 bekannt gemacht (Nachricht mit Kennzeichnung GOODBYE und Inhalt *false*).

Algorithmus 3.21: BearbeiteNachrichtAbm$(L, \widehat{L}, \widetilde{f}, i, tag, j)$
Bearbeitung einer Nachricht während der Endphase

1. **if** (tag = FINAL) **then**
   Recv(p_i, FINAL); $w_j = -1$; Send$(p_i, \text{FINALACK}, w_{own})$;
2. **if** (tag = FINALACK) **then**
   Recv$(p_i, \text{FINALACK}, m)$; $w_j := m$; $abm\_ack_j := true$;
3. **if** (tag = NEWSOLUTION) **then**
   a. Recv$(p_i, \text{NEWSOLUTION}, r)$;
   b. **if** $(r < \widetilde{f})$ **then** $\widetilde{f} := r$; VerteileFmax$(\widetilde{f}, i)$;
4. **if** (tag = WORK) **then**
   a. Recv$(p_i, \text{WORK}, V, \underline{F_V})$; $abm\_ack_j := true$;
   b. **if** $(w_j > 0)$ **then** w_j--;
   c. **if** $(\underline{F_V} < \widetilde{f})$ **then**
      A. **if** $(w_{own} < 0)$ **then** $w_{own} := 1$; **else** w_{own}++;
      B. $L := L + (V, \underline{F_V})$;
      C. **if** $(w_{own} = 1)$ **then**
         1. Send$(p_1, \text{GOODBYE}, false)$;
         2. **for** $j := 1$ **to** k **do**
            $anm\_ack_j := false$; Send$(\tilde{p}_j, \text{INFORM}, -1)$;
5. **if** (tag = INFORM **or** tag = REQUEST) **then**
   a. Recv(p_i, tag, m);
   b. **if** $(m = -1)$ **then** $m := 1$; Send$(p_i, \text{INFORMACK}, w_{own})$;
   c. $w_j := m$; Send$(p_i, \text{REQUEST}, w_{own})$;
6. **if** (tag = INFORMACK) **then**
   Recv$(p_i, \text{INFORMACK}, m)$; $w_j := m$; $anm\_ack_j := true$;

7. **if** (tag = COLLECT) **then**
   a. EmpfangeLösung($i, \widehat{L}, \widetilde{f}$); $abgemeldet_i := true$;
   b. **if** (p_i ist Nachbar von p_1) **then** $w_j := -1$;
8. **if** (tag = GOODBYEACK) **then**
   a. Recv(p_i, GOODBYEACK); $fertig := true$;
   b. **for** $j := 1$ **to** k **do** $w_j := -1$;
9. **M: if** (tag = GOODBYE) **then**
   a. Recv(p_i, GOODBYE, m); $abgemeldet_j := m$;
   b. **if** ($\forall j : abgemeldet_j = true$) **then**
   $fertig := true$; **for** $j := 1$ **to** k **do** $w_j := -1$;
10. **return** $(L, \widehat{L}, \widetilde{f})$;

Die nachfolgend beschriebene Funktion Abmelden() (Algorithmus 3.22) realisiert die zuvor beschriebene zweite Phase der Abmeldung in Schritt 2 und den nachfolgenden Schritten. Allen Nachbarprozessoren wird eine Nachricht mit der Kennzeichnung FINAL übersandt. Anschließend wird auf die Bestätigung durch die Nachbarprozessoren gewartet bzw. neue Arbeit empfangen.

Endgültiges Abmelden beim I/O-Prozessor p_1. Die dritte Phase der Beendigung der Arbeit wird dann in Schritt 6 eingeleitet: An den I/O-Prozessor p_1 wird eine Nachricht mit der Kennzeichnung GOODBYE und dem Inhalt *true* gesendet. Erst wenn p_1 von allen Prozessoren eine derartige Nachricht erhalten hat, versendet p_1 eine Nachricht mit der Kennzeichnung GOODBYEACK an alle Prozessoren. Daraufhin beenden alle Prozessoren die Sammlungsphase und beginnen mit der im nachfolgenden Abschnitt beschriebenen Endphase, dem Sammeln der Lösungslisten auf dem I/O-Prozessor p_1.

Algorithmus 3.22: Abmelden($L, \widehat{L}, \widetilde{f}$)
Endphase des parallelen Algorithmus

1. $w_{own} = -1$;
2. **for** $i := 1$ **to** k **do**
   Send($\tilde{p}_i$, FINAL); abm_ack_i := *false*; anm_ack_i := *true*;
3. **while** ($\exists i : abm\_ack_i =$ *false* **or** $\exists i : anm\_ack_i =$ *false*) **do**
   a. **if** (TestMsg(i:=*any*, *tag* := *any*)) **then**
      A. Bestimme j mit $\tilde{p}_j = p_i$;
      B. BearbeiteNachrichtAbm($L, \widehat{L}, \widetilde{f}, i, tag, j$);
4. **if** ($L \neq \{\}$) **then return** $L, \widehat{L}, \widetilde{f}$;
5. **if** ($\max_{i=1}^{k} w_i \geq 0$) **then goto** 3.a;
6. **if** ($myid \neq 1$) **then if** (**not** *fertig*) **then**
   Send(p_1, GOODBYE, *true*); **goto** 3.a;
   else
   a. $abgemeldet_1$:= *true*;
   b. **if** ($\exists i : abgemeldet_i =$ *false*) **then goto** 3.a;
      else for $i := 2$ **to** p **do** Send(p_i, GOODBYEACK);
7. **return** $L, \widehat{L}, \widetilde{f}$;

3.3.2.5 Endphase: Einsammeln der Lösungen auf dem I/O-Prozessor

Bevor im nächsten Abschnitt der vollständige parallele Algorithmus beschrieben wird, geben wir nachfolgend zwei in der Endphase des parallelen Algorithmus benötigte Funktionen zum Einsammeln der Lösungen auf dem I/O-Prozessor p_1 an.

Der nachfolgende Algorithmus 3.23 empfängt während der laufenden Berechnung auf dem I/O-Prozessor p_1 eine Lösung (F und Lösungsliste L) von einem Nachbarprozessor p_i. Da p_i nicht die gesamte Ausgangsbox X bearbeitet hat, sondern zunächst nur die ihm zugeordnete Teilbox X^i und

später eventuell noch andere Teilboxen, muß F nicht notwendigerweise ein Einschluß für das gesuchte globale Minimum $f^\star$ sein. Zunächst wird in EmpfangeLösung() geprüft, ob $\overline{F}$ die obere Schranke $\widetilde{f}$ auf p_1 verbessern kann. Gilt außerdem noch $(\underline{F} \leq \widetilde{f})$, so wird die Lösungsliste L von p_i in die lokale Lösungsliste $\widehat{L}$ von p_1 eingefügt.

Algorithmus 3.23: EmpfangeLösung$(i, \widehat{L}, \widetilde{f})$
Empfangen einer Lösung von Nachbarprozessor p_i

1. Recv$(p_i, \text{COLLECT}, \tilde{L}, F)$;
2. **if** $(\tilde{L} \neq \{\})$ **then**
   a. **if** $(\overline{F} < \widetilde{f})$ **then** $\widetilde{f} := \overline{F}$;
   b. **if** $(\underline{F} \leq \widetilde{f})$ **then while** $(\tilde{L} \neq \{\})$ **do**
      $(Y, \underline{F_Y}) :=$ Head$(\tilde{L})$; $\tilde{L} := \tilde{L} - (Y, \underline{F_Y})$; $L := L + (Y, \underline{F_Y})$;
3. **return** $\widehat{L}, \widetilde{f}$;

Alle Prozessoren, die nicht schon während der Berechnungsphase des I/O-Prozessors ihre Lösungen an den I/O-Prozessor gesandt haben, versenden dann in der Endphase ihre Lösungen. Erst, wenn sich alle Prozessoren beim I/O-Prozessor abgemeldet haben, kann der parallele Algorithmus durch den nachfolgenden Verifikationsschritt beendet werden.

Auch hier wird vor der Aufnahme der übersandten Lösungsliste L in die lokale Lösungsliste $\widehat{L}$ zunächst das übersandte F mit dem lokal vorliegenden $F^\star$ verglichen. Dieser Vergleich entscheidet darüber, ob L in $\widehat{L}$ einsortiert wird, $\widehat{L}$ ersetzt, oder nicht übernommen wird.

Algorithmus 3.24: SammleLösung($\widehat{L}, F^\star$)
Sammeln der Lösungen auf I/O-Prozessor p_1

1. **while** ($\exists i : abgemeldet_i = false$) **do**
   if (TestMsg(i:=*any*, COLLECT)) **then**
   a. Recv(p_i, COLLECT, L, F); $abgemeldet_i := true$;
   b. **if** ($\underline{F} < \overline{F^\star}$) **then next**$_{while}$;
   c. **if** ($\overline{F} < \underline{F^\star}$) **then** $\widehat{L} := L$; $F^\star := F$; **next**$_{while}$;
   d. $F^\star := [\min(\underline{F}, \underline{F^\star}), \min(\overline{F}, \overline{F^\star})]$;
   e. **while** ($L \neq \{\}$) **do**
   $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$; $\widehat{L} := \widehat{L} + (Y, \underline{F_Y})$;
2. Komprimiere($\widehat{L}$);
3. **return** $\widehat{L}, F^\star$;

3.3.2.6 Der eigentliche parallele Algorithmus

Das Hauptprogramm, das auf allen Prozessoren parallel ausgeführt wird, ist die nachfolgend beschriebene Funktion ParalleleGlobOpt() (Algorithmus 3.25). Nacheinander werden die in den vorigen Abschnitten beschriebenen einzelnen Phasen des parallelen Algorithmus durchlaufen:

In Schritt 1 wird die Funktion StartGO(), die die Startphase realisiert, aufgerufen. In der Startphase wird durch die Funktionsauswertung am Mittelpunkt der den einzelnen Prozessoren zugeordneten Teilboxen und dem anschließenden Aufruf von GlobalesMin() eine gute obere Schranke $\widetilde{f}$ für das globale Minimum $f^\star$ bestimmt. In Schritt 2 wird dann geprüft, ob bereits für die ganze Teilbox Y die Bedingung $\widetilde{f} < \underline{F(Y)}$ erfüllt ist. Falls ja, so kann Y keine globale Minimalstelle enthalten. In diesem Fall wird direkt mit der Sammlungsphase (Anforderung von Arbeit von den Nachbarprozessoren) in Schritt 4 fortgefahren.

Ansonsten folgt auf die Startphase die Berechnungsphase (Schritt 3). Zunächst wird die aktuelle Box Y durch den neuen seriellen Algorithmus mittels der Funktion BearbeiteBox() behandelt. Aufgrund der numerischen Ergebnisse für den neuen seriellen Algorithmus (vgl. Abschnitt 3.2.10) wird dabei stets der Parameter $l = 4$ verwendet. Nach der Bearbeitung einer

Box wird ein Update von $\widetilde{f}$ durchgeführt sowie der Lastverteilungsalgorithmus 3.20 aufgerufen. Anschließend wird die vorderste Teilbox Y aus der Arbeitsliste L entnommen und entweder in die Lösungsliste $\widehat{L}$ eingefügt, falls das Abbruchkriterium erfüllt ist, oder weiter bearbeitet.

An die Berechnungsphase schließt sich die Sammlungsphase an, wenn die lokale Arbeitsliste L keine Teilboxen mehr enthält. Zunächst wird in Schritt 4 weiterhin so lange an der Lastverteilung teilgenommen, bis alle Arbeitslisten der Nachbarprozessoren keine Teilboxen mehr enthalten. Anschließend erfolgt das Abmelden bei den Nachbarprozessoren. Bestätigt der I/O-Prozessor p_1 die Abmeldung bei allen anderen Prozessoren, so wird die $F^\star$ und die Lösungsliste $\widehat{L}$ an den I/O-Prozessor p_1 verschickt. Sind alle Lösungen eingetroffen, so gibt der I/O-Prozessor p_1 das Ergebnis aus.

Algorithmus 3.25: ParalleleGlobOpt$(X, \epsilon, F^\star, \widehat{L})$
Paralleler Algorithmus zur verifizierten globalen Optimierung

1. StartGO$(X, \widetilde{f})$; $\epsilon_{Newton} := 1$; $Y := X$; $L := \{\}$; $\widehat{L} := \{\}$;
2. $F_Y := F(Y)$; **if** $(\widetilde{f} < \underline{F_Y})$ **then** $w_{own} := 0$; **goto** 4;
3. **do**
   a. $\widetilde{f}_{old} := \widetilde{f}$; BearbeiteBox$(\epsilon, \epsilon_{Newton}, 4, Y, F_Y, \nabla F(Y), L, \widetilde{f})$;
   b. EmpfangeFmax$(\widetilde{f}_{old}, src)$;
   c. **if** $(\widetilde{f}_{old} < \widetilde{f})$ **then** $\widetilde{f} := \widetilde{f}_{old}$; VerteileFmax$(\widetilde{f}, src)$;
   d. **if** $(\widetilde{f} < \widetilde{f}_{old})$ **then** VerteileFmax$(\widetilde{f}, myid)$;
   e. Mittelpunktstest$(L, \widetilde{f})$; $w_{old} := w_{own}$; $w_{own} :=$ Length(L);
   f. LastVerteilung$(L, \widehat{L}, \widetilde{f})$; *weiter* := *false*;
   g. **while** $(L \neq \{\})$ **do**
      A. $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$; w_{own}--; $F^\star := [\underline{F_Y}, \widetilde{f}]$;
      B. **if** $((d_{\mathrm{rel}}(F^\star) < \epsilon)$ **or** $(d_{\mathrm{rel}}(Y) < \epsilon))$ **then** $\widehat{L} := \widehat{L} + (Y, \underline{F_Y})$;
         else *weiter* := *true*; **exit**$_{while}$;
   while (*weiter*);

4. **do**
   a. $\widetilde{f}_{old} := \widetilde{f}$; EmpfangeFmax($\widetilde{f}, src$);
   b. **if** ($\widetilde{f} < \widetilde{f}_{old}$) **then** VerteileFmax($\widetilde{f}, src$);
   c. LastVerteilung($L, \widehat{L}, \widetilde{f}$);
   d. **if** ($L \neq \{\}$) **then**
      $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$; w_{own}--; **goto** 3;
   while ($\max_{i=1}^{k} w_i > 0$);
5. Abmelden($L, \widehat{L}, \widetilde{f}$);
6. **if** ($L \neq \{\}$) **then**
      $(Y, \underline{F_Y}) :=$ Head(L); $L := L - (Y, \underline{F_Y})$; w_{own}--; **goto** 3;
7. Mittelpunktstest($\widehat{L}, \widetilde{f}$); $(Y, \underline{F_Y}) :=$ Head($\widehat{L}$); $F^\star := [\underline{F_Y}, \widetilde{f}]$;
8. Send(p_1, COLLECT, $\widehat{L}, F^\star$);
9. **M:** SammleLösungen($\widehat{L}, F^\star$); **return** $\widehat{L}, F^\star$;

3.3.3 Numerische Ergebnisse und Effizienzbetrachtungen für den parallelen Algorithmus

In diesem Abschnitt werden die numerischen Ergebnisse für den im vorigen Abschnitt beschriebenen parallelen Algorithmus vorgestellt. Es werden Laufzeiten (in STU), die Effizienz sowie Graphiken zur Beschleunigung für die in Anhang A beschriebenen Testprobleme angegeben. Alle Testprobleme konnten erfolgreich parallel auf 4, 8, 16, 32, 64 und 96 Prozessoren berechnet werden. Es werden $k = 4$ Nachbarprozessoren im parallelen Algorithmus verwendet. Dabei ist jeder Prozessor p_i über die Lastsituation seiner beiden Vorgänger p_{i-2} ($= \tilde{p}_1$) und p_{i-1} ($= \tilde{p}_2$) sowie seiner beiden Nachfolger p_{i+1} ($= \tilde{p}_3$) und p_{i+2} ($= \tilde{p}_4$) (mit entsprechendem *wrap-around* bei den Randprozessoren) unterrichtet. Man kann sich also die Prozessoren wie in einem Ring angeordnet vorstellen.

Tabelle 3.9 enthält Angaben zu Laufzeit und Effizienz des parallelen Algorithmus für 4, 8 und 16 Prozessoren. Als zusätzliche Information ist in Spalte 2 („It_{ser}“) die Iterationsanzahl (= Anzahl der Aufrufe von BearbeiteBox()) für den seriellen Algorithmus angegeben. In Spalte 3 („$p = 1$“) ist

die Laufzeit t_{ser} des seriellen Algorithmus abgedruckt. Die weiteren Spalten enthalten die jeweiligen parallelen Laufzeiten $t_{\text{par}}(p)$ sowie die erzielte Effizienz $E(p)$.

Wir erkennen, daß sich die Testprobleme L12, L18, G7 und G10 extrem schlecht parallelisieren lassen. Durch die Parallelisierung erhöht sich sogar die Laufzeit im Vergleich mit dem seriellen Algorithmus. Begründen läßt sich dieser Effekt durch die geringe Anzahl an Iterationen zur Lösung des Testproblems. Beispielsweise benötigt L18 nur 45 Iterationen. Um etwa mit $p = 16$ Prozessoren eine optimale Beschleunigung zu erzielen, dürfte der parallele Algorithmus nur ungefähr $\frac{45}{p} \approx 3$ Iterationen auf jedem Prozessor benötigen. Diese Testprobleme wurden daher mit 32, 64 und 96 Prozessoren nicht getestet. Sie sind von den weiteren Betrachtungen ausgenommen.

Eingeschränkt für die Parallelisierung mit einer großen Anzahl an Prozessoren geeignet sind die Testprobleme R4, H6, GEO1, JS und S2.7. Diese Probleme benötigen zwischen 100 und 400 Iterationen im seriellen Algorithmus. Bei mehr als 8 bzw. 16 Prozessoren sinkt die Effizienz stark ab, da der parallele Algorithmus in jedem Fall etwa 15-20 Iterationen zur Lösung eines Problems benötigt, der Quotient $\frac{It_{\text{ser}}}{p}$ aber darunter liegt. Auch diese Probleme wurden mit 32, 64 und 96 Prozessoren nicht mehr getestet.

Prob.	It_{ser}	$p = 1$	$p = 4$ ($E(p)$)	$p = 8$ ($E(p)$)	$p = 16$ ($E(p)$)
R4	260	10.42	3.07 (85%)	1.77 (74%)	1.47 (44%)
L12	58	23.35	33.37 (17%)	26.73 (11%)	30.12 (5%)
L18	45	12.90	14.25 (23%)	12.72 (13%)	14.74 (5%)
G7	60	13.73	15.12 (23%)	15.33 (11%)	16.44 (5%)
G10	68	22.92	23.26 (25%)	25.28 (11%)	25.24 (6%)
GP	2781	61.49	11.73 (131%)	5.88 (131%)	3.26 (118%)
H6	330	25.51	8.71 (73%)	5.62 (57%)	5.39 (30%)
S2.14	854	10.20	3.41 (75%)	2.11 (60%)	1.51 (42%)
GEO1	270	13.85	4.88 (71%)	2.87 (60%)	2.54 (34%)
GEO2	571	28.87	6.67 (108%)	3.96 (91%)	2.43 (74%)
GEO3	544	27.40	5.32 (129%)	4.62 (74%)	2.90 (59%)
JS	223	45.71	15.78 (72%)	7.84 (73%)	8.95 (32%)
S2.7	110	79.49	30.43 (65%)	27.60 (36%)	21.70 (23%)
L3	906	273.46	54.86 (125%)	35.67 (96%)	15.21 (112%)
R8	553	141.63	35.97 (98%)	30.92 (57%)	40.89 (22%)
HM3	840	294.13	69.79 (105%)	40.17 (92%)	20.11 (91%)
HM4	3049	1724.53	431.15 (100%)	219.77 (98%)	109.88 (98%)
KOW	184820	440908.98	4370.80 (2522%)	1586.73 (3473%)	694.22 (3969%)

Tabelle 3.9: Laufzeiten (in STU) und Effizienz $E(p)$ des parallelen Algorithmus für 4, 8 und 16 Prozessoren

Die stark überlineare Beschleunigung des Testproblems KOW ist durch die große Reduzierung der maximalen Arbeitslistenlänge (vgl. Tabelle 3.11) bei der Parallelisierung zu erklären. Im wesentlichen wird beim Testproblem KOW nur Bisektion durchgeführt. Daher wächst während eines Großteils der Berechnung die Arbeitsliste linear an, bis die richtige Lösungsbox gefunden ist. Außerdem ist die verwendete Listenverwaltung (die aus der *Toolbox* [21] stammt) nicht optimal implementiert. Beim Einfügen eines Listenelements muß jedesmal die gesamte Liste durchlaufen werden, dies schlägt bei dem Testproblem KOW dann stark zu Buche. Für Testprobleme, die eine derartig große Arbeitsliste benötigen, wäre eine andere Datenstruktur für die Arbeitsliste (z.B. ein balancierter Baum) geeigneter. Zu beachten ist jedoch, daß der Mittelpunktstest dann nicht mehr so effizient durchzuführen ist. Für kleinere Testprobleme kann sich dies sogar nachteilig in der Laufzeit bemerkbar machen.

In Tabelle 3.10 sind die entsprechenden Laufzeit- und Effizienzangaben für die restlichen Testprobleme mit 32, 64 und 96 Prozessoren aufgeführt.

Prob.	It_{ser}	$p=1$	$p=32$ ($E(p)$)	$p=64$ ($E(p)$)	$p=96$ ($E(p)$)
GP	2781	61.49	1.94 (99%)	1.66 (58%)	1.45 (44%)
S2.14	854	10.20	2.17 (15%)	1.62 (10%)	1.62 (7%)
GEO2	571	28.87	2.45 (37%)	2.11 (21%)	1.55 (19%)
GEO3	544	27.40	1.87 (46%)	2.09 (20%)	1.94 (15%)
L3	906	273.46	11.65 (73%)	11.03 (39%)	13.82 (21%)
R8	553	141.63	31.84 (14%)	27.24 (8%)	27.24 (5%)
HM3	840	294.13	10.95 (84%)	8.37 (55%)	6.54 (47%)
HM4	3049	1724.53	59.36 (91%)	32.90 (82%)	26.20 (69%)
KOW	184820	440908.98	336.38 (4096%)	247.16 (2787%)	205.25 (2238%)

Tabelle 3.10: Laufzeiten (in STU) und Effizienz $E(p)$ des parallelen Algorithmus für 32, 64 und 96 Prozessoren

Aus den Tabellen 3.9 und 3.10 lassen sich zwei Bedingungen an ein zu lösendes Problem ablesen, die eine effiziente Parallelisierung erwarten lassen:

1. Der Quotient $\frac{It_{\text{ser}}}{p}$ darf nicht zu klein werden. Für eine effiziente Parallelisierung müssen mindestens etwa 15 Iterationen auf jedem Prozessor übrig bleiben.
2. Der Quotient $\frac{t_{\text{ser}}}{p}$ darf ebenfalls nicht zu klein werden. Start- und Endphase des Algorithmus benötigen (fast) konstante Zeit, etwa 1

STU. Die theoretisch zu erwartende optimale Laufzeit $\frac{t_{\text{seq}}}{p}$ sollte daher deutlich größer sein. Auf dem verwendeten Parallelrechner entspricht 1 STU etwa 0.85 Sekunden, so daß bei seriellen Laufzeiten in diesem Bereich eine Parallelisierung sowieso noch nicht interessant ist.

Durch die Parallelisierung lassen sich auch größere Probleme lösen, die sehr viel Speicherplatz für die Arbeitsliste L benötigen. In Tabelle 3.11 ist ein Vergleich der maximalen Länge $L_{\max}$ der Arbeitsliste L für den seriellen Algorithmus mit der maximalen Länge für den parallelen Algorithmus angegeben. Die Zahl für den parallelen Algorithmus ist dabei das Maximum der $L_{\max}$ über alle Prozessoren hinweg ermittelt. Auf vielen Prozessoren wird daher noch weniger Speicherplatz benötigt.

Prob.	$p=1$	$p=4$	(%S.1)	$p=8$	(%S.1)	$p=16$	(%S.1)
R4	106	33	(31%)	14	(13%)	**9**	(8%)
L12	87	**55**	(63%)	66	(76%)	69	(79%)
L18	59	35	(59%)	35	(59%)	**34**	(58%)
G7	**76**	84	(111%)	87	(114%)	141	(186%)
G10	**101**	122	(121%)	119	(118%)	116	(115%)
GP	612	217	(35%)	142	(23%)	**100**	(16%)
H6	166	45	(27%)	26	(16%)	**11**	(7%)
S2.14	157	44	(28%)	30	(19%)	**19**	(12%)
GEO1	152	30	(20%)	17	(11%)	**13**	(9%)
GEO2	206	48	(23%)	17	(8%)	**11**	(5%)
GEO3	194	55	(28%)	31	(16%)	**17**	(9%)
JS	86	29	(34%)	**8**	(9%)	11	(13%)
S2.7	68	17	(25%)	**14**	(21%)	16	(24%)
L3	672	211	(31%)	88	(13%)	**54**	(8%)
R8	331	**95**	(29%)	100	(30%)	104	(31%)
HM3	615	201	(33%)	86	(14%)	**53**	(9%)
HM4	1847	930	(50%)	**257**	(14%)	261	(14%)
KOW	184820	21413	(12%)	12192	(7%)	**8316**	(4%)
$\sum$	190355	23664	(12%)	13329	(7%)	9355	(5%)
Ø%			(42%)		(34%)		(32%)

Tabelle 3.11: Maximale Länge der Arbeitsliste L

In den nachfolgenden Abbildungen 3.6, 3.7, 3.8, 3.9 und 3.10 sind die er-

reichten Beschleunigungen durch den parallelen Algorithmus für die verschiedenen Testprobleme dargestellt. Man erkennt, daß sich nur für schwere Probleme der Einsatz von mehr als 32 Prozessoren lohnt. Für solche Probleme läßt sich auch mit 64 oder 96 Prozessoren eine deutliche Beschleunigung erzielen.

Es zeigt sich also, daß der vorgestellte parallele Algorithmus zur verifizierten globalen Optimierung für schwere Probleme gut skalierbar ist.

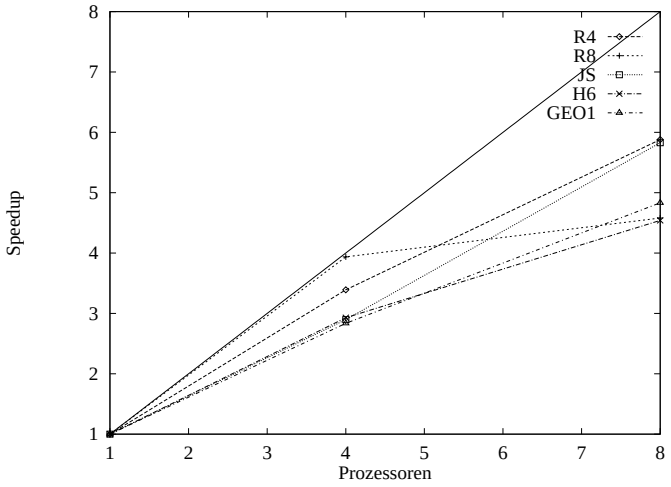


Abbildung 3.6: Beschleunigung für mittelschwere Testprobleme (bis 8 Proz.)

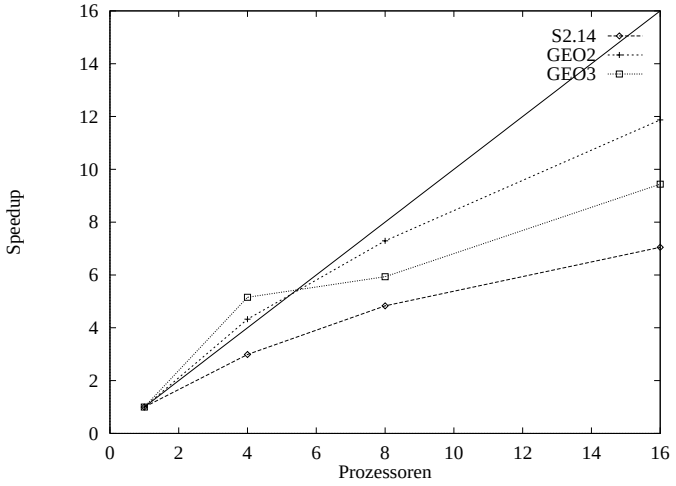


Abbildung 3.7: Beschleunigung für mittelschwere Testprobleme (bis 16 Proz.)

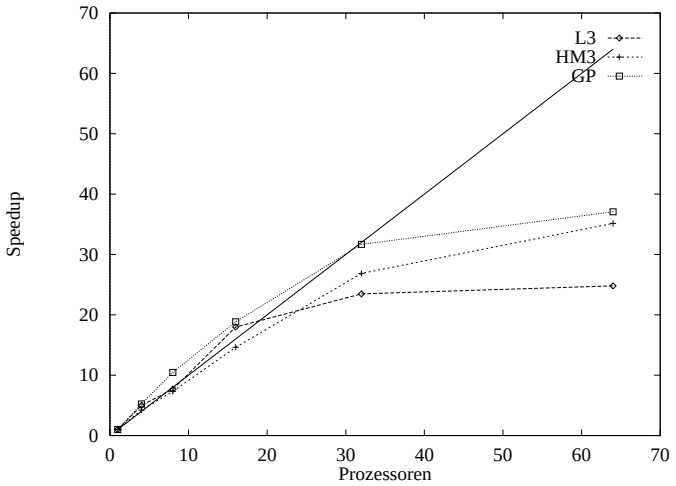


Abbildung 3.8: Beschleunigung für schwere Testprobleme (bis 64 Proz.)

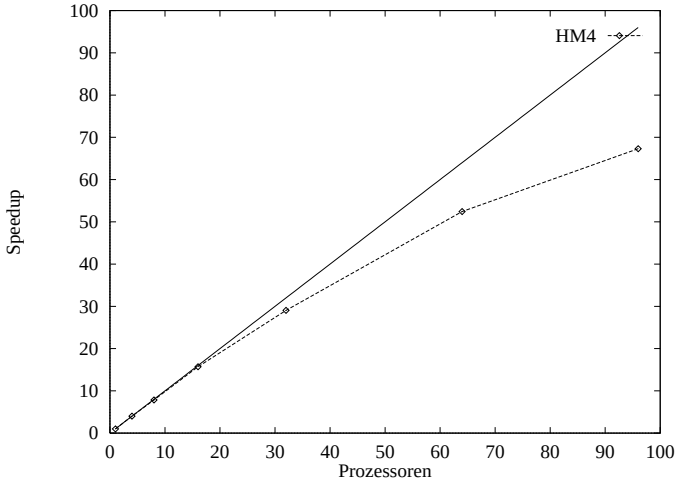


Abbildung 3.9: Beschleunigung für Testproblem HM4 (bis 96 Proz.)

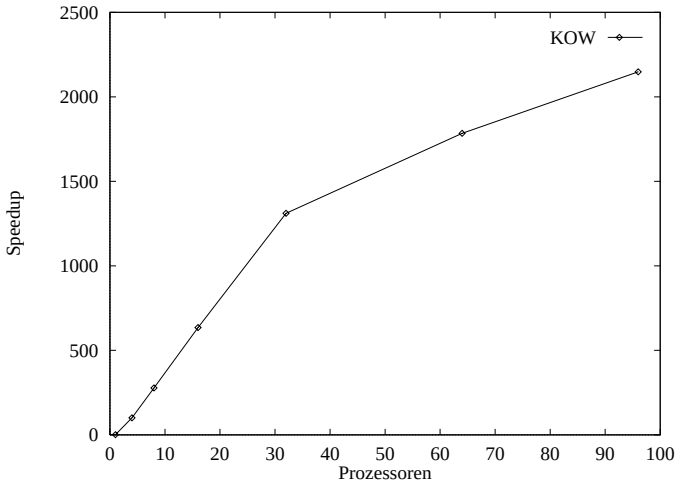


Abbildung 3.10: Beschleunigung für Testproblem KOW (bis 96 Proz.)

3.3.4 Lösung eines scheinbar einfachen Optimierungsproblems durch Parallelisierung

Bei dem in Anhang A angegebenen eindimensionalen Optimierungsproblem WK handelt es sich bei der Zielfunktion um eine rationale Funktion $f = \frac{p}{q}$, wobei p ein Polynom vom Grad 29 und q ein Polynom vom Grad 4 ist. Die ursprüngliche Problemstellung kommt aus dem Bereich von Funktionsapproximationen auf Rechenanlagen. Im vorliegenden Fall ist das globale Minimum $f^\star$ im Intervall $X = [0, 64]$ auf einige Dezimalstellen genau einzuschließen. Um dies zu erreichen, muß der Parameter ϵ für das Abbruchkriterium auf 10^{-12} gesetzt werden. Eine Schwierigkeit dieses Optimierungsproblems besteht darin, daß f auf dem gesamten Intervall X betragsmäßig sehr klein ist (vgl. Abb. auf S. 156). Es gilt nämlich:

$$\max_{x \in [0,64]} |f(x)| < 2 \cdot 10^{-15}.$$

f bewegt sich also im Rahmen der relativen Genauigkeit des verwendeten IEEE-Gleitkommasystems. Daher muß zur Lösung des Optimierungsproblems das Ausgangsintervall X sehr fein zerteilt werden. Die Arbeitsliste L, die die noch zu untersuchenden Teilintervalle speichert, wird also sehr lang und belegt viel Speicher.

Dieser starke Speicherplatzverbrauch führt dazu, daß mit vielen seriellen Algorithmen zur globalen Optimierung die Laufzeiten entweder sehr lang sind oder keine Lösung gefunden wird. Im einzelnen wurden erfolglos die folgenden seriellen Algorithmen getestet:

1. Mit dem originalen *Toolbox*-Algorithmus aus [20] konnte nach 24 Stunden Laufzeit auf einer SUN Sparcstation keine Lösung gefunden werden; das Programm wurde abgebrochen.

2. Der in dieser Arbeit vorgestellte neue serielle Algorithmus wurde nach 8 Stunden Laufzeit auf einem Knoten der IBM RS/6000 SP wegen Zeitüberschreitung ebenfalls abgebrochen.

3. Auch mit dem in [13] implementierten Verfahren von Jansson [34] konnte trotz Variation der Parameter keine Lösung gefunden werden: Der Einschluß von $f^\star$ war nicht genau genug.

4. Das Computeralgebrasystem Maple bricht mit der folgenden Fehlermeldung ab:

```
minimize(sum(p[k]*x^k ,  k=0..29) / sum(q[k]*x^k,k=0..4),x, 0..64);
---------
Error, (in sqrfree)
1st argument must be a polynomial over an algebraic number field
```

Der parallele Algorithmus zur globalen Optimierung ist jedoch in der Lage, das scheinbar einfache eindimensionale Optimierungsproblem zu lösen. Es wurde die im Anhang A angegebene Lösung nach den folgenden Laufzeiten berechnet:

1 Proz.	4 Proz.	8 Proz.	16 Proz.
> 8 Std.	48 min.	41 min.	141 min.

Man erkennt, daß bereits wenige Prozessoren ausreichen, um das Problem zu lösen. Der Einsatz von mehr als 8 Prozessoren lohnt sich bei diesem nur eindimensionalen Problem nicht. Entscheidend für die effiziente Lösbarkeit ist der verringerte Speicherplatzbedarf für die Arbeitsliste L auf den einzelnen Prozessoren.

Kapitel 4

Zusammenfassung und Ausblick

Zur effizienten Lösung globaler Optimierungsprobleme wurde in dieser Arbeit ein neuartiger Algorithmus zur verifizierten globalen Optimierung auf herkömmlichen seriellen Rechnern und auf diesen aufbauend ein vollständig verteilter und somit skalierbarer paralleler Algorithmus entwickelt. Die numerischen Ergebnisse in Abschnitt 3.2.10 belegen, daß durch den neuartigen seriellen Algorithmus deutliche Einsparungen bei Laufzeit und Auswertungsaufwand (fast doppelt so schnelle Laufzeit und mehr als halbierter Auswertungsaufwand) im Vergleich zu bisherigen verifizierenden Algorithmen für die globale Optimierung erzielt werden konnten. Entscheidende Merkmale für die Effizienzvorteile des neuartigen seriellen Algorithmus sind der adaptiv gesteuerte selektive Einsatz des Intervall-Newton-Schritts zur Reduzierung der aufwendigen Hessematrixauswertungen der Zielfunktion, die Verwendung eines Sortierungsvektors bei jeder Aufteilung der aktuell zu untersuchenden Teilbox, die geänderte Aufteilungsstrategie (das *Boxing*-Verfahren) der aktuellen Teilbox in bestimmten Situationen während der Berechnung sowie der Einsatz von Multisektion statt der bislang häufig verwendeten Bisektion.

Der in dieser Arbeit vorgestellte parallele Algorithmus zur verifizierten globalen Optimierung verwendet den neuartigen seriellen Algorithmus zur Bearbeitung von Teilboxen auf jedem Prozessor. Er ist vollständig verteilt und daher gut skalierbar. Die Kommunikation mit einem zentralen I/O-Prozessor findet nur in der Start- und in der Endphase des parallelen Algorithmus statt. Durch den Einsatz eines Arbeits- und eines Kontrollpro-

zesses sowie einer damit verbundenen *Feedback*-Strategie auf jedem Prozessor werden *Trashing*-Effekte vermieden und so eine hohe Effizienz bei der Parallelisierung erreicht. Für hinreichend schwere Optimierungsprobleme wurde eine hohe Beschleunigung bei Einsatz von bis zu 96 Prozessoren erzielt. Durch die Parallelisierung wird die Lösung von Optimierungsproblemen ermöglicht, deren Lösung auf herkömmlichen Rechnern entweder an zu hohem Speicherplatzbedarf oder an zu großer Laufzeit scheitert (vgl. Testprobleme WK (Abschnitt 3.3.4), KOW und HM4). Es ist also durch die Parallelisierung möglich, wesentlich schwierigere bzw. größere Probleme anzugehen als bisher.

Für zukünftige Arbeiten ist sicher eine Erweiterung des seriellen und parallelen Algorithmus für globale Optimierungsprobleme mit Nebenbedingungen denkbar. Zu dieser Problemstellung gibt es in der Literatur einige Arbeiten (z.B. in [24] oder [67]), die dabei verwendet werden können. Daneben sind weitere Verbesserungen an einzelnen Details des seriellen Algorithmus möglich. Es kann noch der Einfluß von verschiedenen Werten für ϵ_{Box} im *Boxing*-Schritt (Algorithmus 3.10) und die Unter- und Oberschranken für ϵ_{Newton} systematisch in einer Studie untersucht werden. Für die Steuerung des selektiven Einsatzes des Intervall-Newton-Schritts ist es auch denkbar, das jeweilige aktuelle ϵ_{Newton} und einen Indikator, ob der letzte Intervall-Newton-Schritt erfolgreich war oder nicht, zusammen mit der aktuellen Teilbox Y und der Unterschranke der Intervallauswertung $\underline{F(Y)}$ in der Arbeitsliste abzuspeichern. Dadurch kann ein besserer Bezug zwischen dem verwendeten ϵ_{Newton} und der untersuchten Teilbox hergestellt werden. Dieses Vorgehen ist jedoch recht aufwendig zu implementieren.

Globale Optimierungsprobleme mit einer nichtdifferenzierbaren Zielfunktion f (f kann etwa $|\cdot|, \min, \max,$ **if**...**then**...**else** enthalten) können mit den in dieser Arbeit beschriebenen Algorithmen nicht behandelt werden, da für die in Abschnitt 3.2.6 beschriebenen beschleunigenden Methoden die zweimalige Differenzierbarkeit von f vorausgesetzt wird. Es ist zwar möglich, auf diese Methoden zu verzichten, jedoch werden dann die Berechnungszeiten wesentlich größer.

Ein kürzlich entwickelter, vielversprechender Ansatz ist die Verwendung eines Algorithmus, der auf einer Steigungsarithmetik (siehe etwa [1], [43], [64]) sowie eines anderen *Pruning*-Schritts zur Verkleinerung der aktuellen Box beruht. Dabei ist im Prinzip nur die Funktion BearbeiteBox() im seriellen Algorithmus komplett auszutauschen. Der für BearbeiteBox() notwendige andere *Pruning*-Schritt ist für den eindimensionalen Fall in [72] dar-

gestellt. Eine Grundidee für den *Pruning*-Schritt findet sich bereits in [25]. Dort wird jedoch die dreimalige Differenzierbarkeit der Zielfunktion vorausgesetzt. Der kompliziertere mehrdimensionale Fall wird in [73] behandelt. Prinzipiell läßt sich also dieser andere serielle Algorithmus mit dem in dieser Arbeit beschriebenen parallelen Lastverteilungsalgorithmus zu einem neuen parallelen Algorithmus zur verifizierten globalen Optimierung für nichtdifferenzierbare Zielfunktionen f kombinieren. Ein solches Vorgehen erscheint recht erfolgversprechend.

Anhang A

Untersuchte Testprobleme

In dieser Arbeit wurden die nachfolgend aufgeführten Standardtestprobleme der globalen Optimierung untersucht. Neben der Angabe der Funktionsdefinition, der Startbox X und des für das Abbruchkriterium verwendeten Parameters ϵ werden die durch den seriellen Algorithmus 3.13 berechneten Einschließungen für das globale Minimum und für die globalen Minimalstellen aufgelistet.

Die behandelten Testprobleme werden nach der seriellen Laufzeit in drei Klassen eingeteilt: Leichte, mittelschwere und schwere Testprobleme. Die leichten Testprobleme mit einer seriellen Laufzeit von unter 10 STU werden für die Parallelisierung nicht untersucht, da die bereits geringe serielle Laufzeit eine Parallelisierung nicht sinnvoll erscheinen läßt.

Alle Testprobleme nehmen ihr globales Minimum im Innern der Startbox X an, so daß eine gesonderte Randbehandlung für den seriellen und parallelen Algorithmus nicht implementiert wurde.

Alle in den Testproblemen auftretende Konstanten wurden bei der Realisierung auf dem Rechner in enge Intervalle eingeschlossen. Dadurch wird das Problem der Nichtdarstellbarkeit von bestimmten Dezimalzahlen auf dem Rechner (z.B. 0.1) gelöst.

A.1 Leichte Testprobleme

S5 ($n = 4$, Shekel 5, vgl. [76])

$$f(x) = -\sum_{i=1}^{5} \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

mit $X = [0, 10]^4$, $\epsilon = 10^{-6}$. Es ist

$$A = \begin{pmatrix} 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 8 & 8 & 8 & 8 \\ 6 & 6 & 6 & 6 \\ 3 & 7 & 3 & 7 \\ 2 & 9 & 2 & 9 \\ 5 & 5 & 3 & 3 \\ 8 & 1 & 8 & 1 \\ 6 & 2 & 6 & 2 \\ 7 & 3.6 & 7 & 3.6 \end{pmatrix} \quad \text{und} \quad c = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[-10.153206961890, -10.153199046665]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[4.000037152819, 4.000037152820]
[4.000133276591, 4.000133276592]
[4.000037152819, 4.000037152820]
[4.000133276591, 4.000133276592]
```

S7 ($n = 4$, Shekel 7, vgl. [76])

$$f(x) = -\sum_{i=1}^{7} \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

mit $X = [0, 10]^4$, $\epsilon = 10^{-6}$, A, c wie bei S5.

Globales Minimum $f^\star \in$ `[-10.402940569572, -10.402939967006]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[4.000572916185, 4.000572916186]
[4.000689366185, 4.000689366186]
[3.999489708859, 3.999489708860]
[3.999606158858, 3.999606158859]
```

S10 ($n = 4$, Shekel 10, vgl. [76])

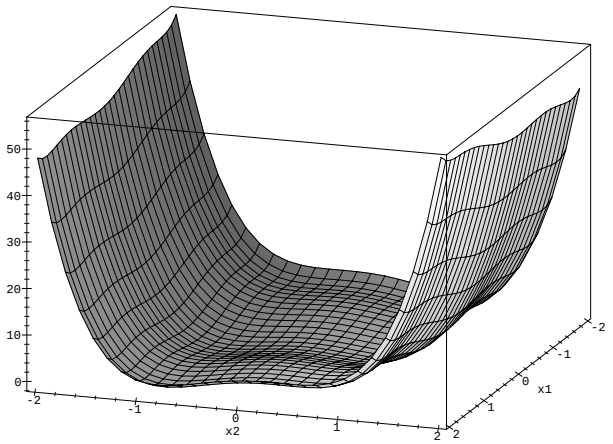
$$f(x) = -\sum_{i=1}^{10} \frac{1}{(x - A_i)(x - A_i)^T + c_i}$$

mit $X = [0, 10]^4$, $\epsilon = 10^{-6}$, A, c wie bei S5.

Globales Minimum $f^\star \in$ `[-10.536409819801, -10.536409213758]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[4.000746531592, 4.000746531593]
[4.000592934138, 4.000592934139]
[3.999663398040, 3.999663398041]
[3.999509800586, 3.999509800587]
```

SHCB ($n = 2$, Six-Hump-Camel-Back, vgl. [76])



$$f(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$$

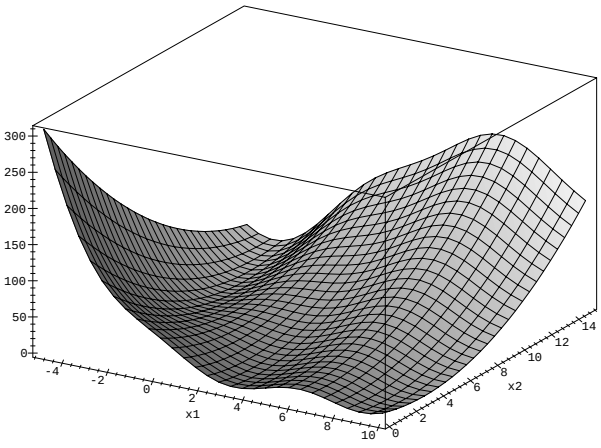
mit $X = [-2, 2]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[-1.031628453501, -1.031628453489]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[-0.089842013101, -0.089842013100]
[ 0.712656403020,  0.712656403021]

[ 0.089842013100,  0.089842013101]
[-0.712656403021, -0.712656403020]
```

BR ($n = 2$, Branin, vgl. [76])



$$f(x) = \left(\frac{5}{\pi}x_1 - \frac{5.1}{4\pi^2}x_1^2 + x_2 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos x_1 + 10$$

mit $X = [-5, 10] \times [0, 15]$, $\epsilon = 10^{-6}$.

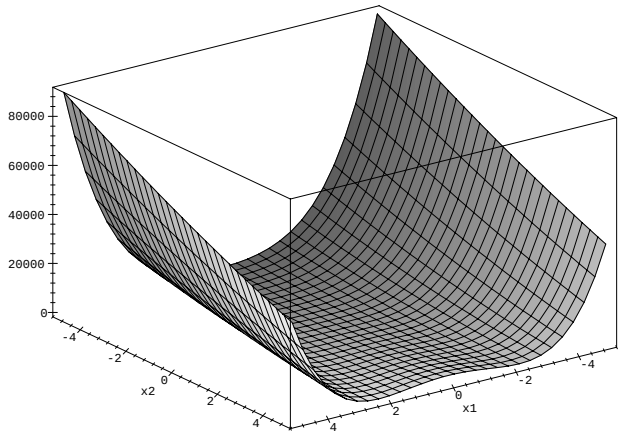
Globales Minimum $f^\star \in$ `[0.397887357729, 0.397887476842]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[ 9.424777876996,  9.424778044874]
[ 2.474999964247,  2.475000035308]

[-3.141592653715, -3.141592653465]
[12.274999999868, 12.275000000132]

[ 3.141592349102,  3.141592956078]
[ 2.274999862723,  2.275000134528]
```

RO ($n = 2$, Rosenbrock, vgl. [24])



$$f(x) = 100(x_2 - x_1^2)^2 + (x_1 - 1)^2$$

mit $X = [-5, 5]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 6.528273647922E-009]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999980650327, 1.000046886831]
[0.999924027806, 1.000079829876]
```

L8 ($n = 3$, Levy 8, vgl. [24])

$$f(x) = \sum_{i=1}^{2}(y_i - 1)^2(1 + 10\sin^2(\pi y_{i+1})) + \sin^2(\pi y_1) + (y_3 - 1)^2$$

mit $y_i = 1 + (x_i - 1)/4$, $i = 1, \ldots, 3$, $X = [-10, 10]^3$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 5.563910617528E-008]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999999999964, 1.000000000047]
[0.999999781755, 1.000000103816]
[0.999999476647, 1.000000375506]
```

L9 ($n = 4$, Levy 9, vgl. [24])

$$f(x) = \sum_{i=1}^{3}(y_i - 1)^2(1 + 10\sin^2(\pi y_{i+1})) + \sin^2(\pi y_1) + (y_4 - 1)^2$$

mit $y_i = 1 + (x_i - 1)/4$, $i = 1, \ldots, 4$, $X = [-10, 10]^4$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 1.356062808590E-009]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999999999999, 1.000000000001]
[0.999999996044, 1.000000005221]
[0.999999987800, 1.000000014771]
[0.999999996926, 1.000000003677]
```

H3 ($n = 3$, Hartman 3, vgl. [76])

$$f(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{3} A_{ij}(x_j - P_{ij})^2\right)$$

mit $X = [0, 1]^3$, $\epsilon = 10^{-6}$. Es ist

$$A = \begin{pmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix},\ c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix} \text{ und } P = \begin{pmatrix} 0.3689 & 0.1170 & 0.2673 \\ 0.4699 & 0.4387 & 0.7470 \\ 0.1091 & 0.8732 & 0.5547 \\ 0.03815 & 0.5743 & 0.8828 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[-3.862782396685, -3.862782138144]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.114614338589, 0.114614338590]
[0.555648849971, 0.555648849972]
[0.852546953520, 0.852546953521]
```

G5 ($n = 5$, Griewank 5, vgl. [76])

$$f(x) = \sum_{i=1}^{5} \frac{x_i^2}{400} - \prod_{i=1}^{5} \cos(\frac{x_i}{\sqrt{i}}) + 1$$

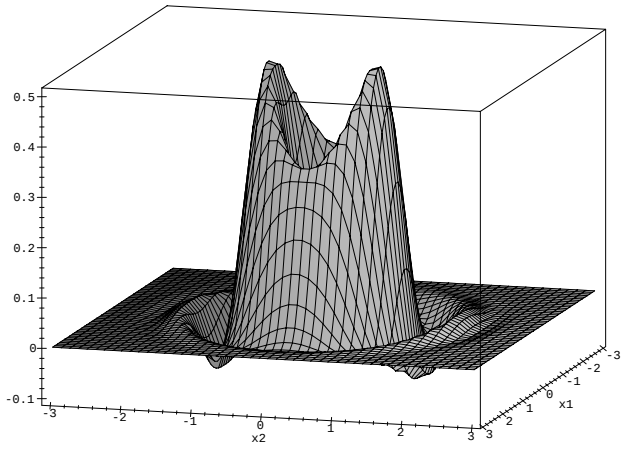
mit $X = [-500, 600]^5$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 2.296284273840E-010]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-2.797956627876E-014, 5.345760370691E-014]
[-3.241088631203E-013, 2.473466123897E-013]
[-2.745348605838E-013, 3.969102946159E-013]
[-1.627064571942E-013, 2.900031323211E-013]
[-1.027615695988E-013, 2.301239914230E-013]
```

A.2 Mittelschwere Testprobleme

R4 ($n = 2$, Ratz 4, vgl. [69])



$$f(x) = sin(x_1^2 + 2x_2^2)\exp(-x_1^2 - x_2^2)$$

mit $X = [-3,3]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[-0.106891375165, -0.106891341407]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[-2.340709061516E-012,  2.188142418052E-012]
[-1.457522104702,      -1.457522104700     ]

[-6.450345731293E-018,  6.833148271772E-018]
[ 1.457522104700,       1.457522104701     ]
```

L12 ($n = 10$, Levy 12, vgl. [24])

$$f(x) = \sum_{i=1}^{9}(y_i - 1)^2(1 + 10\sin^2(\pi y_{i+1})) + \sin^2(\pi y_1) + (y_{10} - 1)^2$$

mit $y_i = 1 + (x_i - 1)/4$, $i = 1, \ldots, 10$, $X = [-10, 10]^{10}$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 5.799987481629E-008]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999999999999, 1.000000000001]
[0.999999984160, 1.000000016770]
[0.999998060937, 1.000001163407]
[0.999993178989, 1.000005512002]
[0.999997041454, 1.000003378170]
[0.999999903275, 1.000000133168]
[0.999999307831, 1.000000980288]
[0.999979243157, 1.000021158259]
[0.999999298681, 1.000000989185]
[0.999999971582, 1.000000032316]
```

L18 ($n = 7$, Levy 18, vgl. [24])

$$\begin{aligned} f(x) \quad = \quad & \sum_{i=1}^{6}(x_i - 1)^2(1 + \sin^2(3\pi x_{i+1})) \\ & +(x_7 - 1)^2(1 + \sin^2(2\pi x_7)) + \sin^2(3\pi x_1) \end{aligned}$$

mit $X = [-5, 5]^7$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 3.332165290331E-008]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999999999999, 1.000000000001]
[0.999999999497, 1.000000000502]
[0.999999974186, 1.000000025925]
[0.999999994400, 1.000000020113]
[0.999999951714, 1.000000063962]
[0.999999945070, 1.000000071267]
[0.999999999186, 1.000000000814]
```

G7 ($n = 7$, Griewank 7, vgl. [76])

$$f(x) = \sum_{i=1}^{7} \frac{x_i^2}{4000} - \prod_{i=1}^{7} \cos(\frac{x_i}{\sqrt{i}}) + 1$$

mit $X = [-500, 600]^7$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 4.870426284498E-010]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-9.608257081401E-014, 1.840908551155E-013]
[-1.384435792461E-012, 1.081147680199E-012]
[-1.383567068694E-012, 1.849689824728E-012]
[-9.632417171562E-013, 1.398014597967E-012]
[-6.695596767998E-013, 1.109301418898E-012]
[-4.836738817891E-013, 9.168256533469E-013]
[-3.612046676572E-013, 7.856639702698E-013]
```

G10 ($n = 10$, Griewank 10, vgl. [76])

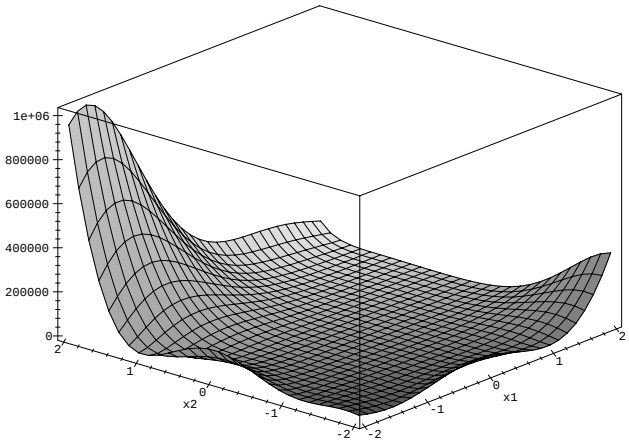
$$f(x) = \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} \cos(\frac{x_i}{\sqrt{i}}) + 1$$

mit $X = [-100.5, 120]^{10}$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 5.970779426435E-013]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-2.403677275192E-016, 1.917052611830E-016]
[-5.250900176840E-017, 7.674229675199E-017]
[-2.970364551520E-017, 6.414644037028E-017]
[-4.623398052481E-016, 6.181354222696E-016]
[-3.008982901168E-016, 2.572048366865E-016]
[-1.850277828224E-016, 1.436839987878E-016]
[-1.019696378388E-016, 6.465095436949E-017]
[-8.013270494205E-017, 4.523282993281E-017]
[-6.609673373134E-017, 3.369687646591E-017]
[-5.787241439041E-017, 2.776049899464E-017]
```

GP ($n = 2$, Goldstein-Price, vgl. [76])



$$f(x) = (1+(x_1+x_2+1)^2(19-14x_1+3x_1^2-14x_2+6x_1x_2+3x_2^2))\cdot \\ (30+(2x_1-3x_2)^2(18-32x_1+12x_1^2+48x_2-36x_1x_2+27x_2^2))$$

mit $X = [-2,2]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[2.999994916166, 3.000000000000]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-2.233669574293E-014,  2.215629362647E-014]
[-1.000000000001,      -0.999999999999     ]
```

H6 ($n = 6$, Hartman 6, vgl. [76])

$$f(x) = -\sum_{i=1}^{4} c_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)$$

mit $X = [0,1]^6$, $\epsilon = 10^{-6}$. Es ist

$$A = \begin{pmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix} \quad \text{und}$$

$$P = \begin{pmatrix} 0.1312 & 0.1696 & 0.5569 & 0.0124 & 0.8283 & 0.5886 \\ 0.2329 & 0.4135 & 0.8307 & 0.3736 & 0.1004 & 0.9991 \\ 0.2348 & 0.1451 & 0.3522 & 0.2883 & 0.3047 & 0.6650 \\ 0.4047 & 0.8828 & 0.8732 & 0.5743 & 0.1091 & 0.0381 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[-3.322368011601, -3.322368011406]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.201689511006, 0.201689511007]
[0.150010691823, 0.150010691824]
[0.476873974221, 0.476873974222]
[0.275332430494, 0.275332430495]
[0.311651616600, 0.311651616601]
[0.657300534065, 0.657300534066]
```

S2.14 ($n = 4$, Schwefel 2.14 (Powell-Problem), vgl. [24])

$$f(x) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$$

mit $X = [-4, 5]^4$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 9.116034405939E-012]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-0.009767841797, 0.007810273438]
[-0.028112159114, 0.015362096247]
[-0.062500000000, 0.218750000000]
[-0.062500000000, 0.078125000000]
```

GEO1 ($n = 3$, Problem aus der Geodäsie, vgl. [5])

$$\begin{aligned} f(x) \quad = \quad & (\sqrt{x_2^2 + x_3^2 - 2c_1x_2x_3} - s_1)^2 + (\sqrt{x_3^2 + x_1^2 - 2c_2x_3x_1} - s_2)^2 + \\ & (\sqrt{x_1^2 + x_2^2 - 2c_3x_1x_2} - s_3)^2 \end{aligned}$$

mit $X = [10^{-13}, 3600] \times [10^{-13}, 3520]^2$, $\epsilon = 10^{-6}$. Es ist

$$c = \begin{pmatrix} 0.846735205 \\ 0.928981803 \\ 0.912299033 \end{pmatrix} \text{ und } s = \begin{pmatrix} 1871.1 \\ 1592.4 \\ 1471.9 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[0.000000000000, 0.000000490132]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[3575.366076116996, 3575.366080169376]
[3412.155702686035, 3412.155703252487]
[2435.715600081947, 2435.715636644915]

[2292.480355828694, 2292.480423031207]
[3225.047002750754, 3225.047005874264]
[3477.180098396315, 3477.180179431853]
```

GEO2 ($n = 3$, Problem aus der Geodäsie, vgl. [5])

f wie in GEO1 mit $X = [10^{-13}, 8.68] \times [10^{-13}, 9.24 \times [10^{-13}, 8.68]$, $\epsilon = 10^{-6}$.
Es ist

$$c = \begin{pmatrix} 0.740824038 \\ 0.817119474 \\ 0.737253644 \end{pmatrix} \text{ und } s = \begin{pmatrix} 6.2 \\ 5.0 \\ 6.3 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[0.000000000000, 0.000000673087]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[4.867582355232, 4.882500000001]
[8.963210764580, 8.966404457464]
[8.114632499050, 8.122808974343]

[8.280704676538, 8.287998398373]
[8.997213390334, 9.001431990829]
[5.286510262869, 5.297462488985]

[8.305905112261, 8.316128909857]
[3.256440670436, 3.285959955199]
[8.214400661052, 8.227087381797]

[8.369591916933, 8.370097642676]
[8.947839892690, 8.948115107352]
[8.150036454104, 8.151149954646]
```

GEO3 ($n = 3$, Problem aus der Geodäsie, vgl. [5])

f wie in GEO1 mit $X = [10^{-13}, 8.0]^3$, $\epsilon = 10^{-6}$. Es ist

$$c = \begin{pmatrix} 0.766044443 \\ 0.766044443 \\ 0.766044443 \end{pmatrix} \text{ und } s = \begin{pmatrix} 5.0 \\ 5.0 \\ 5.0 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[0.000000000000, 1.949236291669E-008]`.
Lokal eindeutige Minimalstellen enthalten in:

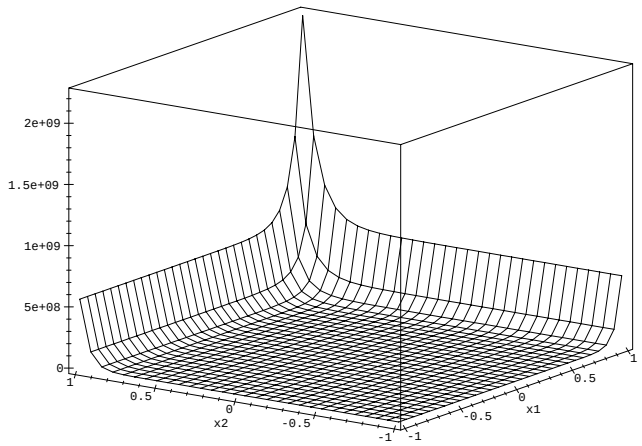
```
[7.307990647238, 7.311000913373]
[7.308222135889, 7.310753097465]
[7.308155051802, 7.310787539910]

[7.307333972823, 7.311712952602]
[3.886561903845, 3.890677066756]
[7.308755383472, 7.310150815954]

[3.875000000000, 3.917548731555]
[7.298833728467, 7.317465643492]
[7.297903945596, 7.319151420849]

[7.307255917141, 7.311650282286]
[7.307351000617, 7.311695924808]
[3.886647392189, 3.890608662605]
```

JS ($n = 2$, Jennrich-Sampson Problem, vgl. [58])



$$f(x) = \sum_{i=1}^{10} \left(2 + 2i - \left(e^{ix_1} + e^{ix_2}\right)\right)^2$$

mit $X = [-1, 1]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[124.362095836981, 124.362182355616]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.257825213638, 0.257825213703]
[0.257825213666, 0.257825213674]
```

A.3 Schwere Testprobleme

S2.7 ($n = 3$, Schwefel 2.7 (Box-3D-Problem), vgl. [24])

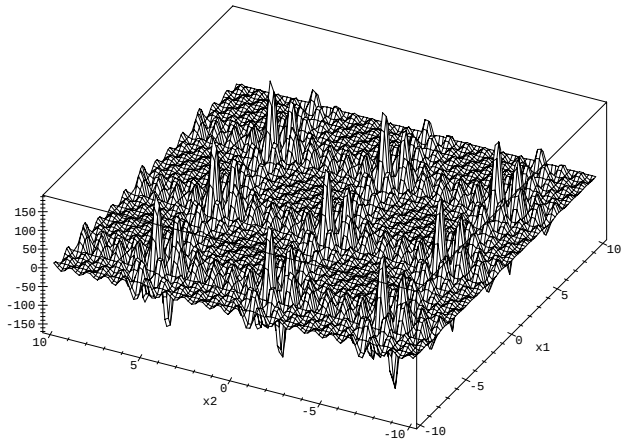
$$f(x) = \sum_{k=1}^{10} \left(\exp\left(\frac{-kx_1}{10}\right) - \exp\left(\frac{-kx_2}{10}\right) - \left(\exp\left(\frac{-k}{10}\right) - \exp\left(-k\right) \right) x_3 \right)^2$$

mit $X = [0,5] \times [8,11] \times [0.5,3]$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 0.000000770443]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.999736630376, 1.000306998085]
[9.998274141632, 10.002358153168]
[0.999881909145, 1.000138647665]
```

L3 ($n = 2$, Levy 3, vgl. [24],[49])



$$f(x) = \sum_{i=1}^{5} i \cos((i+1)x_1 + i) \sum_{j=1}^{5} j \cos((j+1)x_2 + j)$$

mit $X = [-10, 10]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[-186.731857704102, -186.730908830976]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[-7.083506407656, -7.083506407647]
[ 4.858056878859,  4.858056878861]

[-1.425128428328, -1.425128428311]
[ 5.482864206707,  5.482864206708]

[ 4.858056878851,  4.858056878869]
[ 5.482864206707,  5.482864206708]

[-0.800321100473, -0.800321100471]
[-7.708313735500, -7.708313735499]

[-7.083506407652, -7.083506407651]
[-1.425128428320, -1.425128428319]

[ 4.858056878859,  4.858056878860]
[-0.800321100472, -0.800321100471]

[-7.708313735500, -7.708313735499]
[-0.800321100472, -0.800321100471]

[ 5.482864206707,  5.482864206708]
[ 4.858056878859,  4.858056878860]

[ 5.482864206707,  5.482864206708]
[-1.425128428320, -1.425128428319]

[-0.800321100472, -0.800321100471]
[ 4.858056878859,  4.858056878860]

[-1.425128428320, -1.425128428319]
[-0.800321100472, -0.800321100471]

[-0.800321100472, -0.800321100471]
[-1.425128428320, -1.425128428319]

[-7.708313735500, -7.708313735499]
[-7.083506407652, -7.083506407651]

[-1.425128428320, -1.425128428319]
[-7.083506407652, -7.083506407651]

[ 4.858056878859,  4.858056878860]
[-7.083506407652, -7.083506407651]
```

```
[-7.708313735500, -7.708313735499]
[ 5.482864206707,  5.482864206708]

[ 5.482864206707,  5.482864206708]
[-7.708313735500, -7.708313735499]

[-7.083506407652, -7.083506407651]
[-7.708313735500, -7.708313735499]
```

Bemerkung: Die Funktion Levy 3 findet sich zuerst in [49]. Sie hat das oben angegebene globale Minimum und 18 globale Minimalstellen. Viele Autoren (so etwa [5], [69], [24], [33] uvm.) rechnen fälschlicherweise mit dem ersten Term „$i\cos((i-1)x_1+i)$“. Diese Variante hat nur 9 globale Minimalstellen, ein globales Minimum von $-176.54\ldots$ und ist etwas leichter zu berechnen.

R8 ($n = 9$, Ratz 8, vgl. [69])

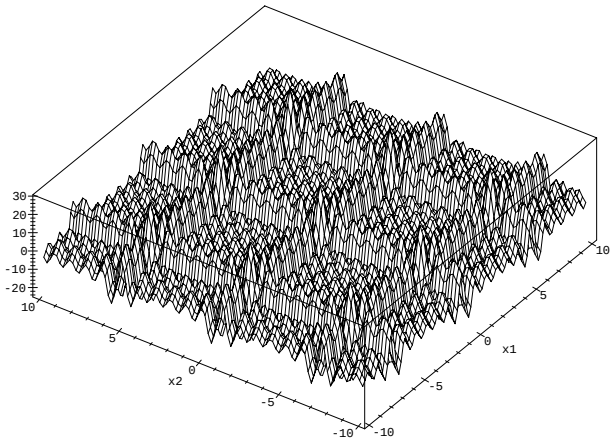
$$f(x) = \left(\sin^2\left(\pi\frac{x_1+3}{4}\right) + \sum_{i=1}^{8}\left(\frac{x_i-1}{4}\right)^2\left(1+10\sin^2\left(\pi\frac{x_{i+1}+3}{4}\right)\right)\right)^2$$

mit $X = [-10,10]^9$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[0.000000000000, 1.005638332378E-008]`.
Nicht verifizierte globale Minimalstelle enthalten in:

```
[  0.999999940395,  1.000000089407]
[  0.999755859375,  1.000366210938]
[  0.937500000000,  1.015625000000]
[  0.937500000000,  1.093750000000]
[  0.937500000000,  1.093750000000]
[  0.937500000000,  1.093750000000]
[  0.937500000000,  1.093750000000]
[  0.937500000000,  1.015625000000]
[-10.000000000000, 10.000000000000]
```

HM3 ($n = 2$, Henriksen und Madsen, vgl. [5])



$$f(x) = -\sum_{i=1}^{2}\sum_{j=1}^{5} j \sin((j+1)x_i + j)$$

mit $X = [-10, 10]^2$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[-24.062499161049, -24.062498883957]`.
Lokal eindeutige Minimalstellen enthalten in:

```
[-6.774576143439, -6.774576143438]
[-0.491390836260, -0.491390836259]

[-0.491390836260, -0.491390836259]
[ 5.791794470920,  5.791794470921]

[ 5.791794470920,  5.791794470921]
[-0.491390836260, -0.491390836259]

[-0.491390836260, -0.491390836259]
[-0.491390836260, -0.491390836259]

[-0.491390836260, -0.491390836259]
[-6.774576143439, -6.774576143438]

[-6.774576143439, -6.774576143438]
[-6.774576143439, -6.774576143438]

[ 5.791794470920,  5.791794470921]
[-6.774576143439, -6.774576143438]
```

```
[-6.774576143439, -6.774576143438]
[ 5.791794470920,  5.791794470921]

[ 5.791794470920,  5.791794470921]
[ 5.791794470920,  5.791794470921]
```

HM4 ($n = 3$, Henriksen und Madsen, vgl. [5])

$$f(x) = -\sum_{i=1}^{2}\sum_{j=1}^{5} j \sin((j+1)x_i + j)$$

mit $X = [-5,5]^3$, $\epsilon = 10^{-6}$.

Globales Minimum $f^\star \in$ `[-36.093748643631, -36.093748066516]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[-0.491390836260, -0.491390836259]
[-0.491390836260, -0.491390836259]
[-0.491390836260, -0.491390836259]
```

KOW ($n = 4$, Kowalik Problem, vgl. [24])

$$f(x) = \sum_{i=1}^{11}\left(a_i - x_1\frac{b_i^2 + b_i x_2}{b_i^2 + b_i x_3 + x_4}\right)^2$$

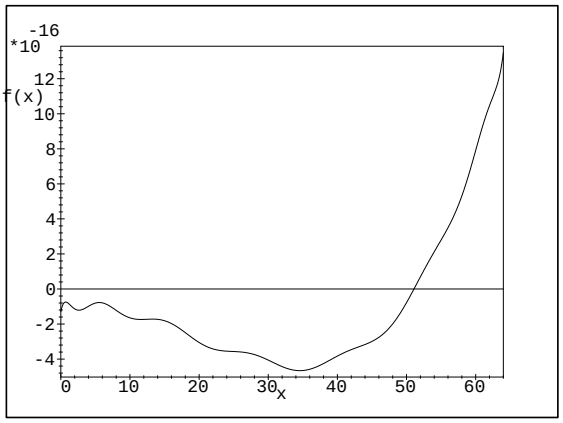
mit $X = [0, 0.42]^4$, $\epsilon = 10^{-6}$. Es ist

$$c = \begin{pmatrix} 0.1957 \\ 0.1947 \\ 0.1735 \\ 0.1600 \\ 0.0844 \\ 0.0627 \\ 0.0456 \\ 0.0342 \\ 0.0323 \\ 0.0235 \\ 0.0246 \end{pmatrix} \text{ und } s = \begin{pmatrix} 4 \\ 2 \\ 1 \\ 0.5 \\ 0.25 \\ 1/6 \\ 0.125 \\ 0.1 \\ 1/12 \\ 1/14 \\ 0.0625 \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[0.000307485965, 0.000307485988]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

```
[0.192833452973, 0.192833452992]
[0.190836237404, 0.190836240165]
[0.123117296249, 0.123117296307]
[0.135765989850, 0.135765990114]
```

WK ($n = 1$, Krämer Problem, vgl. [42])



$$f(x) = -\frac{p(x)}{q(x)} = -\frac{\sum_{i=0}^{29} p_i x^i}{\sum_{i=0}^{4} q_i x^i}$$

mit $X = [0, 64]$, $\epsilon = 10^{-12}$. Es ist

$$q = \begin{pmatrix} 0.58828674632868342934662993 76 \cdot 10^{11} \\ 0.36346749346560087410642370 87 \cdot 10^{9} \\ 0.99635360310006026750272778 24 \cdot 10^{6} \\ 0.14643417762555995397894351 42 \cdot 10^{4} \\ 1 \end{pmatrix}$$

und

$$p = \begin{pmatrix} 7.629394531250000 \cdot 10^{-6} \\ -1.150369644165040 \cdot 10^{-5} \\ 1.372280530631542 \cdot 10^{-5} \\ -6.579421551577981 \cdot 10^{-6} \\ 1.659054419178573 \cdot 10^{-6} \\ -2.521266665667100 \cdot 10^{-7} \\ 2.505680664436719 \cdot 10^{-8} \\ -1.713721655060476 \cdot 10^{-9} \\ 8.330923088212047 \cdot 10^{-11} \\ -2.935023067946825 \cdot 10^{-12} \\ 7.564193999729689 \cdot 10^{-14} \\ -1.426868803614099 \cdot 10^{-15} \\ 1.954186293985405 \cdot 10^{-17} \\ -1.910400220016202 \cdot 10^{-19} \\ 1.299884226135079 \cdot 10^{-21} \\ -5.995712492310049 \cdot 10^{-24} \\ 1.876066147446556 \cdot 10^{-26} \\ -4.291373306373139 \cdot 10^{-29} \\ 7.622481227988642 \cdot 10^{-32} \\ -1.096397325341554 \cdot 10^{-34} \\ 1.315291857866774 \cdot 10^{-37} \\ -1.344664974747858 \cdot 10^{-40} \\ 1.190815536452828 \cdot 10^{-43} \\ -9.253132527171894 \cdot 10^{-47} \\ 6.374582890249432 \cdot 10^{-50} \\ -3.926998956415952 \cdot 10^{-53} \\ 2.170989219023664 \cdot 10^{-56} \\ -1.142719267106732 \cdot 10^{-59} \\ 3.823185031874960 \cdot 10^{-63} \\ -3.691550884472599 \cdot 10^{-66} \end{pmatrix}.$$

Globales Minimum $f^\star \in$ `[-4.66084697735E-016,-4.66081642390E-016]`.
Eindeutige verifizierte globale Minimalstelle enthalten in:

`[34.566830044610, 34.566830494243]`

Anhang B

Verbesserung der Automatischen Differentiation aus der Toolbox

In [21] sind C++ Klassen für automatische Differentiation zur komfortablen Berechnung von Gradienten, Jacobimatrizen sowie Hessematrizen angegeben. Diese Klassen werden in dieser Arbeit in allen untersuchten Programmvarianten verwendet. Die Funktionen zur automatischen Differentiation sind dabei wie folgt bezeichnet:

`fEvalH(f,X,Fx)`: Liefert die Funktionswertseinschließung $F(X)$ über der Box X.

`fgEvalH(f,X,Fx,gFx)`: Liefert Einschließungen für den Funktionswert $F(X)$ sowie den Gradienten $\nabla F(X)$ über der Box X.

`fgEvalH(f,X,Fx,gFx,hFx)`: Liefert Einschließungen für den Funktionswert $F(X)$, den Gradienten $\nabla F(X)$ und die Hessematrix $\nabla^2 F(X)$ über der Box X.

Leider ist der dort angegebene C++ Programmcode nicht effizient implementiert. Dies hat insbesondere zur Folge, daß die Relation der Laufzeiten von Funktionsauswertungen (FA), Gradientenauswertungen (GA) und Hessematrixauswertungen (HA) untereinander nicht stimmig und daher auch nicht mit dem Auswertungsaufwand für die Vorwärtsmethode (Eeff_V) bzw.

Rückwärtsmethode (Eeff_R) in Übereinstimmung zu bringen ist. Außerdem sind Funktions- und Gradientenauswertung erheblich zu langsam.

Für die Laufzeiten $t_{\texttt{fEvalH}}, t_{\texttt{fgEvalH}}, t_{\texttt{fghEvalH}}$ sollte für $f : I\!R^n \to I\!R$ theoretisch gelten (es wird die Vorwärtsmethode verwendet):

$$t_{\texttt{fgEvalH}} \approx \underbrace{(1+n)}_{=:c_\nabla} \cdot t_{\texttt{fEvalH}} \quad \text{und} \quad t_{\texttt{fghEvalH}} \approx \underbrace{\left(1+n+\frac{n(n+1)}{2}\right)}_{=:c_{\nabla^2}} \cdot t_{\texttt{fEvalH}}.$$

Mit dem in [21] angegebenen ursprünglichen C++ Programm ergibt sich jedoch für je 1000 Auswertungen (Zeitangaben in Sekunden):

Prob.	$t_{\texttt{fEvalH}}$	$t_{\texttt{fgEvalH}}$	$t_{\texttt{fghEvalH}}$	$\frac{t_{\texttt{fgEvalH}}}{t_{\texttt{fEvalH}}}$	c_∇	$\frac{t_{\texttt{fghEvalH}}}{t_{\texttt{fEvalH}}}$	c_{∇^2}
S5	22.01	23.82	28.57	1.08	5	1.30	15
H6	64.14	68.62	75.52	1.07	7	1.18	28

Ähnliche Zeitverhältnisse ergeben sich für beliebige andere Funktionen. Gradienten- bzw. Hessematrixauswertungen sind nur unwesentlich „teurer“ als Funktionsauswertungen. Dies ist aus der Theorie nicht zu erwarten.

Ursache dieser ungünstigen Zeitverhältnisse ist überflüssiges Allokieren von Speicherplatz bei Funktions- und Gradientenauswertungen. Die Datei `hess_ari.cpp` muß wie folgt abgeändert werden, um das Laufzeitverhalten zu korrigieren. Die Angabe erfolgt im `diff`-Format: `<` kennzeichnet den ursprünglichen Programmtext, `>` den neuen Programmtext. Ferner sind die entsprechenden Zeilennummern angegeben.

```
155,156c155,156
<   Resize(g,nmax);
<   Resize(h,nmax);
---
>   if (HessOrder > 0) Resize(g,nmax);
>   if (HessOrder > 1) Resize(h,nmax);
163,164c163,164
<   Resize(u.g,dim);
<   Resize(u.h,dim);
---
>   if (HessOrder > 0) Resize(u.g,dim);
```

```
>   if (HessOrder > 1) Resize(u.h,dim);
187c187,191
<   if (nmax > 0) { f = u.f;  g = u.g;  h = u.h; }
---
>   if (nmax > 0) {
>     f = u.f;
>     if (HessOrder > 0) g = u.g;
>     if (HessOrder > 1) h = u.h;
>   }
199c203,205
<   f = x;  g = 0.0;  h = 0.0;
---
>   f = x;
>   if (HessOrder > 0) g = 0.0;
>   if (HessOrder > 1) h = 0.0;
272,274c278,281
<     for (int k = 1; k <= ubd; k++)
<       hht[i].g[k] = (i == k) ? 1.0 : 0.0;
<     hht[i].h = 0.0;
---
>     if (HessOrder > 0)
>       for (int k = 1; k <= ubd; k++)
>         hht[i].g[k] = (i == k) ? 1.0 : 0.0;
>     if (HessOrder > 1) hht[i].h = 0.0;
```

Nimmt man die oben beschriebenen Veränderungen an `hess_ari.cpp` vor, so ergeben sich die folgenden Zeitverhältnisse:

Prob.	$t_{\texttt{fEvalH}}$	$t_{\texttt{fgEvalH}}$	$t_{\texttt{fghEvalH}}$	$\frac{t_{\texttt{fgEvalH}}}{t_{\texttt{fEvalH}}}$	c_∇	$\frac{t_{\texttt{fghEvalH}}}{t_{\texttt{fEvalH}}}$	c_{∇^2}
S5	1.49	5.19	28.57	3.48	5	19.17	15
H6	12.84	18.86	75.52	1.49	7	5.88	28

Funktionsauswertungen sind mit der neuen Programmversion erheblich schneller (Faktor 5–15) als mit der ursprünglichen, für Gradientenauswertungen ergibt sich immerhin noch ein Faktor 3–5. Außerdem werden durch diese Änderungen die theoretischen Zeitverhältnisse wesentlich besser wiedergegeben. Exakt werden die theoretischen Zeitverhältnisse in der Praxis nie angenommen, da in der Theorie der Aufwand für Speicherallokierung usw. unberücksichtigt bleibt.

Eine vergleichbare Unschönheit liegt auch in der in [20] angegebenen Pascal–XSC Implementierung vor.

Anhang C

Zusätzliche Funktionen zur Verwendung von C–XSC und MPI

Um C–XSC unter MPI verwenden zu können, war es notwendig, einige Kommunikationsroutinen zu MPI hinzuzufügen, die die zusätzlichen Datentypen von C–XSC verwenden. Für alle *real* und *interval* Datentypen (skalar, Vektor, Matrix) wurden synchrone und asynchrone Sende- und Empfangsroutinen implementiert. Die Bedeutung der einzelnen Parameter ist in Abschnitt 2.5.2 erläutert. Für die skalaren Datentypen konnten *inline*-Funktionen verwendet werden, z.B.:

```
inline friend int MPI_Send(const real& r, int dest, int tag,
                           MPI_Comm comm)
{
  return MPI_Send((void*)&(r.z), 1, MPI_DOUBLE, dest, tag, comm);
}
```

Beim Empfangen von Vektor- und Matrixdaten wird wie bei der Zuweisung in C–XSC ein implizites *Resize()* durchgeführt.

Nachfolgend geben wir eine vollständige Auflistung aller neu eingeführten Funktionen an:

Synchrones Senden

```
int MPI_Send(const real&, int, int, MPI_Comm);
int MPI_Send(const interval&, int, int, MPI_Comm);
int MPI_Send(const rvector&, int, int, MPI_Comm);
int MPI_Send(const ivector&, int, int, MPI_Comm);
int MPI_Send(const rmatrix&, int, int, MPI_Comm);
int MPI_Send(const imatrix&, int, int, MPI_Comm);
```

Asynchrones Senden

```
int MPI_Issend(const real&, int, int, MPI_Comm, MPI_Request*);
int MPI_Issend(const interval&, int, int, MPI_Comm, MPI_Request*);
int MPI_Issend(const rvector&, int, int, MPI_Comm, MPI_Request*);
int MPI_Issend(const ivector&, int, int, MPI_Comm, MPI_Request*);
int MPI_Issend(const rmatrix&, int, int, MPI_Comm, MPI_Request*);
int MPI_Issend(const imatrix&, int, int, MPI_Comm, MPI_Request*);
```

Synchrones Empfangen

```
int MPI_Recv(real&, int, int, MPI_Comm, MPI_Status*);
int MPI_Recv(interval&, int, int, MPI_Comm, MPI_Status*);
int MPI_Recv(rvector&, int, int, MPI_Comm, MPI_Status*);
int MPI_Recv(ivector&, int, int, MPI_Comm, MPI_Status*);
int MPI_Recv(rmatrix&, int, int, MPI_Comm, MPI_Status*);
int MPI_Recv(imatrix&, int, int, MPI_Comm, MPI_Status*);
```

Literaturverzeichnis

[1] Alefeld, G.; Herzberger, J.: *Einführung in die Intervallrechnung.* Bibliographisches Institut Reihe Informatik, Nr. 12), Mannheim / Wien / Zürich, 1974.

[2] Alefeld, G.; Herzberger, J.: *An Introduction to Interval Computations.* Academic Press, New York, 1983.

[3] Altmann, E.; Marsland, T. A.; Breitkreutz, T.: *Accounting for Parallel Tree Search Overheads.* Proceedings of the International Conference on Parallel Processing, S. 198–201, 1988.

[4] Benson, H. P.: *Concave Minimization: Theory, Applications and Algorithms.* In [29, S. 43–148], 1995.

[5] Berner, S.: *Ein paralleles Verfahren zur verifizierten globalen Optimierung.* Dissertation, Uni Wuppertal, Shaker Verlag, Aachen, 1995.

[6] Bohlender, G.; Davidenkoff, A.: *Accurate Vector and Matrix Arithmetic for Parallel Computers.* Proceedings of the Third Workshop on Parallel and Distributed Processing, Sofia, Bulgaria, April 16–19, 1991, edited by K. Boyanov, Elsevier Science Publishers, 1992.

[7] Bohlender, G.; Kersten, T.; Trier, R.: *Implicit Matrix Multiplication with Maximum Accuracy on Various Transputer Networks.* Computing **52** 1994.

[8] Bomze, I. M. et al. (eds.): *Developments in Global Optimization.* Kluwer Academic Publishers, Dordrecht / Boston / London, 1997.

[9] Csallner, A. E.; Csendes, T.; Markot, M. C.: *Multisection in Interval Methods for Global Optimization.* Manuskript, 1997.

[10] Csendes, T.; Ratz, D.: *Subdivision Direction Selection in Interval Methods for Global Optimization.* Siam J. Numer. Anal., Vol. 34, No. 3, S. 922–938, 1997.

[11] Davidenkoff, A.: *Arithmetische Ausstattung von Parallelrechnern für zuverlässiges numerisches Rechnen.* Dissertation, Universität Karlsruhe, 1992.

[12] Eriksson, J.: *Parallel Global Optimization Using Interval Analysis.* Dissertation, University of Umeå, Schweden, 1991.

[13] Exner, E.: *Intervall-Branch-and-Bound-Methoden zur globalen Optimierung.* Implementierung, Test und Evaluierung eines Verfahrens mit lokaler Minimierungs- und Expansionsstrategie. Diplomarbeit, Inst. f. Angew. Mathematik, Universität Karlsruhe, 1995.

[14] Fischer, H.-C.: *Schnelle automatische Differentiation, Einschließungsmethoden und Anwendungen.* Dissertation, Universität Karlsruhe, 1990.

[15] Flynn, M. J.: *Some computer organizations and their effectiveness.* IEEE Trans. Comput., **C-21**, 1972.

[16] Ge, R. P.; Qin, Y. F.: *A Class of Filled Functions for Finding Global Minimizers of a Function of Several Variables.* Journal of Optimization Theory and Applications, **54**, S. 241–252, 1987.

[17] Geist, A. et al: *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[18] Griewank, A.; Corliss, G. (Eds.): *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia, 1991.

[19] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In [78, S. 467–482], 1990.

[20] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *Numerical Toolbox for Verified Computing I: Basic Numerical Problems.* Springer-Verlag, Berlin / Heidelberg / New York, 1993.

[21] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *C++ Toolbox for Verified Computing: Basic Numerical Problems.* Springer-Verlag, Berlin / Heidelberg / New York, 1995.

[22] Hansen, E.: *Global Optimization Using Interval Analysis: The One-Dimensional Case.* J. Optim. Theory and Appl. **29**, S. 331–344, 1979.

[23] Hansen, E.: *Global Optimization Using Interval Analysis: The Multi-Dimensional Case.* Numerische Mathematik **34**, S. 247–270, 1980.

[24] Hansen, E.: *Global Optimization Using Interval Analysis.* Marcel Dekker Inc., New York / Basel / Hong Kong, 1992.

[25] Hansen, P.; Jaumard, B.; Xiong, J.: *Cord-slope form of Taylor's expansion in univariate global optimization.* J. Optimization Theory Appl. 80, No. 3, S. 441–464, 1994.

[26] Hennart, J.-P. (ed.): *Numerical analysis.* Proc. of the 3rd IIMAS Institute for Research in Applied Mathematics and Systems workshop, held at Cocoyoc, Mexico, Jan. 19-23, 1981. Lecture notes in mathematics **909**, Springer-Verlag, Berlin / Heidelberg / New York, 1981.

[27] Henriksen, T.; Madsen, K.: *Use of a depth-first strategy in parallel global optimization.* Tech. Report 92-10, Institute for Numerical Analysis, Technical University of Denmark, Lyngby, 1992.

[28] Herzberger, J. (ed.): *Topics in Validated Computations.* Elsevier, Amsterdam, 1994.

[29] Horst, R.; Pardalos, P. M.: *Handbook of Global Optimization.* Kluwer Academic Publishers, Dordrecht / Boston / London, 1995.

[30] Hwang, K.; Briggs, F.: *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

[31] Ichida, K.; Fujii, Y.: *An Interval Arithmetic Method for Global Optimization.* Computing **23**, S. 85–97, 1979.

[32] INMOS Ltd.: *ANSI C toolset reference manual.* Doc. 72 TDS 225 00, INMOS, 1990.

[33] Jansson, C.; Knüppel, O.: *A Global Minimization Method: The Multi-Dimensional Case.* Report 92.1 (Berichte des Forschungsschwerpunkts Informations- und Kommunikationstechnik), TU Hamburg-Harburg, 1992.

[34] Jansson, C.: *On Self-Validating Methods for Optimization Problems.* In [28], 1994.

[35] Karp, R. M.; Zhang, Y.: *A randomized Parallel Branch and Bound Algorithm.* Proceedings of the International Conference on Parallel Processing, S. 69–75, 1988.

[36] Kaucher, E.: *Über metrische und algebraische Eigenschaften einiger beim numerischen Rechnen auftretender Räume.* Dissertation, Universität Karlsruhe, 1973.

[37] Kearfott, R. B.: *Preconditioners for the Interval Gauss Seidel Method.* SIAM J. of Num. Anal. **27**, S. 804–822, 1990.

[38] Kearfott, R. B.: *Rigorous Global Search: Continuous Problems.* Kluwer Academic Publishers, Boston, 1996.

[39] Kindervater, G. A. P.; Lenstra, J. K.: *Parallel Computing in Combinatorial Optimization.* Annals of Operations Research, 14, S. 245–289, 1988.

[40] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL–XSC — Language Reference with Examples.* Springer-Verlag, Berlin / Heidelberg / New York, 1992.

[41] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: *C–XSC, A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, Berlin / Heidelberg / New York, 1993.

[42] Krämer, W.: pers. Mitteilung, 1997.

[43] Krawczyk, R.: *Intervallsteigungen für rationale Funktionen und zugeordnete zentrische Formen.* Freiburger Intervall-Berichte 83/2, Institut für Angewandte Mathematik, Universität Freiburg, 1983.

[44] Kulisch, U.: *Grundlagen des Numerischen Rechnens — Mathematische Begründung der Rechnerarithmetik.* Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim / Wien / Zürich, 1976.

[45] Kulisch, U.; Miranker, W. L.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[46] Kulisch, U.; Miranker, W. L.: *The Arithmetic of the Digital Computer: A New Approach.* IBM Research Center RC 10580, S. 1–62, 1984. SIAM Review, Vol. **28**, No. 1, S. 1–40, March 1986.

[47] Kulisch, U. (Ed.): *Wissenschaftliches Rechnen mit Ergebnisverifikation — Eine Einführung.* Ausgearbeitet von S. Geörg, R. Hammer und D. Ratz. Vol. 58. Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[48] Lawler, E. L.; Wood, D. E.: *Branch and Bound Methods: A Survey.* Operations Research **14**, S. 699–719, 1966.

[49] Levy, A. L.; Montalvo, A.; Gomez, S.; Calderon, A.: *Topics in Global Optimization.* In [26, S. 18–33], 1981.

[50] Levy, A. L.; Montalvo, A.: *The Tunneling Algorithm for the Global Minimization of Functions.* SIAM Journal Sci. Stat. Comp., **6**, S. 15–29, 1988.

[51] Lüling, R.; Monien, B.: *Two Strategies for Solving the Vertex Cover Problem on a Transputer Network.* 3rd Int. Workshop on Distributed Algorithms, Lecture Notes in Computer Science 392, S. 160–171, Springer-Verlag, 1989.

[52] Lüling, R.; Monien, B., Ramme, F.: *Load Balancing in Large Networks: A Comparative Study.* Proc. of 3rd IEEE Symposium on Parallel and Distributed Processing, Dallas, 1991.

[53] Lüling, R.; Monien, B.: *Load Balancing for Distributed Branch & Bound Algorithms.* Proc. of the 6th International Parallel Processing Symposium (IPPS '92), S. 543–549, 1992.

[54] Mayer, G.: *Grundbegriffe der Intervallrechnung.* In [47, S. 101–117], 1989.

[55] Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs.* 3rd ed., Springer-Verlag, Berlin / Heidelberg / New York, 1996.

[56] Mockus, J.: *Bayesian Approach to Global Optimization.* Kluwer Academic Publishers, Dordrecht / Boston / London, 1989.

[57] Moore, R. E.: *Interval Analysis.* Prentice Hall Inc., Englewood Cliffs, N. J., 1966.

[58] Moré, J. J.; Garbow, B. S.; Hillstrom, K. E.: *Testing Unconstrained Optimization Software.* ACM Transactions on Mathematical Software, **7**, No. 1, März 1981.

[59] MPI Forum: *MPI: A Message Passing Interface Standard.* Techn. Bericht, University of Tennessee, Knoxville, Tennessee, 1995.

[60] Neumaier, A.: *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, 1990.

[61] Ortega, J. M.: *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York, 1988.

[62] Pinter, J.: *Continuous global optimization software: A brief review.* Optima, Dec. 96, pp. 1–8, 1996.

[63] Quinn, M. J.: *Analysis and Implementation of Branch and Bound Algorithms on a Hypercube Multicomputer.* IEEE Transactions on Computers, Vol. 39, No. 3, 1990.

[64] Ratschek, H.; Rokne, J.: *Computer Methods for the Range of Functions.* Ellis Horwood Limited, Chichester, 1984.

[65] Ratschek, H.; Rokne, J.: *New Computer Methods for Global Optimization.* Ellis Horwood, Chichester, 1988.

[66] Ratschek, H.; Rokne, J.: *The transistor modeling problem again.* Microelectronics and Reliability, **32**, S. 1725–1740, 1992.

[67] Ratschek, H.; Rokne, J.: *Interval Methods.* In [29, S. 751–828], 1995.

[68] Ratz, D.: *Globale Optimierung mit automatischer Ergebnisverifikation.* Dissertation, Universität Karlsruhe, 1992.

[69] Ratz, D.; Csendes, T.: *On the Selection of Subdivision Directions in Interval Branch-and-Bound Methods for Global Optimization.* Journal of Global Optimization **7**, S. 183–207, Kluwer Academic Publishers, Dordrecht / Boston / London, 1995.

[70] Ratz, D.: *On Extended Interval Arithmetic and Inclusion Isotonicity.* Zur Publikation im „SIAM Journal on Numerical Analysis“ eingereicht, 1996.

[71] Ratz, D.: *New Results on Gap-Treating Techniques in Extended Interval Newton Gauss-Seidel Steps for Global Optimization.* In [8, S. 55–72], 1997.

[72] Ratz, D.: *A New Global Optimization Technique Using Slopes – The One-Dimensional Case.* Zur Publikation im „Journal of Global Optimization“ eingereicht, 1997.

[73] Ratz, D.: *Automatic Slope Computation and its Application in Nonsmooth Global Optimization.* Habilitationsschrift, Universität Karlsruhe, 1997.

[74] Skelboe, S.: *Computation of Rational Interval Functions.* BIT **14**, S. 87–95, 1974.

[75] Snir, M.; Otto, S. W.; Huss-Lederman, S.; Walker, D. W.; Dongarra, J.: *MPI The Complete Reference.* MIT Press, 1996.

[76] Törn, A.; Žilinskas, A.: *Global Optimization.* Lecture Notes in Computer Science 350, Springer-Verlag, Berlin / Heidelberg / New York, 1989.

[77] Troya, J. M.; Ortega, M.: *A study of parallel branch and bound algorithms with best bound first search.* Parallel Computing 11, S. 121–126, 1989.

[78] Ullrich, Ch. (Ed.): *Contributions to Computer Arithmetic and Self-Validating Numerical Methods.* (Proceedings of SCAN 89, held in Basel, Oct. 2–6, 1989, submitted papers). IMACS Annals on Computing and Applied Mathematics, Vol. 7, J. C. Baltzer AG, Basel, 1990.

[79] van Laarhoven, P. J. M; Aarts, E. H. L.: *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, Dordrecht / Boston / London, 1987.

[80] Vornberger, O.: *Implementing branch and bound in a ring of processors.* Proceedings of CONPAR 86, Lecture Notes in Computer Science 237, S. 157–164, Springer-Verlag, Berlin / Heidelberg / New York, 1986.

[81] Vornberger, O.: *Load Balancing in a Network of Transputers.* Distributed Algorithms, Lecture Notes in Computer Science 312, S. 116–126, Springer-Verlag, Berlin / Heidelberg / New York, 1987.

Lebenslauf

Andreas Josef Georg Wiethoff

geboren am	15. September 1965 in Münster (Westf.)
Eltern	Manfred Wiethoff und Gisela Wiethoff geb. Davidts
Schulbildung	1971–1975 St. Norbert Grundschule in Münster 1975–1984 Pascal-Gymnasium in Münster
Reifeprüfung	23. Mai 1984
Wehrdienst	Juli 1984 bis September 1985
Studium	der Mathematik (Nebenfach Informatik) an der Universität Münster (WS 85/86 bis März 1988; Vordiplom) und der Universität Karlsruhe bis November 1991
Diplomprüfung	im Studiengang Mathematik am 19.11.1991
Berufstätigkeit	Oktober 1988 bis Februar 1990 als wissenschaftliche Hilfskraft am Rechenzentrum der Universität Karlsruhe Juli 1989 bis September 1989 als Werkstudent bei der SIEMENS AG, München Oktober 1990 bis Dezember 1991 als wissenschaftliche Hilfskraft am Institut für Angewandte Mathematik der Universität Karlsruhe seit Januar 1992 als wissenschaftlicher Mitarbeiter am Institut für Angewandte Mathematik der Universität Karlsruhe
Eheschließung	am 27. Dezember 1996 mit Christine Wiethoff geb. Warneke