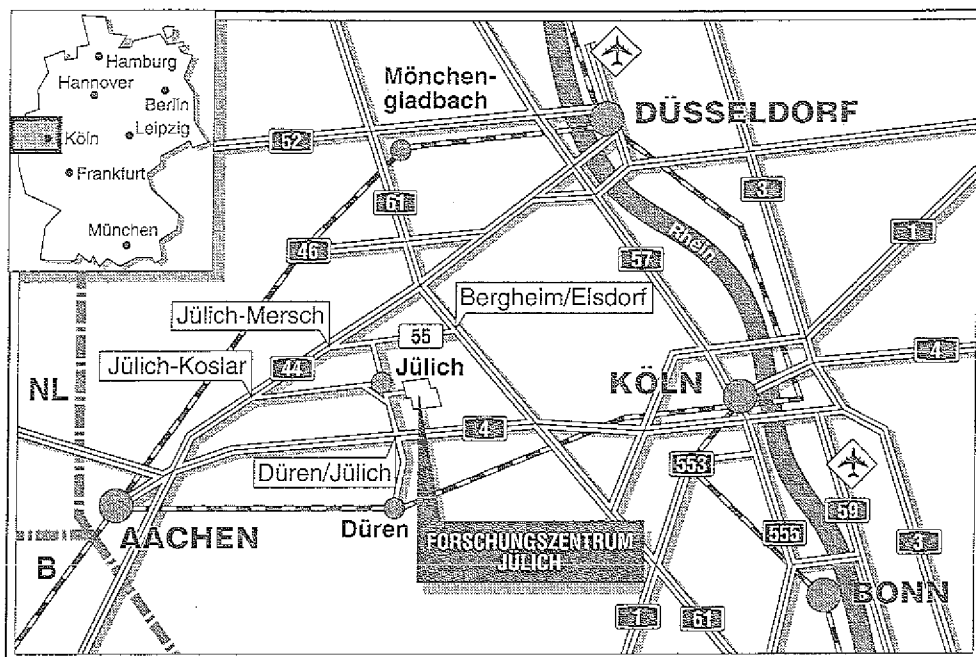


Zentralinstitut für Angewandte Mathematik

Parallele Algorithmen zur nichtlinearen Optimierung

**Mathematische und algorithmische Beschreibung
einiger Verfahren und ihre Implementation
auf CRAY-Rechnern**

Jochen Behrens



Berichte des Forschungszentrums Jülich ; 2705
 ISSN 0366-0885
 Zentralinstitut für Angewandte Mathematik Jül-2705

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
 Postfach 1913 · D-5170 Jülich · Bundesrepublik Deutschland
 Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Handwritten text, likely a list or notes, covering the upper half of the page. The text is written in a cursive script and is mostly illegible due to fading and the quality of the scan.

Handwritten text, possibly a signature or a short note, located in the middle section of the page.

Handwritten text, possibly a signature or a short note, located in the lower middle section of the page.

Handwritten text, possibly a signature or a short note, located in the bottom left corner of the page.

Handwritten text, possibly a signature or a short note, located in the bottom right corner of the page.

Parallele Algorithmen zur nichtlinearen Optimierung

**Mathematische und algorithmische Beschreibung
einiger Verfahren und ihre Implementation
auf CRAY-Rechnern**

Jochen Behrens

Inhalt

Zueignung	v
Notation	vii
Einleitung	1
1 Optimierungsprobleme	5
1.1 Das Grundproblem der Optimierung	5
1.2 Klassifizierung von Optimierungsproblemen	7
1.2.1 Unbeschränkte Optimierungsprobleme	9
1.2.2 Linear beschränkte Probleme	9
1.2.2.1 Lineares Programmieren	9
1.2.2.2 Linear-quadratische Probleme	10
1.2.3 Allgemeine Optimierungsprobleme	10
1.2.4 Konvexe Optimierungsprobleme	11
1.2.5 Separable Probleme	13
2 Klassische Optimierungsverfahren	15
2.1 Unbeschränkte Optimierung	15
2.1.1 Jacobi-Algorithmus	15
2.1.2 Gauß-Seidel-Algorithmus	16
2.1.3 Gradienten-Verfahren	17

2.1.4	Newton- und Quasi-Newton Verfahren	18
2.1.5	Konjugierte Gradienten	20
2.1.6	Abbruchkriterien	21
2.2	Optimierung unter Nebenbedingungen	23
2.2.1	Optimalitätsbedingungen	23
2.2.2	Quadratische Programmierung	24
2.2.3	Verfahren der zulässigen Richtungen	24
2.2.4	Projizierte Gradienten	26
3	Duale Verfahren	29
3.1	Dualität	29
3.2	Strafkosten-Verfahren	32
3.2.1	Penalty-Verfahren	32
3.2.2	Barriere-Verfahren	34
3.2.3	SUMT-Verfahren	35
3.3	Lagrange-Verfahren	36
3.3.1	Gradienten-Verfahren	36
3.3.2	Verfahren von Dantzig	37
3.3.3	Erweiterte Lagrange-Funktion	37
3.4	Quadratische Programmierung	40
3.5	Separable Optimierung	42
4	Parallelisierung	43
4.1	Parallelrechnerkonzepte	43
4.1.1	Pipelining	45
4.1.2	Vektorisierung	47
4.2	Konzepte zur Parallelisierung	49
4.2.1	Grundlegende Begriffe	49

4.2.2	Multitasking auf CRAY-Rechnern	50
4.2.3	Macrotasking	51
4.2.4	Microtasking	52
4.2.5	Autotasking	53
4.2.5.1	FPP: Datenabhängigkeitsanalyse	54
4.2.5.2	FMP: Transformation für Multitasking	56
4.3	Parallelisierung einzelner Verfahren	60
4.3.1	Hilfsmittel	61
4.3.2	Unbeschränkte Optimierung: BFGS	62
4.3.3	Optimierung unter Nebenbedingungen: Sequentielle Quadra- tische Optimierung	69
5	Schlußbemerkungen	77
	Literaturverzeichnis	79
A	Unterprogramme in Programmbibliotheken	85
A.1	Minimierung ohne Nebenbedingungen	86
A.2	Minimierung unter Nebenbedingungen	89
B	Testprobleme	93
B.1	Unbeschränkte Minimierung	93
B.2	Minimierung unter Nebenbedingungen	94

Ich widme seinem Studium
Den Tag mitsamt den Nächten.
Ja, Tag und Nacht studier ich dran,
Will keine Zeit verlieren;
Die Beine werden mir so dünn —
Das kommt vom vielen Studieren.

Heinrich Heine

Notation

Kleinbuchstaben	Vektoren bzw. Funktionen
x	Vektor $(x_1, \dots, x_n)^T$
x^T	transponierter Vektor zu x
x_i	i -te Komponente von x
$x^{(k)}$	k -ter Schritt einer Iterationsfolge zu x
$f(x)$	Zielfunktion, zu optimierende Funktion
$g_i(x)$	Nebenbedingungen
$g(x)$	Vektor der Nebenbedingungen $(g_1(x), \dots, g_m(x))^T$
∇f	Gradientenvektor zu f $(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})^T$
f_{x_k}	partielle Ableitung $\frac{\partial f}{\partial x_k}$
griechische Kleinbuchstaben	Skalare
Ausnahme: λ	Vektor der Lagrange-Multiplikatoren
$\mathcal{L}$	Lagrange-Funktion
Großbuchstaben	Matrizen
	auch: Mengen – aus Kontext ersichtlich
H	Hesse-Matrix $H = \nabla^2 f = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)_{1 \leq i, j \leq n}$
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}^n$	n -dimensionaler euklidischer Raum
$\mathbb{R}^{n \times m}$	Vektorraum der reellen $n \times m$ -Matrizen
$x \leq y$	$x_i \leq y_i, \quad i = 1, \dots, n \quad \text{für } x, y \in \mathbb{R}^n$
$x < y$	$x_i < y_i, \quad i = 1, \dots, n \quad \text{für } x, y \in \mathbb{R}^n$
$[a, b]$	abgeschlossenes Intervall $\{x \in \mathbb{R}^n a \leq x \leq b\}$
(a, b)	offenes Intervall $\{x \in \mathbb{R}^n a < x < b\}$

$n \gg k$	n ist wesentlich größer als k
$x \in X$	x ist Element von X
$x \notin X$	x ist nicht Element von X
$X \subset Y$	X ist Teilmenge von Y
$\ \cdot\ $	Euklidische Vektornorm $\ x\ = \sqrt{\sum_{i=1}^n x_i^2}$

Einleitung

Optimierungsaufgaben stellen sich in einer Vielzahl von Aufgabenfeldern. So beschäftigt man sich etwa im Operations Research mit der Optimierung technisch-ökonomischer Systeme. Hier seien zum Beispiel Transportprobleme, Schedulingverfahren oder die Optimierung von Netzplänen und Netzwerken genannt. Auch in den Ingenieur- und Naturwissenschaften sind häufig Optimierungsprobleme zu lösen, in der Steuer- und Regelungstechnik, bei Konstruktion und Kostenberechnung oder auch bei der Bestimmung der Elemente eines komplexen Gemischs im chemischen Gleichgewicht. In der numerischen Mathematik stellen sich solche Aufgaben bei Approximation und Regression oder auch bei der Suche nach einer geeigneten Triangulation in der Methode der finiten Elemente [BM70, Fle87, Pap91].

Grundsätzlich geht man in der Optimierung in mehreren Schritten vor:

1. Der erste Schritt besteht aus der Aufgabenstellung. Diese liegt in Form einer verbalen oder auch technischen Beschreibung vor.
2. Aus dieser Aufgabenstellung wird in einem Abstraktionsschritt ein mathematisches Modell entwickelt.
3. Dieses mathematische Problem wird mit analytischen oder numerischen Methoden gelöst.
4. Die Lösung des mathematischen Problems bedarf dann noch der Interpretation, sie ist nicht gleichbedeutend mit der Lösung der realen Problemstellung.

Im Operations Research spricht man an Stelle von Optimierung auch von „Programmierung“. *Linear Programming*, *Nonlinear Programming* oder auch *Mathematical Programming* sind in der englischsprachigen Literatur feststehende Begriffe, die, wörtlich übersetzt, auch in deutschsprachige Veröffentlichungen Eingang gefunden

haben. Unter „Mathematischer Programmierung“ wird üblicherweise das Gebiet des Operations Research verstanden, das sich mit der Optimierung unter Nebenbedingungen befaßt [Zim87]. Die Begriffsbildung stammt aus den späten 40er und den 50er Jahren; zu dieser Zeit war *programming* beziehungsweise „Programmierung“ noch nicht so eindeutig mit der Programmierung von Computern belegt. In der vorliegenden Arbeit soll aber im wesentlichen, wie in der Mathematik üblich, der Ausdruck „Optimierung“ verwendet werden.

In dieser Arbeit wird die numerische Lösung des mathematischen Problems betrachtet. Aspekte der anderen Schritte, zum Beispiel entscheidungstheoretischer Natur, werden nicht behandelt. Das besondere Augenmerk gilt der Optimierung unter Nebenbedingungen. Dennoch werden auch Verfahren zur unbeschränkten Optimierung betrachtet. Diese bilden die Grundlage auch für die Lösung beschränkter Problemstellungen; verschiedene Verfahren zu deren Behandlung sind Erweiterungen von Verfahren für unbeschränkte Probleme. Bei anderen Methoden werden beschränkte Probleme in unbeschränkte Ersatzmodelle überführt, die dann mit Hilfe der bekannten Verfahren gelöst werden.

Nichtlineare Aufgabenstellungen sind analytisch wie numerisch in der Regel wesentlich schwieriger zu behandeln als lineare. Daher wird bisher häufig, trotz der Nichtlinearität vieler Problemstellungen, mit linearen Modellen gearbeitet, für die es effiziente Lösungsverfahren gibt. Stünden effektivere Verfahren oder entsprechend größere Rechnerleistungen für die nichtlineare Optimierung zur Verfügung, könnten möglicherweise viele Problemstellungen mit genaueren Modellierungen gelöst werden.

In der Vergangenheit waren immer weitere Steigerungen der Leistungsfähigkeit digitaler Rechner durch Verbesserungen der Technologie möglich. Solche Entwicklungen stoßen jedoch zunehmend an physikalische Grenzen. So stellt die Lichtgeschwindigkeit eine obere Grenze für die Signalausbreitung dar, und auch die Integration einer immer größeren Zahl an Schaltelementen auf einem Chip läßt sich nicht ad infinitum fortführen. Das Erreichen hoher Performance hängt jedoch nicht nur von der Benutzung immer schnellerer Hardware ab, sondern ebenso von wesentlichen Verbesserungen in Computerarchitektur und Verarbeitungstechniken [HB84].

Den modernsten Rechnerarchitekturen ist vor allem das Konzept der Parallelverarbeitung gemein. Neben dem *Pipelining*-Prinzip, das in Form der Vektorarchitekturen

heute noch weitestgehend das Supercomputing beherrscht [Hoß91], wächst die Bedeutung der Mehrprozessorsysteme. Heute verbreitete Supercomputer, wie CRAY X-MP und CRAY Y-MP, bestehen aus mehreren sehr leistungsfähigen Vektorprozessoren, die auf einem gemeinsamen Hauptspeicher arbeiten. Dadurch können zum einen im Mehrbenutzerbetrieb die vorhandenen Ressourcen sehr effizient genutzt werden, zum anderen kann die Verarbeitung einzelner Anwendungen beschleunigt werden.

Ohne geeignete Algorithmen, die das Potential neuer Rechner zu nutzen vermögen, und ohne die zugehörige Software wird jedoch auch die aussichtsreichste Innovation auf der Hardware-Seite gegenstandslos. Dies gilt insbesondere für parallele Rechnersysteme [Bur89]. Man braucht Algorithmen, die einerseits gut die Vektorisierungsmöglichkeiten nutzen, andererseits das jeweilige Problem in möglichst viele parallel zu bearbeitende Teile zerlegen. Manchmal gelingt dies recht gut durch Modifikation bekannter sequentieller Algorithmen, manchmal sind Vektorisierung und Parallelisierung widerstreitende Gesichtspunkte, manchmal sind ganz neue algorithmische Ansätze erforderlich.

Ziel der vorliegenden Arbeit ist sowohl die mathematische Betrachtung als auch die Implementierung einiger Verfahren zur nichtlinearen Optimierung. Nach einem einführenden Kapitel, in dem verschiedene Klassen von Optimierungsproblemen vorgestellt werden, folgen zwei Kapitel, in denen zum einen klassische primale, zum anderen duale Lösungsverfahren dargestellt werden. Im anschließenden Kapitel zur Parallelisierung werden zunächst allgemeine Konzepte und die auf CRAY-Rechnern zur Verfügung stehenden Multitasking-Konzepte vorgestellt, bevor die Parallelisierung spezieller Optimierungsverfahren beschrieben wird. Nach einer kurzen Zusammenfassung und Einordnung der Ergebnisse in Kapitel 5 ist im Anhang eine Zusammenstellung in Bibliotheken implementierter Optimierungsroutinen sowie der verwendeten Testprobleme zu finden.

Kapitel 1

Optimierungsprobleme

1.1 Das Grundproblem der Optimierung

Das Suchen von Extremwerten von Funktionen, die gewissen Restriktionen unterworfen sind, wird Mathematisches Programmieren oder Optimierung unter Nebenbedingungen genannt. Eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ soll unter Nebenbedingungen optimiert werden, die durch Funktionen $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$, $i = 1, \dots, m$ in Gleichungs- oder Ungleichungsform gegeben sind. Außerdem kann es auch obere oder untere Schranken geben; ein häufiger Spezialfall dafür ist die Bedingung $x_i \geq 0$, $i = 1, \dots, n$.

Die verschiedenen Typen von Optimierungsproblemen werden in Kapitel 1.2 beschrieben. Ohne Einschränkung der Allgemeinheit kann man sich auf Minimierungsprobleme beschränken, da $\max f(x) = -\min(-f(x))$. Wegen $g_i(x) \leq 0 \Leftrightarrow -g_i(x) \geq 0$ bedeutet auch die Beschränkung auf Nebenbedingungen der Form $g_i(x) \geq 0$ keine Einschränkung der Allgemeinheit. Auch auf Gleichungsnebenbedingungen sowie obere und untere Schranken könnte verzichtet werden ($g_i(x) = 0 \Leftrightarrow g_i(x) \geq 0 \wedge -g_i(x) \geq 0$). Dies ist jedoch meist nicht sinnvoll, da diese häufig algorithmisch einfacher und von den Ungleichungsbedingungen getrennt behandelt werden können. Im weiteren wird daher das folgende Grundmodell der (endlichdimensionalen kontinuierlichen) Mathematischen Optimierung betrachtet:

MP:

$$\begin{array}{lll}
 \text{minimiere} & f(x) & x \in \mathbb{R}^n \\
 \text{so da\ss} & g_i(x) \geq 0 & i = 1, \dots, l \\
 & g_j(x) = 0 & j = l+1, \dots, m \\
 & \alpha_k \leq x_k \leq \beta_k & \alpha_k, \beta_k \in \mathbb{R}; \quad k = 1, \dots, n
 \end{array} \tag{1.1}$$

1.2 Klassifizierung von Optimierungsproblemen

Optimierungsprobleme lassen sich unter einer Vielzahl von Gesichtspunkten klassifizieren:

- Nach der Art der Grundmenge X , über der optimiert wird

- * X diskret
- * $X \subset \mathbb{R}^n$
- * X unendlich-dimensional

- Optimierung mit oder ohne Nebenbedingungen

- Nebenbedingungen mit Gleichheit oder Ungleichheit

- Nach der Art der Funktionen $f(x)$ und $g_i(x)$

- * $f(x), g_i(x)$ linear
- * $f(x)$ quadratisch, $g_i(x)$ linear
- * $f(x), g_i(x)$ beliebige Funktionen
- * $f(x), g_i(x)$ differenzierbar oder nicht differenzierbar
- * $f(x), g_i(x)$ konvex

- Nach der Struktur des Problems

- * separable Probleme
- * teilweise separable Probleme

- Deterministische oder stochastische Optimierung

- Statische oder dynamische Optimierung

- Nach der Art der Parallelisierungsmöglichkeiten

- * Parallelisierung nach der Struktur des Problems
- * Parallelisierung nach der Struktur der verwendeten Methode.

Auch eine Klassifizierung nach der Problemgröße ist natürlich möglich; in [LR88] findet sich etwa folgende Einteilung für Probleme ohne Nebenbedingungen:

- kleine Probleme: $n \leq 30$
- mittlere Probleme: $30 < n \leq 100$
- große Probleme: $n > 100$.

Eine solche Einteilung ist jedoch mit großer Vorsicht zu betrachten; ob ein Problem in angemessener Zeit lösbar ist, hängt sicherlich sehr viel stärker von der Problemstruktur als von seiner Größe ab. Auch die ständig steigende Rechnerleistung, durch die die handhabbare Problemgröße in den letzten Jahren möglicherweise um Größenordnungen gestiegen ist, stellt diese Art der Einordnung in Frage.

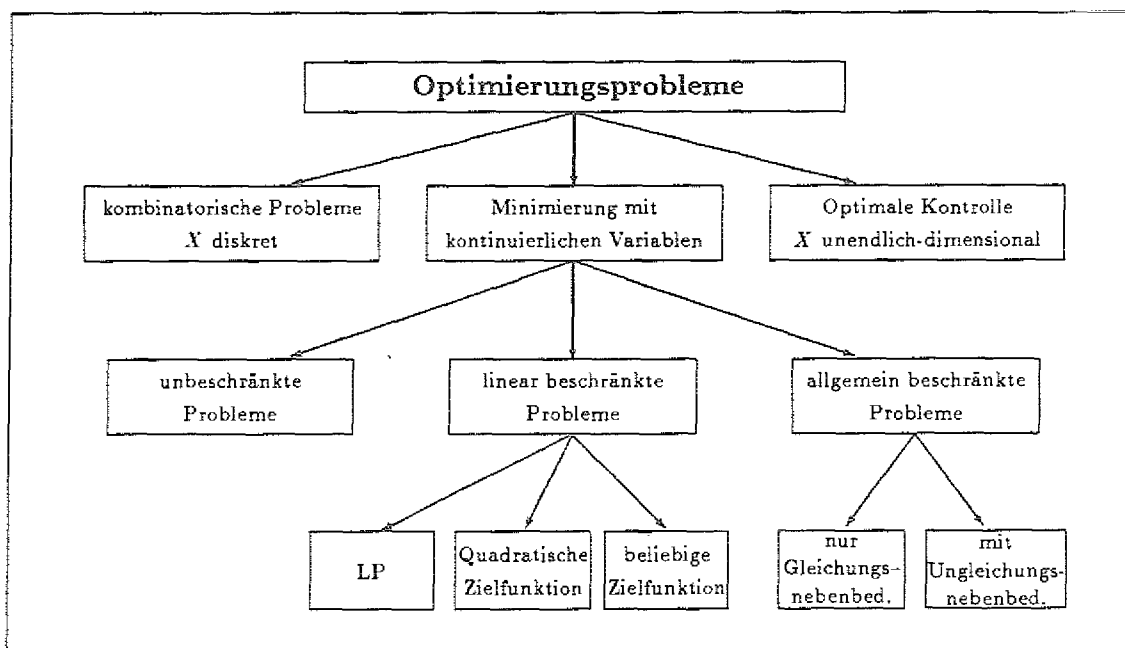


Abbildung 1.1: Klassifizierung nach Lemarechal

In Abbildung 1.1 ist die Klassifizierung von Optimierungsproblemen nach [Lem89] dargestellt. Hier wird zunächst nach der Menge X unterschieden. Auf den diskreten Fall und Kontrollprobleme wird in dieser Arbeit nicht weiter eingegangen. Die Probleme mit kontinuierlichen Variablen werden dann weiter unterschieden in unbeschränkte Probleme, Probleme mit linearen Restriktionen und allgemeine Probleme. Eine kurze Beschreibung dieser Probleme findet sich in den folgenden Abschnitten.

1.2.1 Unbeschränkte Optimierungsprobleme

Die unbeschränkten Optimierungsprobleme bilden die grundlegendste Klasse der nichtlinearen Optimierung. Eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ von n reellen Variablen soll dabei minimiert werden:

$$\boxed{\text{minimiere } f(x), \quad x \in \mathbb{R}^n} \quad (1.2)$$

Meist wird gefordert, daß f mindestens einmal stetig differenzierbar über $\mathbb{R}$ ist. In diesem Fall muß das (im allgemeinen nichtlineare) Gleichungssystem $\nabla f(x) = 0$ gelöst werden; dazu gibt es eine Reihe iterativer Verfahren.

1.2.2 Linear beschränkte Probleme

Die linear beschränkten Probleme sind dadurch charakterisiert, daß ihre Nebenbedingungen lineare Ungleichungen sind ($Ax \leq b$), der Lösungsraum also durch Hyperebenen begrenzt wird und daher eine konvexe Polyeder bildet.

1.2.2.1 Lineares Programmieren

Falls auch die Zielfunktion linear ist, spricht man von Linearem Programmieren:

$$\boxed{\begin{array}{ll} \text{LP:} & \\ \text{minimiere} & z = c^T x \\ \text{so daß} & Ax \leq b \\ & x \geq 0 \end{array}} \quad (1.3)$$

Zur Lösung dieses Problems wird der Simplex-Algorithmus verwendet, der sukzessive in den Ecken des Polyeders nach dem Maximum sucht [SW92]. Wichtige Varianten des Linearen Programmierens sind die ganzzahlige und die gemischt-ganzzahlige lineare Programmierung. Hier werden Branch-and-Bound- oder Schnittebenen-Verfahren (z.B. nach Gomory) verwendet [Zim87].

1.2.2.2 Linear-quadratische Probleme

Nicht wesentlich schwieriger zu lösen ist der Fall einer quadratischen Zielfunktion mit linearen Nebenbedingungen:

QP:		
minimiere	$f(x) = c^T x + \frac{1}{2} x^T Q x$	
so daß	$Ax \leq b$	(1.4)
	$x \geq 0$	
wobei	$c, x \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, Q \in \mathbb{R}^{n \times n}.$	

Die Matrix Q ist symmetrisch und positiv semidefinit. Probleme dieser Klasse haben zwei Eigenschaften, die dazu führen, daß sie relativ einfach zu lösen sind: erstens ist die Zielfunktion konvex, und zweitens ist der Gradient der Zielfunktion linear. Die zweite Eigenschaft führt zum Verfahren von Wolfe, bei dem das quadratische Problem mit Hilfe des Simplex-Algorithmus gelöst wird. Dazu werden die Kuhn-Tucker-Bedingungen zu QP aufgestellt, woraus sich ein lineares Problem mit zusätzlichen Basisbedingungen ergibt. Für dieses wird mit Hilfe des Simplex-Algorithmus eine zulässige Lösung gesucht, die bereits die optimale Lösung zu QP ist.

Probleme mit linearen Nebenbedingungen und einer beliebigen Zielfunktion werden mit ähnlichen Methoden (zulässige Richtungen) wie die Probleme ohne Nebenbedingungen gelöst. Sie können im allgemeinen nicht mehr exakt gelöst werden, sondern müssen mit iterativen Methoden behandelt werden. Bei einer großen Anzahl von Nebenbedingungen kann es sinnvoll sein, diese zu behandeln, als seien sie nichtlinear, also Methoden für allgemeine Optimierungsprobleme anzuwenden. Siehe dazu auch [Lern89].

1.2.3 Allgemeine Optimierungsprobleme

Bei den allgemeinen Problemen werden keine Einschränkungen in Bezug auf Zielfunktion und Nebenbedingungen mehr gemacht, sie können also im Prinzip beliebige nichtlineare Funktionen sein. Daraus ergibt sich, daß der Lösungsraum nicht mehr notwendig ein konvexes Polyeder ist, er muß nicht einmal mehr zusammenhängend sein. Daraus ergibt sich, daß Optimumsstellen (falls vorhanden) nicht mehr unbedingt in Ecken oder am Rand des Lösungsraumes liegen. Wichtig ist weiter, daß es

nicht mehr nur ein globales Optimum geben kann, sondern — wie im unbeschränkten Fall — auch lokale Optimumsstellen.

Viele Verfahren verlangen jedoch, daß alle Funktionen mindestens einmal (stetig) differenzierbar sind oder die Nebenbedingungen einen konvexen Lösungsraum beschreiben. Auch spezielle Eigenschaften der Funktionen, wie etwa Separabilität, können ausgenutzt werden.

Innerhalb dieser Klasse von Optimierungsproblemen läßt sich noch einmal danach unterscheiden, ob nur Gleichungsnebenbedingungen oder auch Ungleichungen vorhanden sind. Erstere sind von der Komplexität her mit den Problemen ohne Nebenbedingungen vergleichbar, während durch das Hinzufügen von Ungleichungen die (theoretische) Komplexität schlagartig erhöht wird [Lem89].

1.2.4 Konvexe Optimierungsprobleme

Neben den bisher vorgestellten Optimierungsproblemen gibt es weitere wichtige Klassen, die jeweils Teilklassen in der bisher angewandten Klassifizierungshierarchie bilden. Die nächst den linearen Problemen wohl am besten untersuchte Klasse von Optimierungsproblemen bilden die konvexen Probleme. Dabei wird eine konvexe oder konkave Zielfunktion über einer konvexen Menge optimiert.

Eine Menge $X \subset \mathbb{R}^n$ heißt *konvex*, wenn für je zwei Punkte $x, y \in X$ gilt:

$$\alpha x + (1 - \alpha)y \in X, \quad \forall \alpha \in [0, 1]. \quad (1.5)$$

Eine Teilmenge des $\mathbb{R}^n$ wird also genau dann konvex genannt, wenn mit je zwei beliebigen Punkten aus dieser Menge auch deren Verbindungsstrecke in der Menge liegt.

Sei nun $X \subset \mathbb{R}^n$ eine konvexe Menge. Eine Funktion $f : X \rightarrow \mathbb{R}$ heißt *konvex*, wenn

$$f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y), \quad \forall x, y \in X, \forall \alpha \in [0, 1]. \quad (1.6)$$

Äquivalent dazu ist für ein differenzierbares f die Bedingung

$$f(x) \geq f(y) + \nabla f(y)^T (x - y) \quad \forall x, y \in X \quad (1.7)$$

f heißt *streng konvex*, falls

$$f(\alpha x + (1 - \alpha)y) < \alpha f(x) + (1 - \alpha)f(y), \quad \forall x, y \in X, \forall \alpha \in (0, 1). \quad (1.8)$$

f heißt (*streng*) *konkav*, wenn $-f$ (*streng*) konvex ist.

Offensichtlich ist jede lineare Funktion sowohl konvex als auch konkav.

Konvexe Mengen und Funktionen haben nun eine Reihe Eigenschaften, die sich bei der Optimierung ausnutzen lassen. Zunächst einmal ist klar, daß ein Minimum x einer konvexen Funktion f ein globales Minimum ist. Ist f sogar streng konvex, so ist x eindeutig. Wenn also $\nabla f(x) = 0$ gilt, so ist x ein globales Minimum von f .

Weiterhin gilt, daß aus der Konvexität von f über einer konvexen Menge X Stetigkeit von f über X folgt (eventuell mit Ausnahme der Randpunkte).

Sei $f : \mathbb{R}^n \rightarrow \mathbb{R}$ zweimal stetig differenzierbar; f ist konvex dann und nur dann, wenn die Hesse-Matrix $\nabla^2 f(x)$ positiv semidefinit ist; f ist streng konvex, wenn $\nabla^2 f(x)$ positiv definit ist.

Zum Beispiel ist jede quadratische Funktion, die sich mit positiv semidefiniter symmetrischer Matrix Q in der Form $f(x) = c^T x + \frac{1}{2} x^T Q x$ schreiben läßt, konvex. (f ist streng konvex, wenn Q positiv definit ist.)

Das Optimierungsproblem

$$\begin{array}{ll} \min & f(x) \\ \text{s.d.} & g_i(x) \leq 0 \end{array}$$

ist genau dann ein Problem der Konvexen Programmierung, wenn sowohl f als auch alle g_i konvexe Funktionen sind. So ist zum Beispiel jedes LP ein konvexes Problem, gleiches gilt für die Probleme der Quadratischen Programmierung. Aber auch in allen anderen bisher genannten Problemklassen bilden die konvexen Probleme eine wichtige Teilklasse. Jedes lokale Minimum eines solchen Problems ist bereits ein globales Minimum. Es reicht also hier aus, genau ein Minimum zu suchen und nicht mehrere lokale Minima zu berechnen. Aufgrund dieser Eigenschaften ist es erheblich einfacher, für die Konvexe Programmierung effiziente Verfahren zu entwickeln, als dies im allgemeinen Fall möglich ist. Hier ist besonders zu beachten, daß die Dualitätstheorie im konvexen Fall besonders weitreichende Ergebnisse liefert, so daß sich duale Verfahren hier besonders erfolgversprechend anwenden lassen.

1.2.5 Separable Probleme

Eine weitere Klasse, die außerhalb der bisher verwendeten Hierarchie liegt, bilden die separablen Probleme. Beim Separablen Programmieren lassen sich Zielfunktion und Nebenbedingungen jeweils in Funktionen einzelner Variablen aufteilen. Dadurch ergibt sich das Modell der Separablen Programmierung:

SP:

$$\begin{aligned} &\text{minimiere} && f(x) = \sum_{j=1}^n f_j(x_j) \\ &\text{so da\ss} && \sum_{j=1}^n g_{ij}(x_j) \geq 0 \quad i = 1, \dots, m \end{aligned} \quad (1.9)$$

Auch hier gibt es, bedingt durch die Struktur des Problems, besonders effiziente Verfahren, und die Dualitätstheorie liefert — insbesondere, wenn die Probleme zusätzlich konvex sind — weitreichende Ergebnisse, die sich in effiziente Lösungsverfahren umsetzen lassen. Ein primales Verfahren verwendet wiederum den Simplex-Algorithmus des Linearen Programmierens. Dazu werden die Zielfunktionen und die Restriktionen stückweise linearisiert. Im nicht-konvexen Fall werden zusätzlich noch „Special Ordered Sets“ verwendet.

Neben den „reinen“ separablen Problemen kann auch aus der Struktur teilweise separabler Probleme Nutzen gezogen werden. Zielfunktion und Nebenbedingungen können etwa in Teilfunktionen zerfallen, die jeweils nur von einigen Variablen abhängen. Im Falle von je zwei Variablen ergibt sich dann zum Beispiel das Problem

$$\begin{aligned} &\text{minimiere} && f(x) = f_{12}(x_1, x_2) + f_{34}(x_3, x_4) + \dots \\ &\text{so da\ss} && g_i(x) = g_{i12}(x_1, x_2) + g_{i34}(x_3, x_4) + \dots \geq 0, \quad i = 1, \dots, m \end{aligned} \quad (1.10)$$

Allgemein lassen sich die teilweise separablen Probleme mit $n_0 := 0$ und $1 \leq n_1 < \dots < n_N = n$ dann wie folgt formulieren:

$$\begin{aligned} &\text{minimiere} && \sum_{j=1}^N f_j(x_{n_{j-1}+1}, \dots, x_{n_j}) \\ &\text{so da\ss} && \sum_{j=1}^N g_{ij}(x_{n_{j-1}+1}, \dots, x_{n_j}) \geq 0, \quad i = 1, \dots, m \end{aligned} \quad (1.11)$$

Wenn ein Optimierungsproblem eine solche Struktur besitzt, so ist die Hesse-Matrix dieser Funktionen (und die der zugehörigen Lagrange-Funktion!) eine Block-Diagonal-Matrix, es ist also zu erwarten, daß man dies rechentechnisch ausnutzen kann.

Andere Varianten sind Probleme, in denen sonst unabhängige Teilprobleme durch wenige verbindende Restriktionen miteinander gekoppelt werden und solche, in denen die im wesentlichen separablen Teilfunktionen durch verbindende Variablen gekoppelt werden. (Die Zielfunktion hat im letzteren Fall etwa das Aussehen $f(x_{[1]}, y) + \dots + f(x_{[N]}, y) + w(y)$, mit $x_{[i]} = x_{n_{i-1}+1}, \dots, x_{n_i}$, analoges gilt für das Aussehen der Restriktionen.)

Kapitel 2

Klassische Optimierungsverfahren

2.1 Unbeschränkte Optimierung

Für die Suche des Minimums einer stetig differenzierbaren Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ gibt es eine Reihe von Algorithmen. Für jeden Vektor x^* , der f (lokal) minimiert, gilt $\nabla f(x^*) = 0$. Das Problem läßt sich also als (nichtlineares) Gleichungssystem $\nabla f(x) = 0$ auffassen, das mit einem Iterationsverfahren gelöst werden kann. Falls f quadratisch ist, sich also in der Form $f(x) = \frac{1}{2}x^T A x - x^T b$ darstellen läßt, so gilt $\nabla f(x) = Ax - b$ und $\nabla^2 f(x) = A$, mit A symmetrisch und positiv definit. Die Bedingung $\nabla f(x) = 0$ wird also zu einem linearen Gleichungssystem, das entweder direkt oder iterativ gelöst werden kann. Die Methoden zur direkten Lösung, wie zum Beispiel die Gauß-Elimination, die LR-Zerlegung oder das Cholesky-Verfahren, sollen hier nicht weiter betrachtet werden. Die klassischen iterativen Verfahren lassen sich auch auf den nichtlinearen Fall übertragen und werden daher im folgenden kurz vorgestellt:

2.1.1 Jacobi-Algorithmus

Eine Möglichkeit, das lineare Gleichungssystem $Ax = b$ iterativ zu lösen, ist der Jacobi-Algorithmus. Wenn D die Matrix der Diagonalelemente von A bezeichnet, gilt mit der Substitution $A = D + B$ (Existenz von D^{-1} vorausgesetzt):

$$\begin{aligned}
Ax &= b \\
\iff (D+B)x &= b \\
\iff Dx &= b - Bx \\
\iff x &= D^{-1}(b - Bx)
\end{aligned}$$

Wenn nun die rechte Seite der Gleichung als Iterationsvorschrift für x angenommen wird, so ergibt sich mit der Rücksubstitution $B = A - D$:

$$\begin{aligned}
x^{(k+1)} &= D^{-1}(b - Bx^{(k)}) \\
\iff x^{(k+1)} &= -D^{-1}(A - D)x^{(k)} + D^{-1}b \\
\iff x^{(k+1)} &= x^{(k)} - D^{-1}(Ax^{(k)} - b)
\end{aligned}$$

Da im Falle einer quadratischen Funktion $Ax^{(k)} - b = \nabla f(x^{(k)})$ und $\nabla^2 f = A$ gilt, legt diese Überlegung die allgemeine Iterationsvorschrift

$$x^{(k+1)} = x^{(k)} - \gamma [D(x^{(k)})]^{-1} \nabla f(x^{(k)}) \quad (2.1)$$

nahe, wobei γ ein Relaxationsparameter ist, mit dem die Schrittweite beeinflusst werden kann und $D(x)$ die Diagonalmatrix mit i -tem Diagonaleintrag $\nabla_{ii}^2 f(x)$, der ungleich 0 sein muß (f muß also sogar zweimal differenzierbar sein).

2.1.2 Gauß-Seidel-Algorithmus

Beim Gauß-Seidel- oder auch *Einzelschritt*-Verfahren werden im Gegensatz zum Jacobi-Algorithmus, der in *Gesamtschritten* durchgeführt wird, zur Berechnung des Wertes von x_i in der $(k+1)$ -ten Iteration die bereits berechneten Werte $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$ benutzt. Die Iterationsschritte sind im Grunde die gleichen wie im Jacobi-Algorithmus, es werden nun aber die jeweils neuesten Werte in die Berechnung einbezogen, wodurch eine schnellere Konvergenz erreicht werden soll. Voraussetzung für das Verfahren ist auch hier, daß f zweimal differenzierbar ist, wobei die zweiten Ableitungen ungleich 0 sein müssen. Die folgende Gleichung beschreibt das Gauß-Seidel-Verfahren:

$$x_i^{(k+1)} = x_i^{(k)} - \gamma \frac{\nabla_i f(z(i, k))}{\nabla_{ii}^2 f(z(i, k))}, \quad i = 1, \dots, n, \quad (2.2)$$

wobei $z(i, k) = (x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}, x_i^{(k)}, \dots, x_n^{(k)})$.

2.1.3 Gradienten-Verfahren

Die Gradienten-Verfahren bilden eine ganze Familie von Methoden. In jedem Iterationsschritt soll eine möglichst gute, verbessernde Richtung gewählt werden. Da der Gradient $\nabla f(x^{(k)})$ der Vektor ist, der im Punkt $f(x_k)$ senkrecht auf der Tangentialebene steht und in die Richtung des steilsten Anstiegs gerichtet ist, liegt es nahe, die entgegengesetzte Richtung als Update zu wählen. In der grundsätzlichen Rekurrenzbeziehung ist dann noch der Relaxationsparameter γ zu wählen:

$$x^{(k+1)} = x^{(k)} - \gamma \nabla f(x^{(k)}). \quad (2.3)$$

Dabei kann entweder mit festen oder variablen Schrittweiten gearbeitet werden.

Auch zum Gradienten-Verfahren gibt es eine Gauß-Seidel Variante:

$$x_i^{(k+1)} = x_i^{(k)} - \gamma \nabla_i f(z(i, k)), \quad i = 1, \dots, n, \quad (2.4)$$

wobei $z(i, k)$ wie im Gauß-Seidel Algorithmus definiert sei.

Eine Variante des Gradienten-Verfahrens ist die **Methode des steilsten Abstiegs**. γ_k wird dabei so gewählt, daß die Funktion $g(\gamma) = f[x^{(k)} - \gamma \nabla f(x^{(k)})]$ minimiert wird. Dazu muß also in jedem Iterationsschritt eine eindimensionale Minimierung durchgeführt werden.

- i) $k = 0$: Wähle einen Startpunkt $x^{(0)}$
- ii) Schritt k : $d^{(k)} = -\nabla f(x^{(k)})$.
Finde γ_k mit $f(x^{(k)} + \gamma_k d^{(k)}) = \min_{\gamma \geq 0} f(x^{(k)} + \gamma d^{(k)})$.
Setze $x^{(k+1)} = x^{(k)} + \gamma_k d^{(k)}$.
- iii) falls Abbruchkriterium erfüllt: Ende, sonst gehe zu ii)

Im Falle einer stetig differenzierbaren Funktion f läßt sich zeigen, daß die Methode des steilsten Abstiegs von einem beliebigen Startpunkt $x^{(0)}$ aus gegen einen stationären Punkt von f konvergiert. Der Hauptnachteil des Verfahrens ist seine oft langsame Konvergenz. Da $\gamma^{(k)}$ die Funktion $g(\gamma)$ minimiert, gilt

$$\frac{dg}{d\gamma}(\gamma_k) = d^{(k)T} \nabla f(x^{(k)} + \gamma_k d^{(k)}) = d^{(k)T} \nabla f(x^{(k+1)}) = 0,$$

woraus folgt, daß

$$d^{(k)T} d^{(k+1)} = 0.$$

Aufeinanderfolgende Suchrichtungen sind also orthogonal zueinander, dadurch kann es passieren, daß der Algorithmus sich in „Zickzack“-Richtung an das Minimum annähert. Bei bestimmten Funktionstypen kann so eine sehr langsame Konvergenzgeschwindigkeit erzeugt werden [Min86].

2.1.4 Newton- und Quasi-Newton Verfahren

Ein sehr wichtiger Algorithmus der nichtlinearen Optimierung ist das Newton-Verfahren. Die Grundidee dazu kommt von der Taylor-Entwicklung um ein lokales Extremum. Unter der Voraussetzung, daß f zweimal stetig differenzierbar ist, gilt in einer Umgebung des Punktes $x^{(k)}$ die quadratische Näherung

$$f(x) \approx q(x) = f(x^{(k)}) + \nabla f(x^{(k)})^T (x - x^{(k)}) + \frac{1}{2} (x - x^{(k)})^T \nabla^2 f(x^{(k)}) (x - x^{(k)}). \quad (2.5)$$

Aus

$$\nabla q(x) = 0 \iff \nabla f(x^{(k)}) + \nabla^2 f(x^{(k)}) (x - x^{(k)}) = 0 \quad (2.6)$$

ergibt sich bei invertierbarer Hesse-Matrix die Iterationsvorschrift

$$x^{(k+1)} = x^{(k)} - \gamma \left[\nabla^2 f(x^{(k)}) \right]^{-1} \nabla f(x^{(k)}). \quad (2.7)$$

Das Verfahren ist jedoch nur dann konvergent, wenn der Startpunkt „hinreichend nahe“ am Optimalpunkt gewählt wird. Ein weiterer Nachteil ist die Forderung der Invertierbarkeit von $\nabla^2 f(x^{(k)})$. Die Inversion der Hesse-Matrix $\nabla^2 f(x^{(k)})$ erfordert bei Funktionen vieler Veränderlicher einen erheblichen Rechenaufwand. Um diesen zu verringern, benutzt man sogenannte **Quasi-Newton-Verfahren** (auch „Variable Metric“-Methoden), bei denen die Inverse der Hesse-Matrix nur approximiert wird. Dies ist auch dann möglich, wenn $\nabla^2 f$ nicht positiv definit ist.

Bei den Quasi-Newton-Verfahren wird die Matrix $\left[\nabla^2 f(x^{(k)}) \right]^{-1}$ in (2.7) durch eine positiv definite Matrix H_k ersetzt. Es ergibt sich eine Iterationsformel der Form

$$x^{(k+1)} = x^{(k)} - \gamma_k \cdot H_k \cdot \nabla f(x^{(k)}), \quad (2.8)$$

wobei γ_k so gewählt wird, daß es $g(\gamma) = f(x^{(k)}) + \gamma_k d^{(k)}$ in Richtung $d^{(k)} = -H_k \times \nabla f(x^{(k)})$ minimiert [Min86].

Das Aktualisieren der Matrix H_k ist auf verschiedene Arten möglich. Es sollte so geschehen, daß H_k gegen die Inverse der Hesse-Matrix konvergiert. Um dies zu erreichen, wird meist das Einhalten der „Quasi-Newton-Gleichung“

$$H_{k+1} (\nabla f(x^{(k+1)}) - \nabla f(x^{(k)})) = x^{(k+1)} - x^{(k)} \quad (2.9)$$

gefordert. Wohl am häufigsten wird die BFGS-Methode (Broyden, Fletcher, Goldfarb, Shanno) verwendet, die traditionell für Probleme mit $n \leq 100$ benutzt wird. Mit schnelleren Rechnern und größerem Speicher wird sie aber auch für größere Probleme nutzbar werden. Der BFGS-Algorithmus arbeitet wie folgt [BSS88]:

gegeben: $x^{(0)}, f(x^{(0)}), \nabla f(x^{(0)}), H_0 \in \mathbb{R}^{n \times n}$ positiv definit (z.B. $H_0 = I$)

Iteration k :

Berechne Suchrichtung:

$$d^{(k)} := -H_k \nabla f(x^{(k)})$$

Liniensuche:

wähle Schrittweite γ_k

$$x^{(k+1)} := x^{(k)} + \gamma_k d^{(k)}$$

werte $f(x^{(k+1)})$ und $\nabla f(x^{(k+1)})$ aus

Entscheide, ob Stop, wenn nicht:

Aktualisiere H_k

$$s^{(k)} := x^{(k+1)} - x^{(k)}, \quad y^{(k)} := \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$$

$$H_{k+1} := H_k + \left(1 + \frac{y^{(k)T} H_k y^{(k)}}{s^{(k)T} y^{(k)}} \right) \frac{s^{(k)T} s^{(k)}}{s^{(k)T} y^{(k)}} - \frac{s^{(k)} y^{(k)T} H_k + H_k y^{(k)} s^{(k)T}}{s^{(k)T} y^{(k)}}$$

Zu diesem Verfahren gibt es mehrere Varianten, was Speicherung und Updates angeht. Man kann dabei entweder mit Näherungen der Inversen der Hesse-Matrix oder der Hesse-Matrix selbst arbeiten. Wenn man nicht wie im oben angegebenen Algorithmus mit Näherungen der Inversen arbeitet, so ist zur Berechnung der Suchrichtung ein Gleichungssystem zu lösen. Außerdem verändert sich die Aktualisierung von H_k . Weiterhin kann man H_k entweder direkt abspeichern oder mit einer Zerlegung in untere Dreiecksmatrizen arbeiten. Nach Byrd, Schnabel und Schultz [BSS88] sind die Performance-Unterschiede zwischen diesen Varianten im allgemeinen vernachlässigbar gering; insbesondere in Hinblick auf Parallelisierung erscheinen jedoch die Methoden mit Update der Inversen der Hesse-Matrix besser geeignet, da sie ausschließlich aus Matrix-Vektor-Operationen und Rang-1-Updates bestehen, die auf modernen Parallelrechnern besonders effizient zu implementieren sind. Von diesen

ist die oben vorgestellte Vorgehensweise mit unfaktorisierter Matrix H_k vorzuziehen, weil dazu deutlich weniger Rechenoperationen durchgeführt werden müssen. (Gill et al. [GMSW85] merken jedoch an, daß eine Speicherung in Cholesky-Faktoren ein zusätzliches Problem beseitigt: durch Rundungsfehler kann es anderenfalls vorkommen, daß die Matrix H_k nicht positiv definit bleibt.)

Die Kosten für dieses Verfahren werden im wesentlichen durch zwei Faktoren bestimmt: die Auswertung von f und ∇f sowie die linear-algebraischen Operationen. Letztere sind jedoch relativ gering, so daß die Funktions- und Ableitungsauswertung meist bei weitem überwiegt [BSS88].

2.1.5 Konjugierte Gradienten

Für große Aufgaben (mit hunderten Variablen [LR88]) besonders geeignet ist die Methode der konjugierten Gradienten. Die Grundidee der Methode besteht darin, die Funktion in der Nähe ihres Minimums durch eine quadratische Funktion zu approximieren und die konjugierten Richtungen der entsprechenden Matrix zu benutzen, ohne Approximation und Matrix explizit zu verwenden [BS86]. Man betrachte dazu zunächst die quadratische Zielfunktion

$$q(x) = \frac{1}{2} x^T H x - b^T x, \quad (2.10)$$

wobei H die positiv definite, symmetrische Hesse-Matrix ist. Die Vektoren $s^1, s^2, \dots, s^k$ heißen (paarweise) *konjugiert* bezüglich H , falls

$$s^i{}^T H s^j = 0, \quad \forall i \neq j. \quad (2.11)$$

Mit $g^{(k)} = \nabla q(x^{(k)})$ wende man, ausgehend vom Startpunkt $x^{(0)}$ und mit $g^{(0)} = \nabla q(x^{(0)}) = Ax^{(0)} + b$, $d^{(0)} = -g^{(0)}$, das folgende Iterationsschema mit Richtung $d^{(k)}$ in Iteration k an:

$$x^{(k+1)} = x^{(k)} + \gamma_k d^{(k)}$$

$$\text{mit } \gamma_k = \frac{g^{(k)T} d^{(k)}}{d^{(k)T} A d^{(k)}}$$

$$d^{(k+1)} = -g^{(k+1)} + \beta_k d^{(k)}$$

$$\text{mit } \beta_k = \frac{g^{(k+1)T} A d^{(k)}}{d^{(k)T} A d^{(k)}}$$

Es läßt sich nun folgendes zeigen [SW92]:

- (i) Die Richtungen $d^{(0)}, \dots, d^{(k)}$ sind paarweise konjugiert.
- (ii) $x^{(k+1)}$ optimiert g über dem von $d^{(0)}, \dots, d^{(k)}$ aufgespannten Teilraum.
- (iii) Der Algorithmus terminiert in $m \leq n$ Schritten an einem Fixpunkt $x^{(m)}$

Dieser Algorithmus läßt sich leicht auf den Fall einer beliebigen nichtlinearen Funktion f verallgemeinern, etwa nach Fletcher-Reeves:

$x^{(0)}$ sei Startpunkt, setze $d^{(0)} = -\nabla f(x^{(0)})$.

Finde γ_k als Minimumsstelle von

$$g(\gamma) = f(x^{(k)} + \gamma d^{(k)})$$

$$x^{(k+1)} = x^{(k)} + \gamma_k d^{(k)}$$

$$d^{(k+1)} = -\nabla f(x^{(k+1)}) + \beta_k d^{(k)}$$

$$\text{mit } \beta_k = \frac{\|\nabla f(x^{(k+1)})\|^2}{\|\nabla f(x^{(k)})\|^2}.$$

Dabei wird die Hesse-Matrix nicht benötigt, die sonst einen erheblichen Rechen- (und Speicher-) Aufwand erfordern würde. In jeder Iteration wird eine *exakte* Liniensuche zur Bestimmung von γ_k ausgeführt. Es gibt jedoch auch Varianten, die diese vermeiden.

2.1.6 Abbruchkriterien

Allen iterativen Algorithmen gemeinsam ist es, daß im allgemeinen keine exakte Lösung zu einem gegebenen Problem gefunden wird, sondern eine solche nur approximativ angegeben werden kann. Es stellt sich also die Frage, nach wie vielen Schritten eine Iteration abgebrochen werden soll, so daß zum einen die bisher gefundene Lösung hinreichend genau ist, zum anderen eine endlose Iteration vermieden wird.

Die einfachste Lösung dieses Problems wäre es, eine Iterationszahl a priori festzulegen. Dies ist jedoch normalerweise nicht praktikabel. Es ist allerdings üblich, eine obere Grenze für die Anzahl der Iterationen anzugeben, um für den Fall gewappnet zu sein, daß die Iterationsfolge divergiert; nicht alle Verfahren können schließlich globale Konvergenz garantieren.

Zur Überprüfung der Konvergenz kann zum einen getestet werden, ob die Folge der $x^{(k)}$ gegen den gesuchten Optimalpunkt konvergiert, zum anderen aber auch, ob die Folge der $f(x^{(k)})$ gegen den minimalen Funktionswert strebt.

Gängige Kriterien sind zum Beispiel die folgenden [Min86]:

$$\text{Kriterium 1:} \quad \max_{i=1 \dots n} \left| \frac{\partial F}{\partial x_i} \right| < \epsilon \quad (\epsilon > 0 \text{ gegeben}).$$

$$\text{Kriterium 2:} \quad \|\nabla f\|^2 = \sum_{i=1}^n \left(\frac{\partial F}{\partial x_i} \right)^2 < \epsilon \quad (\epsilon > 0 \text{ gegeben}).$$

$$\text{Kriterium 3:} \quad |f(x^{(k+1)}) - f(x^{(k)})| < \eta \quad (\eta > 0 \text{ gegeben}).$$

Mit den ersten beiden Kriterien soll garantiert werden, daß die Folge der $x^{(k)}$ tatsächlich gegen eine Optimalstelle strebt, indem sichergestellt wird, daß der Gradient an der Stelle des Abbruchs der Iteration — im Rahmen der Rechengenauigkeit — verschwindet. Mit Kriterium 3 wird hingegen die Konvergenz der Folge der $f(x^{(k)})$ gegen einen — hoffentlich minimalen — Funktionswert abgeprüft. Bei jedem dieser Kriterien ist es angebracht, zu verlangen, daß es bei mehreren aufeinanderfolgenden Iterationen erfüllt ist.

Wichtig ist natürlich auch die Wahl der Genauigkeitsparameter ϵ bzw. η . Diese richtet sich zum einen nach den Ansprüchen an die zu erhaltene Lösung des Optimierungsproblems, zum anderen ist es jedoch auch notwendig, die Rechengenauigkeit der Maschine zu beachten, auf der das Problem gelöst werden soll.

2.2 Optimierung unter Nebenbedingungen

2.2.1 Optimalitätsbedingungen

Man betrachte zunächst das Problem

$$\begin{aligned} &\text{minimiere} && f(x) \\ &\text{so daß} && g_i(x) \leq 0 \quad i = 1, \dots, m \\ &&& x \in \mathbb{R}^n, \quad f, g_i \text{ stetig differenzierbar.} \end{aligned}$$

Die sogenannten *Kuhn-Tucker-Bedingungen*¹

$$\nabla f(x^*) + \sum_{i=1}^m \lambda_i \nabla g_i(x^*) = 0 \quad (2.12)$$

und

$$\lambda_i g_i(x^*) = 0, \quad \lambda_i \geq 0 \quad (2.13)$$

sind notwendig für die Optimalität des Punktes x^* . Bedingung (2.13) wird darin auch als „Complementary-Slackness-Bedingung“ bezeichnet. Zum Beweis wird die Lagrange-Funktion $\mathcal{L}(x, \lambda) = f(x) + \sum_i \lambda_i g_i(x)$ betrachtet. Ein Optimum von f unter den Nebenbedingungen g_i ist gleichbedeutend mit einem Sattelpunkt von $\mathcal{L}$, wenn $\mathcal{L}$ konvex in x und konkav in λ ist. (Siehe Kapitel 3. Dualität). Geometrisch ist (2.12), (2.13) so zu deuten, daß $\nabla f(x^*)$ sich aus den Gradienten der aktiven Nebenbedingungen negativ linear kombinieren lassen muß, die den Kegel der zulässigen Richtungen beschreiben. (Anderenfalls gäbe es eine Richtung r mit $r^T \nabla f < 0$, die auch noch im Kegel der zulässigen Richtungen läge, im Widerspruch zur Minimumseigenschaft.)

Falls f eine konvexe Funktion ist, und die Nebenbedingungen g_i eine konvexe Menge beschreiben, so sind die Kuhn-Tucker-Bedingungen genau dann hinreichend, wenn es in Bezug auf die nichtlinearen Ungleichungsnebenbedingungen einen inneren Punkt gibt (Slater-Bedingung, [Zim87]).

Wenn das Problem um Gleichungsnebenbedingungen $h_j(x) = 0, j = 1, \dots, l$ erweitert wird, so wird Gleichung (2.12) um die Summe $\sum_{j=1}^l \lambda_j \nabla h_j(x^*)$ erweitert, während die Complementary-Slackness-Bedingung unverändert bleibt.

¹In der neueren Literatur teilweise auch als *Karush-Kuhn-Tucker-Bedingungen* bezeichnet, weil W. Karush sie schon 1939, also 11 Jahre vor der Arbeit von Kuhn und Tucker, in seiner Diplomarbeit formulierte [Fra80]

2.2.2 Quadratische Programmierung

Eine quadratische Funktion $q(x) = c^T x + x^T Q x$ soll unter linearen Nebenbedingungen $Ax \leq b$ minimiert werden (QP, Kapitel 1.3.2.2). Neben dem Verfahren von Wolfe, das direkt mit den Kuhn-Tucker-Bedingungen arbeitet und in dem mit dem Simplex-Algorithmus ein LP gelöst wird [Zim87], gibt es auch für dieses Problem iterative Verfahren. Besonders für diese Problemklasse sind duale Algorithmen geeignet, die im nächsten Kapitel beschrieben werden.

Die Quadratische Programmierung wird insbesondere auch dadurch interessant, daß es Methoden zur Lösung allgemeiner NLPs gibt, die mit Sequenzen quadratischer Approximierungen arbeiten [Mur88].

Das Problem

$$\begin{array}{lll} \text{minimiere} & f(x) & x \in \mathbb{R}^n \\ \text{so daß} & g_i(x) = 0 & i = 1, \dots, l \\ & g_j(x) \geq 0 & j = l + 1, \dots, m \end{array}$$

läßt sich zum Beispiel dadurch lösen, daß in jeder Iteration die Suchrichtung $d^{(k)}$ durch das quadratische Teilproblem

$$\begin{array}{ll} \min & q(d) = f(x^{(k)}) + d^T \nabla f(x^{(k)}) + \frac{1}{2} d^T H_k d \\ \text{so daß} & g_i(x^{(k)}) + d^T \nabla g_i(x^{(k)}) = 0, \quad i = 1, \dots, l \\ & g_i(x^{(k)}) + d^T \nabla g_i(x^{(k)}) \geq 0, \quad i = l + 1, \dots, m \end{array}$$

bestimmt wird. Die Matrix H_k , die die Hesse-Matrix der Lagrange-Funktion des Originalproblems approximieren soll, kann dabei zum Beispiel mit der BFGS-Formel berechnet werden [GC88].

2.2.3 Verfahren der zulässigen Richtungen

Idee der Verfahren zulässiger Richtungen ist es, ausgehend von einem zulässigen Startpunkt, Abstiegsrichtungen zu finden, die den durch die Nebenbedingungen beschriebenen zulässigen Bereich nicht verlassen.

Ausgehend vom Punkt $x^{(k)}$ wird ein Vektor $d^{(k)}$ gesucht, für den gilt:

1. $x^{(k)} + \mu d^{(k)}$ ist zulässig für ein $\mu > 0$
2. $f(x^{(k)} + \mu d^{(k)}) < f(x^{(k)})$,

die Richtung $d^{(k)}$ sei also zulässig und verbessernd. Wenn man das Problem

$$\begin{array}{ll} \text{minimiere} & f(x) \\ \text{sa daß} & g_i(x) \leq 0, \quad i = 1, \dots, m \end{array} \quad (2.14)$$

betrachtet, so läßt sich folgendes zeigen[BS79]:

Sei $x^{(k)}$ zulässig, $I = \{i | g_i(x^{(k)}) = 0\}$. Falls $d \in \mathbb{R}^n$ die Ungleichungen

$$\nabla f(x^{(k)})^T d^{(k)} < 0 \quad (2.15)$$

und

$$\nabla g_i(x^{(k)})^T d^{(k)} \leq 0 \quad \forall i \in I, \quad (2.16)$$

erfüllt, dann ist $d^{(k)}$ eine verbessernde, zulässige Richtung.

Ein solcher Vektor läßt sich durch Lösen des folgenden LP finden:

$$\begin{array}{ll} \text{minimiere} & z \\ \text{so daß} & \nabla f(x^{(k)})^T d - z \leq 0 \\ & \nabla g_i(x^{(k)})^T d - z \leq 0, \quad \forall i \in I \\ & |d_j| \leq 1, \quad j = 1, \dots, n \end{array} \quad (2.17)$$

Durch die letzte Restriktion ($|d_j| \leq 1$) soll sichergestellt werden, daß die Lösung beschränkt ist. Zu diesem Zweck werden alternativ auch die Bedingungen $d^T d = 1$ und $\sum |d_j| \leq 1$ vorgeschlagen. Das Verfahren ist in dieser einfachen Form jedoch nicht global konvergent, da das Phänomen der „Zick-Zack-Bewegungen“ auftreten kann. Um dies zu verhindern, nimmt man in die Menge I auch solche Restriktionen auf, die „fast“ aktiv sind, die sogenannten „ α -aktiven“ Restriktionen [BS86], für die $g_i(x^{(k)}) \geq -\alpha$ mit einem zu wählenden $\alpha > 0$ gilt.

2.2.4 Projizierte Gradienten

Die Methoden der projizierten Gradienten sind eine Übertragung der Gradienten-Verfahren für unbeschränkte Optimierung auf den Fall mit Nebenbedingungen. Die Grundidee liegt darin, daß die verbessernde Richtung (der negative Gradient) auf den Rand des zulässigen Bereichs projiziert wird. Dadurch wird ein Weg auf der Grenze des (durch die Nebenbedingungen) erlaubten Bereichs bestimmt. Besonders interessant ist dieses Verfahren im Falle linearer Nebenbedingungen [Min86]. Die Methode ist zwar nicht global konvergent, Nichtkonvergenz kommt aber praktisch nicht vor. Auch die Erweiterung auf nichtlineare Nebenbedingungen ist möglich, jedoch erheblich schwieriger in der Handhabung, und die Effektivität sinkt wesentlich.

Mit linearen Nebenbedingungen betrachtet man das Problem

$$\begin{aligned} &\text{minimiere } f(x) \\ &\text{so daß } \quad a_i^T x \leq b_i, \quad i = 1, \dots, l \\ &\quad \quad a_i^T x = b_i, \quad i = l+1, \dots, m \end{aligned}$$

mit Gleichungs- und Ungleichungsnebenbedingungen. Ausgehend von einem zulässigen Startpunkt (der gegebenenfalls erst gesucht werden muß) wird eine zulässige und verbessernde Richtung gesucht. Dazu betrachtet man in jeder Iteration die Menge der aktiven Restriktionen, d. h. die Restriktionen, für die $a_i^T x^{(k)} = b_i$ gilt. (Die Gleichungsnebenbedingungen werden also immer mit betrachtet!)

Insgesamt läßt sich das Verfahren der projizierten Gradienten mit linearen Nebenbedingungen wie folgt formulieren, wenn A_k die Matrix bezeichnet, deren Zeilen die a_i^T der aktiven Restriktionen bilden, und P_k die „Projektionsmatrix“ der k -ten Iteration darstellt [Lue84]:

1. Bestimme den Teilraum der aktiven Restriktionen und bilde A_k .
2. Berechne $P_k = I - A_k^T (A_k A_k^T)^{-1} A_k$ und $d^{(k)} = -P_k \nabla f(x^{(k)})^T$.
3. Falls $d \neq 0$, suche α_1 und α_2 , die die folgenden Bedingungen erfüllen:

$$\alpha_1 : \max_{\alpha} \{x^{(k)} + \alpha d^{(k)} \text{ zulässig} \}$$

$$\alpha_2 : \min_{0 \leq \alpha \leq \alpha_1} f(x^{(k)} + \alpha d^{(k)})$$

$$x^{(k+1)} = x^{(k)} + \alpha_2 d^{(k)}, \text{ gehe zu Schritt 1}$$

4. Falls $d^{(k)} = 0$ berechne $\lambda = (A_k A_k^T)^{-1} A_k \nabla f(x^{(k+1)})^T$.
- (a) Falls $\lambda_j \geq 0$ für alle j , die aktive Restriktionen darstellen: **stop** ($x^{(k)}$ erfüllt dann die Kuhn-Tucker-Bedingungen).
- (b) Sonst lösche diejenige Zeile aus A_k , die zur kleinsten Komponente von λ gehört und gehe zu Schritt 2.

Man beachte jedoch, daß für das Verfahren die Restriktionen in A_k linear unabhängig sein müssen (A_k muß also vollen Rang haben). Es läßt sich dann zeigen, daß P_k eine Projektionsmatrix ist, für die $P_k = P_k^T$ und $P_k P_k = P_k$ gilt. P_k projiziert die Gradienten der bindenden Restriktionen in den Nullvektor. Die Matrix P_k muß dabei in Schritt 2. nicht in jeder Iteration komplett neu berechnet werden, statt dessen kann mit einer Update-Prozedur die alte Projektionsmatrix aktualisiert werden, wodurch der Rechenaufwand erheblich vermindert wird.

Das Verfahren der projizierten Gradienten läßt sich auch auf den Fall nichtlinearer Nebenbedingungen verallgemeinern. Man betrachte dazu das Problem

$$\begin{array}{lll} \text{minimiere} & f(x) & x \in \mathbb{R}^n \\ \text{so daß} & g_i(x) \geq 0 & i = 1, \dots, l \\ & g_j(x) = 0 & j = l+1, \dots, m \end{array}$$

Im oben angegebenen Algorithmus werden die Zeilen der Matrix A_k nun aus den transponierten Gradienten der bindenden Restriktionen gebildet. (Im Falle linearer Nebenbedingungen sind die a_i gerade die Gradienten.) Die Projektionsmatrix P_k projiziert den Gradienten ∇f nun im allgemeinen jedoch nicht mehr auf den Rand des zulässigen Bereichs, sondern auf die Tangentialebene am Iterationspunkt $x^{(k)}$. Schritt 3. des Algorithmus muß also so modifiziert werden, daß zusätzlich ein Korrektorschritt durchgeführt wird, wie dies in Abbildung 2.1, Seite 28, zu erkennen ist. Auch die Aufteilung der Bestimmung der Schrittweite in Schritt 3. in die Suche nach α_1 und α_2 ist hier nicht mehr möglich, da im allgemeinen $d^{(k)}$ keine zulässige Richtung ist. Es wird also nach α gesucht, so daß $f(x^{(k)} + \alpha d^{(k)})$ minimiert wird. Von dem so gefundenen Punkt aus wird dann ein Korrekturschritt ausgeführt, durch den orthogonal zur Tangentialebene wieder der zulässige Bereich erreicht wird. Besondere Beachtung erfordert hier noch der Fall, daß durch die Korrekturrichtung kein

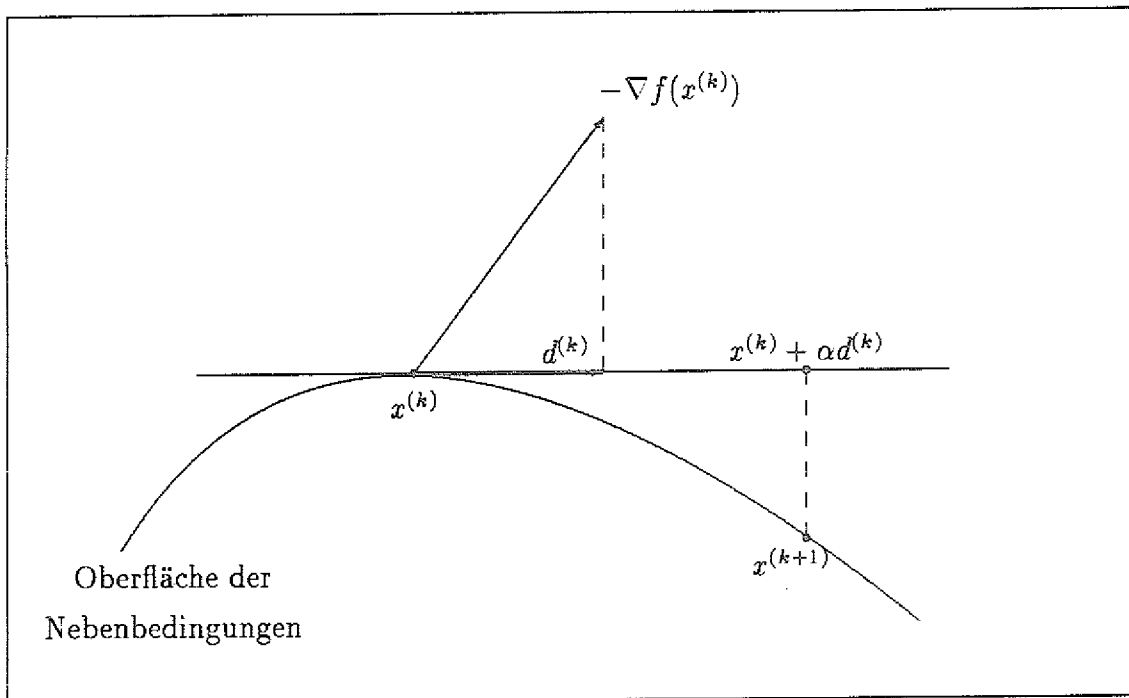


Abbildung 2.1: Methode der projizierten Gradienten

zulässiger Punkt gefunden wird. Dieser Fall kann zum Beispiel auftreten, wenn eine zusätzliche Restriktion aktiv wird. Weitere Einzelheiten dazu sind zum Beispiel in [Lue84] zu finden.

Kapitel 3

Duale Verfahren

3.1 Dualität

Häufig macht es Sinn, an Stelle des formulierten (primalen) Problems das dazu duale Problem zu betrachten. Dieses ist zwar nicht per se einfacher zu lösen als das primale, in einer Reihe von wichtigen Spezialfällen haben duale Algorithmen jedoch Vorteile gegenüber den primalen. Diese Methoden sind normalerweise robuster, und globale Konvergenz ist leichter zu garantieren [Day89]. Die Algorithmen basieren auf der Sattelpunkt- und Lagrange-Theorie, die im folgenden vorgestellt wird.

Dazu wird das folgende Problem betrachtet:

$\begin{array}{ll} \mathbf{P} : & \\ \text{minimiere} & f(x) \quad x \in X \subset \mathbb{R}^n \\ \text{so daß} & g_i(x) \leq 0 \quad i = 1, \dots, m \end{array}$	(3.1)
---	-------

Die zu diesem Problem gehörige *Lagrange-Funktion* ist dann wie folgt definiert:

$$\mathcal{L}(x, \lambda) = f(x) + \sum_{i=1}^m \lambda_i g_i(x) \quad (3.2)$$

Ein Punkt (x^*, λ^*) heißt dann *Sattelpunkt* von $\mathcal{L}$, falls für alle $x \in \mathbb{R}^n, \lambda \in \mathbb{R}^m$ gilt:

$$\mathcal{L}(x^*, \lambda) \leq \mathcal{L}(x^*, \lambda^*) \leq \mathcal{L}(x, \lambda^*) \quad (3.3)$$

Ein Sattelpunkt stellt also zu $\mathcal{L}$ ein Minimum in x und ein Maximum in λ dar. Falls ein solcher Sattelpunkt (x^*, λ^*) zu $\mathcal{L}$ existiert, so ist x^* eine optimale Lösung zu $\mathbf{P}$.

Statt direkt nach x^* zu f zu suchen, kann man also auch nach einem Sattelpunkt zu $\mathcal{L}$ suchen. Mit der Funktion

$$w(\lambda) = \min_{x \in X} \{\mathcal{L}(x, \lambda)\} \quad (3.4)$$

ergibt sich das duale Problem zu P als:

$\begin{aligned} &D : \\ &\text{maximiere } w(\lambda) = \min_{x \in X} \mathcal{L}(x, \lambda) \\ &\text{so daß } \lambda \geq 0 \end{aligned}$	(3.5)
--	-------

Sowohl die duale Funktion w als auch das duale Problem D sind immer definiert, unabhängig davon, ob ein Sattelpunkt zu $\mathcal{L}$ und somit ein Optimum zu f existiert.

Der Zusammenhang zwischen dem primalen und dem dualen Problem wird im wesentlichen durch zwei Sätze charakterisiert:

Satz 1: (Schwache Dualität)

Seien x^*, λ^* Optima zu P bzw. D , ferner $\lambda \geq 0$. Dann gilt:

$$w(\lambda) \leq w(\lambda^*) \leq f(x^*) \quad (3.6)$$

Satz 2: (Starke Dualität)

Wenn die Funktion $\mathcal{L}$ einen Sattelpunkt (x^*, λ^*) besitzt, so gilt:

$$\max(D) = w(\lambda^*) = f(x^*) = \min(P); \quad (3.7)$$

das Maximum des dualen ist also gleich dem Minimum des primalen Problems.

Weiterhin gilt:

Falls es eine Lösung x^* zu P und $\lambda^* \geq 0$ gibt mit

$$w(\lambda^*) = f(x^*), \quad (3.8)$$

dann existiert ein Sattelpunkt zu $\mathcal{L}$ und (x^*, λ^*) ist ein solcher.

Beweise zu diesen Aussagen finden sich zum Beispiel in [Min86].

Außerdem ist es wichtig, festzuhalten, daß die Funktion w konkav in λ ist. Das duale Problem läuft also auf die Maximierung einer konkaven Funktion über einer konvexen Menge ($\lambda \geq 0$) hinaus, was gleichbedeutend mit der Minimierung einer konvexen Funktion über der Menge ist. Der dazu benötigte Gradient $\nabla w(\lambda)$ berechnet sich sehr einfach zu

$$\nabla w(\lambda) = g(x), \quad (3.9)$$

wobei $x = x(\lambda)$ durch $w(\lambda) = \mathcal{L}(x(\lambda), \lambda)$ definiert ist; denn mit

$$w(\lambda) = f(x(\lambda)) + \sum_{i=1}^n \lambda_i g_i(x(\lambda))$$

folgt durch komponentenweises Anwenden der Kettenregel

$$\frac{\partial}{\partial \lambda_j} w(\lambda) = \sum_{k=1}^n \frac{\partial f(x)}{\partial x_k} \cdot \frac{\partial x_k}{\partial \lambda_j} + g_j(x) + \sum_{i=1}^n \lambda_i \sum_{k=1}^n \frac{\partial g_i(x)}{\partial x_k} \cdot \frac{\partial x_k}{\partial \lambda_j}.$$

Da $x(\lambda)$ Minimumsstelle von $\mathcal{L}$ bezüglich x , gilt

$$\frac{\partial}{\partial x_k} \mathcal{L}(x, \lambda) = \frac{\partial f(x)}{\partial x_k} + \sum_{i=1}^n \lambda_i \frac{\partial g_i(x)}{\partial x_k} = 0, \quad k = 1, \dots, n,$$

damit auch

$$\sum_{k=1}^n \frac{\partial f(x)}{\partial x_k} \cdot \frac{\partial x_k}{\partial \lambda_j} = - \sum_{k=1}^n \sum_{i=1}^n \lambda_i \frac{\partial g_i(x)}{\partial x_k} \cdot \frac{\partial x_k}{\partial \lambda_j}$$

und somit

$$\frac{\partial}{\partial \lambda_j} w(\lambda) = g_j(x),$$

also

$$\nabla w(\lambda) = g(x).$$

3.2 Strafkosten-Verfahren

Bei den Strafkosten-Verfahren zur Optimierung unter Nebenbedingungen werden die Nebenbedingungen in die Zielfunktion einbezogen, aus dem beschränkten Problem entsteht also ein unbeschränktes. Die ursprüngliche Zielfunktion wird dabei so modifiziert, daß sie am Rand des zulässigen Bereichs sehr stark ansteigt, dort also „Strafkosten“ entstehen.

Die Strafkosten lassen sich grob in zwei Klassen aufteilen: die Verfahren mit externen Strafkosten („Penalty-Verfahren“) und die mit internen Strafkosten („Barriere-Verfahren“). Bei ersteren nähert man sich von ausserhalb des zulässigen Bereiches an die Optimalstelle an, es wird im allgemeinen eine Folge von unzulässigen Punkten erzeugt, bis ein Minimum gefunden ist. Im Gegensatz dazu beginnt man bei den Barriere-Verfahren mit einem zulässigen Startpunkt und nähert sich im Inneren des Lösungsraumes dem Optimum an.

3.2.1 Penalty-Verfahren

Durch Penalty-Verfahren wird das Problem

$$\begin{array}{ll} \text{minimiere} & f(x) \\ \text{so daß} & g_i(x) \leq 0 \quad i = 1 \dots l \\ & g_j(x) = 0 \quad j = l + 1 \dots m \end{array} \quad (3.10)$$

in eine Folge von unbeschränkten Problemen

$$\text{minimiere } f(x) + \gamma_k \cdot p_k(x) \quad (3.11)$$

überführt. Dabei sind die $p_k : \mathbb{R}^n \rightarrow \mathbb{R}$ die Penalty-Funktionen, während die $\gamma_k \in \mathbb{R}$ eine Folge von Gewichten bilden. Um die Konvergenz der Optimumsstellen einer solchen Folge von Problemen gegen eine Optimumsstelle des ursprünglichen Problems zu erreichen, müssen die folgenden Bedingungen eingehalten werden:

1. $\gamma_k p_k(x)$ ist stetig in $\mathbb{R}^n$
2. $\gamma_k p_k(x) = 0$ für alle zulässigen x
3. $\gamma_{k+1} p_{k+1}(x) > \gamma_k p_k(x) > 0$ für alle nicht zulässigen x

4. $\lim_{k \rightarrow \infty} \gamma_k p_k(x) = \infty$ für alle nicht zulässigen x

Eine Funktion, die diesen Ansprüchen genügt (falls keine Gleichungsnebenbedingungen vorhanden sind), ist etwa

$$p(x) = \sum_{i=1}^l \max\{0, g_i(x)\}^2 \quad (3.12)$$

Man beginnt nun mit einem relativ kleinen γ_0 und läßt in den Iterationen die Folge der γ_k gegen „unendlich“ streben, damit die Lösung der Strafkostenmodelle gegen die (zulässige) Lösung des Ausgangsproblems konvergiert. Solange γ klein ist, liegt das Gewicht des Modells mehr auf der Optimierung der Zielfunktion, bei steigendem γ rückt dann die Zulässigkeit immer mehr in den Vordergrund.

In Abbildung 3.1 ist das Verhalten der Strafkostenfunktionen bei steigendem γ_k anhand der einfachen Restriktionen $g_1(x) = a - x$ und $g_2(x) = x - b$ im eindimensionalen Fall dargestellt.

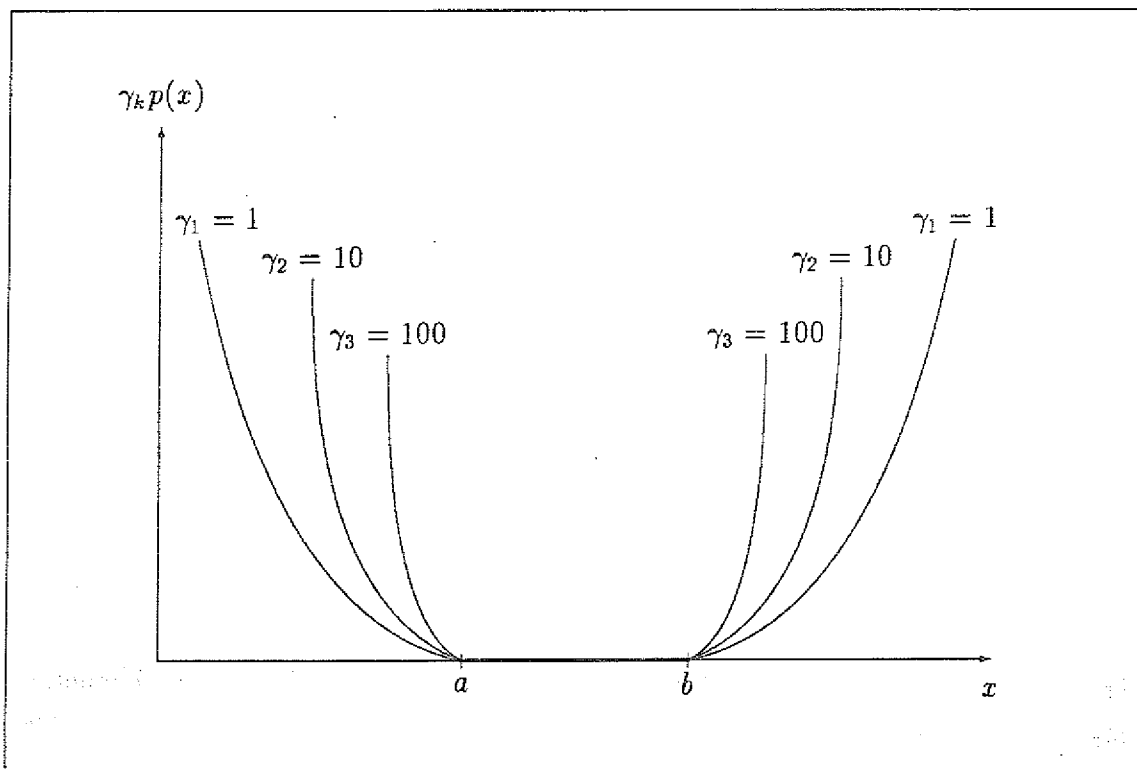


Abbildung 3.1: Penalty-Funktionen

Sollen Probleme betrachtet werden, die auch Gleichungsnebenbedingungen enthalten, so kann die Strafkostenfunktion zum Beispiel wie folgt gewählt werden [Bas89]:

$$p(x) = \sum_{i=1}^l [\max\{0, g_i(x)\}]^r + \sum_{j=l+1}^m |g_j(x)|^r, \quad 1 \leq r \in \mathbb{N} \quad (3.13)$$

3.2.2 Barriere-Verfahren

Statt einer externen Strafkostenfunktion wie im Penalty-Verfahren, die eine Annäherung an den zulässigen Bereich von außen sicherstellen soll, wird beim Barriere-Verfahren eine interne Strafkostenfunktion verwendet. Diese wird so gewählt, daß sie Inneren des Lösungsraums sehr klein ist und an dessen Rändern Polstellen besitzt. In den Iterationen wird dadurch eine Folge innerer Punkte des Lösungsraumes generiert. Da es in Bezug auf Gleichungsnebenbedingungen keinen solchen inneren Punkt geben kann, ist das Verfahren für solche Probleme nicht geeignet. Es wird also für das Problem

$$\begin{array}{ll} \text{minimiere} & f(x) \\ \text{so daß} & g_i(x) \geq 0 \quad i = 1 \dots m \end{array} \quad (3.14)$$

eine Folge von Ersatzproblemen erzeugt. Dabei kann eine Barriere-Funktion der Form

$$b(x) = - \sum_{i=1}^m \frac{1}{g_i(x)} \quad (3.15)$$

verwendet werden. Auch logarithmische Barriere-Funktionen

$$b(x) = \sum_{i=1}^m \ln(g_i(x)) \quad (3.16)$$

werden häufig verwendet. Mit der Folge von Gewichten μ_k mit $\mu_k \rightarrow 0$ für $k \rightarrow \infty$ ergeben sich die Ersatzprobleme

$$\text{minimiere } f(x) + \mu_k \cdot b(x). \quad (3.17)$$

Im Gegensatz zum Penalty-Verfahren werden die Gewichte μ_k hier immer kleiner, um eine Annäherung an den Rand zu erlauben. Abbildung 3.2 zeigt das Verhalten solcher Barriere-Funktionen, wiederum anhand der einfachen Restriktionen $g_1(x) = a - x$ und $g_2(x) = x - b$.

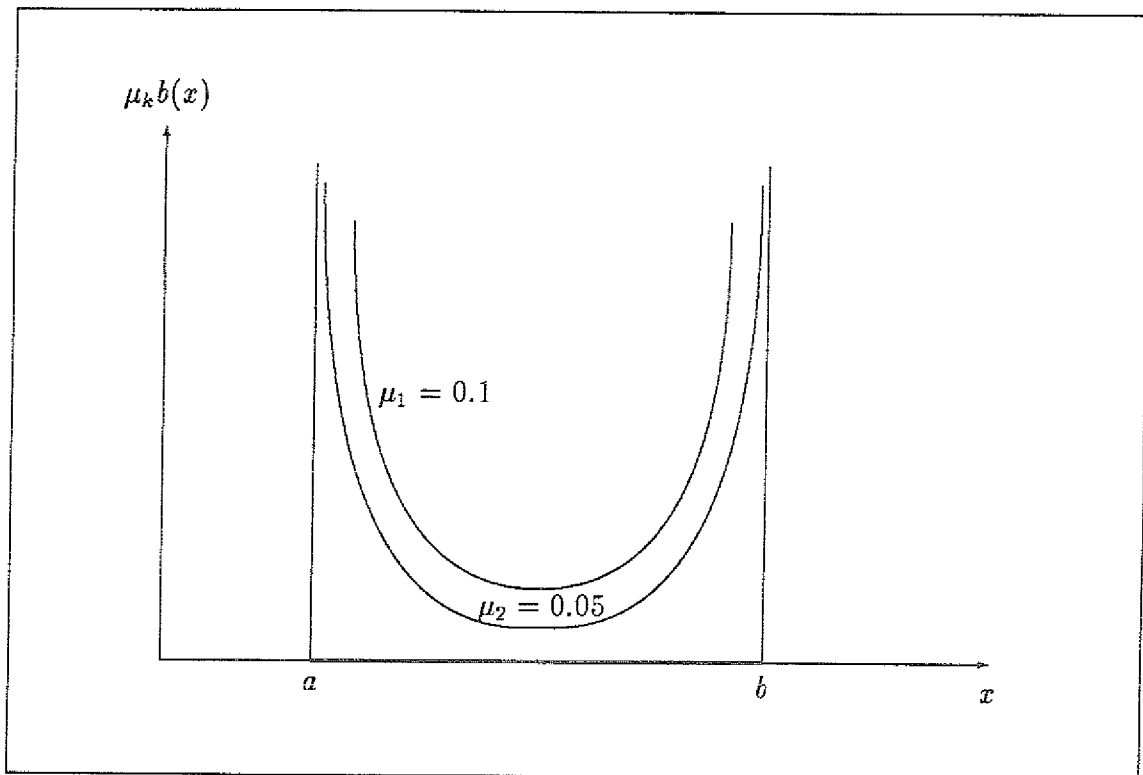


Abbildung 3.2: Barriere-Funktionen

3.2.3 SUMT-Verfahren

SUMT (Sequential Unconstraint Minimization Technique) kombiniert Penalty- und Barriere-Verfahren. Die Strafkostenfunktion erhält einen 'Penalty-Anteil' für die Gleichungsnebenbedingungen und einen 'Barriere-Anteil' für die Ungleichungsnebenbedingungen.

Zur Lösung des Problems (3.10) wird in jedem Schritt die Funktion

$$P(x, \gamma_k) = f(x) + \frac{1}{\gamma_k} \sum_{i=1}^l \frac{1}{g_i(x)} + \gamma_k \sum_{i=l+1}^m g_i(x)^2 \quad (3.18)$$

mit $\gamma_k > 1$ und $\gamma_k < \gamma_{k+1}$ minimiert. Als Startpunkt wird ein $x^{(0)}$ mit $g_i(x^{(0)}) > 0$, $i = 1, \dots, l$ gewählt, die Iterationen werden solange durchgeführt, bis die Nebenbedingungen aus Problem (3.10) — bis auf eine vorgegebene Toleranz — erfüllt werden.

3.3 Lagrange-Verfahren

Die hier als „Lagrange-Verfahren“ bezeichneten Verfahren beruhen auf der Dualität nach Lagrange. Die Lagrange-Funktion wird aufgestellt und zu dieser ein Fixpunkt (Sattelpunkt) gesucht. Die Verfahren dazu sind den primalen Verfahren sehr ähnlich, so daß auf eine genauere Beschreibung verzichtet werden kann. Wie bereits erwähnt, sind die dualen Probleme nicht generell einfacher zu lösen als die primalen. Insbesondere bei konvexen Problemen (und bei anderen Problemen mit speziellen Strukturen) sind jedoch einige Vorteile zu erwarten. So zeigt sich etwa im folgenden Gradienten-Verfahren, daß zwar das duale Problem nicht unbeschränkt ist, aber nur Nichtnegativitätsbedingungen zu erfüllen sind, wodurch die Projektionen auf den zulässigen Bereich durch einfache Maximumsbildung durchzuführen sind.

3.3.1 Gradienten-Verfahren

Beim Gradienten-Verfahren wird — analog zum Gradientenverfahren in der Minimierung unter Nebenbedingungen — der Gradient als Suchrichtung benutzt, da er die Richtung des steilsten Anstiegs darstellt. Das Verfahren arbeitet dann wie folgt:

1. Wähle einen Startpunkt $\lambda^{(0)} \geq 0$, $k := 0$.
2. Berechne in Schritt k bei gegebenem $\lambda^{(k)}$

$$w(\lambda^{(k)}) = \min_{x \in X} \left\{ f(x) + \sum_{i=1}^m \lambda_i^{(k)} g_i(x) \right\} \quad (3.19)$$

$x^{(k)}$ sei die Lösung dieses Problems.

3. $g(x^{(k)})$ ist der Gradient von w . Ist dieser gleich 0 (bzw. hinreichend klein, um ein Abbruchkriterium zu erfüllen), so ist der Optimalpunkt gefunden, **stop**.

Sonst:

$$\lambda_i^{(k+1)} = \max \left\{ 0, \lambda_i^{(k)} + \gamma_k g_i(x^{(k)}) \right\} \quad \forall i = 1, \dots, m \quad (3.20)$$

$k := k+1$, gehe zu Schritt 2.

Die Schrittweite γ_k kann entweder im Voraus festgelegt werden, oder aber so gewählt werden, daß sie $w(\lambda^{(k)})$ in Richtung $g(x^{(k)})$ maximiert. Die Maximumsbildung ist dann die — sehr einfach auszuführende — Projektion auf den positiven Quadranten, um die (duale) Zulässigkeit zu erhalten.

3.3.2 Verfahren von Dantzig

Im Verfahren von Dantzig (auch Schnittebenenverfahren genannt) wird das duale Problem (3.5) in ein äquivalentes lineares Problem überführt:

$$\begin{aligned} &\text{maximiere } z \\ &\text{so daß } f(x) + \sum_{i=1}^m \lambda_i g_i(x) \geq z \quad \forall x \in X \\ &\lambda \geq 0 \end{aligned} \quad (3.21)$$

Da X im allgemeinen nicht endlich ist, enthält dieses Problem jedoch eine unendliche Menge von Restriktionen. Trotzdem läßt sich durch den folgenden Algorithmus — das Verallgemeinerte Lineare Programmieren [Min86] — durch Lösen endlicher linearer Probleme eine Lösung dazu finden:

1. Finde $x^{(0)} \in X$ mit $g(x^{(0)}) \leq 0$, $k := 1$.

2. Löse das Problem

$$\begin{aligned} &\text{maximiere } z \\ &\text{so daß } f(x^{(j)}) + \sum_{i=1}^m \lambda_i g_i(x^{(j)}) \geq z, \quad \text{für } j = 0, \dots, k-1 \\ &\lambda \geq 0 \end{aligned} \quad (3.22)$$

3. Mit der dazu erhaltenen Lösung $(z^{(k)}, \lambda^{(k)})$ berechne die duale Funktion

$$\begin{aligned} w(\lambda^{(k)}) &= \min_{x \in X} \left\{ f(x) + \sum_{i=1}^m \lambda_i^{(k)} g_i(x) \right\} \\ &= f(x^{(k)}) + \sum_{i=1}^m \lambda_i^{(k)} g_i(x^{(k)}) \end{aligned} \quad (3.23)$$

4. Falls $w(\lambda^{(k)}) < z^{(k)}$, dann $k := k + 1$, gehe zu Schritt 2.

Sonst **stop**, denn $\lambda^{(k)}$ ist die optimale Lösung des dualen Problems und somit $x^{(k)}$ die Optimalstelle des primalen Problems.

3.3.3 Erweiterte Lagrange-Funktion

Idee der erweiterten Lagrange-Funktionen ist die Kombination von Dualität und Strafkostenfunktionen. Das Optimierungsproblem mit Nebenbedingungen wird wieder in ein unbeschränktes Problem überführt. Als Gütefunktion (*merit-function*)

verwendet man die Lagrange-Funktion, die zusätzlich durch einen quadratischen Term erweitert wird.

Falls nur Gleichungsnebenbedingungen vorhanden sind, also das Problem

$$\begin{aligned} &\text{minimiere} && f(x) && x \in X \\ &\text{so da\ss} && g_i(x) = 0 && i = 1, \dots, m \end{aligned} \quad (3.24)$$

betrachtet wird, kann man eine Folge von Problemen der Form

$$\min_{x \in X} \left\{ f(x) + \gamma \sum_{i=1}^m [g_i(x)]^2 + \sum_{i=1}^m \lambda_i g_i(x) \right\} \quad (3.25)$$

lösen. Mit dieser Funktion (1969 von Hestenes und Powell unabhängig voneinander vorgeschlagen [Roc73]) wird meistens verhindert, daß γ gegen unendlich streben muß — was bei Penalty-Verfahren der Fall ist und zu numerischen Instabilitäten führen kann.

Diese Methode lässt sich nach [Roc73] leicht auf Ungleichungsnebenbedingungen verallgemeinern. Falls also das Problem

$$\begin{aligned} &\text{minimiere} && f(x), && x \in X \\ &\text{so da\ss} && g_i(x) \leq 0, && i = 1, \dots, m \end{aligned} \quad (3.26)$$

betrachtet wird, so wird dieses durch Schlupfvariablen $z \geq 0$ mit

$$g_i(x) + z_i = 0, \quad i = 1, \dots, m \quad (3.27)$$

in die Form (3.24) überführt und die zugehörige erweiterte Lagrange-Funktion, die dann über $x \in X$ und $z \geq 0$ minimiert wird, erhält die Form

$$f(x) + \gamma \sum_{i=1}^m [g_i(x) + z_i]^2 + \sum_{i=1}^m \lambda_i [g_i(x) + z_i]. \quad (3.28)$$

Die notwendige Minimierung über z kann explizit (komponentenweise!) ausgeführt werden. Dazu betrachte man die Terme, in denen z_i vorkommt. Diese lauten

$$\lambda_i z_i + 2\gamma g_i(x) z_i + \gamma z_i^2. \quad (3.29)$$

Mit $\gamma > 0$ ergibt sich die Minimalstelle in z_i zu

$$z_i = \begin{cases} -g_i(x) - \frac{\lambda_i}{2\gamma} & \text{falls } g_i(x) \leq -\frac{\lambda_i}{2\gamma} \\ 0 & \text{falls } g_i(x) > -\frac{\lambda_i}{2\gamma} \end{cases} \quad (3.30)$$

Damit reduziert sich die erweiterte Lagrange-Funktion auf

$$\mathcal{L}(x, \lambda, \gamma) = f(x) + \sum_{i=1}^m G(g_i(x), \lambda, \gamma) \quad (3.31)$$

mit

$$G(g_i(x), \lambda, \gamma) = \begin{cases} \lambda_i g_i(x) + \gamma [g_i(x)]^2 & \text{falls } g_i(x) \geq -\frac{\lambda_i}{2\gamma} \\ -\frac{\lambda_i^2}{4\gamma} & \text{falls } g_i(x) \leq -\frac{\lambda_i}{2\gamma} \end{cases} \quad (3.32)$$

Man kann leicht einen kombinierten Algorithmus für den Fall angeben, daß sowohl Gleichungs- als auch Ungleichungsnebenbedingungen vorhanden sind. Es kann auch sinnvoll sein, das Verfahren mit anderen zu kombinieren, also etwa für Gleichungsrestriktionen ein Gradientenverfahren zu benutzen, während die Ungleichungen mittels der erweiterten Lagrange-Funktion behandelt werden.

Verfahren mit erweiterten Lagrange-Funktionen gehören zu den effizientesten und robustesten Methoden zur Lösung mathematischer Optimierungsprobleme mit nicht-linearen Ziel- und Nebenbedingungsfunktionen [Min86]. Außer der oben vorgestellten Funktion nach [Roc73] wurden eine Reihe anderer erweiterter Lagrange-Funktionen vorgeschlagen, die alle auf der gleichen Grundidee basieren. Als Nachteile der Verfahren kann es angesehen werden, daß mehrfach ein unbeschränktes Optimierungsproblem gelöst werden muß, was den Aufwand steigen läßt. Außerdem müssen Konstanten a priori geeignet festgelegt werden [Pap91].

3.4 Quadratische Programmierung

Die Lagrange-Funktion zum Problem der quadratischen Programmierung

$$\begin{aligned} &\text{minimiere} && f(x) = c^T x + \frac{1}{2} x^T Q x \\ &\text{so da\ss} && Ax \leq b \end{aligned} \quad (3.33)$$

ist gegeben durch

$$\mathcal{L}(x, \lambda) = c^T x + \frac{1}{2} x^T Q x + \lambda^T (Ax - b). \quad (3.34)$$

Um die duale Funktion $w(\lambda) = \min_{x \in X} \mathcal{L}(x, \lambda)$ zu erhalten, mu\ss diese in x minimiert werden. Notwendige Bedingung f\ur ein Minimum in x ist $\mathcal{L}_x = 0$, also

$$\mathcal{L}_x(x, \lambda) = c + \frac{1}{2}(Q + Q^T)x + (\lambda^T A)^T = 0. \quad (3.35)$$

Da Q symmetrisch, ist $\frac{1}{2}(Q + Q^T) = Q$. Da weiterhin Q positiv definit und somit invertierbar, gilt in der Minimalstelle

$$x = Q^{-1}(-c - A^T \lambda). \quad (3.36)$$

Dieses eingesetzt in $w(\lambda)$ ergibt:

$$\begin{aligned} w(\lambda) &= c^T Q^{-1}(-c - A^T \lambda) + \frac{1}{2}([Q^{-1}(-c - A^T \lambda)]^T Q Q^{-1}(-c - A^T \lambda)) \\ &\quad + \lambda^T (A Q^{-1}(-c - A^T \lambda) - b) \\ &= -c^T Q^{-1}c - c^T Q^{-1}A^T \lambda + \frac{1}{2}(-c^T - \lambda^T A)Q^{-1}(-c - A^T \lambda) \\ &\quad - \lambda^T A Q^{-1}c - \lambda^T A Q^{-1}A^T \lambda - \lambda^T b. \end{aligned}$$

Da $Q^{-1T} = Q^{T^{-1}} = Q^{-1}$ (Q ist symmetrisch) und $\lambda^T A Q^{-1}c$ ein Skalar (somit gleich seinem Tranponierten), ist

$$\begin{aligned} w(\lambda) &= -c^T Q^{-1}c + \frac{1}{2}(c^T Q^{-1}c + c^T Q^{-1}A^T \lambda + \lambda^T A Q^{-1}c + \lambda^T A Q^{-1}A^T \lambda) \\ &\quad - 2\lambda^T A Q^{-1}c - \lambda^T A Q^{-1}A^T \lambda - \lambda^T b \\ &= -\frac{1}{2}c^T Q^{-1}c - (b^T + c^T Q^{-1}A^T)\lambda - \frac{1}{2}\lambda^T A Q^{-1}A^T \lambda. \end{aligned}$$

Der Term $\frac{1}{2}c^T Q^{-1}c$ h\angt nicht von λ ab und hat somit keinen Einflu\ss auf die Minimalstelle. Das duale Problem kann dann — mit Vorzeichenumkehrung, um ein Minimierungsproblem zu erhalten — wie folgt formuliert werden:

$$\begin{aligned} &\text{minimiere} && r^T \lambda + \frac{1}{2} \lambda^T P \lambda \\ &\text{so da\ss} && \lambda \geq 0, \end{aligned} \quad (3.37)$$

wobei

$$P = AQ^{-1}A^T \quad \text{und} \quad r = b + AQ^{-1}c \quad (3.38)$$

Dieses Problem ist einfacher zu lösen — und geeigneter für die Parallelisierung — weil es nur reine Nichtnegativitätsbedingungen hat, die leichter handhabbar sind als die ursprünglichen Nebenbedingungen.

3.5 Separable Optimierung

Im Falle des separablen Optimierungsproblems

$$\begin{aligned} &\text{minimiere} && f(x) = \sum_{j=1}^n f_j(x_j) \\ &\text{so da\ss} && \sum_{j=1}^n g_{ij}(x_j) \geq 0 \quad i = 1, \dots, m \end{aligned} \quad (3.39)$$

hat die Lagrange-Funktion $\mathcal{L}$ die Form

$$\mathcal{L}(x, \lambda) = \sum_{j=1}^n \left\{ f_j(x_j) + \sum_{i=1}^m \lambda_i g_{ij}(x_j) \right\}. \quad (3.40)$$

Diese ist wiederum separabel in x für jedes feste λ und die duale Funktion w kann als

$$w(\lambda) = \sum_{j=1}^n \min_{x_j} \left\{ f_j(x_j) + \sum_{i=1}^m \lambda_i g_{ij}(x_j) \right\} \quad (3.41)$$

geschrieben werden. Unter der Voraussetzung, daß die $f_j(x)$ für alle $j = 1, \dots, n$ streng konvex sind, ist w stetig differenzierbar über $\mathbb{R}$. Um w auszuwerten, müssen also n eindimensionale Minimierungsprobleme gelöst werden. Insbesondere wenn für diese einfache analytische Ausdrücke existieren, kann die Dualisierung schon auf sequentiellen Rechnern eine deutliche Beschleunigung bewirken.

Kapitel 4

Parallelisierung

4.1 Parallelrechnerkonzepte

Die größten Einschränkungen bei der Lösung von Optimierungsproblemen liegen in der Verarbeitungsgeschwindigkeit der dazu verwendeten Computer. In den vergangenen Jahren wurde eine Erhöhung der Rechengeschwindigkeit durch immer raffiniertere elektronische Technik erreicht. Es wird jedoch immer schwieriger, neue leistungsfähigere Systeme nur durch technologische Verbesserungen zu entwickeln; zur Verbesserung der Leistungsfähigkeit ist es nötig, auch die gesamte Computerarchitektur in Betracht zu ziehen. Das wohl wichtigste Konzept zur Beseitigung des „von Neumann-Flaschenhalses“ der klassischen sequentiellen Rechner ist die Parallelverarbeitung.

Nach Hwang und Briggs [HB84] sind parallele Computer in drei Architekturklassen einzuteilen:

- Pipeline-Computer

- Array-Prozessoren

- Multiprozessorsysteme.

Ein Pipeline-Computer führt überlappende Berechnungen aus und nutzt dadurch *zeitlichen Parallelismus* aus. Ein Array-Prozessor nutzt viele synchronisierte Arithmetisch-Logische Einheiten, um *räumlichen Parallelismus* zu erreichen. Ein Multiprozessorsystem erreicht *asynchronen Parallelismus* durch eine Menge interaktiver

Prozessoren mit gemeinsamen Ressourcen (Speicher, Datenbank, etc.). Diese drei Ansätze schließen sich nicht notwendigerweise gegenseitig aus, die meisten heutigen Computer nutzen das Pipelineprinzip und einige haben zusätzlich Array- oder Multiprozessorstruktur.

Die übliche Klassifizierung digitaler Computer nach Instruktions- und Datenfluß, eingeführt von Flynn, enthält die folgenden vier Klassen:

- SISD (Single Instruction Stream – Single Data Stream)
- SIMD (Single Instruction Stream – Multiple Data Stream)
- MISD (Multiple Instruction Stream – Single Data Stream)
- MIMD (Multiple Instruction Stream – Multiple Data Stream)

In die erste dieser Klassen (SISD) fallen die „normalen“ sequentiellen Computer. Die Instruktionen werden nacheinander (aber möglicherweise nach dem Pipelineprinzip überlappend) auf einem einzigen Datenstrom ausgeführt.

Typische Vertreter der Klasse der SIMD-Computer sind die Array-Architekturen. Jeder Prozessor (bzw. jede ALU) erhält den gleichen Befehl, der dann jedoch auf verschiedenen Datenströmen ausgeführt wird.

Die Klasse der MISD-Computer ist hauptsächlich von theoretischer Bedeutung, n essoreinheiten erhalten dabei verschiedene Instruktionen, die auf dem gleichen Datenstrom ausgeführt werden. Es existieren jedoch praktisch keine Realisierungen, so daß die Klasse üblicherweise als leer angesehen wird.

Die meisten modernen Multiprozessorrechner fallen heute in die MIMD-Klasse. Innerhalb dieser Klasse ist noch einmal eine Unterscheidung nach der Organisation des Speichers sinnvoll. Verfügt jeder Prozessor über einen eigenen (Haupt-)Speicher, so spricht man von verteiltem Speicher (*distributed memory*). Im Gegensatz dazu arbeiten bei gemeinsamem Speicher (*shared memory*) alle Prozessoren auf demselben Hauptspeicher. Dies hat natürlich Einfluß auf die Kommunikation und Synchronisation zwischen den Prozessoren, die (heute) bei Shared-Memory-Rechnern leichter zu realisieren sind.

4.1.1 Pipelining

In Rechnern, die nach dem Pipelineprinzip arbeiten, sind die Kontrolleinheit (*control unit, CU*) und/oder die Arithmetisch-Logische Einheit (*arithmetic logic unit, ALU*) praktisch wie ein Fließband angeordnet. Die aufeinanderfolgenden Instruktionen werden dabei überlappend ausgeführt. Das Prinzip des Pipelining wird im folgenden anhand einer Instruktionen-Pipeline erläutert.

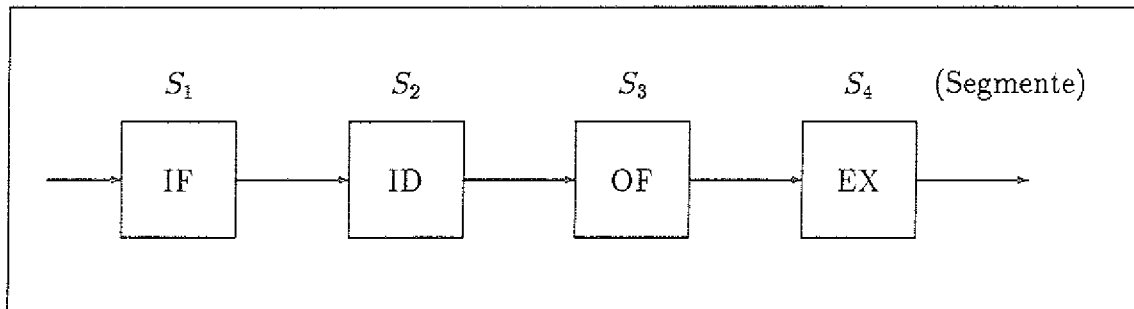


Abbildung 4.1: Autonome Segmente einer Instruktion

Wie in Abbildung 4.1 zu sehen ist, läßt sich eine typische Instruktion eines Computers in vier aufeinanderfolgende Phasen (Segmente) aufteilen:

1. Laden der Instruktion (*instruction fetch, IF*)
2. Decodieren der Instruktion (*instruction decode, ID*)
3. Laden der Operanden (*operand fetch, OF*)
4. Ausführen der Instruktion (*execute, EX*).

Diese Phasen können von verschiedenen Modulen der CPU ausgeführt werden. In herkömmlichen Computern ohne Pipelining laufen diese Phasen streng sequentiell ab. Abbildung (4.2) zeigt, daß dadurch nach jeweils vier Takten ein Ergebnis erzeugt wird. Jede Funktionseinheit ist jedoch drei Viertel der Zeit nicht beschäftigt. In einer Pipeline werden die Instruktionen überlappend ausgeführt. So werden beispielsweise zum gleichen Zeitpunkt die Operanden für Instruktion I_1 geholt, die Instruktion I_2 decodiert und die dritte Instruktion I_3 geladen (Abbildung 4.3). Dadurch liegt — nach einer *Startup*-Phase von vier Takten — in jedem Takt ein Ergebnis vor.

Bei verschiedenen Instruktionen kann die Ausführungsphase noch weiter unterteilt werden. Auch in arithmetischen Pipelines ist die Anzahl der Segmente meist größer.

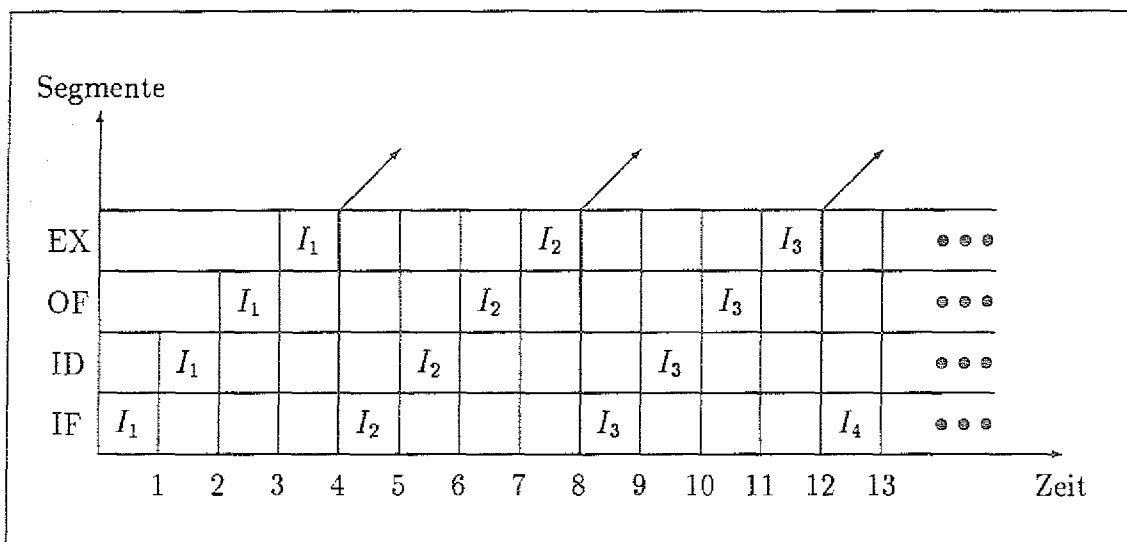


Abbildung 4.2: Raum-Zeit-Diagramm für Verarbeitung ohne Pipeline

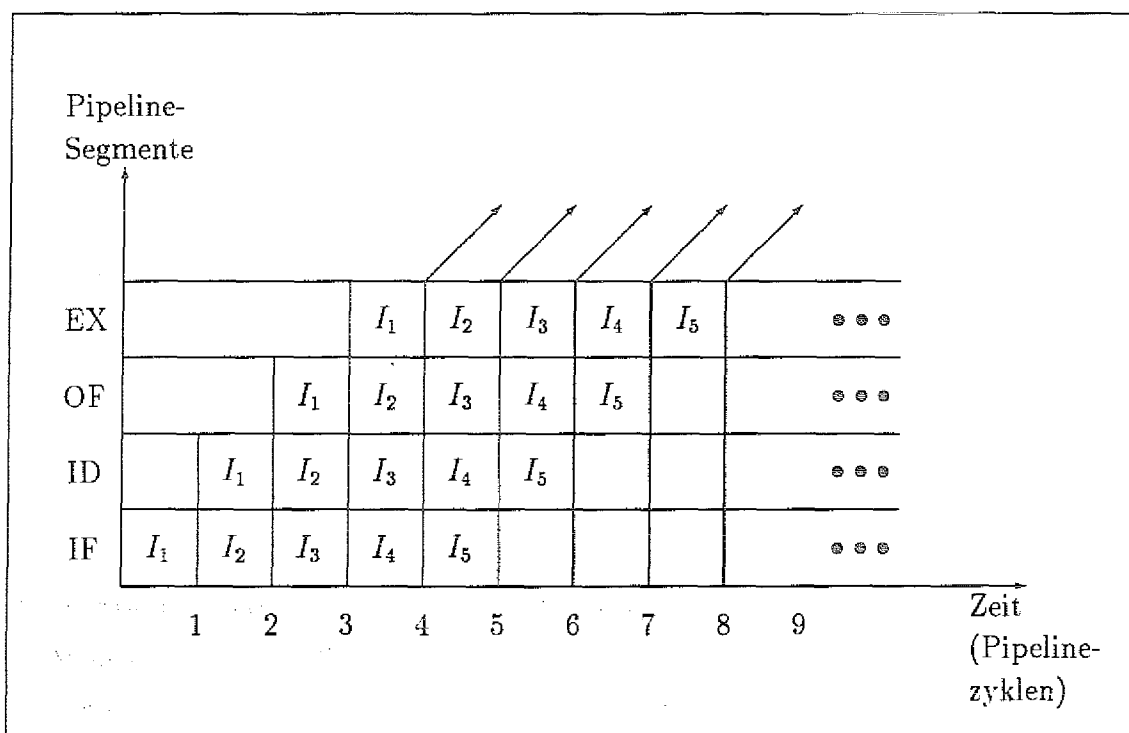


Abbildung 4.3: Raum-Zeit-Diagramm für Pipeline-Verarbeitung

Allgemein besteht eine Pipeline aus k Segmenten. Auf einem seriellen Rechner würde die Ausführung von N identischen Operationen damit also

$$T_1 = k \cdot N \quad (4.1)$$

Zeitschritte benötigen. In einer Pipeline mit k Segmenten ist die Berechnung nach

$$T_k = k + N - 1 \quad (4.2)$$

Schritten beendet. Die erreichbare Beschleunigung — der *Speedup* S_k — ergibt sich dann zu

$$S_k = \frac{T_1}{T_k} = \frac{k \cdot N}{k + N - 1}. \quad (4.3)$$

Dieser Ausdruck ist mit $N \gg k$ etwa gleich k , die Ausführung wird also um das k -fache beschleunigt.

Eine solche Beschleunigung ist jedoch nicht tatsächlich zu erreichen. Sobald z.B. eine Sprunganweisung ausgeführt werden muß, wird die Pipeline unterbrochen, und die Startup-Phase beginnt von neuem.

4.1.2 Vektorisierung

Arithmetisches Pipelining führt dann zu wertvollen Ergebnissen, wenn viele identische Operationen (z. B. Additionen) mit voneinander unabhängigen Daten ausgeführt werden müssen [Hoß88]. Diese Voraussetzung ist normalerweise bei Methoden der linearen Algebra und ihren Datenstrukturen, den Vektoren, erfüllt. Daher ist im Bereich der Supercomputer das Pipelineprinzip eng mit der Vektorverarbeitung verbunden. Um die Pipeline hinreichend schnell mit Operanden zu versorgen, werden als Argumente von arithmetischen Operationen nicht mehr skalare Größen, sondern ganze Vektoren (Felder) verwendet.

Moderne Vektorrechner besitzen getrennte Einheiten für die Verarbeitung von Skalaren und Vektoren. Die Vektoreinheiten bestehen dabei im wesentlichen aus den Vektorregistern, in die typischerweise Felder mit 64 oder 128 Komponenten geladen werden können, und aus den funktionalen Einheiten zu deren Verarbeitung. Diese Einheiten bestehen hauptsächlich aus Pipelines für arithmetische (Addition, Subtraktion, Multiplikation, etc.) und logische Operationen.

Die Vektorisierung eines Programmes wird heute zum großen Teil automatisch vom Compiler durchgeführt. Es ist allerdings eine sorgfältige Programmierung erforderlich, damit der Compiler den vektorisierbaren Code erkennen kann.

Eine Möglichkeit zur weiteren Beschleunigung von Vektoroperationen bietet das Chaining (Verkettung). Falls etwa eine Berechnung der Form $r * x + y$, eine sogenannte

Triade, ausgeführt werden muß, so ist es sinnvoll, mit dem Beginn der Addition nicht zu warten, bis die Multiplikation vollständig abgeschlossen ist. Statt dessen werden die Pipelines direkt hintereinander geschaltet. Dadurch liegt dann — nach den Startup-Zeiten — in jedem Takt ein Ergebnis der Doppeloperation vor.

4.2 Konzepte zur Parallelisierung

4.2.1 Grundlegende Begriffe

Das wichtigste Konzept moderner Betriebssysteme ist das *Multiprogramming* (Mehrprogrammbetrieb). Darunter versteht man die Betriebsart eines Computers, bei dem mehrere (unabhängige) Programme im Arbeitsspeicher gehalten werden, zwischen denen die CPU-Zeit geteilt wird. Dies ist auch auf einem sequentiellen Rechner möglich.

Zum *Multiprocessing* ist hingegen eine Mehrprozessorumgebung erforderlich. Ein *Prozeß* stellt — im Gegensatz zum passiven Programm — eine aktive Einheit dar. Beim Multiprocessing werden mehrere Prozesse gleichzeitig — auf verschiedenen Prozessoren — ausgeführt.

Beim *Multitasking* werden einzelne Prozesse in unabhängig voneinander auszuführende Teilaufgaben — *Tasks* — zerlegt, die dann auf verschiedenen Prozessoren bearbeitet werden. Durch Multitasking soll die Ausführungszeit eines Programmes durch parallele Verarbeitung reduziert werden, im Gegensatz zur Vektorisierung wird jedoch die zu leistende „Arbeit“ nicht verringert — es wird sogar ein gewisser Overhead für die Synchronisation der Tasks erzeugt.

Ein Nachteil von Multitasking nutzender Algorithmen ist, daß sie stark von der Hardware abhängig, also kaum zu portieren sind. Es macht zum Beispiel algorithmisch einen großen Unterschied, ob ein Problem auf einem massiv parallelen System mit Message-Passing-Kommunikation oder einem schwach parallelen Shared-Memory-Computer läuft.

Der *Speedup* gibt an, um wieviel schneller die parallele gegenüber der sequentiellen Verarbeitung ist. Er ist definiert als Quotient aus der Zeit T_1 , die für die sequentielle Ausführung eines Algorithmus benötigt wird, und der für die parallele Verarbeitung benötigten Zeit T_p :

$$S_p = \frac{T_1}{T_p} . \quad (4.4)$$

Der Speedup S_p ist dabei durch die Zahl der Prozessoren p nach oben beschränkt. Eine weitere Beschränkung des Speedup ist durch das Amdahlsche Gesetz gegeben. Da im allgemeinen ein Programm Teile enthält, die sequentiell ausgeführt werden

$f[\%]$	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=\infty$
100	1.00	2.00	4.00	8.00	16.00	32.00	64.00	∞
99	1.00	1.98	3.88	7.48	13.91	24.43	39.26	100.00
98	1.00	1.96	3.77	7.02	12.31	19.75	28.32	50.00
97	1.00	1.94	3.67	6.61	11.03	16.58	22.14	33.33
96	1.00	1.92	3.57	6.26	10.00	14.29	18.18	25.00
95	1.00	1.90	3.48	5.93	9.14	12.55	15.42	20.00
94	1.00	1.89	3.39	5.63	8.42	11.19	13.39	16.67
93	1.00	1.87	3.31	5.37	7.80	10.09	11.83	14.28
92	1.00	1.85	3.23	5.13	7.27	9.19	10.60	12.50
91	1.00	1.83	3.15	4.91	6.81	8.44	9.59	11.11
90	1.00	1.82	3.08	4.71	6.40	7.80	8.77	10.00

Tabelle 4.1: Speedup nach Amdahls Gesetz für parallele Prozesse mit einem Parallelisierungsgrad f zwischen 90% und 100%

müssen, ist eine vollständige Parallelisierung meist nicht möglich. Mit dem Parallelisierungsgrad f , der den Anteil des Programms angibt, der parallel ausgeführt werden kann, lautet das Gesetz von Amdahl

$$S_p \leq \frac{1}{(1-f) + \frac{f}{p}} \leq \frac{1}{1-f}. \quad (4.5)$$

Aus Tabelle 4.1 ist zu ersehen, welche starken Auswirkungen schon ein gering scheinender nichtparallelisierbarer Anteil hat.

Ein weiterer Aspekt ist die *Effizienz*. Diese gibt das Verhältnis des erreichten Speedup zur Zahl der Prozessoren an:

$$E_p = \frac{S_p}{p}. \quad (4.6)$$

Der optimale Wert wäre hier $E_p = 1$, der bei optimalem Speedup von p erreicht würde.

4.2.2 Multitasking auf CRAY-Rechnern

Es kann zwischen fünf verschiedenen Ebenen von Parallelismus unterschieden werden:

1. Parallelverarbeitung von Jobs
2. Parallelverarbeitung auf Jobschritt-Ebene
3. Parallele Verarbeitung von Unterprogrammen
4. Parallelisierung von Schleifen
5. Parallelisierung von Anweisungen.

Die höchste Ebene entspricht dabei dem Multiprocessing. Auf der zweiten Ebene laufen verschiedene Schritte eines Jobs parallel ab; zum Beispiel können Teile eines Programmes parallel compiliert werden, bevor sie zu einem ausführbaren Modul verbunden werden. Parallelismus auf diesen beiden Ebenen wird vor allem genutzt, um den Durchsatz einer Maschine zu erhöhen. Die unterste Ebene, in der zum Beispiel eine Gleitpunkt- und eine Ein-/Ausgabeoperation gleichzeitig durchgeführt werden, wird häufig in der Hardware implementiert. Von Interesse sind also hauptsächlich die Ebenen 3 und 4 [ZM88].

Dazu stellte die Firma CRAY zunächst mit der Einführung der Mehrprozessorfamilie X-MP das Macrotasking-Konzept zur Verfügung, welches es erlaubt, verschiedene Unterprogramme eines Programmes parallel auszuführen. Wenige Jahre später wurde das Konzept Microtasking eingeführt, welches auch die Parallelisierung von Schleifen und einzelnen Anweisungen erlaubt. Das Microtasking wurde inzwischen vom Autotasking abgelöst, das den Parallelismus innerhalb eines Unterprogrammes automatisch erkennen und ausnutzen kann. Es wird festgestellt, ob verschiedene Teile eines Unterprogramms datenunabhängig sind; wenn dies der Fall ist, können zum Beispiel unterschiedliche Iterationen einer Schleife gleichzeitig ausgeführt werden.

4.2.3 Macrotasking

Beim Macrotasking [Kne86, Cra89] hat der Benutzer die Möglichkeit, komplette Unterprogramme parallel ausführen zu lassen. Um ein Unterprogramm als parallel ausführbar zu kennzeichnen, muß er einen Aufruf der Bibliotheksroutine TSK-START in den Quellcode seines Programmes einfügen.

Auch die Synchronisation der Tasks erfolgt durch den Aufruf von Bibliotheksroutinen. Durch Barrieren kann eine festgelegte Anzahl von Tasks an einer bestimmten

Programmstelle synchronisiert werden. Eine weitere Möglichkeit zur Synchronisation sind Ereignisse (*Events*), kritische Regionen können durch *Locks* überwacht werden. Für eine genauere Beschreibung der Verwendung der dazu erforderlichen Routinen sei auf [Kne86] verwiesen.

Da der Overhead relativ groß ist, der zum einen durch die Aufrufe der Bibliotheks-routinen, zum anderen durch die Kommunikation über gemeinsame Variablen im Hauptspeicher entsteht, ist eine effiziente Nutzung der Multitaskingfähigkeiten nur bei relativ grober Granularität der Algorithmen möglich. Dieser Zwang zu grober Granulierung kann zu schlechter Lastverteilung führen, was den Speedup entscheidend senken kann. Insbesondere kann sich dies bei nicht-dediziertem Betrieb auswirken, wenn die Anzahl der zu Verfügung stehenden Prozessoren ständig schwankt [Bur89].

4.2.4 Microtasking

Den nächsten Entwicklungsschritt im Multitasking auf CRAY-Rechnern stellt das Microtasking dar [Cra89, HKN89]. Es bietet die Möglichkeit zur Parallelverarbeitung bis hinunter zur Ebene von Schleifen und Anweisungen. Die Kommunikation erfolgt hier über Register, wodurch der Overhead entscheidend reduziert ist.

Der Benutzer fügt Direktiven in den Quellcode seines Programmes ein, mit denen er Kontrollstrukturen kennzeichnet (z. B. DO-Loops), die parallel ausführbare Programmteile enthalten. Diese werden durch den Präprozessor PREMULTE verarbeitet, der die Konstrukte in Bibliotheks-routinenaufrufe umsetzt. Dabei werden für jedes Unterprogramm, das Microtasking nutzen will, drei getrennte Unterprogramme erzeugt: das Assembler-Unterprogramm MASTER und zwei FORTRAN-Unterprogramme, die den Code für den sequentiellen beziehungsweise parallelen Ablauf enthalten.

Für folgende Aufgaben stehen dem Programmierer Direktiven zur Verfügung [Lie88]:

- Anforderung und Rückgabe von physikalischen CPUs;
- Kennzeichnung von Unterprogrammen, die Microtasking nutzen;
- Definition von Kontrollstrukturen;

- Sicherung kritischer Bereiche.

Als parallelisierbare Kontrollstrukturen kommen zum einen DO-Schleifen in Frage, deren Iterationen voneinander unabhängig sind und somit gleichzeitig ausgeführt werden können. Solche Schleifen werden durch die DO GLOBAL-Direktive gekennzeichnet. Zum anderen kann der Benutzer auch beliebige Code-Sequenzen als Prozesse definieren, diese werden durch die Direktiven PROCESS, ALSO PROCESS und END PROCESS markiert. Damit ist es etwa möglich, daß mehrere verschiedene Schleifen parallel ausgeführt werden. Eine ausführliche Beschreibung der Microtasking-Direktiven und ihrer Funktionsweise findet man etwa in [Lie88].

4.2.5 Autotasking

Den vorläufig letzten Schritt in der Entwicklung des CRAY-Multitasking bildet das Autotasking [Cra90]. Während der Programmierer bisher entweder von Hand das Programm umcodieren (Macrotasking) oder Compiler-Direktiven einfügen mußte (Microtasking), kann Autotasking vollautomatisch ablaufen. Es kann jedoch jederzeit auch hier vom Programmierer eingegriffen werden; neben Autotasking-spezifischen Konzepten können dabei sowohl Macro- als auch Microtasking weiterhin eingesetzt werden.

Das CF77-Compiling-System besteht aus drei Stufen:

1. der Präprozessor *FPP* analysiert Datenabhängigkeiten,
2. *FMP* ersetzt den Microtasking-Präprozessor *PREMULT*
3. der eigentliche Compiler *CFT77* erzeugt den Objectcode, der dann vom Lader *SEGLDR* in ein ausführbares Programm umgesetzt wird.

Jede der Stufen kann auch einzeln aufgerufen werden. So kann etwa zunächst durch den *FPP* ein Programm analysiert werden, das man dann selbst nochmals optimiert, bevor es durch *FMP* und *CFT77* übersetzt wird. Andererseits ist es auch möglich, *FPP* oder *FMP* auszulassen. *FPP* in Verbindung mit *CFT77* hat etwa den Effekt, daß nicht parallelisiert wird, aber möglicherweise eine deutliche Beschleunigung durch bessere Vektorisierung erzielt werden kann.

4.2.5.1 FPP: Datenabhängigkeitsanalyse

Die Datenabhängigkeitsanalyse durch *FPP* ist die erste der drei Stufen des Compiling-Systems. Dabei werden zum einen Autotasking-Direktiven (*CMIC@ ...*) für den *FMP* zum anderen die Direktive *CDIR@ IVDEP* für den Compiler eingefügt, um diesem die Vektorisierbarkeit der folgenden Schleife anzuzeigen. Weiterhin werden auch Umstrukturierungen mit dem Ziel der besseren Vektorisierbarkeit oder Parallelisierbarkeit vorgenommen, die durchaus erhebliches Ausmaß annehmen können. So werden zum Beispiel innere und äußere Schleifen vertauscht, Fallunterscheidungen eingeführt, falls die Schleifenlänge nicht bekannt und je nach Schleifenlänge Vektor- bzw. Skalarverarbeitung effizienter ist, oder Matrixoperationen in optimierte Bibliotheksroutrinenaufrufe umgesetzt. *FPP* kann auch erkennen, ob aufeinanderfolgende Schleifen voneinander datenunabhängig sind und gegebenenfalls gleichzeitig ausgeführt werden können. Vom Benutzer durch *FMP*-Direktiven gekennzeichnete parallele Regionen werden durch *FPP* jedoch nicht mehr optimiert.

Ausgegeben wird wiederum ein FORTRAN-Programm, das — wenn nicht per Compiler-Option anders angegeben — in der Datei *name.m* steht. Hilfreich für eine weitere Optimierung von Hand ist auch das zusätzliche Listing, das optional erzeugt werden kann. Darin wird zu den einzelnen Schleifen angegeben, wie sie behandelt werden und gegebenenfalls auf Grund welcher (scheinbaren) Abhängigkeit keine Vektorisierung oder Parallelisierung durchgeführt werden konnte.

Der *FPP* wird durch Direktiven der Form

CFPP\$ directive scope

gesteuert. Der Direktive folgt also der (optionale) scope-Parameter, der den Gültigkeitsbereich der Direktive angibt. Folgende Werte sind dabei zugelassen:

R : Rest des aktuellen Unterprogramms

L : nächste Schleife

F : Rest der aktuellen Datei

I : an dieser Stelle des Programms

'blank' : wie L

Im folgenden werden die zur Verfügung stehenden Direktiven kurz beschrieben (falls zu einer Direktive ein Default-Wert existiert, so ist dieser unterstrichen). Die erste Gruppe enthält die Direktiven, mit denen die Codetransformation durch *FPP* beeinflußt werden kann:

<u>{NO}</u> VECTOR	<u>Untersuche</u> {nicht} auf Vektorisierungsmöglichkeit.
<u>{NO}</u> CONCUR	<u>Untersuche</u> {nicht} auf Parallelisierungsmöglichkeit.
SKIP	Untersuche weder auf Vektorisierungs- noch auf Parallelisierungsmöglichkeit.
<u>{NO}</u> INNER	Erlaube { <u>keine</u> } Parallelisierung innerer Schleifen.
CNCALL	Erlaube die Parallelisierung von Schleifen, in denen eine CALL-Anweisung auftritt. (Es ist sicherzustellen, daß das aufgerufene Unterprogramm keine Seiteneffekte bewirkt.)
CHOP_HERE	Spalte eine Schleife (an dieser Stelle) in zwei Schleifen auf.
<u>{NO}</u> ALTCODE	<u>Erlaubt</u> {nicht} die Generierung alternativer Codeblöcke, falls Vektor- oder Parallelverarbeitung unter bestimmten Bedingungen möglich ist.
<u>{NO}</u> ASSOC	<u>Erlaubt</u> {keine} assoziativen Transformationen. (Achtung bei numerisch kritischen Operationen!)
<u>{NO}</u> SPLIT	Gibt an, daß ein Unterprogramm { <u>nicht</u> } seiteneffektfrei ist und aus einer optimierten in eine separate Schleife überführt werden kann.
SELECT	Wählt eine Schleife aus, die parallelisiert werden soll.
<u>{NO}</u> LSTVAL	Gibt an, daß der letzte Wert eines transformierten Skalars {nicht} <u>benötigt</u> wird.
<u>{NO}</u> UNROLL	Erlaubt { <u>kein</u> } automatisches <i>Unrolling</i> . Unrolling einer inneren Schleife kann zu besserer Vektorisierung führen.

Eine weitere Gruppe von Direktiven hilft dem Präprozessor, Datenabhängigkeiten zu finden oder scheinbare Abhängigkeiten zu ignorieren:

<code>{NO}DEPCHK</code>	<u>Führe</u> Datenabhängigkeitsanalyse {nicht} <u>durch</u> . Ein Ausschalten der Analyse ist sinnvoll, wenn der Benutzer weiß, daß eine von <i>FPP</i> vermutete Abhängigkeit nicht gegeben ist.
<code>{NO}SYNC</code>	<u>Ignoriere</u> {nicht} potentiell ^{es} Überlappen von Arraybereichen bei paralleler Ausführung. (Verallgemeinerung von {NO} DEPCHK auf Parallelverarbeitung).
<code>{NO}EQVCHK</code>	<u>Untersuche</u> {nicht}, ob durch EQUIVALENCE-Anweisungen Datenabhängigkeiten hervorgerufen werden.
PERMUTATION	Deklariert Integer-Arrays als Permutationen, dadurch kann Parallelverarbeitung ermöglicht werden.
RELATION	Gibt <i>FPP</i> die Information, daß zwischen zwei Integervariablen bzw. einer Variablen und einer Konstante ein bestimmter Zusammenhang besteht (z. B. J.GE.N).

Mit den Direktiven COUNT und ITERATIONS kann die Zahl der Iterationen einer Schleife beziehungsweise einer Klasse von Schleifen bekanntgemacht werden. Weiterhin gibt es noch Direktiven, die die 'Inline'-Übersetzung von Unterprogrammen bewirken und steuern.

4.2.5.2 FMP: Transformation für Multitasking

Der vom *FPP* generierte Code wird dann durch den zweiten Präprozessor, *FMP* weiterverarbeitet. Dabei werden die Auto- (bzw. Micro-)tasking-Direktiven in maschinenspezifische Bibliotheksaufrufe umgesetzt. Ergebnis ist wiederum ein FORTRAN-Programm, das dann vom Compiler übersetzt werden muß.

Direktiven für *FMP* haben folgende Form:

CMICS GENERIC_DIREKTIVE *Parameter*
 CMICS**Parameter_fortführung*

(Vom *FPP* eingefügte Direktiven beginnen mit CMIC@, im Gegensatz zu den CMICS-Direktiven kann die Beachtung dieser nicht per Compiler-Option unterdrückt werden.) Einige der Direktiven sind im Vergleich zum Microtasking unverändert geblieben, andere haben im Grunde die gleiche Bedeutung (CASE — PROCESS, DO

PARALLEL — DO GLOBAL, SOFT EXIT — STOP ALL PROCESS). Im Unterschied zum Microtasking wird beim Autotasking jedoch nicht ein ganzes Unterprogramm zur parallelen Region, sondern nur der Bereich, der vom Benutzer (oder durch *FPP*) spezifiziert wurde. Dies kann im üblichen Mehrbenutzerbetrieb zur Verringerung von Leerlaufzeiten, also zu einem erhöhten Durchsatz führen.

Parallele Regionen können auf verschiedene Art und Weise bestimmt werden:

DO ALL	Die auf die Direktive folgende DO-Schleife kann parallel ausgeführt werden. Es wird eine parallele Region initiiert, die nur die Iterationen der Schleife umfaßt. Beginn (beziehungsweise Ende) einer parallelen Region.
PARALLEL	Markiert den Beginn einer parallelen Region.
END PARALLEL	Ende einer durch PARALLEL begonnenen Region
DO PARALLEL	Die auf die Direktive folgende DO-Schleife kann parallel ausgeführt werden (nur innerhalb einer durch PARALLEL / END PARALLEL markierten Region erlaubt). Mit der folgenden Direktive
END DO	kann die Schleife erweitert werden, um Mechanismen für die Parallelverarbeitung einzubauen.
CASE	Diese Direktive dient als Separator zwischen Codeabschnitten, die von verschiedenen Prozessoren ausgeführt werden können. Mit
END CASE	wird eine Gruppe solcher Abschnitte beendet.

Kritische Bereiche, in denen wechselseitiger Ausschluß nötig ist, werden in GUARD und END GUARD eingeschlossen. Mit CONTINUE wird *FMP* angezeigt, daß ein aufzurufendes Unterprogramm mit Microtasking parallelisiert worden ist. SOFT EXIT veranlaßt den Abbruch der parallelen Ausführung. Dies ist dann erforderlich, wenn durch eine Sprunganweisung eine parallele Region verlassen wird.

Zu den Direktiven, die parallele Bereiche spezifizieren, gibt es eine Reihe von optionalen Parametern:

IF(<i>expr</i>)	Zur Laufzeit wird getestet, ob die Bedingung <i>expr</i> erfüllt ist. Falls dies nicht der Fall ist, wird keine Parallelverarbeitung durchgeführt.
SHARED(<i>var1, var2, ...</i>)	Die in der Liste aufgeführten Variablen sind gemeinsame Variablen, die für jede Task zugreifbar sind.
PRIVATE(<i>var1, var2, ...</i>)	Die hier aufgeführten Variablen sind privat, das heißt jede Task hat eine eigene Kopie dieser Variablen. Dies ist zum Beispiel immer bei der Zählvariablen einer parallelierten Schleife der Fall.
AUTOSCOPE	Die Gültigkeitsbereiche der Variablen, die nicht durch SHARED/PRIVATE bestimmt worden sind, werden automatisch bestimmt. (Diese automatische Bestimmung kann auch falsche Ergebnisse liefern!)
CONTROL(<i>var1, var2, ...</i>)	Die hier aufgelisteten Variablen sind Kontrollvariablen für AUTOSCOPE. Mit solchen Variablen indizierte Felder sind gemeinsame (SHARED) Felder.
SAVELAST	Der Wert, den eine private Variable in der letzten Iteration hat, bleibt erhalten.
MAXCPUS(<i>val</i>)	Es werden maximal <i>val</i> CPUS zur Ausführung der parallelen Region benötigt.

Mit den übrigen Parametern kann die Verteilung der Arbeitslast auf die Prozessoren beeinflusst werden. Nur jeweils einer der folgenden Parameter ist pro Direktive erlaubt (Default: SINGLE).

SINGLE	Die Iterationen werden einzeln von den zur Verfügung stehenden Prozessoren abgearbeitet.
CHUNKSIZE(<i>n</i>)	Die Schleife wird in Blöcke der Größe <i>n</i> aufgeteilt, etwa um in langen Schleifen sowohl Vektorisierung als auch Parallelisierung zu ermöglichen. (<i>n</i> sollte eine Integerkonstante sein!)

NUMCHUNKS(n)	Die Schleife wird in n Blöcke aufgeteilt.
GUIDED	“Guided Self Scheduling” benutzt werden; dieser Mechanismus soll gleichzeitig Synchronisationsoverhead minimieren und die Arbeitslast gut verteilen.
VECTOR	Spezifiziert bestimmten Scheduling-Algorithmus; impliziert GUIDED Blöcke mit minimaler Länge 64.

Eine genauere Beschreibung der Direktiven und Parameter sowie ihrer Benutzung findet sich in Band 4 des *CF77-Systemhandbuchs* [Cra90].

4.3 Parallelisierung einzelner Verfahren

Prinzipiell gibt es zwei verschiedene Ansatzpunkte zur Parallelisierung von Optimierungsverfahren. Zum einen kann man die Struktur der Probleme (bzw. Problemklassen) ausnutzen. Dies führt zur Zerlegung in Teilprobleme, die zum Beispiel bei separablen Problemen möglich ist. Zur globalen Optimierung kann es auch sinnvoll sein, verschiedene lokale Optimalstellen gleichzeitig zu untersuchen. Zum anderen kann man im Optimierungsverfahren selbst Ansätze zur Parallelisierung finden. So ist es etwa möglich, Funktionen parallel auszuwerten oder verschiedene Komponenten des Gradientenvektors gleichzeitig zu berechnen.

Schnabel [Sch85] merkt an, daß Vektorcomputer geeigneter für partielle Differentialgleichungen und Anwendungen in der linearen Algebra als für die Optimierung sind. Dies wird sicherlich durch die Beobachtung unterstützt, daß auf einem sequentiellen Rechner die Funktions- und Gradientenauswertung bei der BFGS-Methode über 90% der Kosten ausmachen [BSS88]. Dies läßt sich zwar möglicherweise nicht ohne weiteres auf andere Methoden übertragen; da jedoch auch bei der Optimierung mit Nebenbedingungen auf die Verfahren zur unbeschränkten Minimierung zurückgegriffen wird, wird sich, je nach verwendetem Verfahren, ein ähnliches Verhalten wahrscheinlich auch dort beobachten lassen.

Andererseits ist die Vektorisierung sowohl von der Hard- als auch von der Softwareseite her sehr ausgereift und gut unterstützt. Bei den Teilaufgaben der Optimierung, die aus linear-algebraischen Operationen bestehen, wird die Vektorverarbeitung also sicherlich signifikanten Nutzen bringen. Dabei müssen allerdings auch die Wechselwirkungen zwischen Vektorisierung und Parallelisierung beachtet werden. Parallelisierung auf Kosten der Vektorlänge kann sich sehr nachteilig auf die erzielbare Leistung auswirken.

Bei der Implementierung ist weiter zu beachten, daß Supercomputer üblicherweise im Mehrbenutzerbetrieb betrieben werden, also zur Lösung von Problemen normalerweise nicht dediziert zur Verfügung stehen. Der Benutzer weiß also im allgemeinen nicht im voraus, wie viele CPUs er tatsächlich zur Verfügung hat. Bei den feingranularen Strukturen, die durch Micro- bzw. Autotasking erzeugt werden, erweist sich dies allerdings nicht als Problem; hier können sehr flexibel die jeweils freien Prozessoren genutzt werden.

Eine allgemeine Aussage, welcher Ansatz zur Parallelisierung den größten Erfolg verspricht, kann aufgrund der Verschiedenartigkeit der Problemstellungen sicherlich nicht getroffen werden. Etwa bei Problemen mit einer großen Anzahl an Variablen, aber mit einfachen (möglicherweise linearen) Nebenbedingungen, liegt die Vektorisierung nahe. Andererseits verspricht sie deutlich weniger Erfolg, wenn die Komplexität der Aufgabenstellung nicht aus der Variablenanzahl, sondern aus der Art der Ziel- und Nebenbedingungsfunktionen herrührt.

Einige Verfahren, wie etwa das Verfahren der projizierten Gradienten sind dann relativ einfach parallelisierbar, wenn die Nebenbedingungen einen „Quader“ definieren [BT88]. Auf solche Spezialfälle soll hier aber nicht weiter eingegangen werden, im folgenden wird die Parallelisierung zweier allgemein verwendbarer Bibliotheksroutinen zur Minimierung mit und ohne Nebenbedingungen beschrieben.

4.3.1 Hilfsmittel

Zur Parallelisierung vorhandener sequentieller Programme ist es sinnvoll, diese zunächst auf ihr Laufzeitverhalten hin zu untersuchen. Dazu kann im Forschungszentrum Jülich (KFA) zum einen LOOPTOOL, zum anderen das CRAY-Produkt Flowtrace/Flowview verwendet werden.

LOPTOOL wertet dazu in einem Testlauf die Anzahl der arithmetischen Operationen aus, die in den DO-Schleifen des Programmes ausgeführt werden. Zu jeder Programmeinheit wird angegeben, wie viele solcher Operationen ausgeführt wurden und welchen Anteil am gesamten Programm sie ausmachten. Zu jeder Schleife wird die Anzahl der Iterationen sowie die Vektorlänge, die Schachtelungstiefe und die Anzahl der arithmetischen Operationen ausgegeben.

Die daraus möglichen Rückschlüsse sind jedoch zur Laufzeitanalyse nicht ausreichend. Eine weitere Untersuchung mittels Flowtrace, deren Ergebnisse dann mittels Flowview verarbeitet werden, ist daher sinnvoll.

Neben generellen Informationen zum Programm und zum Rechner liefert Flowtrace Informationen zum Verhalten der einzelnen Unterprogramme. Zu jedem Unterprogramm wird die Häufigkeit des Aufrufs, die insgesamt und die durchschnittlich pro Aufruf verbrauchte Zeit sowie der prozentuale Anteil an der Gesamtzeit angegeben. Zu den einzelnen Schleifen gibt es jedoch keine Informationen.

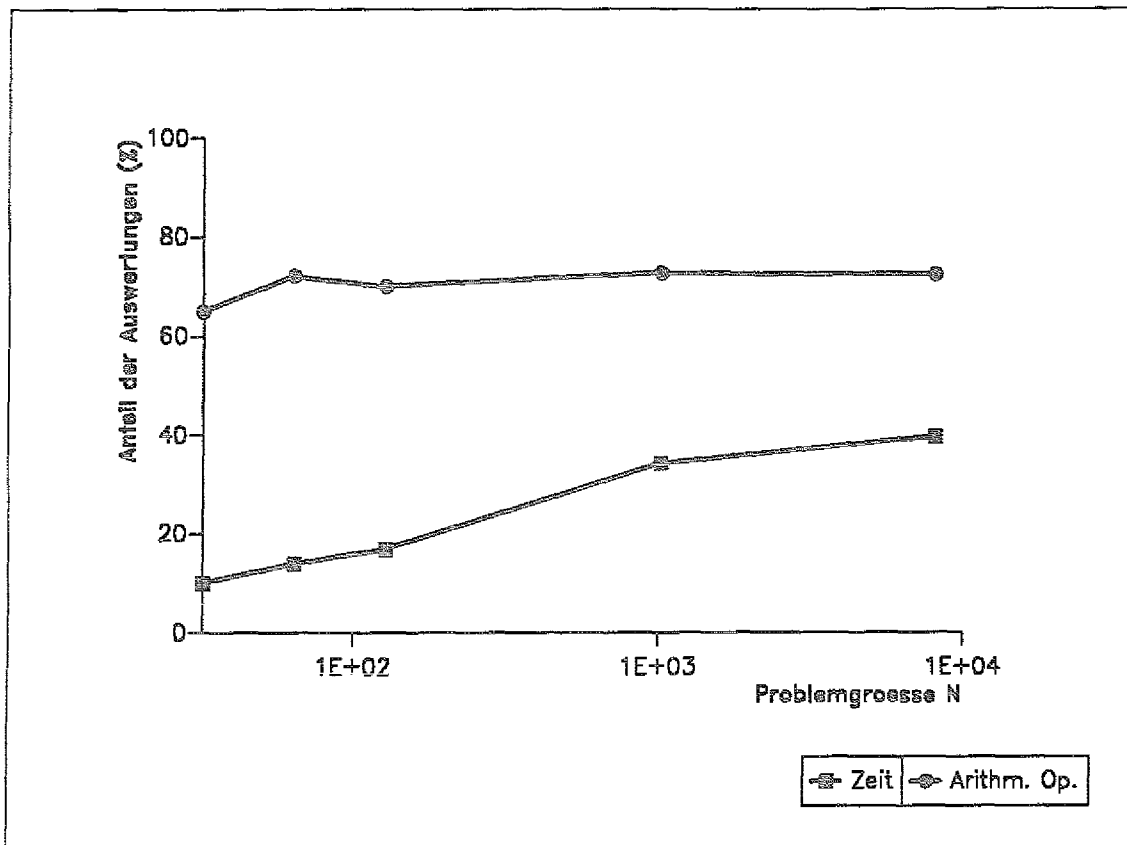


Abbildung 4.4: Zeitanteil der Funktions- und Gradientenauswertungen, Testproblem 2, BFGS

Abbildung 4.4 zeigt anhand eines Testproblems, wie weit die Ergebnisse von LOOP-TOOL und Flowtrace auseinander liegen können. Während der Anteil der Auswertungen der Funktion und ihrer Gradienten an den arithmetischen Operationen nahezu konstant bei etwa 70% liegt, ist der Zeitanteil wesentlich geringer. Letzterer nähert sich bei wachsender Problemgröße nur langsam dem Anteil der Operationen an.

4.3.2 Unbeschränkte Optimierung: BFGS

Zur unbeschränkten Optimierung wurde das Quasi-Newton-Verfahren nach Broyden, Fletcher, Goldfarb und Shanno (BFGS) untersucht. Dieses ist im Unterprogramm VA15A in der Harwell Subroutine Library (HSL) implementiert (vergleiche Anhang A) [HSL90]. Da die Unterprogramme der HSL im Quelltext vorliegen, ist es möglich, diese mittels LOOPTOOL und Flowtrace/Flowview zu untersuchen. Da-

zu wurden drei Testprobleme entwickelt, deren Problemgröße veränderbar ist (siehe Anhang B). Zum Vergleich wurden alle Messungen auch durchgeführt mit der erweiterten Rosenbrock-Funktion, die in der Literatur als Testproblem weit verbreitet ist.

Der Benutzer muss die Prozedur VA15A in jedem Iterationsschritt neu aufrufen und dabei die aktuellen Funktions- und Gradientenwerte übergeben. VA15A führt dann die Hauptiteration durch, berechnet die Suchrichtung durch das Produkt $-H_k \nabla f$ und überprüft, ob das Verfahren terminiert. Intern werden dabei folgende Unterprogramme aufgerufen:

- VA15B druckt im vom Benutzer gewählten Umfang Informationen aus,
- VA15E bestimmt die Schrittweite,
- VA15F führt eine Liniensuche durch.

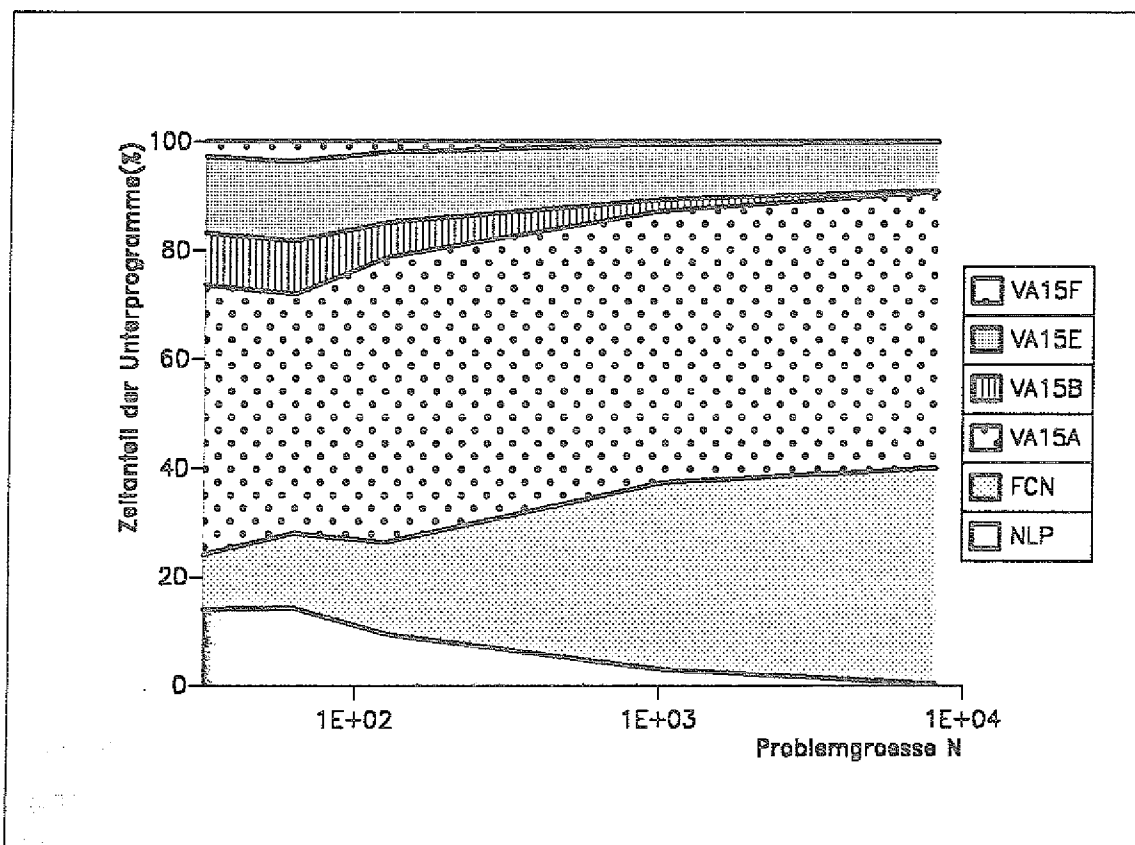


Abbildung 4.5: Anteil der Unterprogramme an der Ausführungszeit, Testproblem 2

Zunächst wurde das Verhalten für den Fall untersucht, daß die Gradienten analytisch berechnet wurden und ein entsprechendes Unterprogramm zur Verfügung steht. Abbildung 4.5 zeigt, welchen Anteil an der Ausführungszeit bei Testproblem 2 die einzelnen Routinen mit steigender Problemgröße beanspruchen. Während bei kleiner Problemgröße der Anteil von NLP, dem aufrufenden Programm, in dem Initialisierungen vorgenommen werden und das Endergebnis ausgedruckt wird, sowie der Druckroutine VA15B nicht unerheblich sind, wird deren Anteil mit zunehmender Problemgröße vernachlässigbar gering. Dafür steigt der Anteil der Funktions- und Gradientenauswertungen (durch die Prozedur FCN) kontinuierlich an, während die Anteile von VA15A und VA15E nur langsam geringer werden.

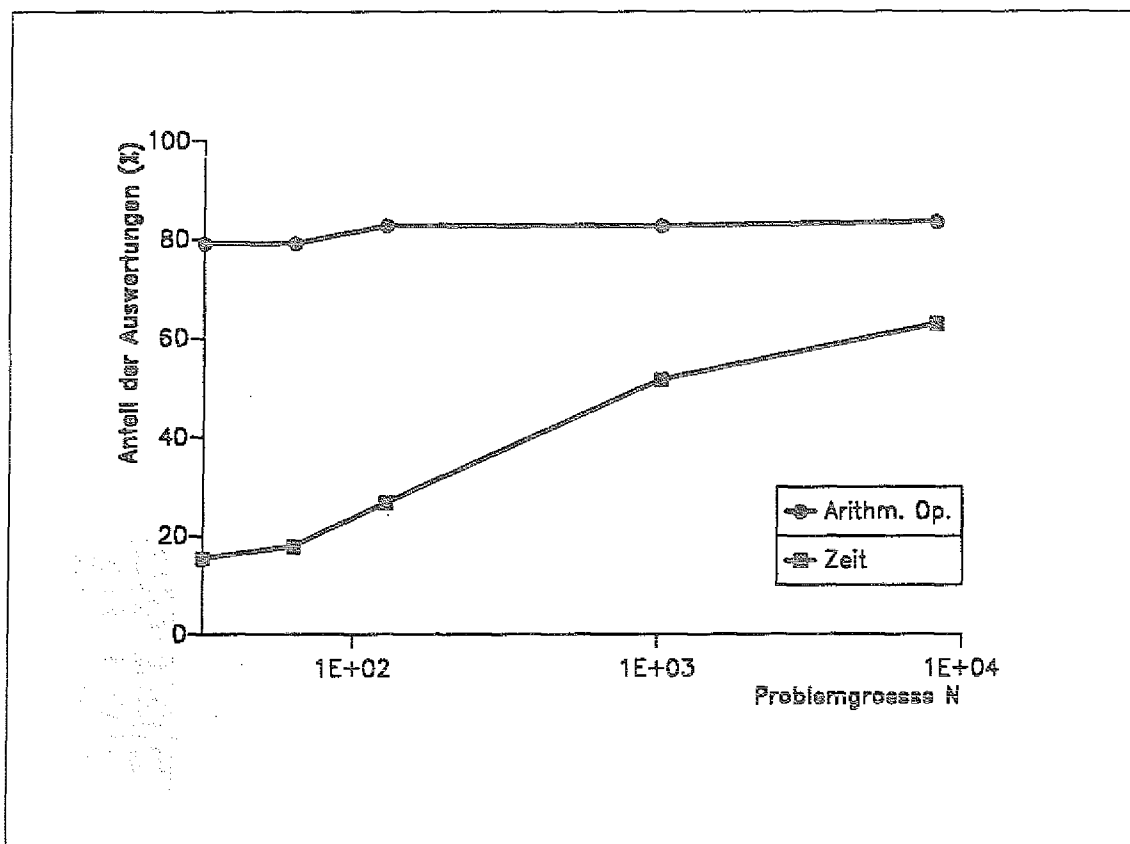


Abbildung 4.6: Zeitanteil der Funktions- und Gradientenauswertungen, Testproblem 3, BFGS

Bei Testproblem 3, mit einer komplexeren Zielfunktion, steigt der Anteil der Funktions- und Gradientenauswertung erheblich schneller an (Abbildung 4.6), während gleichzeitig der Anteil von VA15A und VA15E deutlicher zurückgeht. Bei $n = 1024$ wird hier bereits mehr als die Hälfte der Zeit für die Auswertungen verbraucht, mit steigender Tendenz. Der Anteil an den arithmetischen Operationen liegt sogar über

80%. Im Gegensatz dazu nehmen die Auswertungen mit dem einfacheren Testproblem 1 und der Rosenbrock-Funktion einen erheblich geringeren Anteil ein, wobei auch die insgesamt benötigte Zeit geringer ist.

Zum weiteren Vergleich wurden diese Messungen auch mit der Routine VA14A aus der HSL durchgeführt, die mit konjugierten Richtungen arbeitet. Die Verwendung dieses Verfahrens ist insbesondere für sehr große Probleme angezeigt, da es weniger Speicherplatz benötigt. VA14A terminiert mit den angegebenen Testproblemen schneller als VA15A, andererseits sind die Ergebnisse aber auch weniger genau. In

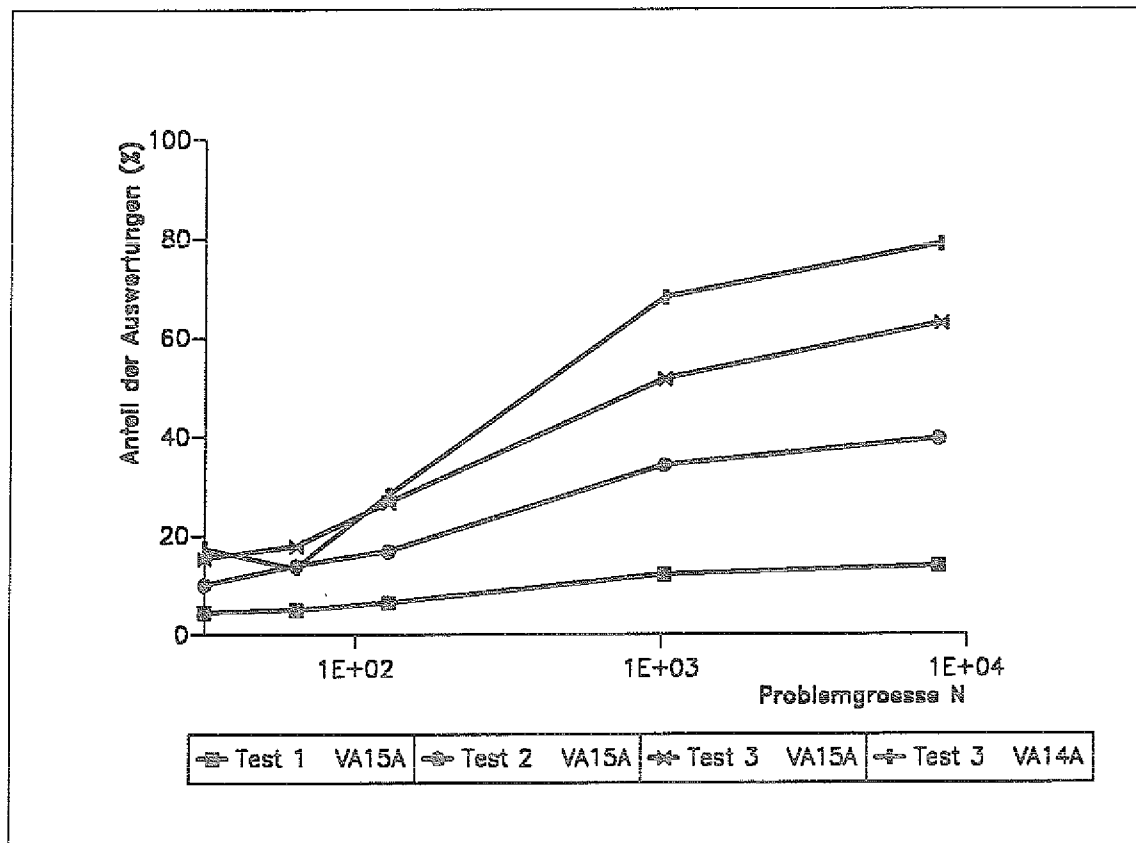


Abbildung 4.7: Zeitanteil der Funktions- und Gradientenauswertungen

Abbildung 4.7 sind die Zeitanteile mit den verschiedenen Testproblemen vergleichend aufgetragen. Hier zeigt sich nochmals, daß mit steigender Komplexität der Zielfunktion der Anteil der Auswertungen zunimmt. Im Verfahren der konjugierten Richtungen nehmen sie sogar noch stärkeren Raum ein als bei BFGS.

Insgesamt läßt sich aus diesen Ergebnissen der Schluß ziehen, daß es sinnlos ist, allein die Bibliotheksroutinen zu parallelisieren. Nach Amdahls Gesetz läßt sich dadurch kein akzeptabler Speedup — und somit erst recht keine gute Auslastung mehrerer

Prozessoren — erreichen. Für die Parallelisierung nur der Funktions- und Gradientenauswertung läßt sich bei den gegebenen Testproblemen der gleiche Schluß ziehen.

In VA15A sind die Vektor- und Matrixoperationen zum großen Teil durch Aufrufe von BLAS-Routinen (BLAS = Basic Linear Algebra Subprograms [DDHH88, DDHD90]) realisiert. Diese liegen auf den CRAY-Rechnern als Bestandteil der SCILIB (SCientific Applications LIBrary [Cra91]) bereits in parallelisierter Form vor. Der Einsatz der BLAS-1 Routinen wirkt sich allerdings eher negativ aus. Der CRAY-FORTRAN-Compiler würde für die Vektor-Vektor-Operationen besser optimierte Programmteile erzeugen, der durch die Unterprogrammaufrufe erzeugte Overhead würde wegfallen.

Das Bereitstellen der analytischen Gradienten wird jedoch bei zunehmender Problemkomplexität immer schwieriger und ist sehr fehlerträchtig. Es ist daher üblich, Gradienten automatisch zu erzeugen. Hier gibt es zum einen Ansätze zur symbolischen Differentiation (Maple, Reduce, Macsyma, Mathematica) beziehungsweise zur Erzeugung von Ableitungs-Code aus FORTRAN-Unterprogrammen (ADIFOR) [BCC⁺92]. Zum anderen kann man den Gradienten durch die Berechnung von Differenzenquotienten approximieren:

$$\left. \frac{\partial f(x)}{\partial x_i} \right|_{x=x_0} \approx \frac{f(x_0 + he_i) - f(x_0 - he_i)}{2h},$$

wobei e_i der i -te Einheitsvektor sei. Auch einseitige Differenzen sind möglich. Der größte Nachteil dieser Methode ist die Schwierigkeit, eine angemessene Schrittweite h zu finden. Ein zu groß gewähltes h kann zu ungenauen Ergebnissen führen, aber auch ein sehr kleines h kann durch Rundungsfehler problematisch werden.

Zu Testzwecken erscheint es jedoch angemessen, Differenzenquotienten zu verwenden, die auch in der Praxis üblicherweise angewandt werden. Weiterhin ist durch andere automatische Methoden im allgemeinen keine Verringerung des Aufwands zur Gradientenberechnung um Größenordnungen zu erwarten.

Abbildung 4.8 zeigt, daß schon bei relativ kleinen Problemgrößen der Zeitaufwand für Zielfunktion und Gradienten mittels Differenzenquotienten überwiegt. Die Auswertung der Funktionen macht bis über 99% der insgesamt zur Minimierung benötigten Zeit aus. Dies geht einher mit einem ganz erheblichen Anstieg der erforderlichen Zeit. Dieser Mehraufwand ist jedoch sicher nicht untypisch für reale Anwendungen.

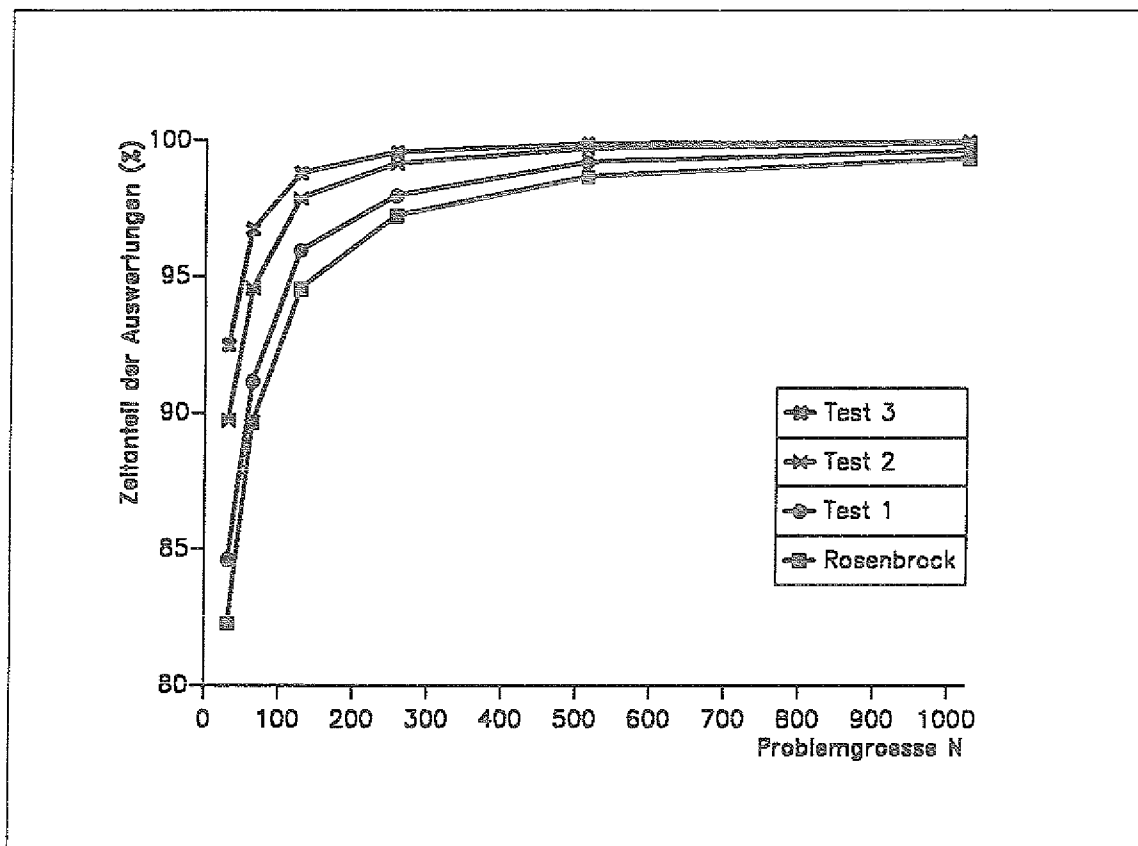


Abbildung 4.8: Zeitanteil der Funktions- und Gradientenauswertungen mit Differenzenquotienten

Zur Parallelisierung bieten sich nun zwei grundsätzliche Alternativen an:

- Parallelisierung in der Funktionsauswertung.
- Parallele Auswertung der einzelnen Komponenten des Gradienten und der Zielfunktion.

Die erste Möglichkeit ist jedoch sehr problemabhängig, daher wurde hier zur Implementierung die zweite Variante gewählt. Diese ist zum einen recht einfach zu realisieren, zum anderen ist sie — unabhängig vom gegebenen Problem — praktisch immer möglich.

Die Hauptiteration der Prozedur zur Berechnung des Gradienten an gegebener Stelle x_0 hat dann folgendes Aussehen:

```

CMIC$ DO ALL SHARED(N, X, G, H) PRIVATE(I, J, FPH, FMH, Y)
  DO 20, I=1, N
    DO 10, J=1, N
      Y(J) = X(J)
10  CONTINUE
      Y(I) = Y(I) + H
      CALL FCN(N, Y, FPH)
      Y(I) = Y(I) - 2.0*H
      CALL FCN(N, Y, FMH)
      G(I) = (FPH - FMH) / (2.0*H)
      Y(I) = Y(I) + H
20 CONTINUE

```

X ist dabei ein Feld der Länge N, das den Punkt x_0 enthält. Aus diesem Feld wird nur gelesen, so daß es als SHARED-Variable deklariert werden kann. G ist ebenfalls ein Feld mit Länge N, es nimmt die Werte der Komponenten des Gradienten auf. Da in jeder Iteration genau eine eindeutige Komponente beschrieben wird, ist auch dieses Feld „SHARED“. Das Feld Y muß hingegen „PRIVATE“ deklariert werden, da es in jeder Iteration nur an genau einer Komponente verändert werden darf. Die aufgerufene Funktion FCN(N, X, F) liefert für die N-dimensionale Funktion an der Stelle X den Funktionswert F.

Wie Abbildung 4.9 zeigt, lassen sich auf diese Weise sehr gute Speedup-Werte erreichen. Schon bei einer Problemgröße von 128 liegt der Speedup für die Testprobleme 2 und 3 auf der X-MP mit vier Prozessoren über 3,5. Bei weiter steigender Problemgröße steigt er bis über 3,9; selbst bei der Rosenbrock-Funktion wird dann ein Speedup von 3,8 erreicht.

Diese Methode ist auch dann anwendbar, wenn die Dimension des Problems wesentlich geringer ist. Dann ist ein guter Speedup zu erwarten, wenn die Funktionsauswertung sehr lange dauert. Auch die Auswertung analytischer Gradienten kann sehr komplex sein, bei Bischof et al. [BCC⁺92] wird etwa von Beispielen aus praktischen Anwendungen berichtet, in denen der Code zur Berechnung der Jacobi-Matrix über 1000 Zeilen umfasst.

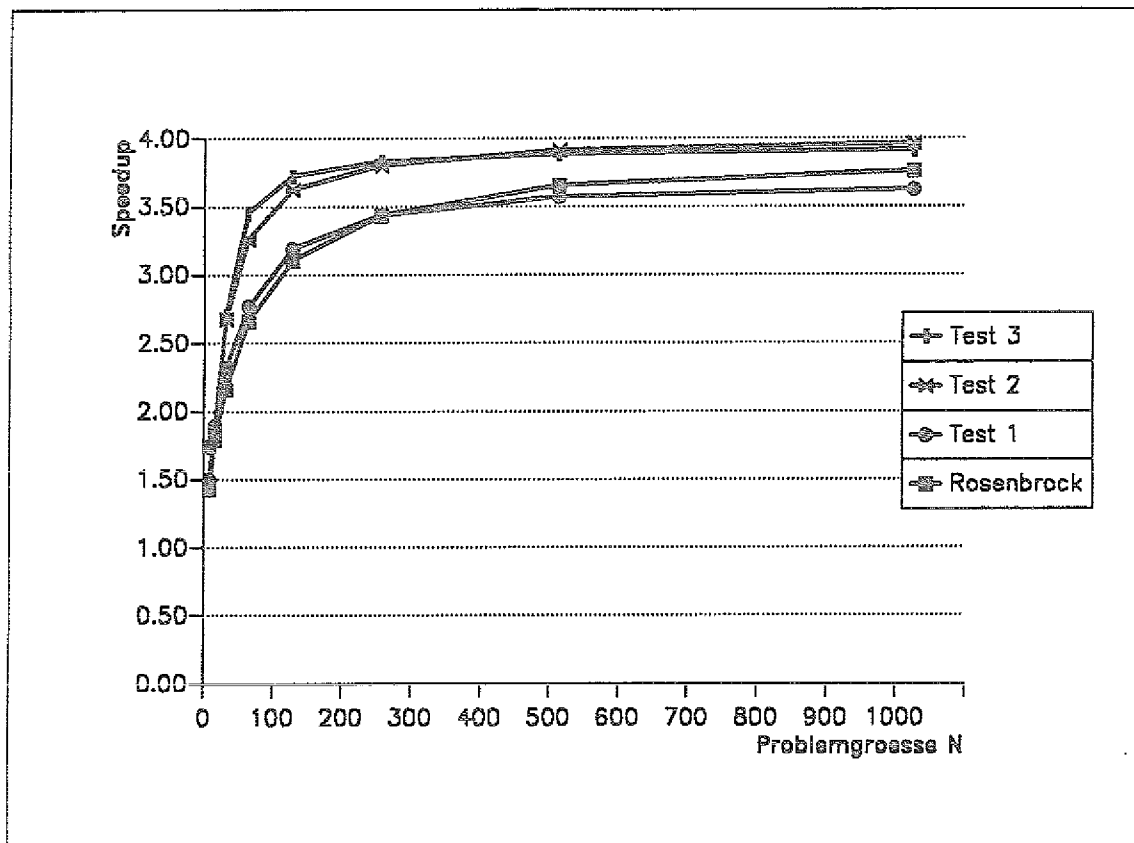


Abbildung 4.9: Beschleunigung auf CRAY X-MP , 4 Prozessoren

Falls für die einzelnen Komponenten des Gradienten jeweils unabhängige Codeabschnitte vorhanden sind, so sind diese innerhalb einer parallelen Region (PARALLEL / END PARALLEL) durch CASE-Direktiven zu trennen.

4.3.3 Optimierung unter Nebenbedingungen: Sequentielle Quadratische Optimierung

Zur Optimierung unter Nebenbedingungen wurde die Methode der Sequentiellen Quadratischen Optimierung (*Sequential Quadratic Programming*) ausgewählt. Grundidee dieser Methode ist die Minimierung der Lagrange-Funktion mit einer speziellen Implementierung der BFGS-Methode. Zum Bestimmen der Suchrichtung wird in jeder Iteration eine quadratische Optimierungsaufgabe gelöst (siehe Kapitel 2.2.2, Seite 24).

Vorteil dieser Methode ist die geringe Anzahl der nötigen Funktions- und Gradientenauswertungen, in jeder Iteration ist jeweils eine solche erforderlich. Weiterhin ist

die Sequentielle Quadratische Programmierung zur nichtlinearen Optimierung mit nichtlinearen Nebenbedingungen universell einsetzbar und nicht nur auf spezielle Problemklassen anwendbar. Daher steht das Verfahren sowohl in der HSL als auch in den Bibliotheken IMSL und NAG zur Verfügung.

In der HSL ist die Sequentielle Quadratische Programmierung durch die Routine VF13A realisiert. Wiederum muß der Benutzer die Routine zu jeder Iteration mit den jeweils aktuellen Werten der Ziel- und Nebenbedingungsfunktionen sowie der dazugehörigen Gradienten aufrufen. In VF13A werden jedoch nur einige Initialisierungen durchgeführt, die Iteration zur Optimierung wird durch die intern aufgerufene Prozedur VF13B ausgeführt. Das quadratische Problem zur Bestimmung der Suchrichtung wird durch die Routine VE17A gelöst, die dazu nötigen Parameter werden in der von VF13B aufgerufenen Prozedur VF13C gesetzt.

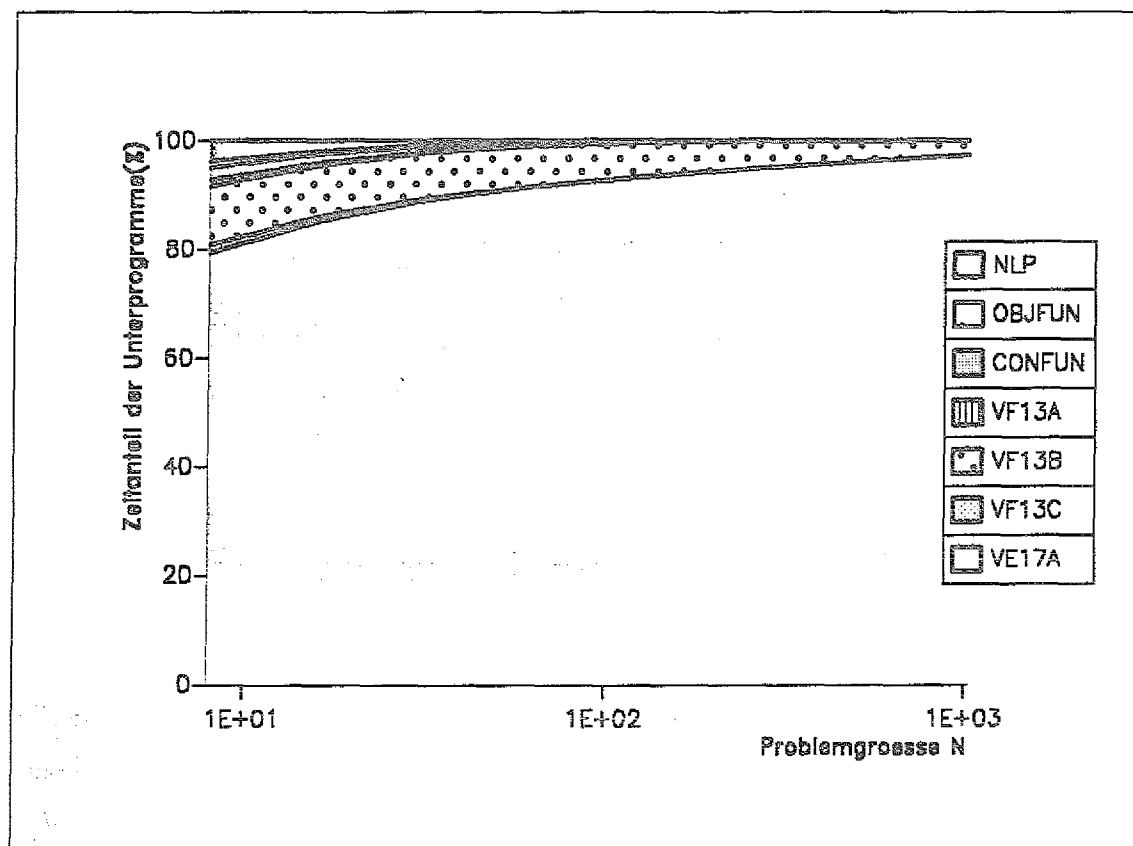


Abbildung 4.10: Anteil der Unterprogramme an der Ausführungszeit, Testproblem 1

Abbildung 4.10 zeigt, daß der bei weitem überwiegende Anteil der Zeit zur Lösung des Testproblems 1 (Anhang B.2) von der Routine VE17A, die das quadratische

Teilproblem löst, verbraucht wird. Schon bei einer Problemgröße von $n = 8$ macht dieser Anteil nahezu 80% aus. Bis zu $n = 1024$ steigt er auf über 97%. Dieses Verhalten zeigte sich ohne signifikante Unterschiede auch bei anderen Testproblemen. Der Anteil der Funktionen und Gradienten (implementiert in den Unterprogrammen OBJFUN und CONFUN) ist hier vernachlässigbar gering. Dies ist in praktischen Anwendungen häufig nicht der Fall. Dennoch erscheint es für die hier vorgenommenen Untersuchungen sinnvoll zu sein, Testprobleme mit dieser Eigenschaft zu wählen; der im Verfahren selbst erzielte Gewinn durch Parallelisierung wird dadurch klarer erkennbar. Falls die Auswertungen der Funktionen in einer Anwendung einen wesentlich höheren Zeitanteil beanspruchen, ist hier eine zusätzliche Parallelisierung wie im vorherigen Abschnitt nötig.

Der Name des Verfahrens, Sequentielle Quadratische Optimierung, deutet bereits an, daß sich das Verfahren nicht „natürlich“ zur Parallelisierung anbietet. Die wesentlichen Schritte müssen tatsächlich nacheinander durchgeführt werden, es gibt kaum Aufgaben, die unabhängig voneinander ausgeführt werden können. Andererseits ist es möglich, die feingranularen Konstrukte von Autotasking zu nutzen und so die auszuführenden Schleifen, also die linear-algebraischen Operationen, zu parallelisieren.

Da der weitaus größte Teil der Zeit in der Routine zur quadratischen Optimierung, VE17A, verbraucht wird, ist es vor allem wichtig, hier Möglichkeiten zur Parallelisierung zu erkennen und umzusetzen. Die Routine realisiert das duale Verfahren nach Goldfarb und Idnani [GI83] zur Lösung einer positiv definiten quadratischen Optimierungsaufgabe. Die Matrizen werden dabei faktorisiert, um die numerische Stabilität zu erhöhen.

Die Untersuchung des Programmes mittels LOOPTOOL ergab bei Testproblem 1 mit 3 Nebenbedingungen und großen n einen Anteil der Cholesky-Zerlegung sowie der anschließenden Inversion der daraus erhaltenen Matrix von jeweils über 45% an den arithmetischen Operationen. Bei Testproblemen mit mehr Nebenbedingungen steigt jedoch der Anteil anderer Programmabschnitte, dann fallen auch die Aktionen ins Gewicht, die durchgeführt werden, wenn eine Nebenbedingung in die Menge der aktiven Restriktionen aufgenommen oder aus dieser entfernt werden. Dabei werden zum Beispiel Givens-Rotationen durchgeführt.

Der FORTRAN-Code von VF13A erscheint optimiert für herkömmliche, sequentielle Rechner. Alle benötigten Matrizen und Vektoren werden in dem *Workspace*-Array W gehalten. Dadurch erkennt das CF77-Compiling-System häufig nicht, wo die verwendeten Daten unabhängig voneinander sind. Dies führt dazu, daß an vielen Stellen, an denen Vektorisierung oder Parallelisierung möglich ist, diese nicht durchgeführt wird. Die Cholesky-Zerlegung zum Beispiel ist wie folgt realisiert (die faktorisierte Matrix wird dabei im Workspace W ab der Position IWR+1 abgelegt):

```

90 IR = IWR
   DO 130 J=1,N
     IRA=IWR
     IRB=IR+1
     DO 120 I=1,J
       TEMP=G(I,J)
       IF (I .EQ. 1) GOTO 110
       DO 100 K=IRB,IR
         IRA=IRA+1
100    TEMP=TEMP-W(K)*W(IRA)
110    IR=IR+1
       IRA=IRA+1
       IF (I .LT. J) W(IR)=TEMP/W(IRA)
120    CONTINUE
       IF (TEMP .LT. VSMALL) GOTO 140
130    W(IR)= SQRT(TEMP)
       GOTO 170

```

Hier vektorisiert nur die innere Schleife (Label 100); durch die Benutzung des Zählers IRA, der sowohl in der K- als auch in der J-Schleife inkrementiert wird, wird eine Vektorisierung oder Parallelisierung der mittleren Schleife verhindert. Über die äußerste Schleife kann nicht parallelisiert werden, da in jeder Iteration die Ergebnisse der bereits durchgeführten Iterationen benötigt werden.

Da bekannt ist, daß die Cholesky-Faktorisierung parallelisiert werden kann [Ort88], wurden zwei Alternativen getestet: Zum einen wurde der „Lehrbuch-Algorithmus“ verwandt, der nur auf der Eingabematrix G arbeitet, zum anderen wurde eine Bibliotheksroutine verwendet.

Die erste Variante wurde durch die folgende Codesequenz realisiert:

```
90 CONTINUE
  DO 96, K=1, N
    DO 94, J=1, K-1
      DO 93, I=K, N
         $G(I,K) = G(I,K) - G(K,J) * G(I,J)$ 
93    CONTINUE
94    CONTINUE
    IF (G(K,K).LE.ZERO) GOTO 140
     $G(K,K) = \text{SQRT}(G(K,K))$ 
     $C = 1.0 / G(K,K)$ 
    DO 95, I=K+1, N
       $G(I,K) = G(I,K) * C$ 
95    CONTINUE
96 CONTINUE
  GOTO 170
```

Dazu müssen zusätzlich zunächst die ursprünglichen Werte der Hauptdiagonale zwischengespeichert werden. Nach Durchführung der Faktorisierung muß das Ergebnis zur weiteren Verwendung dem Arbeitsarray W zugewiesen sowie die Matrix G wiederhergestellt werden, da diese im weiteren noch benötigt wird.

Der gleiche Overhead entsteht auch in der zweiten getesteten Alternative. Hierzu wurde der Code zur Faktorisierung durch einen Aufruf der LAPACK-Routine SPOTRF [ABB⁺92] ersetzt, die in der SCILIB enthalten ist:

```
CALL SPOTRF( 'U', N, G, N, INFO)
```

Der Parameter 'U' bewirkt hier, daß als Ergebnis die obere rechte Dreiecksmatrix geliefert wird, G bezeichnet die Eingabematrix, N bezeichnet einmal die Zahl der Spalten von G und beim zweiten Auftreten die erste Dimension, mit der G im aufrufenden Programm deklariert ist; INFO zeigt an, ob die Faktorisierung erfolgreich durchgeführt wurde oder wo gegebenenfalls ein Fehler auftrat [Cra91].

Im oben vorgestellten Code wird nun sowohl die I-Schleife (Label 93) als auch die J-Schleife (Label 94) vektorisiert. Der *FPP* erkennt diese Möglichkeit automatisch und realisiert sie durch einen Aufruf der BLAS-2-Routine SGEMVX.

Auch Parallelisierung wird auf diese Weise leicht möglich, da sowohl die LAPACK- als auch die BLAS-Routinen in der SCILIB in parallelisierter Form enthalten sind. Die damit erreichten Speedup-Werte stiegen bei großen Problemgrößen auf deutlich über 3 auf der 4-Prozessor-Maschine CRAY X-MP an. Die Unterschiede zwischen den beiden Varianten sind sehr gering; außer bei kleinen Problemgrößen erwies sich meist die erste Variante als geringfügig schneller.

Zu beachten ist hier, daß auch die Behandlung einer nicht positiv definiten Matrix G zu ändern ist. Diese wird durch den Sprung „GOTO 140“ aufgerufen. Zwar ist dieser Fall theoretisch ausgeschlossen, aufgrund von Rundungsfehlern kann er jedoch in praktischen Anwendungen dennoch auftreten.

Die anschließende Invertierung des Cholesky-Faktors ist in VE17A wie folgt realisiert:

```
170 NM=N-1
    DO 220 I=1,N
        IZ=IWZ+I
        IF (I .EQ. 1) GOTO 190
        DO 180 J=2,I
            W(IZ)=ZERO
180  IZ=IZ+N
190  IR=IWR+(I+I*I)/2
        W(IZ)=ONE/W(IR)
        IF (I .EQ. N) GOTO 220
        IZA=IZ
        DO 210 J=I,NM
            IR=IR+I
            SUM=ZERO
            DO 200 K=IZA,IZ,N
                SUM=SUM+W(K)*W(IR)
200  IR=IR+1
            IZ=IZ+N
210  W(IZ)=-SUM/W(IR)
220  CONTINUE
```

Auch hier wird automatisch lediglich die K-Schleife (Label 200) vektorisiert, Parallelismus wird nicht automatisch erkannt. Eine parallele Ausführung der äußersten Schleife ist hier jedoch ohne weitere Änderungen möglich, dazu muß die Direktive

```
CMIC$ DO ALL PRIVATE(I,J,K,IZ,IZA,IR,SUM)
CMIC$*      SHARED(N,IWZ,W,ZERO,ONE,IWR,NM)
```

vor der Schleife eingefügt werden. Dadurch ist für die Invertierung auf den 4 Prozessoren der X-MP ein nahezu optimaler Speedup zu erreichen.

Auf ähnliche Art und Weise ist auch an anderen Stellen Parallelisierung möglich; die meisten der vom *FPP* vermuteten Datenabhängigkeiten sind nicht tatsächlich vorhanden, so daß durch Einfügen entsprechender Direktiven (CFPP\$ NODEPCHK oder CDIR\$ IVDEP) auch eine wesentlich bessere Vektorisierung erreicht wird.

Schon bei sequentieller Ausführung des modifizierten Codes läßt sich so, im Vergleich zum ursprünglichen HSL-Code, eine Beschleunigung um mehr als das Doppelte erzielen.

Der Speedup bei paralleler Ausführung auf der X-MP mit 4 Prozessoren ist abhängig von der Struktur des zu lösenden Problems. Abbildung 4.11 zeigt, daß mit Testproblem 1 ein deutlich besserer Speedup als mit Problem 2 erreicht wird. Weitere Vergleichsmessungen ergaben, daß der Speedup mit steigender Anzahl an Nebenbedingungen abnimmt. Der Speedup liegt jedoch auch für Problem 2 bei entsprechend großen Problemen über 3, bei Testproblem 1 steigt er sogar deutlich über 3,5.

In praktischen Anwendungen ist sogar eine weitere Verbesserung möglich. Wie zu Anfang des Abschnitts erwähnt, sind die Testprobleme hier so gewählt, daß der Anteil der Funktions- und Gradientenauswertung an der benötigten Zeit vernachlässigbar gering ist. In Anwendungen ist dies jedoch häufig nicht der Fall. Nur wenn es gelingt, die Funktions- und Gradientenauswertung ebenfalls zu parallelisieren, kann dann eine stärkere Beschleunigung erreicht werden.

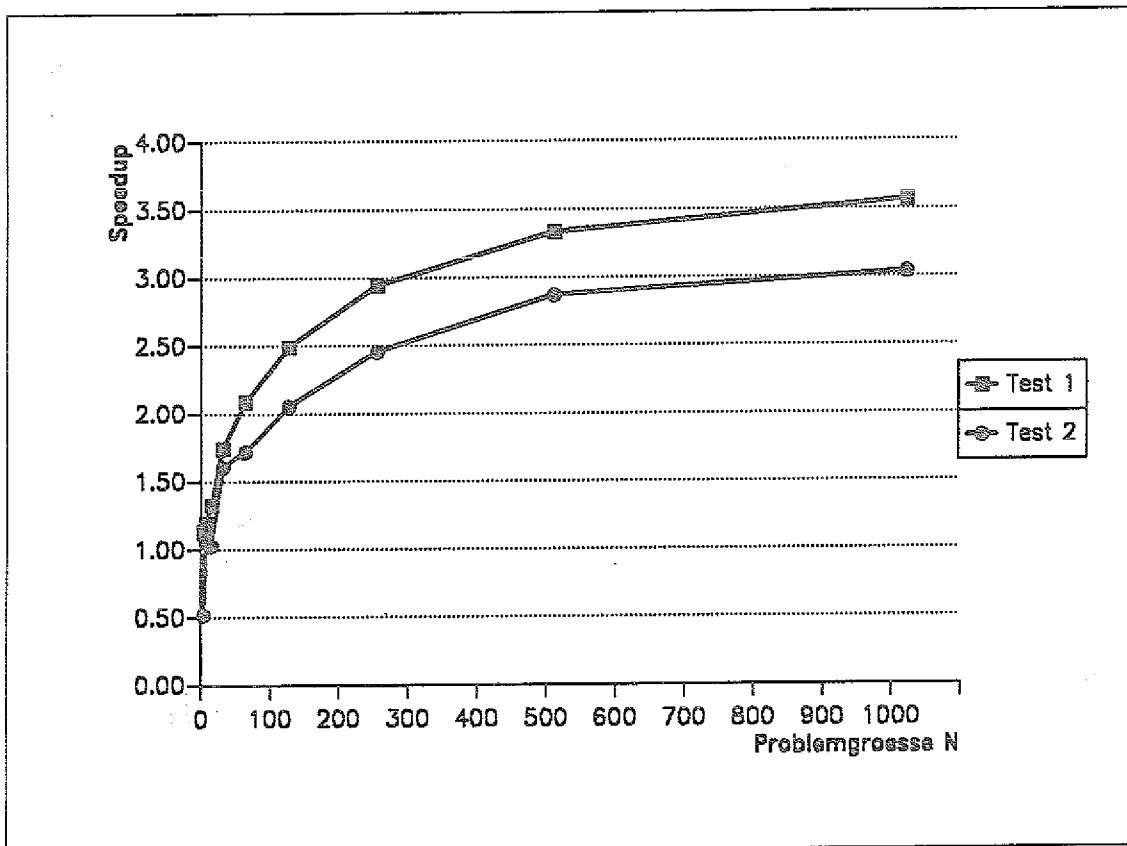


Abbildung 4.11: Beschleunigung auf CRAY X-MP, 4 Prozessoren

Kapitel 5

Schlußbemerkungen

Bei der Entwicklung von Algorithmen gibt es zwei grundsätzliche Vorgehensweisen: Zum einen werden existierende sequentielle Algorithmen parallelisiert; zum anderen ist es möglich, neue parallele Algorithmen zu entwerfen. Hier gilt es vor allem auch, gewohnte sequentielle Denkweisen zu überwinden.

In dieser Arbeit wurde mit der Parallelisierung der Sequentiellen Quadratischen Optimierung der erste Ansatz gewählt. Damit lassen sich auf schwach-parallelen Computern, wie den CRAY-Rechnern, durchaus akzeptable Ergebnisse erzielen. Es konnte zwar kein optimaler Speedup in Höhe der Prozessorzahl erreicht werden; andererseits werden Parallelrechner in Rechenzentrumsumgebungen üblicherweise im Multiprogramming-Betrieb genutzt. Multitasking kann hier insbesondere auch dann zu substantiellen Verbesserungen des Durchsatzes führen, wenn es durch Engpässe bei Betriebsmitteln — wie etwa im Hauptspeicher bei Rechnern mit realem Speicherkonzept — zu einer schlechten Auslastung des Gesamtsystems kommt [Nag90a].

Die für die unbeschränkte Minimierung getestete — auch in der Optimierung unter Nebenbedingungen nutzbare — parallele Auswertung von Funktionen und Gradienten weist eher in die Richtung der Ausnutzung inhärent vorhandener paralleler Strukturen. Ähnlich sind auch Versuche einzuordnen, in denen die Struktur des dualen Problems in der separablen Optimierung ausgenutzt werden sollten (vergleiche Kapitel 3.5). In ersten Ansätzen wurden damit durchaus erfolgversprechende Ergebnisse erzielt. Der vorgegebene Rahmen dieser Arbeit ließ jedoch für die Untersuchung dieser Fragen keinen Raum.

Bei parallelen Algorithmen dieser Art ist eher eine Übertragbarkeit auf massiv-parallele Computer gegeben. Auf solchen Systemen tritt jedoch in sehr viel stärkerem Umfang auch der Aufwand für Kommunikation und Synchronisation zwischen den Prozessoren in den Vordergrund, da derartige Rechner üblicherweise mit verteiltem Speicher ausgerüstet sind. Aus diesem Grund bergen hier asynchrone Algorithmen sicherlich auch für die Optimierung ein erhebliches Leistungspotential.

Wenn auch heute noch ein Mangel an Erfahrung in Programmierung und Betrieb einen breiten Einsatz massiv-paralleler Rechner verhindert, so steht ihr Durchbruch doch bevor oder ist bereits vollzogen. Massiv-parallele Rechner werden so mit Sicherheit die zukünftigen Leistungsträger des Supercomputing stellen, zunächst eigenständig und zunehmend als integrale Systemkomponenten heterogen strukturierter Supercomputerkomplexe [Hoß91]. So muß sich auch die Entwicklung paralleler Algorithmen zur Optimierung in Zukunft verstärkt auf Konzepte dieser Art konzentrieren.

Literaturverzeichnis

- [ABB⁺92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov und D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.
- [Bas89] M. Bastian. Operations Research II. Skriptum zur Vorlesung, Lehrstuhl für Unternehmensforschung, RWTH Aachen, 4. Auflage 1989.
- [BCC⁺92] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank und Paul Hovland. ADIFOR Generating Derivative Codes from Fortran Programs. Preprint MCS-P263-0991, Argonne National Laboratory, 1992.
- [BM70] Jerome Bracken und Garth P. McCormick. *Ausgewählte Beispiele für Nichtlineares Programmieren*. Verlag Berliner Union, Stuttgart, 1970.
- [BS79] Mokhar S. Bazaraa und C. M. Shetty. *Nonlinear Programming*. John Wiley & Sons, New York, 1979.
- [BS86] I.N. Bronstein und K.A. Semendjajew. *Taschenbuch der Mathematik, Ergänzende Kapitel*. Verlag Harry Deutsch, Thun, 4. Auflage, 1986.
- [BSS88] R.H. Byrd, R.B. Schnabel und G.A. Shultz. Parallel Quasi-Newton Methods for Unconstrained Optimization. *Mathematical Programming B*, 42, Nr. 2, Seiten 273–306, 1988.
- [BT88] Dimitri P. Bertsekas und John N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

- [Bur89] Heribert Burg. Implementierung und Untersuchung paralleler Algorithmen zur Schnellen Fourier-Transformation auf CRAY-Mehrprozessorrechnern. Interner Bericht Jül-Spez-520, Forschungszentrum Jülich (KFA), Jülich, August 1989.
- [Cra89] Cray Research, Inc., Mendota Heights. *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual*, 1989. SR-0222 F.
- [Cra90] Cray Research, Inc., Mendota Heights. *CF77 Compiling System, Volume 4: Parallel Processing Guide*, Juni 1990. SG-3074 4.0.
- [Cra91] Cray Research, Inc., Mendota Heights. *Volume 3: UNICOS Math and Scientific Library Reference Manual*, Januar 1991. SR-2081 6.0.
- [Day89] M. Dayde. Parallel Algorithms for Nonlinear Programming Problems. *Journal of Optimization Theory and Applications*, 61, Nr. 1, Seiten 23–46, April 1989.
- [DDHD90] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling und Iain Duff. An Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16, Nr. 1, Seiten 1–17, März 1990.
- [DDHH88] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling und Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14, Nr. 1, Seiten 1–17, März 1988.
- [DP82] L.C.W. Dixon und K.D. Patel. The Place of Parallel Computation in Numerical Optimisation; IV Algorithms for Nonlinear Optimisation. Technical Report 125, The Hatfield Polytechnic Numerical Optimisation Centre, Hatfield, Februar 1982.
- [Fle87] Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester, 2. Auflage, 1987.
- [Fra80] Joel Franklin. *Methods of Mathematical Economics*. Springer-Verlag, New York, 1980.

- [GC88] L. Grandinetti und D. Conforti. Numerical Comparisons of Nonlinear Programming Algorithms on Serial and Vektor Processors Using Automatic Differentiation. *Mathematical Programming B*, 42, Nr. 2, Seiten 375–389, 1988.
- [GI83] D. Goldfarb und A. Idnani. A Numerically Stable Dual Method for Solving Strictly Convex Quadratic Programs. *Mathematical Programming*, 27, Seiten 1–33, 1983.
- [GMSW85] P.E. Gill, W. Murray, M.A. Saunders und M.H. Wright. Model Building and Practical Aspects of Nonlinear Programming. In: Klaus Schittkowski (Hrsg.), *Computational Mathematical Programming*, Band 15 von *Series F: Computer and Systems Sciences*, Seiten 209–247. Springer-Verlag, 1985.
- [HB84] Kai Hwang und Fayé Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.
- [Hei85] Heinrich Heine. *Ein Lesebuch für unsere Zeit*. Lesebücher für unsere Zeit. Aufbau-Verlag, Berlin und Weimar, 28. Auflage, 1985.
- [HKN89] Friedel Hoßfeld, Renate Knecht und Wolfgang E. Nagel. Multitasking: Experience with applications on a CRAY X-MP. *Parallel Computing*, 12, Seiten 259–283, 1989.
- [Hoß88] Friedel Hoßfeld. Vector-supercomputers. *Parallel Computing*, 7, Seiten 373–385, 1988.
- [Hoß91] Friedel Hoßfeld. "Grand Challenges" – wie weit tragen die Antworten des Supercomputing. Interner Bericht KFA-ZAM-IB-9117, Forschungszentrum Jülich – ZAM, Jülich, November 1991.
- [HS81] Willi Hock und Klaus Schittkowski. *Test Examples for Nonlinear Programming Codes*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin, 1981.
- [HSL90] Advanced Computing Department; AEA Industrial Technology; Harwell Laboratory, Oxfordshire. *Harwell Subroutine Library Specifications*, 1. Auflage, September 1990.

- [IMS87] IMSL, Inc., Houston, Texas. *IMSL Math/Library User's Manual 1.0*, April 1987.
- [IMS92] IMSL, Inc., Sugar Land, Texas. *The IMSL Parallel Libraries Supplemental Manual*, März 1992.
- [Kne86] Siegfried Knecht. Möglichkeiten des Multitasking zur Beschleunigung von Standardalgorithmen. Interner Bericht Jül-Spez-361, Forschungszentrum Jülich (KFA), Jülich, 1986.
- [Lem89] Claude Lemarechal. Mathematical Classification of Optimization Problems. In: Jean-Paul Penot (Hrsg.), *New Methods in Optimization and their Industrial Uses*, Seiten 73–87, Basel, 1989. Birkhäuser Verlag.
- [Lie88] Arno Liegmann. Die Strategie des Microtasking als Mittel zur Beschleunigung von Programmen auf dem Vektorrechner CRAY X-MP. Interner Bericht Jül-Spez-453, Forschungszentrum Jülich (KFA), Jülich, 1988.
- [LR88] F.A. Lootsma und K.M. Ragsdell. State-of-the-art in parallel nonlinear optimization. *Parallel Computing*, 6, Seiten 133–155, 1988.
- [Lue84] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley, Reading, MA, 1984.
- [Min86] Michel Minoux. *Mathematical Programming: Theory and Algorithms*. John Wiley & Sons, New York, 1986.
- [Mur88] Katta Gopalakrishna Murty. *Linear Complementary, Linear and Nonlinear Programming*. Sigma Series in Applied Mathematics. Heldermann Verlag, Berlin, 1988.
- [Nag90a] Wolfgang Nagel. Prinzipien der Parallelverarbeitung auf Rechnern mit gemeinsamem Speicher. In: A. Reuter (Hrsg.), *Proceedings der 20. GI-Jahrestagung 1990, Informatik Fachbericht 257*. Springer Verlag, 1990.
- [NAG90b] The Numerical Algorithms Group Limited, Oxford. *NAG Fortran Library Manual, Mark 14*, 1. Auflage, April 1990.
- [Ort88] James M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, New York, 1988.

- [Pap91] Markos Papageorgiou. *Optimierung: statische, dynamische, stochastische Verfahren für die Anwendung*. R. Oldenbourg Verlag, München, 1991.
- [Roc73] R. Tyrell Rockafellar. A Dual Approach to Solving Nonlinear Programming Problems by Unconstrained Optimization. *Mathematical Programming*, 5, Seiten 354–373, 1973.
- [Sch85] Robert B. Schnabel. Parallel Computing in Optimization. In: Klaus Schittkowski (Hrsg.), *Computational Mathematical Programming*, Seiten 357–381. Springer-Verlag, Berlin, 1985.
- [SW92] Robert Schaback und Helmut Werner. *Numerische Mathematik*. Springer-Verlag, Berlin, 4. Auflage, 1992.
- [Zim87] Hans-Jürgen Zimmermann. *Methoden und Modelle des Operations Research*. Rechnerorientierte Ingenieurmathematik. Friedr. Vieweg & Sohn, Braunschweig, 1987.
- [ZM88] Stavros A. Zenios und John M. Mulvey. Vectorization and Multitasking of Nonlinear Network Programming Algorithms. *Mathematical Programming B*, 42, Seiten 449–470, 1988.

Anhang A

Unterprogramme in Programmbibliotheken

Unterprogramme zur Optimierung werden vom Zentralinstitut für Angewandte Mathematik im Forschungszentrum Jülich aus folgenden Programmbibliotheken angeboten:

- IMSL
- NAG
- HSL (Harwell Subroutine Library)
- Collected Algorithms from ACM
- CERNLIB

Die Routinen aus HSL, Collected Algorithms of ACM und MINUIT aus der Cernlib liegen auch im Sourcecode vor, während von IMSL und NAG nur der Objectcode verfügbar ist.

Die Unterprogramme der IMSL liegen auch auf den CRAY-Rechnern in „double precision“ vor. Diese sind dann durch ein dem Namen vorangestelltes D gekennzeichnet (z.B. NCONF / DNCONF). Es wird jedoch davon abgeraten, diese zu benutzen, da die double precision-Arithmetik auf den CRAY-Rechnern sehr ineffizient ist [IMS92]. Man beachte jedoch, daß die einfache Arithmetik hier bereits 64-Bit-Arithmetik ist, also double precision auf anderen Rechnern entspricht.

Seit Juni 1992 ist die Version 2.0 IMSL-Bibliothek auch in einer parallelisierten Version verfügbar. Die Unterprogramme zur nichtlinearen Optimierung zeigen jedoch keine effiziente Ausnutzung der Mehrprozessorumgebung, hier ist sicherlich keine wesentliche Arbeit zur Parallelisierung geleistet worden.

Häufig werden für die gleiche Problemklasse mehrere Unterprogramme angeboten, die im wesentlichen den gleichen Algorithmus nutzen. Diese unterscheiden sich dann in den vom Benutzer erwarteten Angaben. Der Benutzer muß als Parameter immer eine Prozedur angeben, mit der die Funktionswerte der Zielfunktion berechnet werden können (ggf. auch Prozeduren für die Nebenbedingungen). Unterschiede liegen in den Erfordernissen an Gradienten, Hesse- und (bei Nebenbedingungen) Jacobimatrix. Diese können entweder ebenfalls als Prozeduren angegeben oder durch Differenzenverfahren angenähert werden. Ziel- und Nebenbedingungsfunktionen müssen stetig und meist auch stetig differenzierbar sein. Manche der Bibliotheksprozeduren erlauben allerdings auch, daß die Funktionen (und Gradienten) einzelne Unstetigkeitsstellen besitzen. (Es gibt auch spezielle Prozeduren für nicht stetige Zielfunktionen, diese werden hier jedoch nicht weiter betrachtet.)

A.1 Minimierung ohne Nebenbedingungen

Zur Minimierung ohne Nebenbedingungen werden hauptsächlich drei Algorithmentypen verwendet: Newton- und Quasi-Newton-Verfahren sowie Konjugierte-Gradienten-Verfahren. Dabei sollten die letzteren insbesondere bei großen Problemen verwendet werden, während die Quasi-Newton Verfahren bei kleinen und mittleren Problemgrößen besser geeignet sind. Für den Fall, daß nur Funktionswerte bekannt sind (also keine Informationen über Ableitungen vorliegen), werden Algorithmen mit direkter Suche, sogenannte Simplex-Verfahren, verwendet, die zwar langsam konvergieren, aber robust in der Anwendung sind.

Von nahezu allen Bibliotheksroutinen zur Minimierung ohne Nebenbedingungen wird erwartet, daß eine Ausgangsnäherung angegeben wird (vergleiche Tabelle A.2, Init.).

Prozeduren, bei denen als Restriktionen nur obere und untere Schranken vorhanden sind, werden in diesem Abschnitt mit aufgeführt, da die Behandlung der Schranken algorithmisch sehr einfach ist. Im Prinzip werden hierzu die gleichen Algorithmen

Name	Algorithmus	Schranken
IMSL		
BCOAH	modifiziertes Newton-Verfahren	Ja
BCODH	modifiziertes Newton-Verfahren	Ja
BCONF	Quasi-Newton-Verfahren	Ja
BCONG	Quasi-Newton-Verfahren	Ja
BCPOL	direkte Suche	Ja
UMCGF	Konjugierte Gradienten	Nein
UMCGG	Konjugierte Gradienten	Nein
UMIAH	modifiziertes Newton-Verfahren	Nein
UMIDH	modifiziertes Newton-Verfahren	Nein
UMINF	Quasi-Newton-Verfahren	Nein
UMING	Quasi-Newton-Verfahren	Nein
UMPOL	direkte Suche / Polytope	Nein
NAG		
E04CCF	Simplex, nur Funktionswerte	Nein
E04JAF	Quasi-Newton-Verfahren	Ja
E04JBF	Quasi-Newton-Verfahren	Ja
E04KAF	Quasi-Newton-Verfahren	Ja
E04KBF	Quasi-Newton-Verfahren	Ja
E04KCF	modifiziertes Newton-Verfahren	Ja
E04KDF	modifiziertes Newton-Verfahren	Ja
E04LAF	modifiziertes Newton-Verfahren	Ja
E04LBF	modifiziertes Newton-Verfahren	Ja
HSL		
VA04A	Powell	Nein
VA06A	Kombination aus steilstem Abstieg und verallgem. Newton	Nein
VA08A	Konjugierte Gradienten	Nein
VA09A	Quasi-Newton-Verfahren	Nein
VA10A	Quasi-Newton-Verfahren	Nein
VA13A	Variable Metric — BFGS	Nein
VA14A	Konjugierte Gradienten	Nein
Collected Algorithms from ACM		
Alg.500	Quasi-Newton-Verfahren, BFS-Update	Nein
Alg.611	Trust-Regions, Quasi-Newton, BFGS	Nein
Alg.630	Variable Storage (CG, Quasi-Newton)	Nein
Alg.702	Truncated Newton, Preconditioned CG	Nein
Cernlib		
MINUIT	Variable Metric (Fletcher)	Nein

Tabelle A.1: Prozeduren zur Optimierung ohne Nebenbedingungen

Name	f	∇f	$\nabla^2 f$	Init.	Bemerkungen
IMSL					
BCOAH	P	P	P	ja	
BCODH	P	P	fd	ja	
BCONF	P	fd	BFGS	ja	
BCONG	P	P	BFGS	ja	
BCPOL	P	–	–	ja	
UMCGF	P	fd	–	ja	
UMCGG	P	P	–	ja	
UMIAH	P	P	P	ja	
UMIDH	P	P	–	ja	
UMINF	P	fd	BFGS	ja	
UMING	P	P	BFGS	ja	
UMPOL	P	–	–	ja	
NAG					
E04CCF	P	–	–	ja	comprehensive
E04JAF	P	–	–	ja	easy-to-use
E04JBF	P	–	–	ja	comprehensive
E04KAF	P	P	–	ja	easy-to-use
E04KBF	P	P	–	ja	comprehensive
E04KCF	P	P	–	ja	easy-to-use
E04KDF	P	P	–	ja	comprehensive
E04LAF	P	P	P	ja	easy-to-use
E04LBF	P	P	P	ja	comprehensive
HSL					
VA04A	P	–	–	ja	
VA06A	P	P	–	ja	
VA08A	P	P	–	ja	Fletcher, Reeves
VA09A	P	P	–	ja	besser als VA06A
VA10A	P	fd	fd	ja	
VA13A	P	P	BFGS	ja	
VA14A	P	P	–	ja	
Collected Algorithms from ACM					
Alg.500	P	P	–	ja	Shanno, Phua 1974
Alg.611	P	P	BFGS	ja	
Alg.630	P	–	–	ja	
Alg.702	P	P	P	ja	

P: Prozedur fd: finite Differenzen

Tabelle A.2: Benutzer-Angaben und Methode des Update

(Newton- und Quasi-Newton-Verfahren) verwendet, gegebenenfalls wird eine Variable an ihrer Schranke festgehalten, und die Funktion wird über den freien Variablen weiter minimiert. Während in der IMSL-Bibliothek getrennte Unterprogramme zur Optimierung mit und ohne Schranken zur Verfügung gestellt werden, gibt es in der NAG-Bibliothek nur kombinierte Prozeduren. Zur Minimierung ohne Schranken sollen hier dieselben so groß (bzw. klein) gewählt werden, daß sie praktisch keine Beschränkung bedeuten.

Die Prozeduren E04JBF und E04KBF werden in absehbarer Zukunft nicht mehr im Lieferumfang der NAG-Bibliothek enthalten sein, statt dessen soll E04UCF benutzt werden. Dieses Unterprogramm ist zur Optimierung sowohl mit als auch ohne Nebenbedingungen geeignet.

A.2 Minimierung unter Nebenbedingungen

Zur allgemeinen nichtlinearen Optimierung unter Nebenbedingungen wird überwiegend die Sequentielle Quadratische Programmierung verwendet. Bei diesem Verfahren wird ein stationärer Punkt der zum Problem gehörigen Lagrange-Funktion gesucht, wobei die Suchrichtung in jeder Iteration durch Lösen eines Problems der quadratischen Programmierung bestimmt wird (vergleiche 2.2.2). In der HSL ist jedoch auch eine Routine enthalten, die ein Lagrange-Penalty-Verfahren verwendet.

In den Tabellen A.3 und A.4 ist eine Aufstellung der Routinen zu finden. Es sind zwar — insbesondere in der HSL — einige weitere Unterprogramme vorhanden, diese sind jedoch meist älteren Datums, und die Verwendung der hier aufgeführten Unterprogramme wird empfohlen.

In der Version 2.0 der IMSL Library sind zur Optimierung unter Nebenbedingungen zusätzlich die Prozeduren LCONF und LCONG enthalten. Die erste der beiden schätzt die Gradienten mittels finiter Differenzen ab, während die zweite vom Benutzer eine Prozedur zur exakten Bestimmung der Gradienten erfordert. Auf CRAY-Rechnern werden diese voraussichtlich in der zweiten Jahreshälfte 1992 verfügbar sein.

Die Unterprogramme zur Lösung von Problemen der quadratischen Programmierung **QP** benötigen keine vom Benutzer erstellten Prozeduren zur Funktions- und

Name	Problem	verwendeter Algorithmus
IMSL		
NCONF	NLP	Sequentielle Quadratische Programmierung
NCONG	NLP	Sequentielle Quadratische Programmierung
QPROG	QP	dualer Algorithmus nach Goldfarb und Idnani (1983)
NAG		
E04NAF	QP	Working Set, Proj. Gradienten; auch für LP
E04UCF	NLP	Sequentielle Quadratische Programmierung
E04VCF	NLP	Sequentielle Quadratische Programmierung
E04VDF	NLP	Sequentielle Quadratische Programmierung
HSL		
VE01A	NLP	Davidon (lineare NB)
VE02A	QP	Fletcher
VE03A	NLP	Fletcher (lineare NB)
VE09	QP	Methode mit aktiver Menge
VE11	NLP	Aktive Menge / BFGS (lineare NB)
VE17	QP	Goldfarb, Idnani
VF01A	NLP	Lagrange-Penalty-Verfahren nach Fletcher
VF02AD	NLP	quadratische Approximation an Lagrangefunktion
VF04A	NLP	BFGS für Lagrange-Penalty-Funktion
VF13A	NLP	quadratische Approximation an Lagrangefunktion
Collected Algorithms from ACM		
Alg.559	QP	orthogonale Dekomposition

Tabelle A.3: Prozeduren zur Optimierung unter Nebenbedingungen

Name	f	g	∇f	∇g	Init.	Bemerkungen
IMSL						
NCONF	P	P	–	–	ja	
NCONG	P	P	P	P	ja	
NAG						
E04UCF	P	P	OP	OP	ja	comprehensive
E04VCF	P	P	P	P	ja (od. 0)	comprehensive
E04VDF	P	P	P	P	ja	easy-to-use
HSL						
VE01A	P	V	P	–	zul.	
VE02A	–	–	–	–	ja	(Fletcher)
VE03A	P	–	P	–	opt.	(Fletcher)
VE11A	P	–	P	–	ja	
VF01A	P	P	P	P	ja	
VF02AD	P	P	P	P	ja	
VF04A	P	–	–	–	ja	
VF13A	A	A	A	A	ja	

P: Prozedur OP: Optionale Prozedur A: aufrufendes Programm

Tabelle A.4: Benutzer-Angaben

Gradientenauswertung, sie sind deshalb in Tabelle A.4 nicht mehr gesondert aufgeführt. Die Zielfunktion wird in zwei Teilen, der symmetrischen, positiv definiten Matrix Q und dem Vektor c , übergeben. Die linearen Nebenbedingungen werden ebenfalls als Matrix (für die linke Seite) und Vektor (für die rechte Seite) mitgeteilt. Meist sind auch hier zusätzlich obere und untere Schranken erlaubt.

Anhang B

Testprobleme

B.1 Unbeschränkte Minimierung

Testproblem 1:

$$f(x) = \sum_{i=1}^{n/2} (1 + (x_{2i} - a_{2i})^4) \cdot (1 + (x_{2i-1} - a_{2i-1})^4)$$

Testproblem 2:

$$f(x) = \sum_{i=1}^{n/2} (1 + (x_{2i} - a_{2i})^4) \cdot \log(e + (x_{2i-1} - a_{2i-1})^4)$$

Testproblem 3:

$$f(x) = \sum_{i=1}^{n/2} \arctan(1 + (x_{2i} - a_{2i})^4) \cdot \log(e + (x_{2i-1} - a_{2i-1})^4)$$

Rosenbrock-Funktion [DP82]:

$$f(x) = \sum_{i=1}^{n/2} [100(x_{2i-1}^2 - x_{2i})^2 + (1 - x_{2i-1})^2]$$

B.2 Minimierung unter Nebenbedingungen

Testproblem 1:

$$\begin{aligned} \text{minimiere} \quad & f(x) = \sum_{i=1}^{n/2} (1 + (x_{2i} - a_{2i})^4) \cdot \log(e + (x_{2i-1} - a_{2i-1})^4) \\ \text{so da\ss} \quad & (x - a) \cdot v_j + \varepsilon_j \geq 0, \quad j = 1, \dots, m \end{aligned}$$

Testproblem 2:

$$\begin{aligned} \text{minimiere} \quad & f(x) = \sum_{i=1}^n \log(e + (x_i - a_i)^4) \\ \text{so da\ss} \quad & (x_{2j-1} - j)^2 + (x_{2j} - a_{2j})^2 - (j - a_{2j})^2 \geq 0, \quad j = 1, \dots, \frac{n}{2} \end{aligned}$$

In beiden Testproblemen sind die Parameter a_i frei zu whlen. Die Minimalstellen beider Testprobleme liegen bei $x_i = a_i$, $i = 1, \dots, n$.

Bei Testproblem 1 sind zustzlich die Vektoren v sowie $\varepsilon_i \geq 0$ fr die Nebenbedingungen frei whlbar.

Beide Testprobleme sind in ihrer Dimension leicht skalierbar, bei immer existierender Minimalstelle. Problem 1 ist zustzlich in der Anzahl der Nebenbedingungen variabel. Bei Problem 2 sind alle Nebenbedingungen im Minimalpunkt bindend, auch hier ist eine Vernderung der Anzahl der Nebenbedingungen leicht mglich.

Weitere Testprobleme findet man bei Hock und Schittkowsky [HS81]. Die dort aufgefhrten Probleme haben jedoch hchstens die Dimension $n = 16$, so da\ss sie zwar geeignet erscheinen, die Korrektheit eines Verfahrens zu berprfen, fr Performanceanalysen auf modernen Parallelrechnern jedoch zu klein sind.

Diese Arbeit wurde als Diplomarbeit in Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen angefertigt.

Dem Lehrstuhlinhaber Herrn Prof. Dr. Friedel Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich anfertigen zu können. Die erfahrene hervorragende Unterstützung durch Mitarbeiter des Instituts war wichtige Voraussetzung für das Gelingen der Arbeit. Mein besonderer Dank gilt Herrn Dr. Edgar M. E. Wermuth für die konstruktiven Diskussionen und die Betreuung der Arbeit.

Jüli-2705
Dezember 1992
ISSN 0366-0885