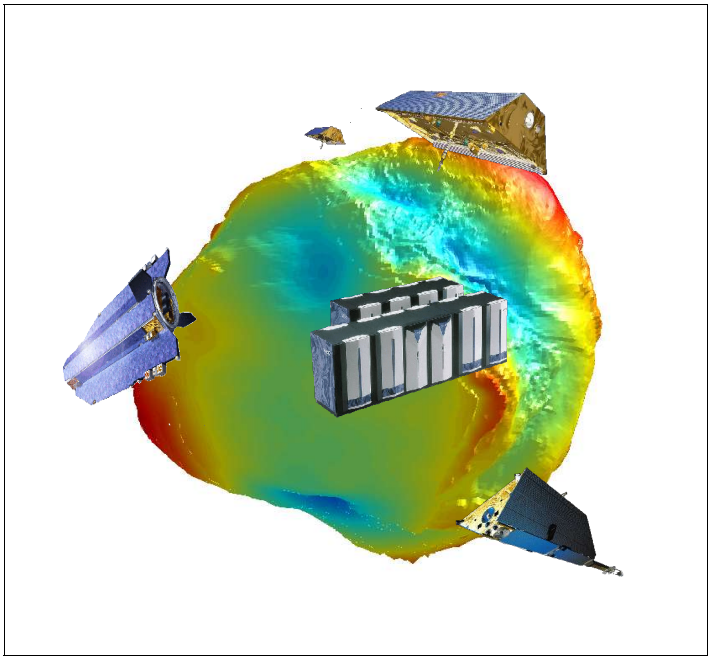
Universität Stuttgart

Geodätisches Institut

High Performance Computing im Einsatz zur Schwerefeldanalyse mit CHAMP, GRACE und GOCE



Diplomarbeit im Studiengang

Geodäsie und Geoinformatik

an der Universität Stuttgart

Tobias Wittwer

Stuttgart, Juli 2004

Betreuer:

Dipl.-Ing. Gerrit Austen, Dipl.-Ing. Oliver Baur

Prüfer:

Prof. Dr. sc. techn. Wolfgang Keller

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir eingereichte Diplomarbeit zum Thema

„High Performance Computing
im Einsatz zur Schwerefeldanalyse
mit CHAMP, GRACE und GOCE“

selbstständig und nur unter der Benutzung der in der Arbeit angegeben Literatur angefertigt habe.

Stuttgart, den 30. Juli 2004

Tobias Wittwer

Symbole und Bezeichnungen

a	große Halbachse
$\mathbf{A}$	Designmatrix
α	Regularisierungsparameter
$\mathbf{b}$	rechte Seite des Normalgleichungssystems
β	rechte Seite des Normalgleichungssystems aus Zufallsbeobachtungen
$c_{k,l}$	Cosinus-Koeffizienten der harmonischen Reihenentwicklung
$\mathbf{C}_{\hat{x}\hat{x}}$	Kovarianzmatrix der ausgeglichenen Unbekannten
$\mathbf{C}_{yy}$	Kovarianzmatrix der Beobachtungen
e	Exzentrizität
GM	geozentrische Konstante
$\mathbf{G}$	Cholesky-Dreieckszerlegung von $\mathbf{P}$
i	Inklination
$\mathbf{I}$	Einheitsmatrix
k	Grad der Reihenentwicklung
k_{max}	maximaler Entwicklungsgrad
$\mathbf{K}$	Regularisierungsmatrix
l	Ordnung der Reihenentwicklung
λ	geografische Länge
M	mittlere Anomalie
n	Anzahl der Beobachtungen
$\mathbf{N}$	Normalgleichungsmatrix
$\mathbf{N}_{bd}$	Vorkonditionierungsmatrix
ω	Argument des Perigäums
Ω	Rektaszension des aufsteigenden Knotens
$\mathbf{P}$	Gewichtsmatrix
$\overline{P}_{k,l}$	normierte Legendresche Funktionen
φ	geografische Breite
r	Gesamtredundanz
r_i	Redundanzanteil
R	mittlerer Erdradius
ρ	Korrelationskoeffizient
s^2	a-posteriori Varianz der Gewichtseinheit
s_i^2	Varianzkomponente
$s_{k,l}$	Sinus-Koeffizienten der harmonischen Reihenentwicklung
σ_i^2	a-priori Varianz einer Beobachtungsgruppe
u	Anzahl der Unbekannten
$\mathbf{V}$	Gravitationstensor
V	Gravitationspotential
$\mathbf{v}$	Verbesserungsvektor
$\mathbf{w}$	Pseudozufallsvektor bei der MCVCE
$\mathbf{x}$	Vektor der Unbekannten
$\widehat{\mathbf{x}}$	Vektor der ausgeglichenen Unbekannten
$\widehat{\xi}$	Vektor der ausgeglichenen Unbekannten aus Zufallsbeobachtungen
$\mathbf{y}$	Beobachtungsvektor
$\widehat{\eta}$	Beobachtungsvektor aus Zufallszahlen
$\mathbf{z}$	Pseudozufallsvektor bei der MCVCE

Abkürzungen

ACML	AMD Core Math Library
ALU	Arithmetic/Logical Unit
AMD	Advanced Micro Devices
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subroutines
BLACS	Basic Linear Algebra Communication Subroutines
ccNUMA	cache-coherent Non-Uniform Memory Access
CHAMP	Challenging Minisatellite Payload for Geophysical Research and Application
CPU	Central Processing Unit
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DM	Distributed Memory
EGM96	Earth Gravity Model 1996
ESA	European Space Agency
FLOPS	Floating Point Operations per Second
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GCV	General Cross Validation
GFZ	Geoforschungszentrum Potsdam
GNU	GNU's not UNIX
GOCE	Gravity Field and Steady-State Ocean Circulation Explorer
GPS	Global Positioning System
GRACE	Gravity Recovery and Climate Experiment
HLRS	Höchstleistungsrechenzentrum Stuttgart
HL-SST	High-Low Satellite to Satellite Tracking
JPL	Jet Propulsion Laboratory
LAPACK	Linear Algebra Package
LEO	Low Earth Orbiter
LL-SST	Low-Low Satellite to Satellite Tracking
LSQR	Least Squares basierend auf QR-Zerlegung
MCVCE	Monte Carlo Variance Component Estimation
MIMD	Multiple Instruction - Multiple Data
MKL	Math Kernel Library
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NASA	National Aeronautic and Space Administration
NOW	Network of Workstations
PBLAS	Parallel Basic Linear Algebra Subprograms
PC-LSQR	Pre-Conditioned LSQR
RAM	Random Access Memory
ScaLAPACK	Scalable Linear Algebra Package
SGG	Satellite Gravity Gradiometry
SIMD	Single Instruction - Multiple Data
SISD	Single Instruction - Single Data
SLR	Satellite Laser Ranging
SM	Shared Memory
SMP	Symmetric Multiprocessing
SPMD	Single Program - Multiple Data
SSE	Streaming SIMD Extensions
SST	Satellite to Satellite Tracking
VCE	Variance Component Estimation

Inhalt

Kapitel 1

Einleitung

1.1 Motivation und Aufgabenstellung

Das Schwerefeld der Erde ist seit einigen Jahren verstärkt zum Forschungsgegenstand geworden. Eine genaue Kenntnis des Schwerefeldes ist für viele Anwendungen von Bedeutung, besonders in der Ozeanographie, in der Geodynamik und in der Geodäsie.

Um das Erdschwerefeld in der heutzutage erforderlichen Genauigkeit bestimmen zu können, wurden seitens des GFZ, des DLR, der NASA und der ESA drei Satellitenmissionen initiiert, von denen zwei bereits erfolgreich operieren. Die Auswertung der bei diesen Missionen entstehenden Beobachtungen ist jedoch mit Schwierigkeiten verbunden. So treten bei der Bestimmung der unbekannten Schwerefeldkoeffizienten sehr große Gleichungssysteme auf. Diese sind aufgrund des großen Bedarfs an Speicherplatz und Rechenzeit nur schwierig zu handhaben.

Im Rahmen dieser Diplomarbeit sollen zwei unabhängige Algorithmen zur Bestimmung von Schwerefeldkoeffizienten auf verschiedenen Hochleistungsrechner-Architekturen, die am Höchstleistungsrechenzentrum Stuttgart zur Verfügung stehen, implementiert werden. Ziel sind portable Programme, die auch Gleichungssysteme in der maximal zu erwartenden Problemgröße in überschaubarer Zeit aufstellen und lösen können. Zusätzlich soll es möglich sein, eine Analyse des Datenmaterials bezüglich seiner Genauigkeit durchzuführen.

Es ist auf existierenden Programmen für die drei Satellitenmissionen CHAMP, GRACE und GOCE aufzubauen, welche den genannten Anforderungen entsprechend zu erweitern sind. Der Schwerpunkt soll dabei auf der parallelen Implementierung und der Effizienzsteigerung liegen.

1.2 Gliederung

Kapitel 2, „Grundlagen der Schwerefeldanalyse“, gibt einen kurzen Einblick in die Motivation für die Bestimmung des Schwerefeldes und die dazu verwendete Methodik. Es werden die drei für diese Arbeit relevanten Satellitenmissionen CHAMP, GRACE und GOCE beschrieben .

In Kapitel 3, „Grundlagen des High Performance Computing“, wird ausführlich das für das Verständnis dieser Diplomarbeit benötigte Wissen über Hard- und Software im Bereich Hochleistungsrechner vermittelt. Die verwendeten Computersysteme und Softwarepakete werden kurz vorgestellt.

Kapitel 4, „Der Brute-Force-Ansatz“, widmet sich dessen Implementierung und Parallelisierung. Neben der Einführung einer Gewichtsmatrix und der Regularisierung ist die Varianzkomponentenschätzung von besonderer Bedeutung.

Der LSQR-Algorithmus mit seinen Varianten wird in Kapitel 5 vorgestellt. Schwerpunkt bilden dabei die Parallelisierung, die Vorkonditionierung im PC-LSQR-Algorithmus und die Regularisierung.

Den Hauptteil dieser Arbeit bildet ab Seite 53 die Präsentation und Diskussion der Ergebnisse der zahlreichen durchgeführten Berechnungen. Die relevanten Punkte werden einzeln vorgestellt und in Form von Tabellen und Grafiken erläutert.

Den Abschluss bildet Kapitel 7 mit der Zusammenfassung aller Inhalte und einem Ausblick.

1.3 Danksagung

An erster Stelle möchte ich meinen Betreuern Gerrit Austen und Oliver Baur danken. Im fast täglichen Dialog mit ihnen konnte die Arbeit zielgerichtet und auf die Wünsche beider Seiten ausgerichtet bearbeitet werden. Durch die kurzen Wege und die hervorragende Kommunikation waren die vereinzelten Schwierigkeiten schnell aus dem Weg geräumt. Mein Dank gilt auch Herrn Keller, der das Projekt von Anfang an unterstützte und sich als Prüfer zur Verfügung stellte.

Ein besonderer Dank geht an die Herren Küster, Altmann und Beisel vom Höchstleistungsrechenzentrum Stuttgart und Herrn Berger von der Firma NEC. Sie waren von großer Hilfe beim Einstieg in das High Performance Computing, den Wirren der Account-Beantragung und dem Support bei auftretenden Problemen.

Das Vollenden der Diplomarbeit bedeutet gleichzeitig auch das Ende meines Studiums. Ich möchte hier all meinen Kommilitonen danken, durch die das manchmal doch sehr trockene und nicht immer einfache Studium nicht nur erträglich wurde, sondern auch eine schöne Zeit, auf die ich mit Freude zurückblicken kann.

Kapitel 2

Grundlagen der Schwerefeldanalyse

Dieses Kapitel beschreibt die für diese Diplomarbeit benötigten Grundlagen der Schwerefeldanalyse. Der Schwerpunkt liegt auf der Methodik der Schwerefeldmodellierung hinsichtlich der drei Satellitenmissionen CHAMP, GRACE und GOCE, welche die auszuwertenden Daten liefern oder liefern werden. Dieses Kapitel ist dabei relativ kompakt gehalten und geht nur auf die wichtigsten Sachverhalte ein. Eine wesentlich detailiertere Beschreibung aller Aspekte findet sich in [REUBELT, AUSTEN 2003].

2.1 Zweck der Schwerefeldbestimmung

In der Einleitung wurde bereits kurz angesprochen, welche Gebiete die wesentliche Motivation für die Schwereefeldbestimmung schaffen [ESA 2000]. Abbildung 2.1 zeigt einen Gesamtüberblick. Die drei wichtigsten Themen sollen hier noch einmal detailierter angesprochen werden.

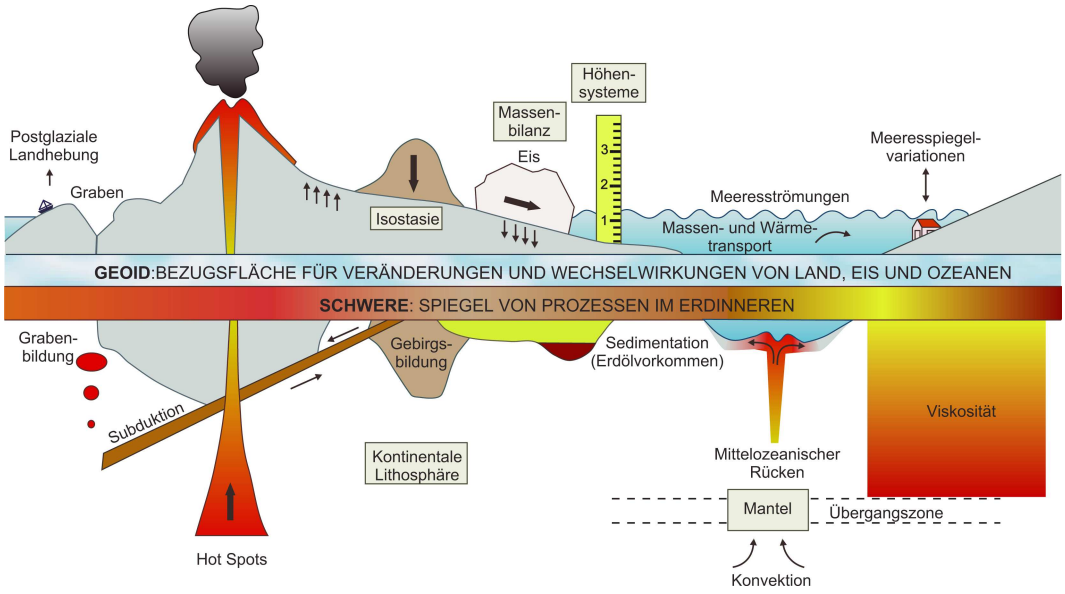


Abbildung 2.1: Bedeutung des Geoids © GOCE Projektbüro München

2.1.1 Ozeanographie

Die Wissenschaft der Ozeanographie befasst sich mit den dynamischen Vorgängen in den Weltmeeren. Von besonderer Bedeutung sind dabei die Strömungen in den Ozeanen, da durch diese gewaltige Energiemengen transportiert werden.

Strömungen äußern sich in Änderungen der Meeresoberfläche. Die Meeresoberfläche wird von Satellitenaltimetriemissionen wie z.B. TOPEX/POSEIDON auf wenige Zentimeter genau erfasst. Durch den Vergleich mit einer ruhenden Referenzfläche (dem Geoid) lässt sich daraus der Massentransport und damit die Stärke der Strömungen ermitteln. Für ein solches Vorgehen ist aber eine Geoidgenauigkeit von wenigen Zentimetern erforderlich.

Durch den großen Energietransport haben die Meeresströmungen einen ganz wesentlichen Einfluss auf das globale Klima. Die genaue Kenntnis der Meeresströmungen ist daher von großer Bedeutung für längerfristige Wettervorhersagen. Zusätzlich ist die Änderung der Meeresströmungen ein Indikator für den globalen Klimawandel. Aus diesen Gründen handelt es sich hierbei um ein sehr wichtiges Forschungsgebiet. Diese Bedeutung findet sich auch im zweiten Teil des Namens der Satellitenmission GOCE (*Steady-State Ocean Circulation Explorer*, Abschnitt 2.5) wieder.

2.1.2 Geodynamik

Die Geodynamik beleuchtet Prozesse innerhalb des Erdkörpers. Interessant sind Kenntnisse über die Zusammensetzung der kontinentalen Lithosphäre (also der Erdkruste), da sich so Rohstoffvorkommen lokalisieren lassen. Die Dynamik der ozeanischen Lithosphäre ist eng verknüpft mit tektonischen Prozessen und folglich mit der Entstehung von Erdbeben. Hinzu kommen Vorgänge wie die Reaktion der Erde auf das Verschwinden von Eislasten (was wiederum ein Indiz für den Klimawandel ist) und Konvektionsströme im Erdinneren.

All diese Zustände und Prozesse äußern sich in der Form des Geoids. Eine genaue Kenntnis des Geoids ist deshalb für die Geodynamik von großer Bedeutung.

2.1.3 Geodäsie

Das Geoid als Äquipotentialfläche des Erdschwerefeldes fungiert als Referenzfläche verschiedener Höhensysteme (Normal Null, NN). Hingegen können mit GPS ausschließlich ellipsoidische Höhen bestimmt werden, die sich auf eine rein mathematische Referenzfläche beziehen. Für die Umrechung auf schwerebezogene Höhen ist die Kenntnis der Geoidundulationen (der Differenzen zwischen Referenzellipsoid und Geoid) vonnöten, welche mit der Kenntnis des Geoids berechnet werden können.

Zu diesem Zweck werden traditionell lokale Schweremessungen verwendet. Im globalen Maßstab ergeben sich dabei jedoch beträchtliche Fehler. In Kombination mit aus Satellitenmissionen bestimmten, genauen globalen Schwerefeldern ist es möglich, flächendeckend auch über Entfernungen von einigen hundert Kilometern hinweg Höhenunterschiede mit einer Genauigkeit von wenigen Zentimetern zu bestimmen.

Heutzutage definiert sich praktisch jedes Land sein eigenes Höhensystem mit individuellem Referenzniveau. Dadurch ergeben sich beim Grenzübergang Differenzen, die bei Nichtberücksichtigung (oder falscher Berücksichtigung) zu erheblichen Problemen führen können. Durch die Kenntnis einer globalen Referenzfläche können die unterschiedlichen Höhensysteme vereinheitlicht werden.

2.2 Schwerefeldmodelle

2.2.1 Das Gravitationspotential

Die Größe, welche der Modellierung des Schwerefeldes der Erde zugrunde liegt, ist das Gravitationspotential V (Einheit $\frac{m^2}{s^2}$). Das Gravitationspotential an einem beliebigen Punkt $\mathbf{x}$ wird bestimmt durch das Integral über alle Massen:

$$V(\mathbf{x}) = G \cdot \int_B \frac{\rho(\mathbf{x}')}{|\mathbf{x} - \mathbf{x}'|} \, d\mathbf{x}' \tag{2.1}$$

Das so entstehende Potential V erfüllt außerhalb von B die Laplace-Gleichung. Laut Gleichung 2.1 benötigt man zur Berechnung des Gravitationspotentials die Kenntnis über die Dichteverteilung ρ in der gesamten Erde. Selbst wenn man diese kennen würde (was nicht der Fall ist), so wäre dies bei vernünftiger Auflösung eine ungeheure Datenmenge.

Es wird deshalb ein anderer Ansatz gewählt. Man macht sich dabei zu nutze, dass das Gravitationspotential keineswegs sprunghafte Änderungen, sondern nur geringe Variationen aufweist.

Funktionen, welche die LAPLACE-Gleichung erfüllen, können in eine Reihe entwickelt werden (analog zur Entwicklung von reelen periodischen Funktionen einer Variablen in eine Fourierreihe). Dies führt auf die Kugelfunktionsentwicklung des Gravitationspotentials. Gleichung 2.2 zeigt diese Reihe für den Außenraum der Erde:

$$V = \frac{GM}{r} + \frac{GM}{R} \sum_{k=2}^{\infty} \sum_{l=0}^{k} \left(\frac{R}{r}\right)^{k+1} \overline{P}_{k,l}(\sin\varphi)(\overline{c}_{k,l}\cos(l\lambda) + \overline{s}_{k,l}\sin(l\lambda)) \tag{2.2}$$

Dabei ist GM die geozentrische Konstante (das Produkt von Gravitationskonstante und Erdmasse), R der Erdradius und r der Abstand des Punktes, an dem das Gravitationspotential berechnet werden soll, vom Massenmittelpunkt der Erde. λ und φ sind die geografische Länge und Breite des Punktes.

$\overline{c}_{k,l}$ und $\overline{s}_{k,l}$ sind die gesuchten Schwerefeldkoeffizienten, $\overline{P}_{k,l}(\sin\varphi)$ sind die normierten Legendreschen Funktionen.

Grad und Ordnung der Reihenentwicklung werden dabei durch k und l angegeben. Je höher die Reihe entwickelt wird, desto besser nähert sie den tatsächlichen Verlauf des Gravitationspotentials an. Einen Satz von Schwerefeldkoeffizienten bezeichnet man als Schwerefeldmodell.

2.2.2 Bestimmung der Schwerefeldkoeffizienten

Informationen über das Schwerefeld können aus mehreren Quellen gewonnen werden. In der Vergangenheit wurden die langwelligen Anteile aus der Satellitenaltimetrie und der Beobachtung von Satellitenbahnen abgeleitet, während die kurzwelligen Anteile durch Gravimeter-Messungen bestimmt wurden. Auf diese Weise wurden z.B. das EGM96 gewonnen.

Heute wird die Gewinnung globaler Informationen aus niedrigfliegenden Satellitenmissionen (low earth orbiter - LEO) favorisiert. Dadurch können konsistente Modelle mit einer engmaschigen räumlichen Abdeckung (im EGM96 ist z.B. die Antarktis nur unzureichend erfasst) gewonnen werden. Um die gewünschte hohe Auflösung der Schwerefeldmodelle zu erhalten, wurden drei Satellitenmissionen aus der Taufe gehoben, die in den folgenden Abschnitten beschrieben werden.

2.2.3 Ausgleichungsansatz

Die Satellitenbeobachtungen $\mathbf{y}$ stehen in einem linearen Zusammenhang mit den unbekannten Schwerefeldkoeffizienten, welche hier im Vektor $\mathbf{x}$ zusammengefasst sind. Die Designmatrix $\mathbf{A}$

beschreibt den funktionalen Zusammenhang zwischen Unbekannten und Beobachtungen. Die Messfehler sind durch den Vektor $\mathbf{v}$ repräsentiert:

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{v} \tag{2.3}$$

Zur Lösung des linearen Gleichungssystems verwendet man einen L_2-Norm-Schätzer. Der entsprechende Algorithmus, bekannt als „Methode der kleinsten Quadrate“, ergibt sich nach GAUSS zu

$$\widehat{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}\mathbf{y}. \tag{2.4}$$

Erweitert um eine Gewichtung der verschiedenen Beobachtungen ergibt sich der Algorithmus nach GAUSS und MARKOV:

$$\widehat{\mathbf{x}} = (\mathbf{A}^T\mathbf{P}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{P}\mathbf{y} = \mathbf{N}^{-1}\mathbf{b} \tag{2.5}$$

$\widehat{\mathbf{x}}$ ist eine erwartungstreue Schätzung der unbekannten Parameter $\mathbf{x}$. Dieses Gleichungssystem muss aufgestellt und gelöst werden.

2.3 Die Satellitenmission CHAMP

2.3.1 Beschreibung

Der Satellit CHAMP (*Challenging Minisatellite Payload for Geophysical Research and Application*, Abbildung 2.2) dient vorrangig der Erforschung des Schwerefelds und des Magnetfelds der Erde, sowie der Atmosphäre und der Ionosphäre. Für diese Arbeit ist nur der erste Auftrag von Bedeutung. Mehr Informationen finden sich im WWW auf der Seite des GFZ unter `http://op.gfz-potsdam.de/champ/`.

Wie der Name bereits ausdrückt, handelt es sich bei CHAMP um einen kleinen Satelliten von lediglich 4 m Länge (8 m mit Ausleger) und 522 kg Masse. Dadurch waren die Kosten für Bau und Start relativ gering. Andererseits ist natürlich die Nutzlastkapazität begrenzt. An Bord befinden sich Magnetometer und Ionendriftmeter zur Erforschung des Magnet- und des elektrischen Felds der Erde. Nichtgravitative Beschleunigungen werden durch ein Akzelerometer erfasst. Die Orientierung des Akzelerometers und der Magnetometer wird durch jeweils zwei Sternenkameras ermittelt.

Von besonderer Bedeutung für die Ermittlung des Einflusses des Schwerefeldes ist die Bestimmung der Bahn des Satelliten. Dies geschieht bei CHAMP durch GPS (sogenanntes high-low satellite to satellite tracking, HL-SST) und einen Laser-Retroreflektor für SLR (satellite laser ranging).

CHAMP wurde am 15. Juli 2000 gestartet. Die Missionsdauer war ursprünglich auf fünf Jahre angesetzt, mittlerweile werden jedoch acht Jahre erwartet. Der Satellit liefert zuverlässig Daten, die vom Geoforschungszentrum Potsdam verarbeitet und zur Verfügung gestellt werden.

Aus den CHAMP-Beobachtungen lassen sich in erster Linie langwellige Schwerefeldmodelle bestimmen. Die momentan verfügbaren aus CHAMP-Messungen abgeleiteteten Schwerefelder sind vollständig bis Grad und Ordnung 120 und für einige Koeffizienzen bis Grad und Ordnung 140 entwickelt, allerdings mit geringer Genauigkeit für die Koeffizienten jenseits von Grad und Ordnung 70. Da der Satellit im Lauf seiner Mission weiter absinkt und damit die Sensitivität für Schwereeinflüsse steigt, ist davon auszugehen, dass der Entwicklungsgrad noch leicht gesteigert werden kann.

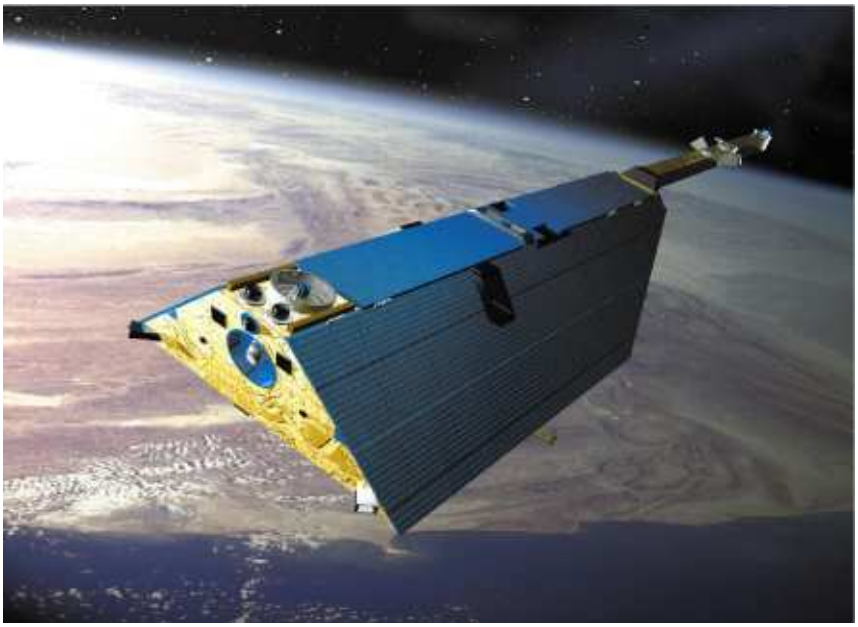


Abbildung 2.2: Der Satellit CHAMP © Astrium

2.3.2 Beobachtungsmodell

Beobachtet werden die Positionen $\mathbf{x}(t)$ des Satelliten. Durch zweimalige numerische Differentiation können daraus die auf den Satelliten wirkenden Beschleunigungen $\ddot{\mathbf{x}}$ berechnet werden. Dabei entsprechen die Beschleunigungen gerade dem Gradienten des Gravitationspotentials.
Es ergibt sich die Beobachtungsgleichung:

$$\ddot{\mathbf{x}} = grad\, V \tag{2.6}$$

Der Gradientenoperator in Kugelkoordinaten ist definiert als

$$grad_{(\lambda,\varphi,r)} = \mathbf{e}_r \frac{\partial}{\partial r} + \mathbf{e}_\lambda \frac{1}{r\cos\lambda}\frac{\partial}{\partial \lambda} + \mathbf{e}_\varphi \frac{1}{r}\frac{\partial}{\partial \varphi} \tag{2.7}$$

Der Gradient ergibt sich mit der Anwendung dieses Gradientenoperators auf Gleichung 2.2 zu:

$$\begin{aligned} grad_{(\lambda,\varphi,r)}\, V \;=\; & \frac{GM}{R^2}\sum_{k=0}^{\infty}\sum_{l=0}^{k}\left(\frac{R}{r}\right)^{k+2} \Big[-(k+1)\overline{P}_{k,l}(\sin\varphi)\left(\overline{c}_{k,l}\cos(l\lambda)+\overline{s}_{k,l}\sin(l\lambda)\right)\mathbf{e}_r \\ & + \frac{1}{\cos\varphi}\overline{P}_{k,l}(\sin\varphi)\left(-\overline{c}_{k,l}\, l\sin(l\lambda)+\overline{s}_{k,l}\, l\cos(l\lambda)\right)\mathbf{e}_\lambda \\ & + \frac{\partial \overline{P}_{k,l}(\sin\varphi)}{\partial\varphi}\left(\overline{c}_{k,l}\cos(l\lambda)+\overline{s}_{k,l}\sin(l\lambda)\right)\mathbf{e}_\varphi\Big] \end{aligned} \tag{2.8}$$

2.4 Die Satellitenmission GRACE

2.4.1 Beschreibung

GRACE (*Gravity Recovery and Climate Experiment*, Abbildung 2.3) ist eine Erweiterung der CHAMP-Mission. Sie dient noch konsequenter der Bestimmung des Erdschwerefeldes und vor allen Dingen dessen zeitlicher Variation. Zu diesem Zweck fliegen zwei Satelliten im Abstand von etwa 220 km auf der gleichen Bahn. Die Entfernungsänderung zwischen den Satelliten wird dabei permanent über Mikrowellen bestimmt (low-low satellite to satellite tracking, LL-SST). Die Entfernungsänderung kann mit einer Genauigkeit von ~1 μm/s bestimmt werden.

Die beiden Satelliten, die CHAMP ähneln und jeweils etwa 500 kg wiegen, sind bis auf die Funkfrequenzen für den Mikrowellen-Entfernungsmesser und den Datenlink zur Bodenstation identisch aufgebaut. Beide sind mit einem Akzelerometer, einem GPS-Empfänger und einem Laser-Retroreflektor ausgestattet. Zur Bestimmung der Orientierung besitzen die Satelliten jeweils zwei Sternenkameras und einen einfachen Orientierungssensor, der die Erwärmung des Satelliten und damit die Orientierung gegenüber der Sonne misst.

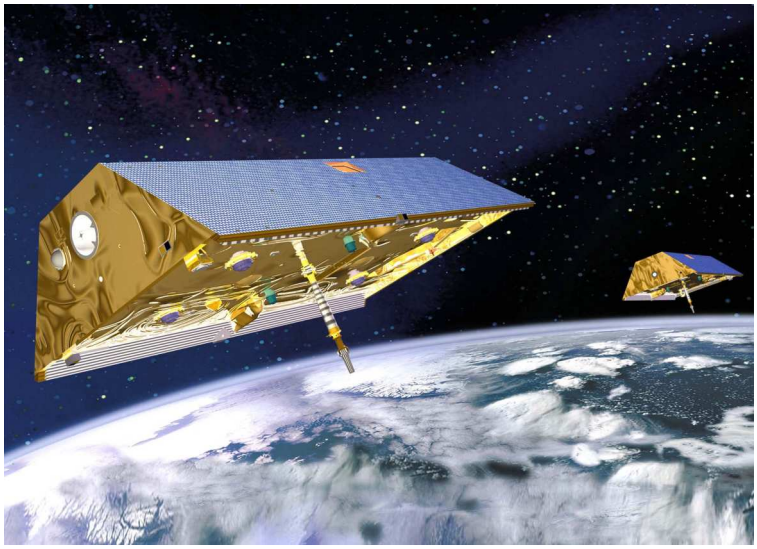


Abbildung 2.3: Die Satelliten GRACE 1 und 2 © Astrium

Die beiden GRACE-Satelliten wurden am 17.3.2002 in ihren Orbit gebracht. Die Missionsdauer ist wie bei CHAMP auf fünf Jahre angesetzt. Die Daten werden vom Jet Propulsion Laboratory (JPL) der NASA und vom GFZ Potsdam ausgewertet.

Das zur Zeit verfügbare vom GFZ Potsdam bestimmte Schwerefeld ist bis Grad und Ordnung 120 und für einige Koeffizienten bis 140 aufgelöst. Für GRACE wird eine Auflösung bis über Grad 150 erwartet, wobei die Genauigkeit der langwelligen Anteile deutlich höher als bei CHAMP sein wird. Das erste Schwerefeld wurde aus lediglich 39 Tagen Messungen hergeleitet und weist bereits eine fünfmal höhere Genauigkeit als das bisherige CHAMP-Schwerefeld auf.

Mehr Informationen findet man auch hier auf den Webseiten des GFZ unter
`http://op.gfz-potsdam.de/grace/index_GRACE.html`.

2.4.2 Beobachtungsmodell

Beobachtet wird die Entfernungsänderung zwischen den beiden Satelliten $\dot{\rho}(t)$. Aus dieser resultiert die Relativbeschleunigung $\ddot{\rho}(t)$ zwischen den Satelliten. Diese ergibt sich aus der Differenz der Beschleunigung zwischen den beiden Satelliten. Die Differenz der Beschleunigung ist wiederum die Differenz des Gradienten des Gravitationspotentials, und zwar gerade in Richtung der beiden Satelliten. Es ergibt sich also folgende Beobachtungsgleichung (in kartesischen Koordinaten):

$$\ddot{\rho}(t) - \dot{\mathbf{e}}^T(t) \cdot \Delta\dot{\mathbf{x}}_{12}(t) = \mathbf{e}^T(t) \cdot \mathbf{R}_{\mathbf{xx}}^T(t) \left[grad_{\mathbf{x}_2} V(\mathbf{x}_2(t)) - grad_{\mathbf{x}_1} V(\mathbf{x}_1(t)) \right]. \tag{2.9}$$

Dabei ist $\mathbf{R}_{\mathbf{xx}}(t)$ die Rotationsmatrix zum Übergang auf sphärische Koordinaten. $\mathbf{e}(t)$ ist der Normaleneinheitsvektor zwischen den beiden Satelliten und berechnet sich folgendermaßen:

$$\mathbf{e}(t) = \frac{\mathbf{x}_2(t) - \mathbf{x}_1(t)}{\|\mathbf{x}_2(t) - \mathbf{x}_1(t)\|} = \frac{\Delta\mathbf{x}_{12}(t)}{\|\Delta\mathbf{x}_{12}(t)\|} \tag{2.10}$$

$\Delta\mathbf{x}_{12}(t)$ ist die Differenz zwischen den beiden Satellitenpositionen zum Zeitpunkt t.

2.5 Die Satellitenmission GOCE

2.5.1 Beschreibung

Die letzte im Trio der Schwerefeld-Satellitenmissionen ist GOCE (*Gravity Field and Steady-State Ocean Circulation Explorer*, Abbildung 2.4). Ziel ist die Gewinnung eines im Vergleich zu den Vorgängermissionen noch höher aufgelösten und genaueren Schwerefeldmodells. Zu diesem Zweck wird GOCE mit einem Gradiometer, einem GPS-Empfänger und einem Laser-Retroreflektor ausgestattet. Beim Gradiometer handelt es sich um eine sehr sensitive Konfiguration aus sechs Akzelerometern, deren differentielle Messungen den zweiten Ableitungen des Gravitationspotentials entsprechen (satellite gravity gradiometry - SGG).

Um auch kurzwellige Anteile des Schwerefeldes erfassen zu können, wird der Satellit auf einer geringen Bahnhöhe von nur 250 km fliegen (zum Vergleich: CHAMP und GRACE haben eine Anfangsbahnhöhe von ca. 450 km und werden bei Erreichen einer Bahnhöhe von 250 km zum Ende ihrer Mission schnell abstürzen). Um den Einfluß der Atmosphärenreibung zu kompensieren, die sowohl als unerwünschte Störbeschleunigung auftritt, als auch für das Absinken der Satelliten verantwortlich ist, wird GOCE mit einem Ionenstrahltriebwerk ausgestattet. Zusätzlich ist der Satellit aerodynamisch günstig geformt.

Durch das Ionenstrahltriebwerk hat GOCE ein großen Energiebedarf, weshalb der auf einer sonnensynchronen Bahn fliegen wird. CHAMP und GRACE dagegen fliegen nicht auf sonnensynchronen Bahnen und können dadurch auch Einflüsse von Tag- und Nachtunterschieden ermitteln.

Der Start von GOCE ist für 2006 geplant. Aufgrund der geringen Flughöhe und dem damit verbundenen hohen Treibstoffbedarf des Satelliten wird die Missionsdauer lediglich 20 Monate betragen.

Erwartet wird ein statisches Schwerefeldmodell, das bis Grad und Ordnung 250 aufgelöst ist, eventuell bis Grad und Ordnung 300. Dies markiert damit gleichzeitig auch die Obergrenze dessen, was aus rechentechnischer Sicht als Problemgröße zu erwarten ist. Weitere Details zu GOCE gibt es auf der Webseite des GOCE Projektbüro unter `http://www.goce-projektbuero.de/`.

Abbildung 2.4: Der Satellit GOCE © GOCE Projektbüro München

2.5.2 Beobachtungsmodell

Beobachtet werden die zweiten Ableitungen des Gravitationspotentials, diese werden im sogenannten Gravitationstensor $\mathbf{V}$ zusammengefasst. Es ergibt sich die Beobachtungsgleichung

$$\mathbf{V}(\lambda,\varphi,r) = V_{ij}\mathbf{e}_i \otimes \mathbf{e}_j = \begin{bmatrix} V_{11} & V_{12} & V_{13} \\ V_{12} & V_{22} & V_{23} \\ V_{13} & V_{23} & V_{33} \end{bmatrix} (\mathbf{e}_i \otimes \mathbf{e}_j) = grad_{(\lambda,\varphi,r)} \otimes grad_{(\lambda,\varphi,r)} V(\lambda,\varphi,r)$$

$$= grad_{(\lambda,\varphi,r)} \otimes grad_{(\lambda,\varphi,r)} \left[\frac{GM}{R} \sum_{k=0}^{\infty} \sum_{l=0}^{k} \left(\frac{R}{r} \right)^{k+1} \overline{P}_{k,l}(sin\varphi)(\overline{c}_{k,l} cos(l\lambda) + \overline{s}_{k,l} sin(l\lambda)) \right] \quad (2.11)$$

mit $i,j = \{\lambda,\varphi,r\}$. Wie zu sehen ist, wird auf Gleichung 2.8 nochmals der Gradientenoperator (Gleichung 2.7) angewendet.

Kapitel 3

Grundlagen des High Performance Computing

Dieses Kapitel erläutert die für diese Diplomarbeit benötigten Grundlagen des High Performance Computing. Der Text wird durch vereinfachte, erläuternde Abbildungen ergänzt. Für detailiertere Informationen sei auf die zahlreich vorhandene Literatur verwiesen. Einen sehr guten Überblick, besonders auch über die Programmierung, bietet [DOWD 1998]. Ebenfalls vorgestellt werden die Hochleistungsrechner, die im Rahmen dieser Diplomarbeit zum Einsatz kamen.

3.1 Was ist ein Hochleistungsrechner?

Als Hochleistungsrechner (was High Performance Computer ja bedeutet) werden sehr leistungsfähige Computer bezeichnet, die (im Gegensatz zu Servern) für wissenschaftliche Berechnungen verwendet werden, primär zur Simulation von verschiedensten Prozessen: Wettervorhersage, Strömungen an Flugzeugen, Fahrzeugen und in Motoren (*Computational Fluid Dynamics* - CFD), Crash- und Belastungstests (*Finite Element Modeling* - FEM) und vieles mehr (Medizin, Physik, Astronomie...). Drei Beispiele (Globale Wettersimulation, Strömung an einem Formel 1-Rennwagen, Crashtest eines PKW) sind in Abbildung 3.1 dargestellt.

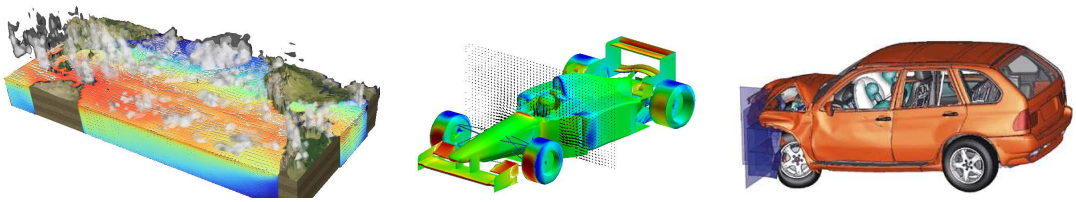


Abbildung 3.1: Beispiele für Simulationen auf Hochleistungsrechnern

Die schnellsten Hochleistungsrechner werden als Supercomputer bezeichnet. Um diese außerordentliche Leistungsfähigkeit zu erreichen, gibt es mehrere Wege. Diese sollen in diesem Kapitel vorgestellt und erläutert werden.

3.1.1 FLOPS

Als Maß für die Geschwindigkeit von Supercomputern werden häufig *FLOPS* verwendet. Diese Abkürzung steht für *floating point operations per second*, gibt also an, wie viele Fließkommaoperationen ein Computer pro Sekunde durchführen kann. Die schnellsten Supercomputer sind mittlerweile in den Teraflop-Bereich (10^{12} FLOPS) vorgedrungen. Diese sogenannte Peak-Leistung ist jedoch mit Vorsicht zu genießen - die tatsächlich erzielbare Leistung ist oft deutlich geringer. Man spricht häufig von 50% erzielbarer Leistung bei Vektorrechnern und 5% bei Skalarrechnern. So schreibt [SCHÖNAUER 2000] von bis zu 69% Effizienz auf einer Cray T90 (einem Vektorrechner). Bei der gleichen Operation auf einer Cray T3E (einem massiv-parallelen System mit Alpha-Skalarprozessoren) ermittelte er hingegen eine Effizienz von nur 2,4%.

3.1.2 Die Top 500-Liste

Zweimal pro Jahr erscheint die Top 500-Liste, in der die weltweit 500 schnellsten Supercomputer (nach dem LINPACK-Benchmark) verzeichnet sind. Diese Liste zeichnet ein interessantes Bild der momentanen Situation im Supercomputing-Bereich, und über die älteren Listen lassen sich Rückschlüsse auf die Entwicklung in den letzten Jahren ziehen. Die Top 500-Liste ist on-line unter `http://www.top500.org` einsehbar.

Bereits seit 2002 wird die Spitzenposition vom Earth Simulator in Yokohama, Japan (`http://www.es.jamstec.go.jp/`, Abbildung 3.2) gehalten. Dieser aus 640 Knoten aufgebaute Supercomputer, welcher auf der SX-6 von NEC basiert, erzielt eine Spitzenleistung von fast 40 TFLOPS.

Abbildung 3.2: Der Earth Simulator © The Earth Simulator Center

3.2 Systemarchitekturen

Ein System zur Unterscheidung der Systemarchitekturen von Computern wurde in [FLYNN 1972] präsentiert. Auch über 30 Jahre später hat es prinzipiell noch Gültigkeit und wird in fast jedem Buch zum Thema High Performance Computing zitiert. So soll es auch hier (mit Erweiterungen) herangezogen werden.

3.2.1 Singe Instruction - Single Data (SISD)

Der einfachste Typ eines Computers führt pro Taktzyklus eine Instruktion aus (z.B. Auslesen eines Speicherbereichs, Addition zweier Werte), und dies mit lediglich einem Datensatz oder Operanden (im Beispiel also einer Speicheradresse, oder einem Paar von Zahlen). Man spricht dann von einem *Skalarrechner*.

Abbildung 3.3: Ablauf bei der Addition zweier Zahlen

Abbildung 3.3 zeigt in vereinfachter Weise, wie in einem Skalarrechner die Addition zweier Zahlen abläuft. Für den kompletten Ablauf werden fünf Schritte benötigt - davon nur einer für die tatsächliche Addition. Um n Zahlenpaare zu addieren, wären also $n \cdot 5$ Schritte notwendig. Noch aufwändiger wird diese Tatsache dadurch, dass in der Realität jeder dieser Schritte aus mehreren Taktzyklen besteht.

Die Lösung für dieses Problem der ineffizienten Verwendung von Rechenleistung heißt *Pipeline*. Wenn für jeden der fünf benötigten Schritte eine eigene Funktionseinheit zur Verfügung steht, benötigt die Berechnung der Addition zwar immer noch fünf Schritte, aber bei gleichzeitiger Auslastung aller Funktionseinheiten, die für jede Addition der Reihe nach durchlaufen werden, liefert jeder Schritt ein Ergebnis. Für die Addition von n Zahlenpaaren sind dann nur noch $(n\text{-}1)+5$ Schritte notwendig. Abbildung 3.4 zeigt schematisch die Addition in einer Pipeline.

Da in der Realität die Ausführung der meisten Instruktionen mehr als fünf Schritte erfordert, werden auch die Pipelines länger gewählt. Auch für das Erzielen hoher Taktfrequenzen sind lange Pipelines von Vorteil. Dadurch ergibt sich jedoch ein neues Problem. Findet in den Instruktionen plötzlich ein Sprung statt (z.B. durch eine if-Anweisung), so muss die Pipeline geleert und erneut gefüllt werden, und es vergeht eine Anzahl von Schritten entsprechend der Länge der Pipeline, bis diese wieder Ergebnisse liefert. Es ist also darauf zu achten, dass möglichst keine Sprünge auftreten, um die Pipeline gefüllt zu halten.

Die Leistungsfähigkeit eines Prozessors kann weiter gesteigert werden, wenn er mehrere Pipelines hat. Man spricht dann von einem *superskalaren* Prozessor. Üblich ist die Trennung von Festkomma- und Logikrechnung (*ALU* - Arithmetic/Logical Unit) und Fließkommarechnung (*FPU* - Floating Point Unit). Die FPU ist in der Regel weiter aufgeteilt in eine Einheit für die Addition und eine für die Multiplikation. Diese Einheiten sind manchmal auch mehrfach vorhanden, einige Prozessoren besitzen auch zusätzliche Einheiten für Division und Berechnung von Quadratwurzeln.

Um die zusätzlichen Pipelines ausnutzen zu können, müssen diese jedoch gleichzeitig ausgelastet sein. Hier ist also schon *Parallelisierung* notwendig, ein Konzept, das in 3.2.3 erläutert wird.

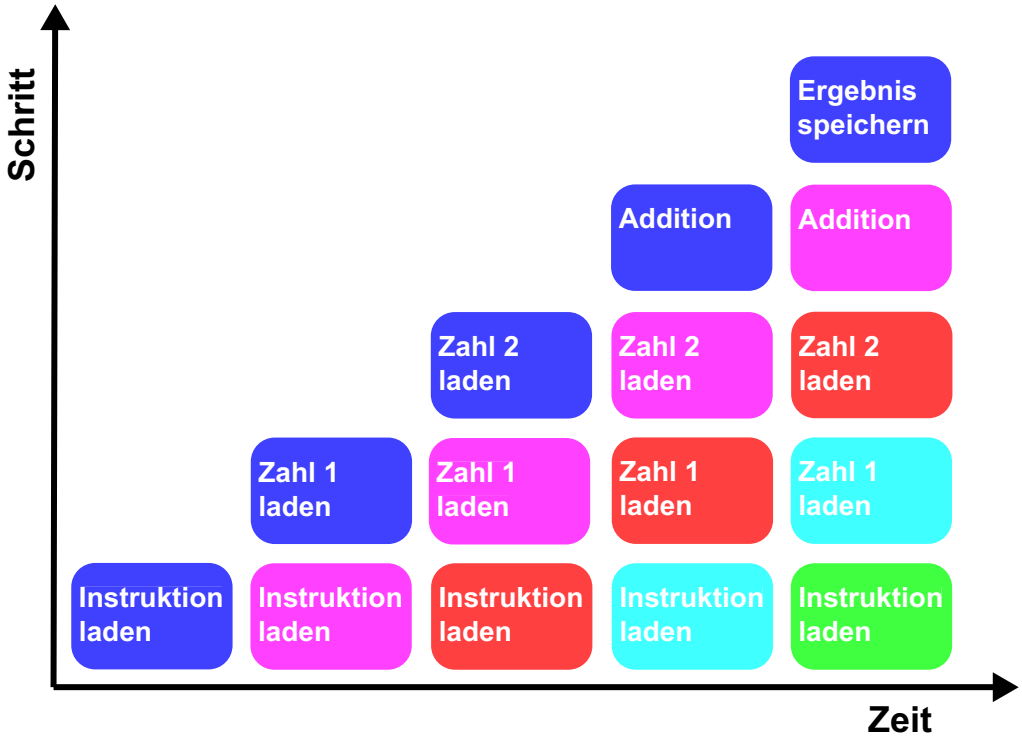


Abbildung 3.4: Addition zweier Zahlen in einer Pipeline

3.2.2 Single Instruction - Multiple Data (SIMD)

Der im vorigen Abschnitt betrachtete Skalarrechner führt eine Instruktion immer nur mit einem Datensatz aus. Bei wissenschaftlichen Berechnungen hat man jedoch oft mit einer großen Anzahl von Datensätzen zu tun, auf die die jeweils gleiche Operation (die gleiche Instruktion) ausgeführt werden soll. Einen Rechner, der eine Instruktion mit mehreren Datensätzen durchführt, bezeichnet man als *Vektorrechner*.

In Abbildung 3.5 ist vereinfacht dargestellt, wie die Addition zweier Vektoren in einer Vektor-Pipeline erfolgt. Die Abbildung entspricht Abbildung 3.4 mit einem Unterschied - es werden nicht einzelne Zahlen, sondern komplette Vektoren verarbeitet.

Die Anzahl der Zahlen (der Vektorelemente), die gleichzeitig verarbeitet werden kann, ist begrenzt. Trotzdem macht ein Vektorrechner deutliche Leistungssteigerungen möglich. Ein Vektorprozessor, der gleichzeitig 64 Vektorelemente verarbeiten kann, liefert damit auch 64 Ergebnisse pro Schritt. Der Skalarrechner aus dem vorherigen Abschnitt würde dazu mindestens 64 Schritte benötigen.

Um die theoretische Leistungsfähigkeit eines Vektorrechners ausnutzen zu können, müssen die Berechnungen dementsprechend *vektorisiert* werden. Wenn ein Vektorprozessor nur mit einzelnen Zahlen gefüttert wird, kann er seine Leistung nicht ausspielen, und wie beim Skalarrechner ist darauf zu achten, dass die Pipelines (auch Vektorprozessoren arbeiten mit mehreren Pipelines, die wie bei superskalaren Prozessoren unterschiedliche Aufgaben haben) gefüllt bleiben.

Ende der 90er Jahre des vergangengen Jahrhunderts wurde, bedingt durch die massiven Leistungssteigerungen bei kostengünstigen Mikroprozessoren, das Ende des Vektorrechners postuliert. Tatsächlich lässt sich die Leistungsfähigkeit eines Vektorrechners aber wesentlich effizienter nutzen, was im nächsten Abschnitt erläutert wird. Interessanterweise beinhalten moderne Mikroprozessoren auch Vektoreinheiten (*SSE* - Streaming SIMD Extensions in Intel und AMD

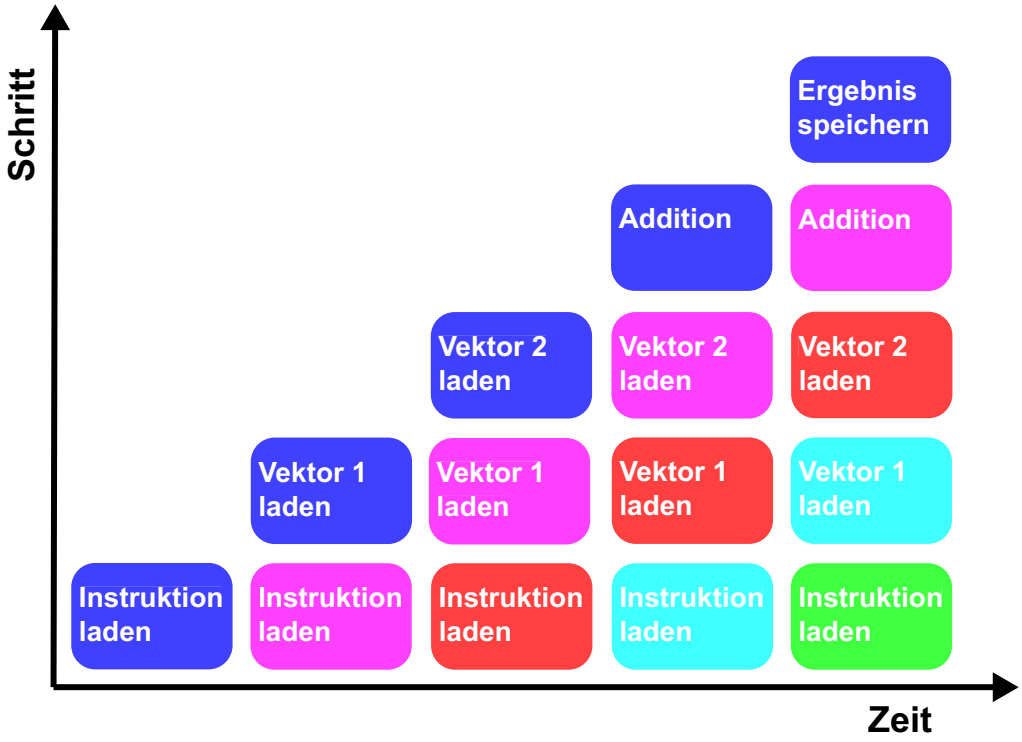


Abbildung 3.5: Addition zweier Vektoren in einer Vektor-Pipeline

Prozessoren sowie *AltiVec* in Motorola/IBM PowerPC Prozessoren), die zum Teil gewaltige Leistungssteigerungen bei bestimmten Anwendungen ermöglichen.

3.2.3 Multiple Instruction - Multiple Data (MIMD)

Bisher wurden lediglich Systeme betrachtet, die pro Taktzyklus eine Instruktion verarbeiten, was Systemen mit nur einem Prozessor (*CPU* - Central Processing Unit) entspricht. Kombiniert man mehrere dieser Prozessoren (egal ob Skalar- oder Vektorprozessoren), erhält man ein System, das pro Taktzyklus mehrere Instruktionen und Datensätze verarbeiten kann. Da praktisch alle Supercomputer (und in nicht allzu ferner Zukunft wohl alle Computer) diesem Typ angehören, soll diese Architektur in den folgenden Abschnitten etwas ausführlicher erläutert werden. MIMD-Systeme unterscheiden sich in erster Linie hinsichtlich des Speichersystems.

3.2.3.1 Parallelisierung

Bevor die unterschiedlichen Speicherarchitekturen erläutert werden, soll jedoch kurz das Konzept der Parallelisierung erklärt werden. Dazu bleiben wir bei dem im vorigen Abschnitt erwähnten Beispiel der Vektoraddition.

Statt mehrere Werte gleichzeitig in einem Prozessor zu addieren, ist auch ein anderes Vorgehen denkbar: Der Vektor wird in mehrere Teile zerlegt und auf die verfügbaren Prozessoren verteilt. Diese berechnen *parallel* jeweils die Addition für ihren Teilvektor und geben das Ergebnis am Ende zurück, woraus sich dann der komplette Ergebnisvektor rekonstruieren läßt. In der Realität wird man dieses Vorgehen jedoch nicht wählen.

Das Problem besteht darin, dass für die Kommunikation zwischen den Prozessoren (Zerlegung und Verteilung des Urvektors, Zusammensetzen des Ergebnisvektors) in den meisten Fällen we-

sentlich mehr Zeit benötigt wird, als für die eigentliche Berechnung. Man zerlegt das Problem deshalb an anderer Stelle. Wenn man z.B. viermal eine Vektoraddition zu berechnen hätte, so würde man diese Vektoren im Ganzen auf die einzelnen Prozessoren verteilen, um das Verhältnis von Rechenzeit zu Kommunikationszeit besser zu gestalten.

In der Praxis ist es wesentlich schwieriger, ein Programm geschickt zu parallelisieren, als es zu vektorisieren. Besonders der Zeitverlust durch Kommunikation kann beachtliche Ausmaße annehmen. Aus diesem Grund erreichen Vektorrechner bei gut vektorisierbaren Problemen eine wesentlich höhere Effizienz als Systeme aus vielen Skalarprozessoren.

3.2.3.2 MISD

Der aufmerksame Leser hat vielleicht festgestellt, dass eine Systemarchitektur in der Aufzählung fehlt: *Multiple Instruction - Single Data* (MISD). Das liegt daran, dass ein solcher Computer weder theoretisch noch praktisch sinnvoll realisierbar ist. OPENSHAW und TURTON schreiben [OPENSHAW 1999]: „We found it hard to figure out why you would want to do this (die gleichzeitige Manipulation eines Datensatzes mit mehreren Operationen) unless you are a computer scientist interested in weird computing! It is a highly specialised and seemingly a very restrictive form of parallelism that is often impractical, not to mention useless, as the basis for a general-purpose machine.“

3.2.4 Shared Memory

MIMD-Systeme mit gemeinsamem Speicher (shared memory) werden kurz als SM-MIMD Systeme bezeichnet. Alle Prozessoren sind über eine Verbindung an den gemeinsamen Hauptspeicher (*RAM* - Random Access Memory) angeschlossen (Abbildung 3.6). In der Regel sind alle Prozessoren identisch und haben gleichberechtigen Zugriff auf den Hauptspeicher. Man spricht dann von *Symmetric Multiprocessing* (SMP).

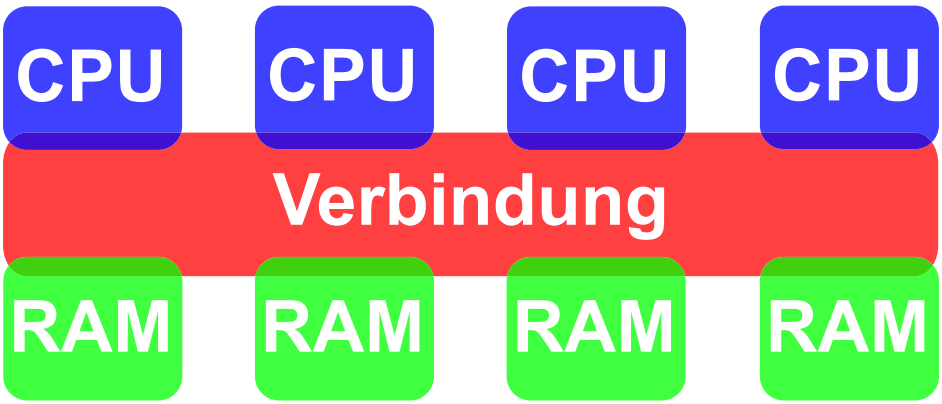


Abbildung 3.6: Schematischer Aufbau eines Shared Memory-Systems

Von wesentlicher Bedeutung ist die Verbindung zwischen Prozessoren und Speicher. In Abbildung 3.7 ist ein Shared Memory-System dargestellt, in dem die Verbindung über einen Bus erfolgt. Ein Bus hat den Vorteil, dass er sich sehr leicht erweitern lässt. Nachteilig ist, dass sich alle angeschlossenen Prozessoren die zur Verfügung stehende Bandbreite teilen müssen, auch wenn sie auf unterschiedliche Speichermodule zugreifen. Bus-Systeme sind besonders in Desktop-Rechnern (Frontside-Bus) vorhanden.

Um das Problem der begrenzten Bandbreite zu umgehen, ist eine direkte Verbindung eines jeden Prozessors mit jedem Speichermodul wünschenswert. Dies erreicht man durch einen so-

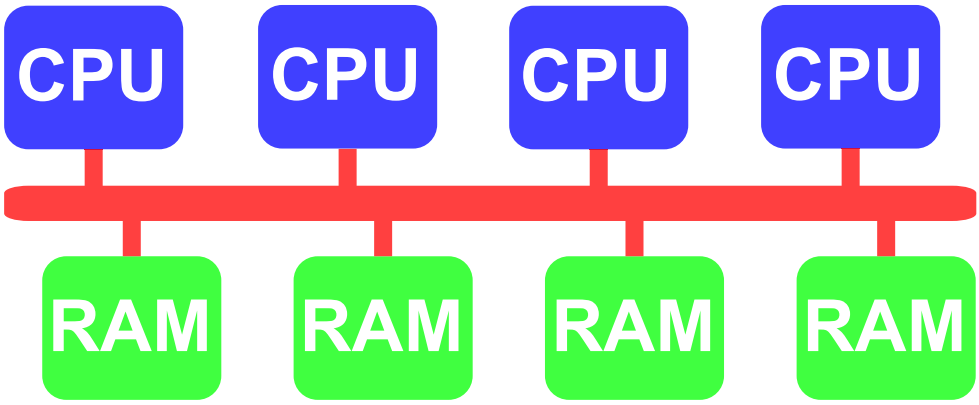


Abbildung 3.7: Shared Memory-System mit Bus

genannten Crossbar-Switch (Abbildung 3.8). Crossbar-Switches finden sich in leistungsfähigen Workstations sowie in Supercomputern.

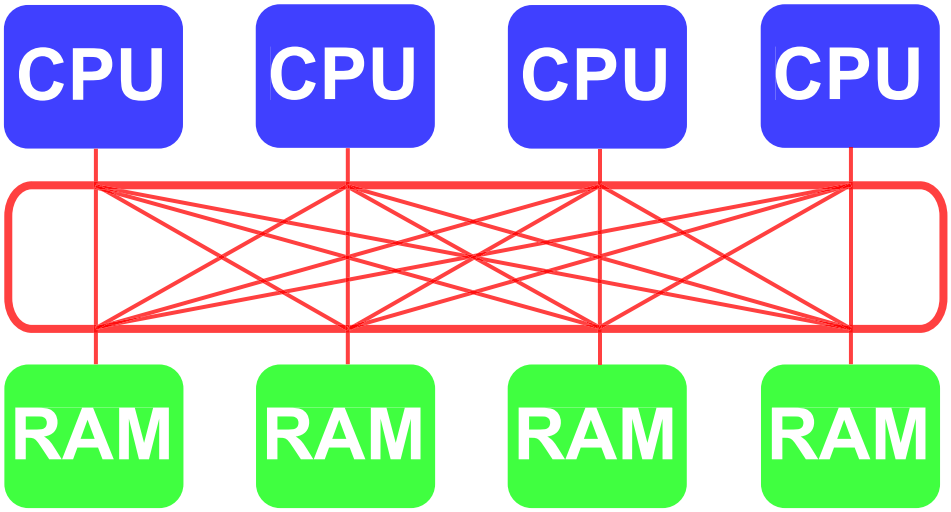


Abbildung 3.8: Shared Memory-System mit Crossbar-Switch

Das Problem bei Crossbar-Switches besteht darin, dass diese mit ansteigender Zahl der benötigten Verbindungen immer komplexer werden. Man verwendet deshalb mehrstufige Crossbar-Switches, was jedoch wieder längere Kommunikationszeiten bedeutet. Aus diesem Grund ist in der Praxis die maximal mögliche Anzahl von Prozessoren und Speichermodulen, die über einen Crossbar-Switch verbunden werden können, begrenzt.

Shared Memory-Systeme haben den Vorteil, dass alle Prozessoren auf den gleichen Speicher zugreifen können. Dadurch sind sie einfach zu programmieren und effizient einsetzbar. Begrenzender Faktor für ihre Leistungsfähigkeit ist die maximale Anzahl von Prozessoren und Speichermodulen, die noch effizient miteinander verschaltet werden können. In der Regel arbeiten Shared Memory-Systeme nur mit wenigen Prozessoren.

3.2.5 Distributed Memory

Wie im vorigen Abschnitt zu sehen war, kann die Anzahl von Prozessoren und Speichermodulen in einem Shared Memory-System nicht beliebig gesteigert werden. Ein zweiter Weg ist deshalb die Verteilung des Hauptspeichers (Distributed Memory). MIMD-Systeme mit verteiltem Hauptspeicher werden auch als DM-MIMD Systeme bezeichnet.

Bei Distributed Memory-Systemen wird jeder Prozessor mit seinem eigenen lokalen Speicher verschaltet. Die Prozessoren sind wiederum untereinander verbunden (Abbildung 3.9). An das Verbindungsnetzwerk werden hier nicht ganz so hohe Anforderungen gestellt wie bei Shared Memory-Systemen, da die Kommunikation zwischen den Prozessoren langsamer erfolgen kann als zwischen Prozessor und Hauptspeicher.

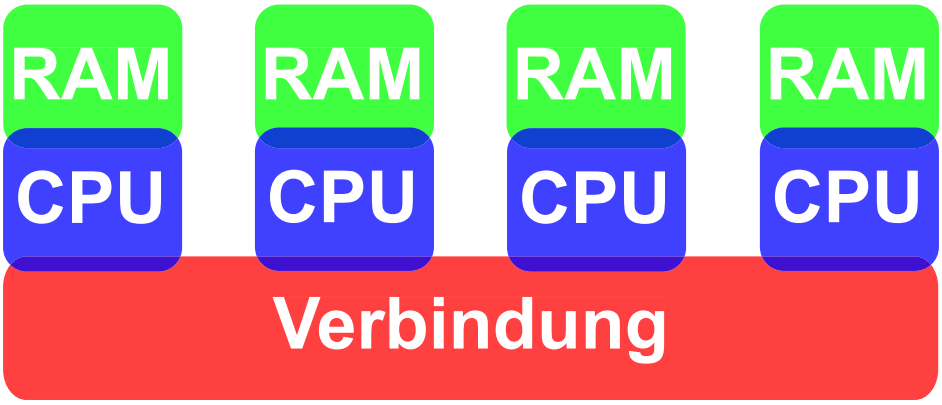


Abbildung 3.9: Schematischer Aufbau eines Distributed Memory-Systems

Distributed Memory-Systeme lassen sich sehr stark erweitern. Mehrere tausend Prozessoren sind nicht unüblich, man spricht dann von *massively parallel processing* (MPP). Um die theoretische Leistung ausnützen zu können, ist jedoch wesentlich mehr Programmieraufwand nötig als bei Shared Memory-Systemen. Das Problem muss in Teile zerlegt werden, die möglichst wenig Kommunikation untereinander erfordern. Die Prozessoren können nur auf ihren lokalen Speicher zugreifen - benötigen sie Daten aus dem Speicher eines anderen Prozessors, so müssen diese umkopiert werden. Dies ist aufgrund der geringeren Kommunikationsgeschwindigkeit zwischen den Prozessoren soweit möglich zu vermeiden.

3.2.6 ccNUMA

In den beiden vorhergehenden Abschnitten wurde erläutert, das Shared Memory-Systeme am Problem der begrenzten Maximalgröße leiden, Distributed Memory-Systeme hingegen am Problem der Kommunikation zwischen den Hauptspeichern der einzelnen Prozessoren. Einen Mittelweg stellt die ccNUMA (*cache coherent non-uniform memory access*) Architektur dar.

Ein ccNUMA-System (Abbildung 3.10) besteht im Prinzip aus mehreren SMP-Systemen (den sogenannten Knoten oder *Nodes*). Diese sind miteinander über ein schnelles Kommunikationsnetzwerk (häufig mit Crossbar-Switches) verbunden. Auf den gesamten verteilten Speicher (non-unified memory) kann über einen gemeinsamen Zwischenspeicher (*Cache*) zugegriffen werden.

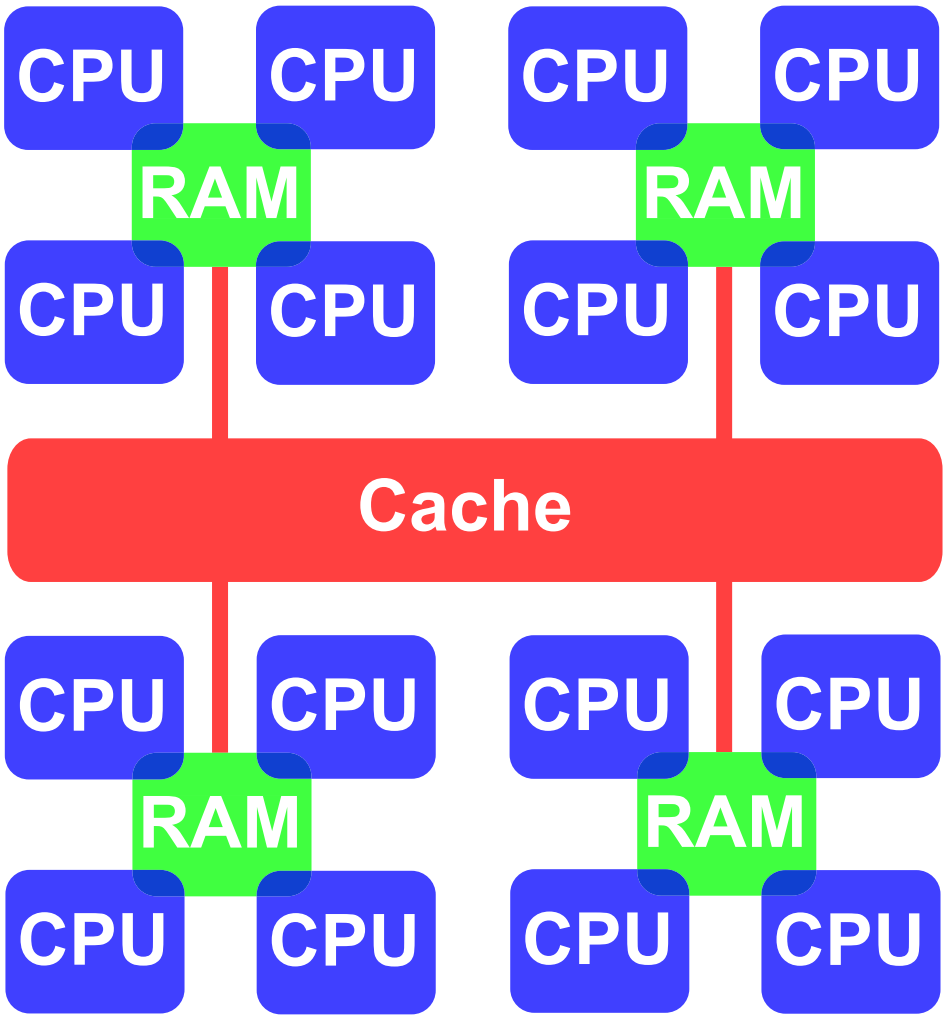


Abbildung 3.10: Schematischer Aufbau eines ccNUMA-Systems

Ein ccNUMA-System läßt sich dadurch ähnlich einfach verwenden wie ein Shared Memory-System, es kann aber gleichzeitig wesentlich stärker erweitert werden. Für optimale Leistung ist darauf zu achten, dass in erster Linie der lokale Speicher verwendet wird und nicht der langsamer angebundene Speicher der übrigen Knoten. Der modulare Aufbau ist ein weiterer Vorteil dieser Architektur. Die meisten ccNUMA-Systeme bestehen aus Modulen, die zu unterschiedlich großen Systemen zusammengeschaltet werden könne.

3.2.7 Cluster

Cluster sind seit einigen Jahren im Bereich des High Performance Computing äußerst beliebt. In einem Cluster (engl. Haufen oder Gruppe) werden viele preisgünstige Rechner (Knoten) zusammengeschaltet. Im einfachsten Fall sind das normale Arbeitsplatzrechner - dann spricht man von einem *Network of Workstations* (NOW). In der Regel verwendet man wegen des guten Preis-Leistungs-Verhältnisses jedoch SMP-Systeme (meistens Dualprozessorsysteme mit Intel- oder AMD-CPUs). Es handelt sich also um *hybride* Systeme. Die Knoten sind Shared Memory-Systeme, bauen aber wieder ein Distributed Memory-System auf (Abbildung 3.11). Innerhalb eines Knoten kann dann über OpenMP (Abschnitt 3.5.4) parallelisiert werden, für die Kommunikation der Knoten untereinander wird Message Passing (3.5.3) verwendet.

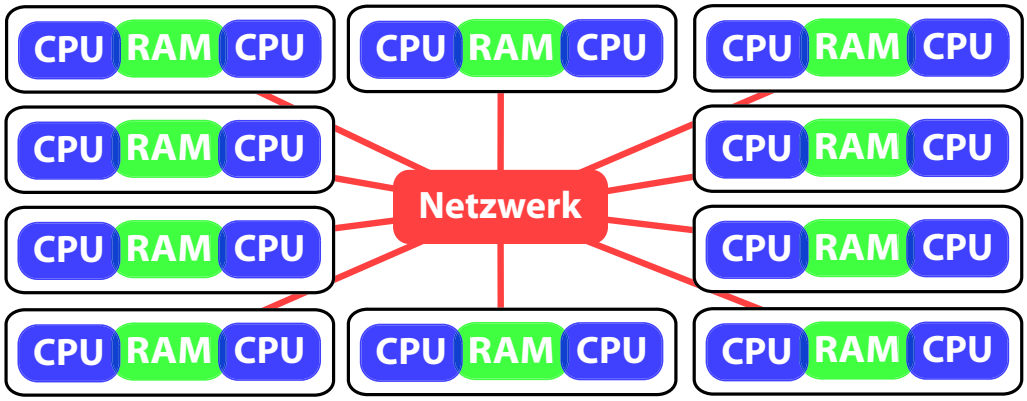


Abbildung 3.11: Schematischer Aufbau eines Clusters aus SMP-Knoten

Die Knoten werden über ein schnelles Netzwerk verbunden. Dafür wird meistens *Myrinet* verwendet. Gigabit-Ethernet hat zwar in etwa die gleiche Bandbreite im Bereich von 100 MB/s und ist kostengünstiger, die Latenz (die Dauer, die ein Datenpaket unterwegs ist) beträgt aber etwa 100 μs, während Myrinet lediglich 10 - 20 μs Latenz aufweist. Auch das ist noch eine Menge Zeit - bei einer Taktfrequenz von 2 GHz dauert ein Taktzyklus 0,5 ns - bei einer Latenz von 10 μs vergehen also 20.000 Taktzyklen, bis das Paket das Ziel erreicht.

Cluster haben den großen Vorteil, dass sie viel Rechenleistung für verhältnismäßig wenig Geld bieten. Allerdings ist es nicht ganz einfach, diese Rechenleistung auch zu nutzen. Die Kommunikation zwischen den einzelnen Rechnern ist langsam, und wie bei Distributed Memory-Systemen hat jeder Rechner nur Zugriff auf seinen lokalen Speicher. Auch sind durch die verwendete PC-Architektur Grenzen gesetzt. 32-bit Rechner können nicht mit mehr als 4 GB Hauptspeicher ausgestattet werden, und auch 64-bit Rechner haben mangels ausreichender Steckplätze und großer Speichermodule selten deutlich mehr Hauptspeicher. Trotz dieser Nachteile sind Cluster sehr erfolgreich und bedrängen das Segment der traditionellen Distributed Memory-Systeme stark.

3.3 Software

3.3.1 Timing und Profiling

Um ein Programm beurteilen und optimieren zu können, muss man wissen, welche Laufzeit das Programm benötigt. Man unterscheidet dabei drei unterschiedliche Zeitspannen:

- *wall time*: Die Zeit, die „die Uhr an der Wand“ ermittelt, also die Zeit zwischen Start und Ende des Programms, die der Benutzer registriert. In der Regel ist es das Ziel, diese Zeit zu minimieren.
- *user time*: Die Rechenzeit, die das Programm tatsächlich verbraucht. Ist diese deutlich geringer als die wall time, so wartet das Programm sehr häufig, z.B. auf die Zuteilung von Rechenzeit durch das Betriebssystem oder auf Daten aus dem Hauptspeicher oder (im schlimmsten Fall) von der Festplatte. Dies ist ein Hinweis auf eine notwendige Optimierung.
- *system time*: Die Rechenzeit, die nicht vom eigentlichen Programm, sondern vom Betriebssystem verbraucht wird, z.B. beim Reservieren von Speicher und beim Dateizugriff.

Eine Möglichkeit der Zeitmessung (*Timing*) ist unter Unix/Linux-Betriebssystemen der `time` Befehl:

```
time Programmname
real 1m50.371s
user 1m45.720s
sys 0m2.510s
```

Die andere Möglichkeit ist das direkte Abfragen der Laufzeit im Programm. Dazu kann z.B. über Standard-Funktionen der verwendeten Programmiersprache die Uhr des Rechners abgefragt werden (einen hilfreichen Überblick über die Zeitmessung mit verschiedenen Betriebssystemen liefert [RIEPE 2004]). *MPI* (Abschnitt 3.5.3) bietet die Funktion `MPI_Wtime()`, *OpenMP* (Abschnitt 3.5.4) die Funktion `omp_get_wtime()`, die jeweils die „wall time“ in Sekunden als double-Zahl zurückgibt.

Interessant und wichtig ist jedoch nicht nur, welche Laufzeit das Programm insgesamt benötigt. Um gezielt optimieren zu können, muss man wissen, an welcher Stelle das Programm viel Rechenzeit benötigt. Diese eingehende Untersuchung der Laufzeit der verschiedenen Programmteile wird als *Profiling* bezeichnet. Dafür steht unter Unix/Linux das Programm `gprof` zur Verfügung.

Um ein Profiling durchführen zu können, muss das Programm mit Profiling-Informationen kompiliert werden. Dies geschieht in der Regel mit der Option `-pg` bzw. mit `-qp` beim Compiler der Firma Intel. Dann muss das Programm normal ausgeführt werden, es erzeugt dabei die Datei `gmon.out`, die vom Profiler benötigt wird. Durch `gprof Programmname > prof.txt` erzeugt der Profiler eine Textdatei mit den gewünschten Informationen. Nachfolgend ist daraus ein Ausschnitt, das sog. *flat profile* dargestellt.

```
%   cumulative self              self    total
time  seconds seconds calls us/call us/call name
59.04  58.40   58.40                      ATL_dJIK40x40x40TN40x40x0_a1_b1
13.17  71.43   13.03  40000  325.75  325.75 legendre1kind
10.51  81.83   10.40                      ATL_dgpMBmm_b1
4.42   86.20    4.37                      main
3.50   89.66    3.46                      ATL_dJIK0x0x20TN20x20x0_a1_bX
```

Die wichtigsten Informationen stehen in den Spalten `% time`, `self seconds` und `name`. Die Spalte `name` gibt die Funktion an, deren Werte in der jeweiligen Zeile angezeigt werden. `% time` gibt an, wieviel Prozent der gesamten Laufzeit auf diese Funktion entfallen. Dies entspricht der in `self seconds` angegebenen Zeit von Sekunden. Im Falle des Beispiels werden 59% der Laufzeit, was

58,4 Sekunden entspricht, von der Funktion `ATL_dJIK40x40x40TN40x40x0_a1_b1`, einer Funktion der ATLAS-Bibliothek, verbraucht.

Der Profiler ermittelt außerdem, welche Funktion welche Unterfunktionen aufruft, und stellt dies im *call tree* dar. Für x86- und IA-64-Systeme (Rechner mit Intel-kompatiblen CPUs) gibt es das Programm *VTune* der Firma Intel, das noch wesentlich mehr Informationen ermitteln kann. Zum Beispiel lässt sich herausfinden, wieviele Fließkomma-Operationen der Prozessor ausführt. Daraus lässt sich die Effizienz als Verhältnis von tatsächlich erzielter zu theoretisch erreichbarer Rechenleistung bestimmen. Der NEC-Compiler *ecc* ermöglicht dies auch direkt auf den Systemen AzusA und TX-7 (siehe 3.4.1 und 3.4.2). Supercomputer wie die NEC SX-6 (Abschnitt 3.4.3) besitzen interne Zähler, die einfache Performancemessungen ermöglichen.

3.3.2 Message Passing und SPMD

Distributed Memory-Systeme werden mittels *Message Passing* programmiert. Da die Prozessoren nicht direkt auf den Speicher anderer Prozessoren zugreifen können, müssen Daten explizit durch den Austausch von Nachrichten transferiert werden. Dabei läuft auf allen Prozessoren das gleiche Programm, jeder Prozessor arbeitet jedoch mit anderen Daten. Daher spricht man von *Single Program - Multiple Data* (SPMD).

Quellcode 3-1 Message Passing-Beispiel

```
MP_initialisieren()
id = MP_Prozessor_Id()
Teilvektoren_laden(id)
Berechnen_Skalarprodukt()
MP_Sammeln_Aufsummieren_Skalarprodukt()
wenn id = 0
  Ergebnis_speichern()
Ende wenn
```

Quellcode 3-1 zeigt in Pseudocode ein Beispiel für die Verwendung von Message Passing und SPMD anhand der Berechnung eines Skalarprodukts. Alle Funktionen, die Message Passing verwenden, sind durch ein vorangestelltes `MP_` gekennzeichnet.

Zu Beginn wird das Message Passing initialisiert und die Prozessor-Id (ein Wert zwischen 0 und (Anzahl der beteiligten Prozessoren - 1)) abgefragt. Dann werden die Vektoren geladen, und zwar in Abhängigkeit von der Prozessor-Id, so dass jeder Prozessor seinen eigenen Datensatz erhält. Für diesen berechnet er dann das Skalarprodukt. Die Funktion
`MP_Sammeln_Aufsummieren_Skalarprodukt()` spricht alle Prozessoren an. Sie sammelt die Werte alle Prozessoren und summiert diese auf. Da diese Kommunikation alle Prozessoren involviert, spricht man von *collective communication*. Das Speichern des Ergebnisses führt dann nur der Rechner mit der `id` 0 durch.

Das Beispielprogramm kommt mit sehr wenig Kommunikation aus. Denkbar wäre auch, dass nur ein Prozessor die Daten lädt, und diese dann an die anderen Prozessoren verteilt. Prinzipiell ist jedoch darauf zu achten, dass möglichst wenig Kommunikation stattfindet, da diese sehr langsam ist.

3.4 Verwendete Hardware

3.4.1 NEC AzusA

Bei der NEC AzusA (Abbildung 3.12, technische Daten in Tabelle 3.1) handelt es sich um ein ccNUMA-System. Es besteht aus vier Boards mit jeweils vier Itanium-Prozessoren und einem lokalen Speicher von je 8 GB. Die Boards sind über einen schnellen Crossbar-Switch miteinander verbunden, wodurch der Zugriff auf den entfernten Speicher nur unwesentlich langsamer ist als der Zugriff auf den lokalen Speicher. Dadurch kann der Rechner praktisch als Shared Memory-System angesehen werden.

Abbildung 3.12: NEC AzusA

Systemarchitektur	ccNUMA
Anzahl Prozessoren	16
Prozessortyp	Intel Itanium (800 MHz)
theoretische Spitzenleistung	16·3,2 = 51,2 GFLOPS
Hauptspeicher	32 GB
Betriebssystem	Red Hat Linux

Tabelle 3.1: Technische Daten der NEC AzusA

Die NEC AzusA dient als Entwicklungssystem. Zu diesem Zweck steht eine Vielzahl von Compilern und Bibliotheken zur Verfügung. Im Rahmen dieser Diplomarbeit diente die AzusA als Entwicklungsplattform für die Implementierung und Parallelisierung mit OpenMP und MPI. Berechnungen können interaktiv und über ein Batch-System gestartet werden. Durch den relativ großen Hauptspeicher eignet sich das System auch zum Testen von speicherintensiven Programmen.

3.4.2 NEC TX-7

Die NEC TX-7 (auch bekannt unter dem Codenamen AsAmA, Abbildung 3.13, technische Daten in Tabelle 3.2) ist ein ccNUMA-System. Sie ähnelt im Aufbau stark der NEC AzusA, verwendet jedoch die zweite Generation des Intel 64 bit-Prozessors Itanium.

Abbildung 3.13: NEC TX-7

Systemarchitektur	ccNUMA
Anzahl Prozessoren	16
Prozessortyp	Intel Itanium 2 (1,5 GHz)
theoretische Spitzenleistung	16 · 6 GFLOPS = 96 GFLOPS
Hauptspeicher	240 GB
Betriebssystem	Red Hat Linux

Tabelle 3.2: Technische Daten der NEC TX-7

Die TX-7 wird am HLRS als Entwicklungssystem für die NEC SX-6 verwendet. Hierzu sind verschiedene Compiler, darunter ein Cross-Compiler für die SX-6, installiert. Ausserdem dient die TX-7 als Front End für die SX-6. Über sie läuft der Zugriff auf die SX-6 und die Verwaltung der Rechenjobs.

Im Rahmen dieser Diplomarbeit wurde die NEC TX-7 für viele Berechnungen verwendet. Sie ist in der theoretischen Spitzenleistung etwa um den Faktor 2 schneller als die NEC AzusA, und aufgrund ihres außerordentlich großen Hauptspeichers ist die Lösung größerer Probleme möglich.

3.4.3 NEC SX-6

Bei der NEC SX-6 handelt es sich um den neuen Vektorrechner des Höchstleistungsrechenzentrums Stuttgart. Die SX-6 ist modular aufgebaut. Einzelne Rechnerschränke (die sogenannten Cabinets) mit je 8 Prozessoren sind über einen Crossbar-Switch verbunden. In seiner jetzigen Ausbaustufe (seit März 2004, Abbildung 3.14, technische Daten in Tabelle 3.3) besteht das System aus 6 Cabinets. Im Jahr 2005 wird ein System aus 64 Cabinets des nochmals deutlich schnelleren Nachfolgesystems in Betrieb gehen.

Abbildung 3.14: NEC SX-6

Systemarchitektur	Cluster aus SMP-Knoten
Anzahl Prozessoren	6·8
Prozessortyp	NEC SX-6 (565 MHz)
theoretische Spitzenleistung	$6 \cdot 8 \cdot 9$ GFLOPS = 432 GFLOPS
Hauptspeicher	$6 \cdot 64$ GB
Betriebssystem	NEC Super-UX

Tabelle 3.3: Technische Daten der NEC SX-6

Aufgrund ihrer Rechenleistung (geplant ist eine Spitzenleistung von 11 TFLOPS), ihrer Hauptspeichergröße (das endgültige System am HLRS wird mit 128 GB Speicher pro Cabinet ausgestattet sein) und ihrer gerade erst beginnenden Lebensspanne am HLRS ist die NEC SX-6 das optimale System für die Bestimmung von Schwerefeldkoeffizienten mit dem Brute-Force-Ansatz. Ziel dieser Diplomarbeit war deshalb unter anderem, diesen Algorithmus auf diesem System lauffähig zu machen.

3.4.4 Cray Strider

Der Name Cray hat im Supercomputing eine lange Geschichte. Am HLRS war bis März 2003 eine Cray T3E-900 als massiv paralleles System im Einsatz. In den letzten Jahren sind die traditionellen massiv parallelen distributed-memory Systeme jedoch durch preisgünstige Cluster stark bedrängt worden. Auch Cray trägt dieser Tatsache Rechnung und verkauft mittlerweile ein Cluster-System. Dieses ist aus SMP-Knoten mit je zwei Prozessoren aufgebaut, die Vernetzung der Knoten erfolgt über Myrinet. Das System kann aus bis zu 1024 Knoten bestehen. Die Installation am HLRS (Abbildung 3.15, technische Daten in Tabelle 3.4) besteht aus 128 Knoten, von denen 125 dem Nutzer als Rechenplatformen zur Verfügung stehen. Die drei übrigen Knoten übernehmen Verwaltungsaufgaben.

Abbildung 3.15: Cray Strider

Systemarchitektur	Cluster aus SMP-Knoten
Anzahl Prozessoren	125·2
Prozessortyp	AMD Opteron 246 (2 GHz)
theoretische Spitzenleistung	125 · 2 · 4 GFLOPS = 1 TFLOPS
Hauptspeicher	125 · 4 GB
Betriebssystem	SuSE Linux

Tabelle 3.4: Technische Daten des Cray Strider

Der Cray Strider-Cluster ersetzt am HLRS das bisherige MPP-Systeme Cray T3E-900. Er ist das System mit der momentan höchsten Rechenleistung am HLRS. Seine Einsatzmöglichkeiten im Rahmen dieser Diplomarbeit sind jedoch etwas begrenzt, da die Speichergröße eines Knoten relativ gering ist (zumindest im Vergleich zu den ccNUMA-Systemen und der NEC SX-6). Aufgrund seiner hohen Rechenleistung eignet er sich jedoch besonders für Algorithmen mit geringem Bedarf an Speicher bei gleichzeitig hohem Rechenaufwand (z.B. iterative Algorithmen, siehe Kapitel 5), zur schnellen Lösung von kleineren Problemen und zum Test von Message Passing.

3.5 Verwendete Software

3.5.1 Compiler

NEC ecc

Beim Compiler ecc handelt es sich um einen von NEC modifizierten Intel-Compiler. Er kommt auf den Itanium-Systemen AzusA (Abschnitt 3.4.1) und TX-7 (3.4.2) zum Einsatz. Der zugrundeliegende Compiler der Firma Intel gilt allgemein als der schnellste für Itanium-Systeme verfügbare Compiler. Gleichzeitig ist er sowohl von der Befehlssyntax als auch von den erzeugten Binärdateien her weitgehend kompatibel zum unter Linux weit verbreiteten GNU-Compiler *gcc*. Die wichtigsten Kommandozeilenparameter sind in Tabelle 3.5 aufgeführt.

Parameter	Funktion
`-ftrace`	Performancemessung aktivieren
`-openmp`	OpenMP aktivieren
`-O2, -O3`	Optimierungen einschalten
`-static`	statisches Linken

Tabelle 3.5: Wichtige Parameter für den Compiler ecc

NEC sxcc

Beim Compiler sxcc handelt es sich um einen Crosscompiler. Mit diesem kann auf der NEC TX-7 (Abschnitt 3.4.2) für die NEC SX-6 (3.4.3) kompiliert werden. Das hat den großen Vorteil, dass die für Kompilierung ungeeignete und sehr langsame SX-6 entlastet wird. Das Itanium-System TX-7 ist hierfür wesentlich besser geeignet. Die wichtigsten Parameter für den sxcc sind in Tabelle 3.6 aufgelistet.

Parameter	Funktion
`-Popenmp`	OpenMP aktivieren
`-size_t64`	64 bit-Pointer aktivieren (um Speicherblöcke > 2GB reservieren zu können)
`-f90lib`	gegen Fortran-Bibliotheken linken (für LAPACK)

Tabelle 3.6: Wichtige Parameter für den Compiler sxcc

PGI

PGI ist ein Linux-Compiler der Portland Group. Er unterstützt Fortran, C und C++ und ist in Versionen für 32 bit- und 64 bit-Systeme basierend auf Intels x86-Architektur verfügbar. Die 64 bit-Version ist an die AMD64-Architektur angepasst, weshalb dieser Compiler auf dem Cray Strider-Cluster (siehe Abschnitt 3.4.4) zum Einsatz kommt. Die wichtigsten Parameter sind in Tabelle 3.7 zusammengefasst.

3.5.2 Bibliotheken für lineare Algebra

ATLAS

ATLAS (Automatically Tuned Linear Algebra Software) ist eine in C implementierte Bibliothek für lineare Algebra. Sie enthält die BLAS (Basic Linear Algebra Subroutines) Routinen und einen

Parameter	Funktion
-Mcache_align	Objekte im Cache ausrichten (zur Geschwindigkeitssteigerung)
-mp	OpenMP aktivieren
-O2, -O3	Optimierungen einschalten
-tp=k8-64	Prozessortyp Opteron (64 bit) auswählen

Tabelle 3.7: Wichtige Parameter für den PGI-Compiler

Teil der LAPACK (Linear Algebra Package) Funktionen. Die Besonderheit der ATLAS-Bibliothek liegt darin, dass bei der Kompilierung automatisch die Algorithmen und Optimierungen verwendet werden, die auf der vorliegenden Rechnerarchitektur die bestmögliche Performance bieten. Dazu gehört auch das automatische Parallelisieren auf Multiprozessor-Systemen.

Die ATLAS-Bibliothek steht unter `http://www.netlib.org` zum kostenlosen Download zur Verfügung.

Intel MKL

Die Intel Math Kernel Library ist eine Bibliothek, die die Funktionen der Standards BLAS, LINPACK und LAPACK zur Verfügung stellt. Dabei gibt es optimierte Versionen für Pentium III-, Pentium IV- und Itanium-Systeme, weshalb die MKL die bevorzugte Bibliothek für lineare Algebra auf x86- und IA64-Systemen ist. Am HLRS steht die MKL-Bibliothek auf allen Intel-basierten Systemen zur Verfügung und wurde für die hier behandelten Programme verwendet.

NEC MathKeisan

Bei der MathKeisan handelt es sich um eine Mathematik-Bibliothek der Firma NEC, die auf den Systemen SX-6 und TX-7 zur Verfügung steht. Sie enthält unter anderem Bibliotheken zur verteilten linearen Algebra (BLACS, PBLAS, ScaLAPACK) und zur Fast Fourier Transformation (FFT). Auf der SX-6 beinhaltet sie außerdem die Bibliotheken BLAS und LAPACK. Auf der TX-7 wird diese Funktionalität durch die Intel MKL bereitgestellt.

ACML

Die AMD Core Math Library ist die Mathematik-Bibliothek der Firma AMD. Sie kommt auf dem Cray Strider-Cluster zum Einsatz. Es gibt sie in 32 bit- und 64 bit-Versionen, jeweils für die Compiler gcc und PGI. Sie enthält die BLAS und LAPACK-Funktionen sowie Routinen zur Fast Fourier Transformation

3.5.3 MPI

MPI (*Message Passing Interface*) ist ein Standard für Message Passing [SNIR 1996]. Er beschreibt die Kommunikationsroutinen (Namen, Parameter, Funktionalität für die Programmiersprachen C und Fortran), die tatsächliche Implementierung bleibt dem Softwarehersteller überlassen. Eine frei verfügbar MPI-Bibliothek ist *mpich*, die für viele Architekturen (besonders auch PCs unter Unix/Linux) zur Verfügung steht.

MPI umfasst eine Vielzahl von Routinen, die alle Aspekte des Message Passing abdecken. Für viele Anwendungsfälle wird man jedoch mit wenigen Funktionen auskommen. Quellcode 3-2 zeigt das Beispiel aus Quellcode 3-1 in konkreten C-Code mit MPI umgesetzt.

Nach der Initialisierung wird die Anzahl der Prozessoren und die Prozessor-Id (in MPI *rank* genannt) abgefragt. `MPI_COMM_WORLD` bezeichnet dabei die Gruppe, die abgefragt wird - die beteilig-

Quellcode 3-2 Beispiel für Message Passing mit MPI

```
#include <mpi.h>
...
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &anzahl_proz);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
... Laden der Vektoren a und b mit der Länge vektor_groesse ...
summe = 0;
for(i=myrank; i<vektor_groesse; i += anzahl_proz)
{
  summe += a[i]*b[i];
}
MPI_Allreduce(&summe, &gesamtsumme, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
if(myrank==0)
  printf("Ergebnis: %lf", gesamtsumme);
MPI_Finalize();
```

ten Prozessoren lassen sich in Gruppen aufteilen, `MPI_COMM_WORLD` bezeichnet alle Prozessoren, auf denen das Programm läuft.

Die Schleife sorgt dafür, dass jeder Prozessor einen Teil der Vektoren $\mathbf{a}$ und $\mathbf{b}$ verarbeitet. Danach werden alle Daten gesammelt, es wird von jedem Prozessor ein Wert des Typs `double` (nämlich dessen Teilsumme) geladen und im Wert `gesamtsumme` aufsummiert. Gleichzeitig wird das Ergebnis wieder an alle Prozessoren verteilt. Das ist zwar in diesem Fall nicht notwendig, da nur Prozessor 0 die Ausgabe erledigt, würde aber auf allen Prozessoren das Weiterverarbeiten des Ergebnisses ermöglichen.

Für die Verwendung von MPI müssen lediglich die oben beschriebenen Funktionen verwendet und das Programm gegen die MPI-Bibliotheken gelinkt werden. Um das Programm dann auf mehreren Prozessoren zu starten, wird das Programm über `mpirun -np` n `Programmname` gestartet. n bezeichnet dabei die Anzahl der Prozessoren, auf denen das Programm laufen soll. Um was für Rechner es sich dabei handelt, ist unerheblich - das Programm (und damit auch die MPI-Bibliothek) muss nur auf all diesen Rechnern zur Verfügung stehen.

3.5.4 OpenMP

Das im vorhergehenden Abschnitt beschriebene MPI dient der parallelen Programmierung von Distributed Memory-Systemen. Prinzipiell ist es auch auf Shared Memory-Systemen einsetzbar. Dadurch entstehen jedoch zwei Probleme:

- Es findet unnötige Kommunikation zwischen den Prozessen statt, da diese ja auf den gleichen Speicher zugreifen können und so keine Daten explizit austauschen müssen.
- Da jeder Prozess seinen eigenen Speicherbereich verwendet, steht jedem Prozess auch nur der Speicher $\frac{\text{gesamter Speicher}}{\text{Anzahl der Prozesse}}$ zur Verfügung.

Es ist also eine Möglichkeit vonnöten, ohne Message Passing zu parallelisieren. Dafür hat sich in den letzten Jahren *OpenMP* [OPENMP 2002] als Standard herauskristallisiert.

OpenMP muss vom Compiler unterstützt werden. Diesem teilt der Programmierer durch sogenannte *Pragmas* mit, welche Teile des Programms parallel ausgeführt werden können. Das Betriebssystem sorgt dann für eine Aufteilung der parallelen Programmteile in *Threads* und deren Verteilung auf die Prozessoren.

Quellcode 3-3 zeigt das Beispiel aus Quellcode 3-2, diesmal mit OpenMP parallelisiert. Das Pragma weist den Compiler darauf hin, dass die folgende for-Schleife parallelisiert werden kann

Quellcode 3-3 Beispiel für die Verwendung von OpenMP

```
... Laden der Vektoren a und b mit der Länge vektor_groesse ...
summe = 0;
#pragma omp parallel for
for(i=0; i<vektor_groesse; ++i)
{
  summe += a[i]*b[i];
}
printf(”Ergebnis: %lf”, summe);
```

(ist der verwendete Compiler nicht OpenMP-tauglich, so wird das Pragma ignoriert, es handelt sich ja lediglich um eine Präprozessoranweisung). Das Betriebssystem erledigt dann automatisch die Verteilung der Schleifendurchläufe auf die verschiedenen Threads.

Alle ausserhalb des parallelen Bereichs deklarierten Variablen sind *shared*, also für alle Threads zugänglich, und haben damit auch in allen Threads den gleichen Wert. Variablen, die innerhalb eines parallelen Bereichs deklariert werden, sind automatisch *private*. Jeder Thread besitzt seine eigene, nur für ihn sichtbare Kopie dieser Variablen. Es ist auch möglich, außerhalb des parallelen Bereichs deklarierte Variablen zu privaten Variablen zu erklären, im Fall des Beispiels 3-3 z.B. mit `#pragma omp parallel for private(j, k)`.

Die Anzahl der Threads wird normalerweise vom Betriebssystem festgelegt. Sie lässt sich aber auch manuell setzen. So kann z.B. die Verwendung mehrerer Threads auch auf einem System mit nur einem Prozessor getestet werden.

Quellcode 3-4 Setzen und Abfragen der Thread-Anzahl

```
#include <omp.h>
...
omp_set_dynamic(0);
omp_set_num_threads(2);
...
#pragma omp parallel
{
  anz_threads = omp_get_num_threads();
  thread_num = omp_get_thread_num();
}
```

Quellcode 3-4 zeigt das manuelle Setzen und Abfragen der Thread-Anzahl. Zuerst muss das dynamische Setzen der Thread-Anzahl deaktiviert werden, dann kann sie manuell gesetzt werden (im Beispiel auf 2). In den parallelen Bereichen lässt sich, analog zur Abfrage der Anzahl der Prozesse und des Rangs bei MPI, auch die Anzahl der Threads und die Nummer des Threads ermitteln.

Kapitel 4

Der Brute-Force-Ansatz

4.1 Algorithmus

Zur Lösung des Gleichungssystems wird der in Gleichung 2.5 dargestellte Gauß-Markov-Algorithmus verwendet. Da hierbei die gesamte Normalgleichungsmatrix aufgebaut und invertiert wird, wird dieser Ansatz in dieser Arbeit als „Brute-Force-Ansatz“ bezeichnet.

Unter der Annahme, dass das Schwerefeldmodell bis Grad und Ordnung k_{max} entwickelt werden soll, ergibt sich die Zahl der Unbekannten aus der Doppelsumme in Gleichung 2.8. Die Summation von 1 bis k_{max} entspricht einer Dreiecksmatrix der Dimension $k_{max} \times k_{max}$, und zwar jeweils für die Cosinus- und die Sinus-Terme, also $k_{max}^2 + k_{max}$ Unbekannte. Dazu kommen noch die k_{max} Cosinus-Terme der Ordnung 0.Die drei Terme vom Grad 1, welche die Abweichung des Ursprungs des Koordinatensystems vom Massenmittelpunkt der Erde beschreiben, werden zu Null angenommen und damit nicht mitgeschätzt. Der Zentralterm (Grad und Ordnung 0) wird mitgeschätzt. Wir erhalten also

$$u = k_{max}^2 + 2 \cdot k_{max} - 2 \tag{4.1}$$

Unbekannte. Die Dimension der Normalgleichungsmatrix $\mathbf{N}$ ist $u \times u$. Bei Fließkommazahlen doppelter Genauigkeit (Länge = 8 byte) ergibt sich damit in Abhängigkeit vom Entwicklungsgrad die in Tabelle 4.1 dargestellte Anzahl von Unbekannten und der dazugehörige Speicherbedarf für die Normalgleichungsmatrix.

Entwicklungsgrad	**Anzahl der Unbekannten** u	**Speicherbedarf für** $\mathbf{N}$
20	438	1,46 MB
50	2598	51,50 MB
100	10198	793,45 MB
150	22798	3,87 GB
200	40398	12,16 GB
250	62998	29,57 GB
300	90598	61,15 GB

Tabelle 4.1: Anzahl der Unbekannten und Speicherbedarf in Abhängigkeit vom Entwicklungsgrad

Der Speicherbedarf für die Designmatrix $\mathbf{A}$ ist jedoch deutlich höher, diese hat $n \times u$ Elemente, wobei n die Anzahl der Beobachtungen ist. Bei einer klein gewählten Menge von 500.000 beobachteten Satellitenpositionen (was in etwa einem Monats-Datensatz bei einer Samplingrate von

5 Sekunden entspricht) und einem maximalen Entwicklungsgrad von 100 ergibt sich bereits ein Speicherbedarf von 114 GB.

Die Lösung dieses Problems bietet die Zerlegung von $\mathbf{A}$. Unter der Annahme $\mathbf{P} = \mathbf{I}$ (alle Beobachtungen gleich genau und untereinander nicht korreliert) ergibt sich

$$\mathbf{N} = \mathbf{A}^T\mathbf{A}. \tag{4.2}$$

$\mathbf{A}$ wird jetzt in p Blöcke $\mathbf{A}_1 \ldots \mathbf{A}_p$ zerlegt, gemäß den Rechenregeln für Blockmatrizen gilt dann

$$\mathbf{N} = \begin{pmatrix} \mathbf{A}_1^T & \cdots & \mathbf{A}_p^T \end{pmatrix} \cdot \begin{pmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_p \end{pmatrix} = \mathbf{A}_1^T\mathbf{A}_1 + \ldots + \mathbf{A}_p^T\mathbf{A}_p = \sum_{i=1}^{p} \mathbf{A}_i^T\mathbf{A}_i. \tag{4.3}$$

Die Blockgröße kann dabei beliebig klein gewählt werden, so dass der Speicherbedarf für einen Block minimal $u \cdot 24$ byte beträgt (24 = 3· 8, da eine Position, und damit ein Datensatz, aus drei Koordinaten besteht).

Oft wird man aber Korrelationen zwischen den Beobachtungen berücksichtigen wollen. Da der Speicherplatz begrenzt ist, kann die Gewichtsmatrix nur entsprechend der Blockgröße gewählt werden. Die Korrelationen der Beobachtungen innerhalb eines Blocks werden damit berücksichtigt. Beobachtungen in unterschiedlichen Blöcken bleiben aber zueinander unkorreliert. Ein Ansatz zur Lösung dieses Problems findet sich in Abschnitt 4.6.

4.2 Aufbau des Programms

Quellcode 4-1 zeigt die Grundstruktur des Programms. Es ist klar ersichtlich, dass das Programm eine sehr einfache Struktur hat, die im wesentlichen durch zwei Schleifen bestimmt wird. Diese bilden später den Ansatzpunkt für die Parallelisierung.

Quellcode 4-1 Grundstruktur des Programms

```
einlesen der Beobachtungen
Schleife über alle Blöcke B_i
    Schleife über alle Datensätze x(t_j) im Block B_i
        berechnen der zu x(t_j) gehörenden Zeilen der Designmatrix A_i
        besetzen des Beobachtungsvektors y_i
    Ende Schleife über die Beobachtungen
    N = A_i^T A_i + N
    b = A_i^T y_i + b
Ende Schleife über die Blöcke
x̂ = N^-1 b
```

4.3 Serielle Version

Quellcode 4-2 Serielle Version (vereinfacht) in C

```
 1:  #include <atlas_enum.h>
 2:  #include <cblas.h>
 3:  #include <clapack.h>
 4:  ...
 5:  #define kmax 20
 6:  #define block_am 80
 7:  #define block_size 500
 8:  ...
 9:  int main(int argc, char* argv[])
10:  {
11:    ... Variablendeklaration ...
12:    ... Allokierung von Speicher ...
13:    ... Einlesen der Beobachtungen ...
14:    for(i=0; i<block_am; i++)
15:    {
16:      for(j=i*block_size; j<(i+1)*block_size; j++)
17:      {
18:        ... Besetzen von y mit den Werten des Datensatzes j
19:        legendre1kind(kmax, lat[j], nL, dnL);
20:        ... Berechnen der Stelle in A für c_{0,0} und Datensatz j ...
21:        for(k=2; k<=kmax; k++)
22:        {
23:          for(l=0; l<=k; l++)
24:          {
25:            ... Berechnen der entsprechenden Stelle in A zum Datensatz j ...
26:          }
27:        }
28:      }
29:      cblas_dsyrk(...);
30:      cblas_dgemv(...);
31:    }
32:    clapack_dposv(...);
33:    ... Ausgabe der Ergebnisse ...
34:    return 0;
35:  }
```

Quellcode 4-2 zeigt vereinfacht die serielle Version des Programms zur Bestimmung von Schwerefeldkoeffizienzen. Es besteht aus folgenden Teilen:

- In Zeile 1 bis 3 finden sich die Header-Dateien der verwendeten ATLAS-Bibliothek (3.5.2).
- In Zeile 5 bis 7 sind der maximale Entwicklungsgrad, die Blockanzahl und die Blockgröße definiert.
- In Zeile 14 beginnt die Schleife über alle Beobachtungsblöcke, in Zeile 16 die Schleife über die Beobachtungen des jeweiligen Blocks.
- Mit der Funktion `legendre1kind()` in Zeile 19 werden die normierten Legendreschen Funktionen 1. Art und deren Ableitung berechnet und in die Arrays `nL` und `dnL` geschrieben.
- Die Zeilen 21 bis 27 beinhalten die beiden Schleifen über die Unbekannten, in denen die Designmatrix besetzt wird.

- In Zeile 29 wird mittels der Funktion `dsyrk()` der BLAS-Bibliothek (siehe 3.5.2) $\mathbf{N} = \mathbf{A}^T\mathbf{A} + \mathbf{N}$ berechnet.
- In Zeile 30 wird mit der Funktion `dgemv()` $\mathbf{b} = \mathbf{A}^T\mathbf{y} + \mathbf{b}$ berechnet.
- In Zeile 32 wird mittels der Funktion `dposv()` der LAPACK-Bibliothek $\hat{\mathbf{x}} = \mathbf{N}^{-1}\mathbf{b}$ berechnet.

Es ist klar erkennbar, dass das Programm im wesentlichen aus vier Schleifen und vier Funktionsaufrufen besteht. Um jetzt mit der Optimierung beginnen zu können, ist jedoch wichtig zu wissen, welche Programmteile am meisten Rechenzeit verbrauchen. Dazu wurden Timing und Profiling, wie in 3.3.1 beschrieben, durchgeführt. Dies ergab für den vorliegenden Fall (Entwicklungsgrad 20, 40.000 Datensätze) dass nahezu 100% der Rechenzeit auf das Berechnen der Normalgleichungsmatrix entfallen, und davon wiederum 13% auf die Berechnung der Legendreschen Funktionen. Die übrige Zeit wurde von den ATLAS/BLAS-Funktionen verbraucht.

Da sowohl die BLAS-Funktionen als auch die `legendre1kind()` Funktion bereits optimiert sind, bieten sich hier keine Ansatzpunkte mehr. Weitere Geschwindigkeitssteigerungen sind nur noch durch Parallelisierung zu erreichen.

4.4 Parallelisierung mit MPI

Quellcode 4-3 zeigt die mit MPI parallelisierte Version des Programms. Es sind nur die wichtigsten Teile dargestellt und Änderungen gegenüber 4-2 farbig hervorgehoben.

Quellcode 4-3 Mit MPI parallelisierte Version des Programms

```
 1:  #include <mpi.h>
 2:  ...
 3:  int main(int argc, char* argv[])
 4:  {
 5:    ...
 6:    MPI_Init(&argc, &argv);
 7:    MPI_Comm_size(MPI_COMM_WORLD, &np);
 8:    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 9:    ...
10:    for(i=myrank; i<block_am; i+=np)
11:    {
12:      ...
13:    }
14:    ...
15:    MPI_Reduce(N, N2, u*u, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
16:    MPI_Reduce(b, b2, u, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
17:    if(myrank==0)
18:    {
19:      clapack_dposv(...);
20:      ... Ausgabe der Ergebnisse ...
21:    }
22:    ...
23:  }
```

Es gibt folgende Änderungen:

- In Zeile 1 wird zusätzlich die Header-Datei für MPI eingebunden.
- In den Zeilen 6 bis 8 wird MPI initialisiert und die Anzahl der Prozessoren und die Id des eigenen Prozessors abgefragt.

- Die abgeänderte Schleife in Zeile 10 sorgt für die Verteilung der Berechnung der Blöcke auf die unterschiedlichen Prozessoren. Abbildung 4.1 zeigt, wie die Verteilung erfolgt.
- In Zeile 15 und 16 werden die Normalgleichungsmatrizen und die rechten Seiten des Normalgleichungssystems von allen Prozessoren eingesammelt und auf dem Master-Prozessor (dem Prozessor mit dem Rang 0) aufsummiert.
- Die Abfrage in Zeile 17 sorgt dafür, dass nur der Prozessor 0 das Gleichungssystem löst und die Ergebnisse ausgibt.

Damit ist das Programm für die Verwendung auf Distributed Memory-Systemen parallelisiert. Die Zeit zur Berechnung der Designmatrix sollte bis auf die Kommunikationszeit in den `MPI_Allreduce()` Funktionen umgekehrt proportional zur Anzahl der Prozessoren sein.

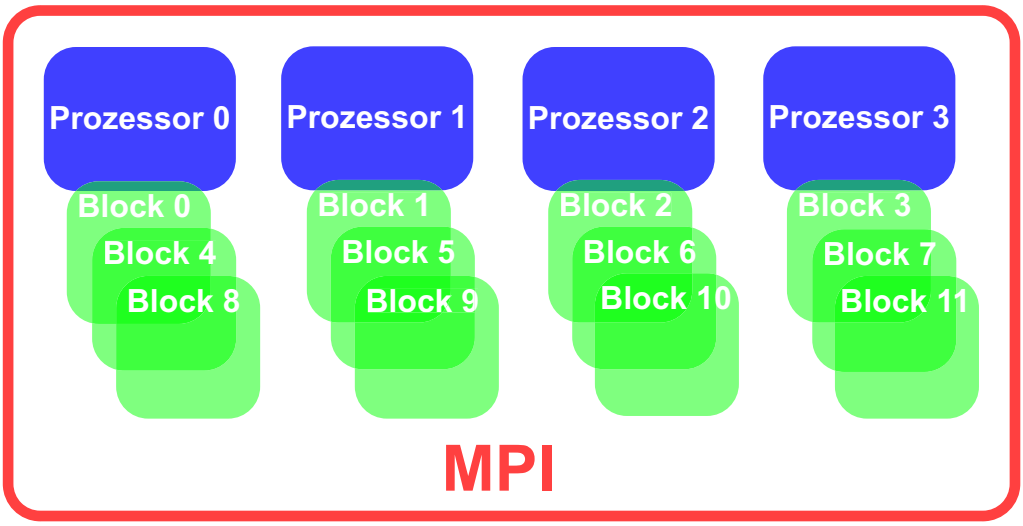


Abbildung 4.1: Verteilung der Blöcke bei einem Distributed Memory-System mit vier Prozessoren

4.5 Parallelisierung mit OpenMP

Im vorherigen Abschnitt wurde das Programm mittels MPI für Distributed Memory-Systeme parallelisiert. Wie in 3.5.4 erläutert wurde, kann dieser Ansatz zwar prinzipiell auch auf Shared Memory-Systemen verwendet werden, ist aber mit dem Nachteil der unnötigen Kommunikation und der Teilung des Speichers verbunden. Deshalb wurde zusätzlich die in Quellcode 4-4 gezeigte Parallelisierung mittels OpenMP durchgeführt. Dabei wurde an anderer Stelle als bei der Parallelisierung mittels MPI angesetzt. So kann das Programm auf Distributed Memory-Systemen laufen (durch Verwendung von lediglich einem Thread), auf Shared Memory-Systemen (durch Verwenden von nur einem MPI-Prozessor) und natürlich auf hybriden Systeme. Die enstprechende Aufteilung ist in Abbildung 4.2 dargestellt.

In Quellcode 4-4 sind die Änderungen sind wieder farbig hervorgehoben.

- In Zeile 1 wird der OpenMP-Header eingebunden.
- In Zeile 2 wird die Anzahl der Threads auf 2 definiert (wie man es bei einem System mit Doppelprozessor-Knoten machen würde).
- In Zeile 3 wird definiert, dass die Variablen nL und dnL in jedem Thread *private* sein sollen.

Quellcode 4-4 Mit OpenMP parallelisierte Version des Programms

```
 1:  #include <omp.h>
 2:  #define NUMTHREADS 2
 3:  #pragma omp threadprivate(nL,dnL)
 4:  ...
 5:  int main(int argc, char* argv[])
 6:  {
 7:    ...
 8:    omp_set_dynamic(0);
 9:    omp_set_num_threads(NUMTHREADS);
10:    #pragma omp parallel
11:    {
12:      ... Allokierung von Speicher für nL und dnL ...
13:    }
14:    ...
15:    for(i=myrank; i<block_am; i+=np)
16:    {
17:      #pragma omp parallel
18:      {
19:        ... Variablendeklaration ...
20:        #pragma omp for
21:        for(j=i*block_size; j<(i+1)*block_size;j++)
22:        {
23:          ...
24:        }
25:      }
26:    }
27:    ...
28:  }
```

- In Zeile 8 wird die dynamische Threadverteilung deaktiviert, und in Zeile 9 die Anzahl der Threads in den parallelen Bereichen auf den in Zeile 2 definierten Wert gesetzt.
- In den Zeilen 10 bis 13 wird der Speicher für nL und dnL allokiert. Jeder Thread benötigt seine eigene Kopie dieser Variablen. Deshalb müssen diese auch vorab als *threadprivate* deklariert werden. Durch das Deaktivieren der dynamischen Threadverteilung wird verhindert, dass später mehr Threads tatsächlich arbeiten, als bei der Speicherreservierung berücksichtigt wurden.
- In Zeile 17 wird der eigentliche parallele Bereich definiert, in dem die Berechnung stattfindet. In Zeile 19 werden die Variablen deklariert, die in den einzelnen Threads private sein müssen. Prinzipiell wäre die Deklaration und Allokierung von nL und dnL auch hier möglich. Es würde die Zeilen 3 und 8 bis 13 unnötig machen, aber gleichzeitig das Allokieren und Freigeben des Speichers für jeden Block erfordern, was zu Lasten der Laufzeit geht.
- In Zeile 20 wird definiert, dass die folgende Schleife parallel auszuführen ist. Das Betriebssystem verteilt die Berechnung der einzelnen Zeilen der Designmatrix auf die Threads.

Das Programm liegt jetzt in einer für alle Systemarchitekturen geeigneten parallelen Version vor.

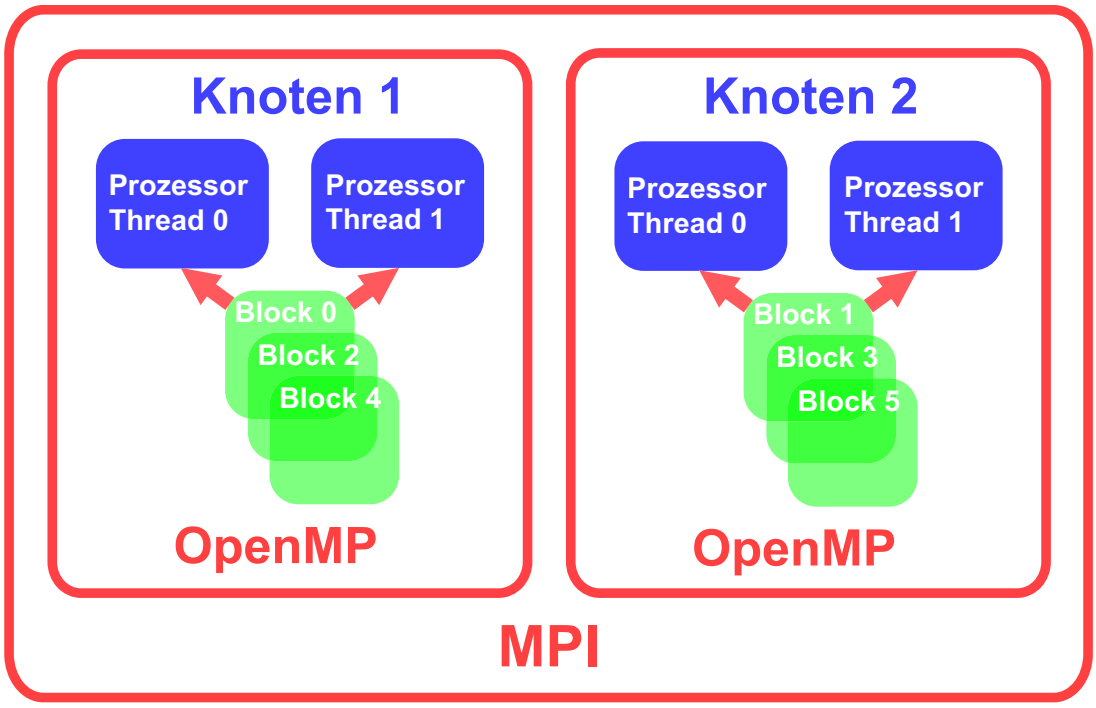


Abbildung 4.2: Verteilung der Blöcke bei einem hybriden System mit zwei Knoten und zwei Prozessoren pro Knoten

4.6 Einführung einer Gewichtsmatrix

4.6.1 Blockweise Berechnung der Normalgleichungsmatrix

Wie in Abschnitt 4.1 erläutert wurde, kann im ungewichteten Fall die Berechnung der Normalgleichungsmatrix $\mathbf{N}$ in Blöcke zerlegt werden. Bei höheren Entwicklungsgraden ist dies sogar unumgänglich, da die Designmatrix $\mathbf{A}$ dann nicht gesamt im Speicher gehalten werden kann. Erweitert man Gleichung 4.3 um eine Gewichtsmatrix ergibt sich

$$\mathbf{N} = \sum_{i=1}^{p} \mathbf{A}_i^T \mathbf{P}_i \mathbf{A}_i \tag{4.4}$$

für die Normalgleichungsmatrix bzw.

$$\mathbf{b} = \sum_{i=1}^{p} \mathbf{A}_i^T \mathbf{P}_i \mathbf{y}_i \tag{4.5}$$

für die rechte Seite der Normalgleichungen (hier ist p die Anzahl der Blöcke). Diese Vorgehensweise macht die Verwendung einer Gewichtsmatrix auch bei blockweiser Berechnung der Normalgleichungsmatrix möglich. Sie ist jedoch mit einem großen Nachteil verbunden: Es werden zwar die Korrelationen zwischen den Beobachtungen innerhalb eines Blocks berücksichtigt, aber nicht die Korrelationen zwischen den Beobachtungen in verschiedenen Blöcken.

4.6.2 Struktur der Gewichtsmatrix

Auf der Suche nach einer Lösung des im vorigen Abschnitt genannten Problems untersuchen wir zuerst die Struktur einer geeigneten Gewichtsmatrix. Diese beruht auf theoretischen Erwägungen. Wir gehen davon aus, dass aufeinanderfolgende Satellitenpositionen hoch miteinander korreliert sind. Zusätzlich gehen wir davon aus, dass es sich bei der Korrelationsfunktion um einen Markov-Prozeß handelt, also die Korrelation mit zunehmendem zeitlichen und örtlichen Abstand zwischen zwei Beobachtungen immer weiter abnimmt.

Für die Korrelationsfunktion

$$\rho_{i,j} = \rho^{|i-j|} \tag{4.6}$$

mit der Korrelation $\rho_{i,j}$ zwischen den Beobachtungen i und j mit einer Distanz von $i-j$ Epochen ergibt sich bei einen angenommenen Korrelationskoeffizienten von ρ = 0,99 die in Abbildung 4.3 dargestellte Korrelationsfunktion.

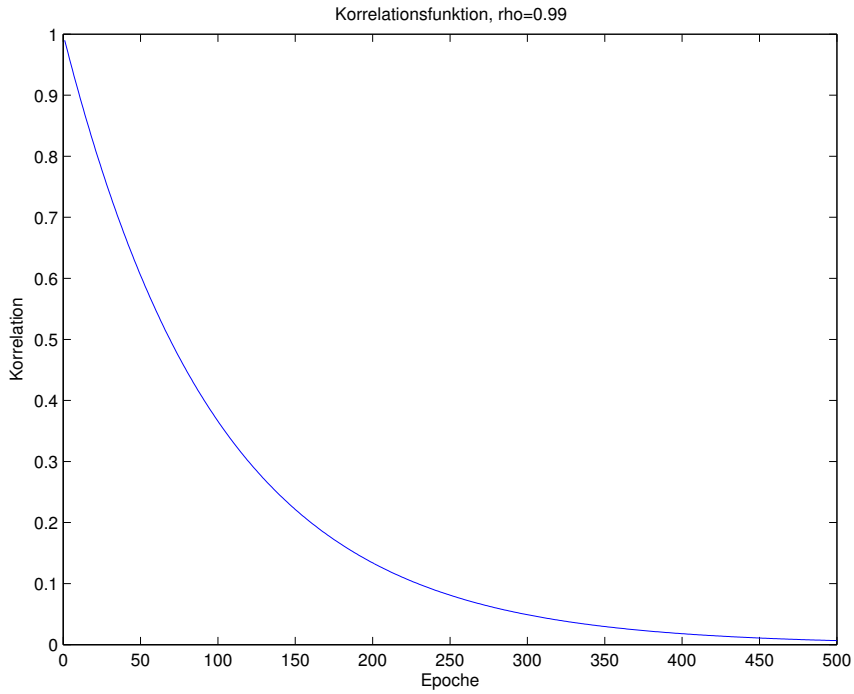


Abbildung 4.3: Korrelationsfunktion bei ρ = 0,99

Mit dieser Korrelationsfunktion und einer Blockgröße von q Beobachtungen ergibt sich die nachfolgend dargestellte Struktur der Kovarianzmatrix $\mathbf{C}_{yy}$ der Beobachtungen:

$$\mathbf{C}_{yy} = \begin{pmatrix} 1 & \rho & \rho^2 & \cdots & \rho^{q-1} \\ \rho & 1 & \rho & & \\ \rho^2 & \rho & 1 & & \\ \vdots & & & \ddots & \\ \rho^{q-1} & & & & 1 \end{pmatrix} \tag{4.7}$$

Aus dieser ergibt sich die Gewichtsmatrix:

$$\mathbf{P} = \mathbf{C}_{yy}^{-1} \tag{4.8}$$

Im Falle eines Korrelationskoeffizienten von ρ = 0,99 ergibt sich die in Abbildung 4.4 dargestellte Struktur der Gewichtsmatrix.

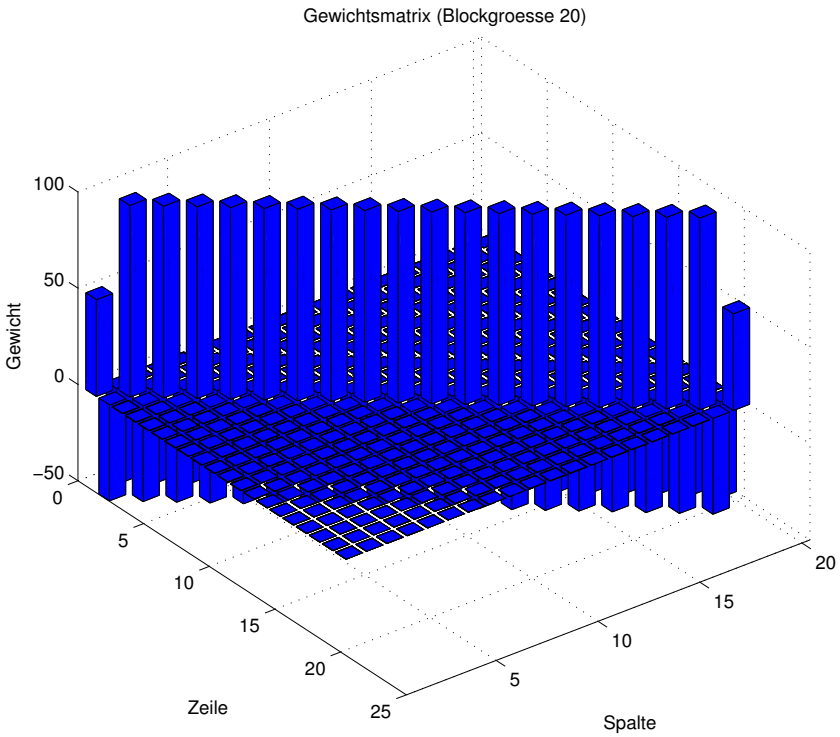


Abbildung 4.4: Gewichtsmatrix für 20 Beobachtungen

Wie in der Beobachtungsgleichung 2.8 des HL-SST zu sehen, gehen nicht die Positionen als Beobachtungen in das funktionale Modell ein, sondern Beschleunigungen. Die Fehlerfortpflanzung der numerischen Differentiation bringt letztlich die Kovarianzmatrix der Beschleunigungen.

In Abbildung 4.5 ist die resultierende Gewichtsmatrix der Beschleunigungen für eine Blockgröße von 20 Beobachtungen und einen Korrelationskoeffizienten von ρ = 0,99 dargestellt. Es ist zu erkennen, dass analog zur Korrelationsfunktion das Gewicht zwischen weiter entfernten Beobachtungen immer weiter abnimmt. Bei einer Blockgröße von 500 Beobachtungen ist der Unterschied zwischen größtem und kleinstem Gewicht größer als 10^5. Daraus ergibt sich die in Abbildung 4.6 dargestellte Struktur der Gewichtsmatrix.

Die Elemente außerhalb der „Korrelationslänge“ sind gleich Null. Sie müssen bei der Berechnung von $\mathbf{N}$ nicht mit berücksichtigt werden. Wählt man die Blockgröße entsprechend der doppelten Korrelationslänge (in Abbildung 4.7 grün dargestellt) werden nur die rot dargestellten Elemente der Gewichtsmatrix vernachlässigt.

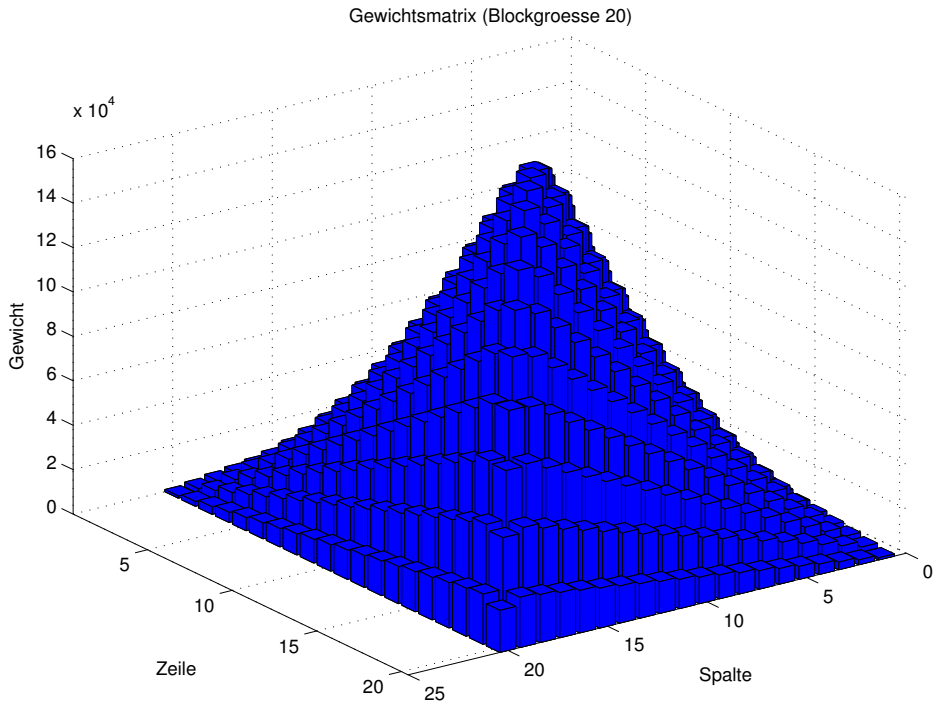


Abbildung 4.5: Gewichtsmatrix für 20 Beobachtungen nach Fehlerfortpflanzung

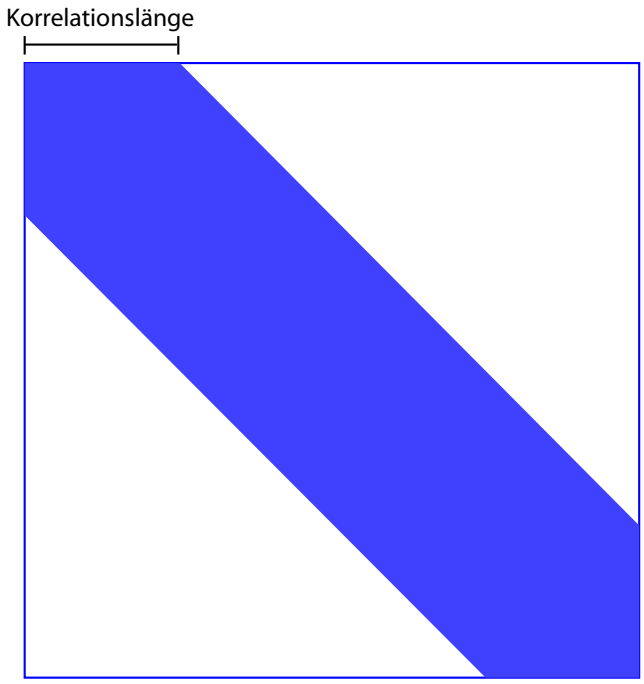


Abbildung 4.6: Struktur der Gewichtsmatrix

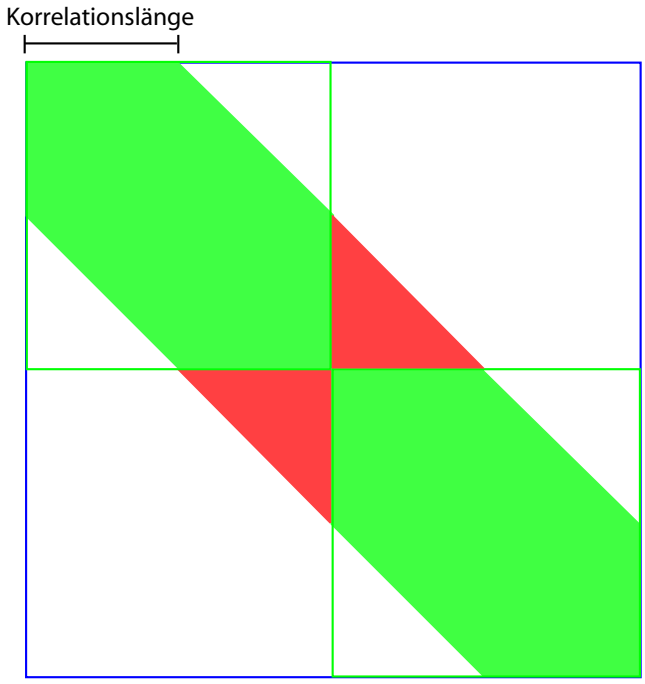


Abbildung 4.7: Blockweise Berücksichtigung der Gewichtsmatrix

4.6.3 Berechnung der Normalgleichungsmatrix in zwei Schritten

Führt man in einem zweiten Schritt eine um die Korrelationslänge verschobene Berechnung durch, bei der das linke obere und das rechte untere Viertel von $\mathbf{P}$ auf Null gesetzt sind, werden sämtliche Korrelationen zwischen den Beobachtungen berücksichtigt (Abbildung 4.8, Schritt 1 blau, Schritt 2 grün).

Es ergibt sich als Formel für die Berechnung der Normalgleichungsmatrix

$$\mathbf{N} = \sum_{i=1}^{p} \mathbf{A}_i^T \mathbf{P}_1 \mathbf{A}_i + \sum_{j=1}^{p-1} \mathbf{A}_j^T \mathbf{P}_2 \mathbf{A}_j \tag{4.9}$$

und für die rechte Seite des Gleichungssystems

$$\mathbf{b} = \sum_{i=1}^{p} \mathbf{A}_i^T \mathbf{P}_1 \mathbf{y}_i + \sum_{j=1}^{p-1} \mathbf{A}_j^T \mathbf{P}_2 \mathbf{y}_j. \tag{4.10}$$

Dabei ist $\mathbf{P}_1$ die voll besetzte Gewichtsmatrix, während in $\mathbf{P}_2$ das linke obere und das rechte untere Viertel auf Null gesetzt sind.

In der Regel wird man die Blockgröße so klein wie möglich, also entsprechend der doppelten Korrelationslänge, wählen. Das sorgt nicht nur für möglichst geringen Speicherbedarf, sondern minimiert auch den Rechenaufwand, da unnötige Multiplikationen (deren Resultat ja Null ist) vermieden werden.

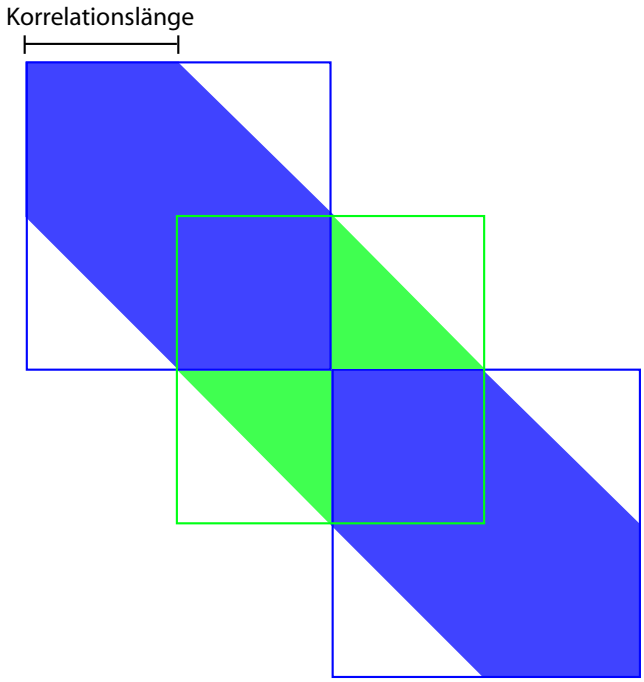


Abbildung 4.8: Berücksichtigung der Gewichtsmatrix durch zwei Berechnungsschritte

4.7 Varianzkomponentenschätzung

Bei der Kombination mehrerer Beobachtungstypen muss jeder Beobachtungstyp mit seiner individuellen Genauigkeit eingeführt werden, um die Ergebnisse nicht zu verfälschen. Das Ermitteln dieser Genauigkeiten ist Aufgabe der Varianzkomponentenschätzung (*variance component estimation* - VCE) [KUSCHE 2003/1]. Ausgangspunkt dafür ist eine Lösung $\widehat{\mathbf{x}}$ unter Verwendung aller Beobachtungen:

$$\widehat{\mathbf{x}} = (\mathbf{A}^T\mathbf{P}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{P}\mathbf{y} = \mathbf{N}^{-1}\mathbf{b} \tag{4.11}$$

4.7.1 Strenge Varianzkomponentenschätzung

Zur Berechnung der Varianzkomponenten werden die Verbesserungen pro Beobachtungstyp benötigt, die sich gemäß Gleichung 4.12 berechnen. Dabei ist i der Index der q Beobachtungsgruppen. Die Berechnung kann blockweise erfolgen.

$$\mathbf{v}_i = \mathbf{A}_i\widehat{\mathbf{x}} - \mathbf{y}_i \tag{4.12}$$

Neben den Verbesserungen werden die Redundanzanteile r_i benötigt. Diese ergeben in der Summe die Gesamtredundanz $r = n - u$. Die Berechnung erfolgt laut Gleichung 4.13:

$$r_i = n_i - \frac{1}{\sigma_i^2}\mathrm{trace}\left(\left(\sum_{i=1}^{p}\frac{1}{\sigma_i^2}\mathbf{A}_i^T\mathbf{P}_i\mathbf{A}_i\right)^{-1}\mathbf{A}_i^T\mathbf{P}_i\mathbf{A}_i\right)$$

$$= n_i - \frac{1}{\sigma_i^2}\mathrm{trace}\left(\left(\sum_{i=1}^{p}\frac{1}{\sigma_i^2}\mathbf{N}_i\right)^{-1}\cdot\mathbf{N}_i\right) \tag{4.13}$$

Es ist n_i die Anzahl der Beobachtungen der jeweiligen Beobachtungsgruppe i, und σ_i^2 die a-priori Varianz derselben. Die Berechung der $\mathbf{N}_i$ kann blockweise erfolgen, wenn $\mathbf{P} \neq \mathbf{I}$ ist muss jedoch die Berechnung wie in Abschnitt 4.6 beschrieben erfolgen.

Sind die Redundanzanteile berechnet, können daraus zusammen mit der Verbesserungsquadratsumme die Varianzkomponenten bestimmt werden:

$$s_i^2 = \frac{\mathbf{v}_i^T\mathbf{P}_i\mathbf{v}_i}{r_i} \tag{4.14}$$

Die berechneten s_i^2 können nun in einer neuen Berechnung als a-priori-Werteσ_i^2 verwendet werden. Es handelt sich hierbei um einen iterativen Prozess, der mehr als einen Iterationsschritt benötigt.

Aus der Summe der einzelnen Verbesserungsquadratsummen und Redundanzanteile lässt sich die Varianz der Gewichtseinheit berechnen, die z.B. benötigt wird, um empirische Genauigkeiten der geschätzten Parameter zu berechnen (Gleichung 4.16):

$$s^2 = \frac{\Sigma\mathbf{v}_i^T\mathbf{P}_i\mathbf{v}_i}{\Sigma r_i} \tag{4.15}$$

$$\mathbf{C}_{\hat{x}\hat{x}} = s^2\mathbf{N}^{-1} \tag{4.16}$$

4.7.2 Monte Carlo Varianzkomponentenschätzung

Wie aus Gleichung 4.13 hervorgeht, benötigt die strenge Varianzkomponentenschätzung sehr viel Hauptspeicher, da für jede Beobachtungsgruppe eine Kopie $\mathbf{N}_i$ der Normalgleichungsmatrix benötigt wird. Die benötigte Speichergröße für die strenge Berechnung wird in der Regel nicht zu Verfügung stehen. Einen Ausweg stellt die Monte Carlo Varianzkomponentenschätzung (*Monte Carlo variance component estimation* - MCVCE) dar. Es handelt sich dabei um eine sog. Monte Carlo-Simulation, wobei zufällig generierte Werte verwendet werden, die in ihrem (angenommenen) stochastischen Verhalten den tatsächlichen Beobachtungen entsprechen [KUSCHE 2003/2].

Bei der MCVCE ändert sich gegenüber der strengen Varianzkomponentenschätzung lediglich die Berechnung der Redundanzanteile:

$$r_i = n_i\frac{1}{\sigma_i^4}\mathbf{w}_i^T\mathbf{P}_i\mathbf{A}_i\widehat{\xi}_i \tag{4.17}$$

Es gilt für den Vektor $\mathbf{w}_i$:

$$\mathbf{w}_i = \left(\mathbf{G}_i^{-1}\right)^T\sigma_i\mathbf{z}_i \tag{4.18}$$

Dabei ist $\mathbf{G}_i$ die Cholesky-Dreieckszerlegung der Gewichtsmatrix $\mathbf{P}_i$. Der Vektor $\mathbf{z}_i$ ist zufällig, aber gleichverteilt mit den Werten 1 und -1 besetzt. $\widehat{\xi}_i$ ergibt sich aus

$$\widehat{\xi}_i = \mathbf{N}^{-1}\mathbf{A}_i^T\mathbf{P}_i\eta_i = \mathbf{N}^{-1}\beta_i \tag{4.19}$$

In η_i sind alle Beobachtungen der Beobachtungsgruppe i mit den Werten aus $\mathbf{w}_i$ besetzt. Die übrigen Elemente sind gleich null.

Wie aus Gleichung 4.17 ersichtlich ist, müssen hierbei bei blockweiser Berechnung keine großen Matrizen gespeichert werden. Zusätzlich entfällt die aufwändige Berechnung des Produkts $\mathbf{A}_i^T\mathbf{P}_i\mathbf{A}_i$, was die Rechenzeit gegenüber der strengen Variante deutlich reduziert. Die mehrfache Lösung des Gleichungssystems in Gleichung 4.19 stellt keinen zusätzlichen Rechenaufwand dar, da $\mathbf{N}^{-1}$ nur einmal berechnet werden muss, und sich die unterschiedlichen Lösungen $\widehat{\xi}_i$ lediglich aus unterschiedlichen rechten Seiten β_i ergeben.

Die übrige Berechnung der Varianzkomponenten erfolgt wie in den Gleichungen 4.12 und 4.14 beschrieben.

4.8 Gewichtung ohne Gewichtsmatrix

Im vorigen Abschnitt wurde erläutert, wie mit Hilfe der Varianzkomponentenschätzung das tatsächliche Varianzniveau der einzelnen Beobachtungsgruppen ermittelt werden kann. Dieses Varianzniveau wird normalerweise durch die Gewichtsmatrix $\mathbf{P}$ berücksichtigt:

$$\mathbf{P}_i = \left(\sigma_i^2 \cdot \mathbf{C}_{yy,i}\right)^{-1} \tag{4.20}$$

Will man jedoch auf eine Gewichtsmatrix verzichten (z.B. um die Rechenzeit niedrig zu halten), muss die Gewichtung auf eine andere Weise erfolgen. Ausgehend von $\mathbf{C}_{yy} = \mathbf{I}$ ergibt sich

$$\mathbf{N} = \sum_{i=1}^{q} \mathbf{A}_i^T\mathbf{P}_i\mathbf{A}_i = \sum_{i=1}^{q} \mathbf{A}_i^T \frac{1}{\sigma_i^2}\mathbf{I}\mathbf{A}_i = \sum_{i=1}^{q} \frac{1}{\sigma_i^2} \cdot \mathbf{A}_i^T\mathbf{A}_i \tag{4.21}$$

für q Beobachtungsgruppen.

4.9 Regularisierung

Sehr große Gleichungssystene können schlecht konditioniert und damit instabil werden. Sie lassen sich dann nicht mehr zuverlässig lösen. Um dieses Problem zu umgehen, muss das Gleichungssystem *regularisier*t werden. Dazu wird die Normalgleichungsmatrix manipuliert gemäß:

$$\mathbf{N} = \mathbf{N} + \alpha \cdot \mathbf{K} \tag{4.22}$$

Dabei ist α der Regularisierungsparameter und $\mathbf{K}$ die Regularisierungsmatrix. Letztere ist häufig eine Diagonalmatrix, Gleichung 4.22 kann dann zu

$$N_{ii} = N_{ii} + \alpha \cdot K_{ii}. \tag{4.23}$$

vereinfacht werden. Folglich muss auch nicht die gesamte Matrix $\mathbf{K}$ in der Dimension $u \times u$ im Speicher gehalten werden, sondern es genügt, deren Hauptdiagonale zu speichern und zu verarbeiten.

Man unterscheidet verschiedene Möglichkeiten, die Regularisierungsmatrix $\mathbf{K}$ aufzubauen. Für die sog. KAULA-Stabilisierung beispielsweise werden die Elemente entsprechend der inversen Kaula-Regel zu $10^{10} \cdot k^4$ gewählt (der Wert ist also vom Grad abhängig). Für α werden oft Erfahrungswerte angenommen. Genaue Parameter können über das L-Curve Kriterium oder über General Cross Validation (GCV) bestimmt werden [KUSCHE, KLEES 2002].

Kapitel 5

Der LSQR-Ansatz

5.1 Zweck

Im vorherigen Kapitel wurde der Brute-Force-Ansatz erläutert. Dieser weist den Nachteil auf, dass auf jedem beteiligten Rechnerknoten die gesamte Normalgleichungsmatrix im Speicher gehalten werden muss. Deshalb wurde zusätzlich ein zweites, iteratives Verfahren betrachtet. Dieser LSQR-Algorithmus (least squares basierend auf QR-Zerlegung) verarbeitet immer nur eine Zeile der Designmatrix und kommt so mit sehr wenig Speicher aus. Das macht ihn auch für Cluster gut geeignet.

Besonderes Augenmerk wurde auf die vorkonditionierte Variante des LSQR gelegt. Durch die Vorkonditionierung benötigt der PC-LSQR (preconditioned LSQR) Algorithmus wesentlich weniger Iterationen als der herkömmliche LSQR-Algorithmus, um eine Lösung des Gleichungssystems zu erzielen.

Hier wird der Algorithmus nur kurz beschrieben. Eine detailierte Beschreibung findet sich in [PAIGE, SAUNDERS 1982/1, PAIGE, SAUNDERS 1982/2].

5.2 Algorithmus LSQR

Abbildung 5.1 zeigt in einem Flussdiagramm den Aufbau des LSQR-Algorithmus. Rechenzeitlich relevant ist besonders der erste Schritt der Lanczos-Bidiagonalisierung, da in diesem die komplette Designmatrix $\mathbf{A}$ zeilenweise abgearbeitet wird.

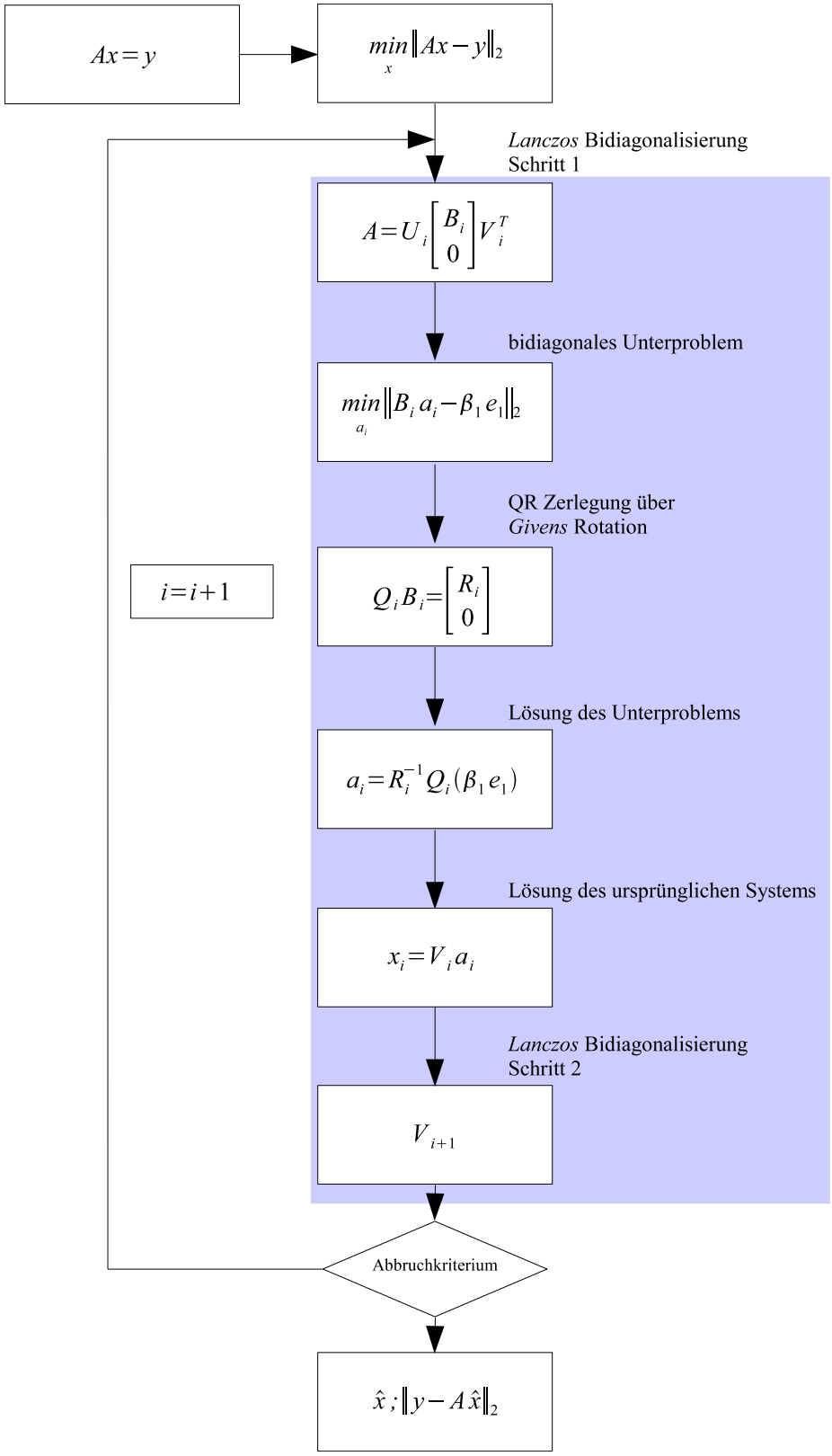


Abbildung 5.1: Flussdiagramm des LSQR-Algorithmus

5.3 Algorithmus PC-LSQR

Abbildung 5.2 zeigt die Struktur des PC-LSQR-Algorithmus. Wesentliche Änderung ist die Vorkonditionierung mittels der Matrix $\mathbf{N}_{bd}$. Dies zieht auch eine Änderung im eigentlichen LSQR-Algorithmus nach sich. Im zweiten Schritt der Lanczos-Bidiagonalisierung wird die Designmatrix $\mathbf{A}$ ein zweites Mal benötigt.

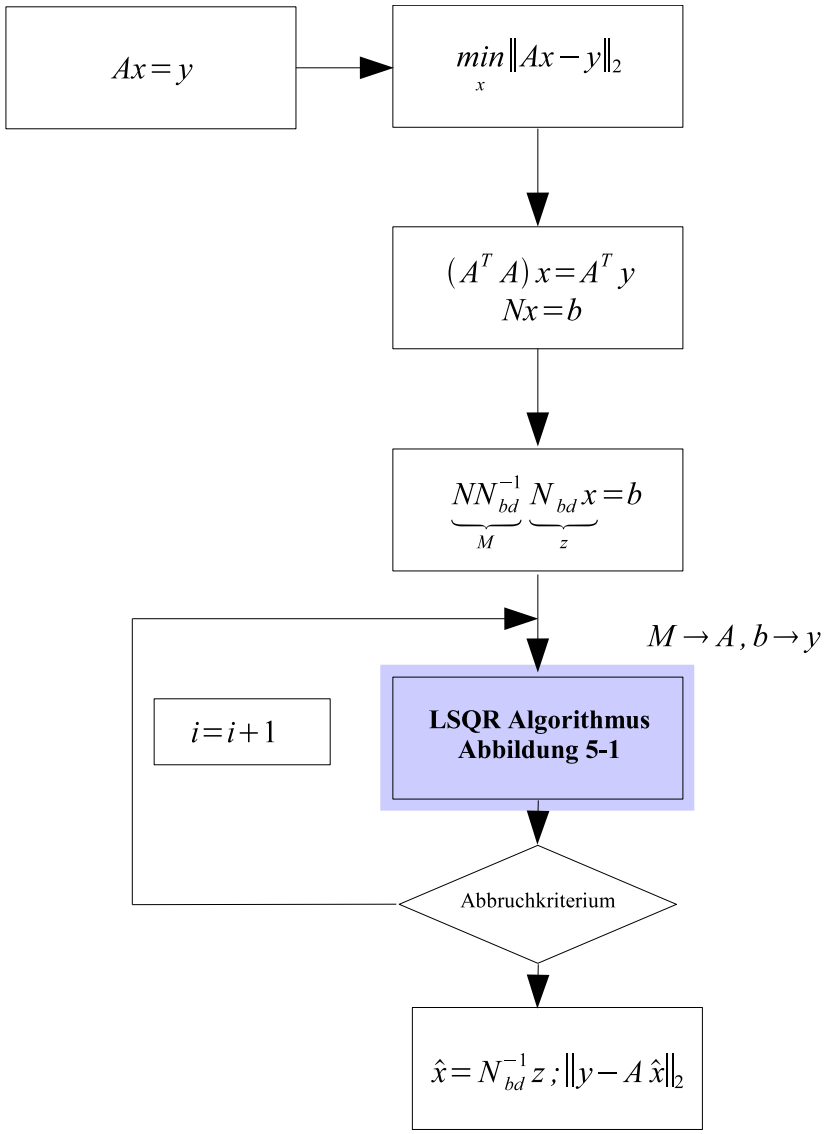


Abbildung 5.2: Flussdiagramm des PC-LSQR-Algorithmus

5.4 Abbruchkriterium

Wie bereits erläutert, handelt es sich beim LSQR-Algorithmus um ein iteratives Verfahren. Die Berechnung kann abgebrochen werden, sobald eine hinreichende Genauigkeit erreicht worden ist. Als Abbruchkriterium dient die sog. *gcv*-Funktion (*general cross validation*) [GOLUB, VON MATT 1997]. Sobald diese ihr globales Minimum erreicht, ist Konvergenz erreicht, und die Berechnung kann abgebrochen werden. Sie berechnet sich folgendermaßen:

$$gcv_i = \frac{\|\mathbf{A}\widehat{\mathbf{x}}_i - \mathbf{y}\|}{n - i} \tag{5.1}$$

Der Wert der gcv-Funktion ergibt sich also aus der Verbesserungsquadratsumme geteilt durch die Differenz von Beobachtungsanzahl und der aktuellen Iterationsanzahl i.

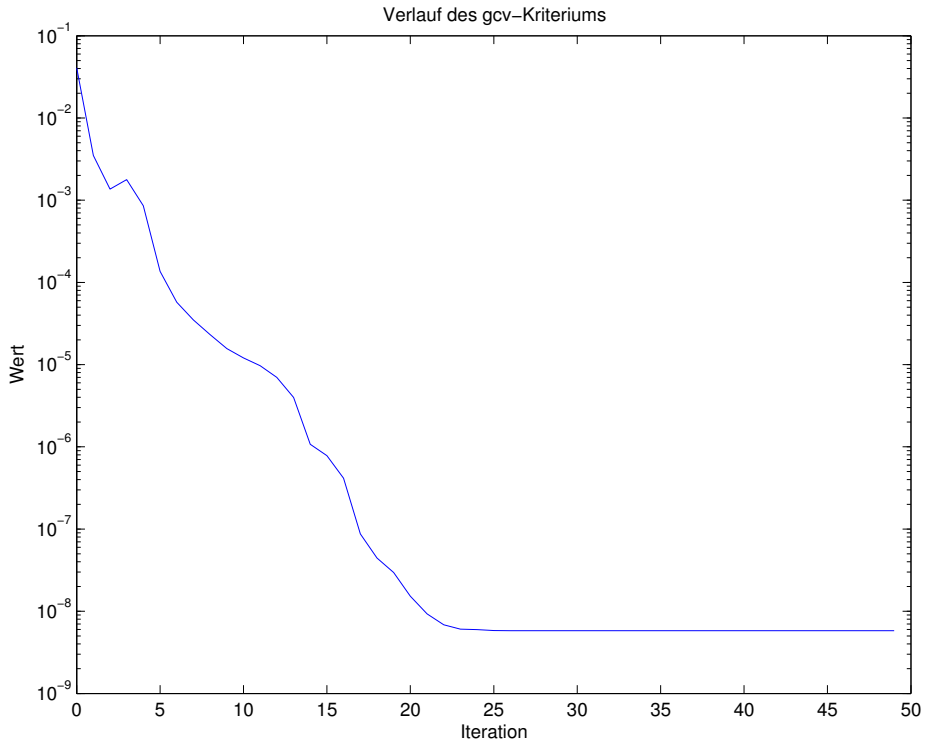


Abbildung 5.3: Verlauf der gcv-Funktion

Tatsächlich ist der Verlauf der gcv-Funktion wie in Abbildung 5.3 dargestellt. Es können mehrere lokale Minima auftreten. Deshalb wird nach Feststellung eines lokalen Minimums noch eine frei definierbare Anzahl von Iterationen berechnet, um zu untersuchen, ob in diesen die gcv-Funktion einen noch geringeren Wert annimmt. Ist dies der Fall, so wird dieses als neues Minimum angenommen. Dieser Ablauf ist in Abbildung 5.4 dargestellt.

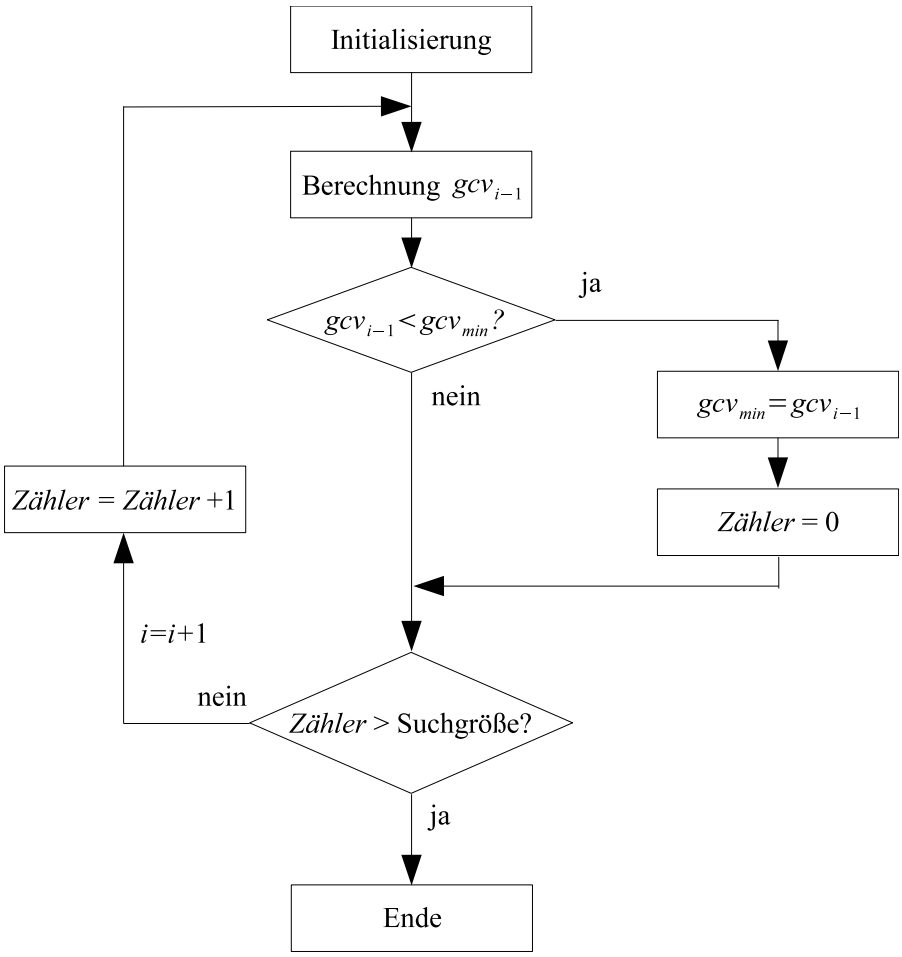


Abbildung 5.4: Ablauf der Minimumssuche

5.5 Aufbau der Vorkonditionierungs-Matrix

Wie in Abschnitt 5.3 erläutert wurde, wird bei der vorkonditionierten Variante des LSQR- Algorithmus die Matrix $\mathbf{N}_{bd}$ benötigt. Durch diese werden vorab Informationen über das Gleichungssystem in die Berechnung eingeführt, was für eine drastische Reduzierung der für eine Lösung benötigten Iterationsanzahl sorgt.

Bei der Matrix $\mathbf{N}_{bd}$ handelt es sich um die Normalgleichungsmatrix $\mathbf{N}$ in einer Blockdiagonalstruktur. Die Blöcke bestehen aus den zu einer Ordnung der Reihenentwicklung gehörenden Elementen (für eine Erklärung dieses Vorgehens siehe z.B. [DITMAR, KLEES 2002]). Dadurch sind die meisten Elemente der resultierenden Matrix gleich Null, bei geschickter Speicherung sinkt der Speicherbedarf gegenüber einer vollbesetzten Normalgleichungsmatrix erheblich. Auch sinkt die zur Berechnung benötigte Rechenzeit, da nicht das komplette Produkt $\mathbf{A}^T\mathbf{A}$ berechnet werden muss, sondern nur das Produkt zwischen den Elementen einer Ordnung. Die Struktur von $\mathbf{N}_{bd}$ ist in Abbildung 5.5 dargestellt.

Benötigt wird nicht nur die Matrix $\mathbf{N}_{bd}$, sondern auch deren Inverse. Für deren Berechnung lassen sich die einzelnen Blöcke von $\mathbf{N}_{bd}$ einzeln invertieren, so dass die Inversion sehr schnell erfolgt.

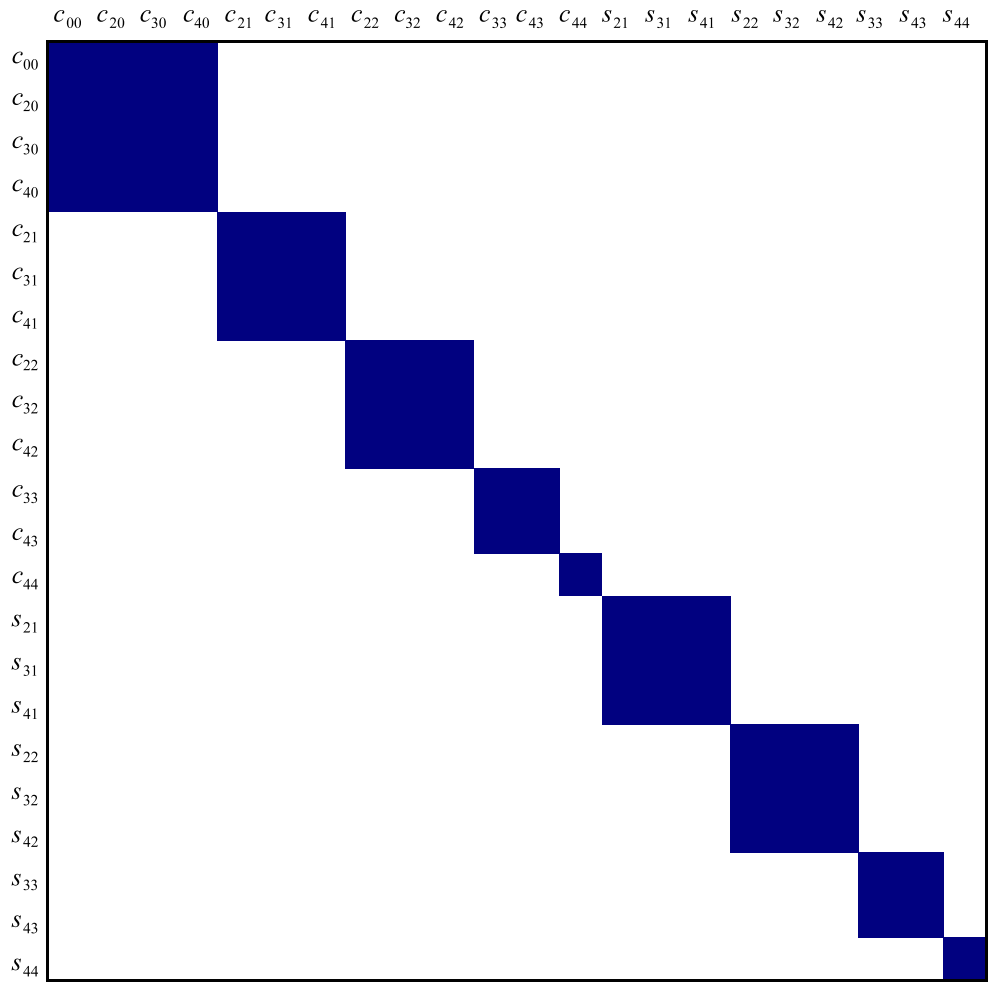


Abbildung 5.5: Struktur der Vorkonditionierungs-Matrix $\mathbf{N}_{bd}$

5.6 Parallelisierung

Um Ansatzpunkte für die Parallelisierung zu finden, wurde eine Zeitmessung an den interessanten Stellen innerhalb der seriellen Programme durchgeführt. Dabei wurde festgestellt, dass praktisch die gesamte Rechenzeit für eine Iteration im *Lanczos*-Schritt in der Schleife über alle Beobachtungen verbraucht wird. Diese Schleife war also zu parallelisieren.

Da die Schleife über alle Beobachtungen in der vorkonditionierten Variante PC-LSQR zweimal auftritt, dauert die Berechnung eines Iterationsschritts auch doppelt so lange wie beim einfachen LSQR-Algorithmus. Damit der PC-LSQR eine Geschwindigkeitssteigerung gegenüber dem normalen LSQR-Algorithmus bringt, muss durch die Vorkonditionierung die Anzahl der für die Lösung des Gleichungssystems benötigten Iterationsanzahl also mindestens halbiert werden.

5.6.1 Parallelisierung mit OpenMP

Die Rechenzeit in einem Iterationsschritt entfällt, wie oben beschrieben, auf die Schleife über alle Beobachtungen, und zwar auf die Berechnung der Zeile i der Designmatrix $\mathbf{A}$.

Die Parallelisierung mit OpenMP ist, wie in Quellcode 5-1 dargestellt, sehr einfach. Die Beobachtungen werden dazu mittels `#pragma omp parallel for` auf die einzelnen Prozessoren verteilt.

Beim PC-LSQR-Algorithmus erfolgt dies für den zweiten *Lanczos*-Schritt in gleicher Weise.

Quellcode 5-1 Parallelisierung des LSQR-Algorithmus mittels OpenMP

```
...
#pragma omp parallel for
for(i=0; i<n; i++)
{
  ...
  Berechnung des Lanczos-Schrittes
  ..
}
...
```

5.6.2 Parallelisierung mit MPI

Zusätzlich wurden beide Algorithmen auch noch mit MPI parallelisiert, um sie für die Verwendung auf Clustern geeignet zu machen. Dazu berechnet jeder Knoten den *Lanczos*-Schritt für einen Teil der Beobachtungen. Wenn alle Knoten ihre Berechnung beendet haben, kombinieren sie ihre Einzelergebnisse (hier einen Vektor $\mathbf{v}$) und generieren daraus ein Gesamtergebnis, das wieder an alle Prozessoren verteilt wird (mit der Funktion `MPI_Allreduce()`, siehe Quellcode 5-2). Die Berechnungen außerhalb der Schleifen über alle Beobachtungen werden von allen Prozessoren durchgeführt, da der Zeitaufwand für die Kommunikation hier größer wäre als der für die Berechnung.

Quellcode 5-2 Parallelisierung des LSQR-Algorithmus mittels MPI

```
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
worksize = n/np;

#pragma omp parallel for
for(i=myrank*worksize; i<(myrank+1)*worksize; i++)
{
  ...
  Berechnung des Lanczos-Schrittes
  ..
}

MPI_Allreduce(v, v2, u, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
...
```

5.7 Regularisierung

Auch bei Verwendung des LSQR-Algorithmus kommt es bei großen Gleichungssystemen zu Instabilitäten, die das Lösen des Systems erschweren. Aus diesem Grund muss hier auch gegebenenfalls regularisiert werden, wie es in Abschnitt 4.9 für den Gauss-Markov-Algorithmus erklärt wurde.

Da der optimale Regularisierungsparameter α nicht bekannt ist, wird das System mit mehreren Regularisierungsparametern α_j gelöst. Für jeden Regularisierungsparameter wird eine eigene Lösung mit dem dazugehörigen gcv-Funktionswert berechnet. Die Lösung mit dem geringsten gcv-Wert wird als endgültige Lösung herangezogen. Die Stärke des LSQR-Algorithmus besteht

darin, dass im Gegensatz zu dem häufig verwendeten iterativen Verfahren der konjugierten Gradienten die Hinzunahme mehrerer Regularisierungsparameter α_j den Rechenaufwand nur unwesentlich steigert (siehe dazu auch Abschnitt 6.11).

5.7.1 Abbruchkriterium

Die bisherige gcv-Funktion, wie in Gleichung 5.1 dargestellt, ist bei der regularisierten Variante des LSQR-Algorithmus nicht mehr als Abbruchkriterium verwendbar. Es muss die in Gleichung 5.2 dargestellte Abwandlung verwendet werden [KILMER, O'LEARY 2001].

$$gcv_{i,j} = \frac{\|\mathbf{A} \cdot \widehat{\mathbf{x}}_{i,j} - \mathbf{y}\|}{n - \eta^T \cdot \mathbf{A} \cdot \xi_{i,j}} \tag{5.2}$$

η ist ein Pseudozufallsvektor, der mit den Werten +1 und -1 mit einer Wahrscheinlichkeit von je 0,5 besetzt ist. ξ ist die Lösung des Systems, die sich ergibt, wenn statt $\mathbf{y}$ der Zufallsvektor η als Beobachtungsvektor verwendet wird. Das Abbruchkriterium muss nicht nur für jede Iteration, sondern auch für jeden Regularisierungsparameter berechnet werden.

Kapitel 6

Ergebnisse

6.1 Übersicht

Produkt dieser Diplomarbeit sind fünf Programme:

- BRUTUS (Brute Force Untersuchung des Schwerefeldes): Implementierung des Brute Force-Ansatzes für High-Low SST und SGG zur Verwendung für CHAMP und GOCE.
- GMAP (Grace Mission Analysis Program): Implementierung des Brute Force-Ansatzes für High-Low SST und Low-Low SST zur Verwendung für GRACE.
- LSQR: Implementierung des LSQR-Algorithmus für High-Low SST.
- PC-LSQR: Implementierung des PC-LSQR-Algorithmus für High-Low SST.
- PC-LSQR regularisiert: Implementierung des PC-LSQR-Algorithmus für High-Low SST mit der Option zur Regularisierung.

Mit diesen fünf Programmen wurden umfangreiche Berechnungen durchgeführt, um sämtliche Funktionalitäten zu testen und die unterschiedlichen Ansätze und Rechnersysteme beurteilen zu können. Um welche Berechnungen es sich dabei handelt, wird nachfolgend kurz erläutert. Die detailierten Ergebnisse dieser Berechnungen sind ab Abschnitt 6.4 zusammengefasst.

6.1.1 Auswirkung der Parallelisierung

Es sollte untersucht werden, inwieweit die Verwendung mehrerer CPUs eine Geschwindigkeitssteigerung bringt. Die wesentlichen Parameter der durchgeführten Berechnungen sind in Tabelle 6.1 aufgeführt. Die Ergebnisse finden sich in Abschnitt 6.4.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Rechner	CPU-Anzahl
BRUTUS	HL-SST	100	TX-7	1,2,4,8,12,15
GMAP	LL-SST	50	TX-7	1,2,4,8,12,15
LSQR	HL-SST	50	TX-7	1,2,4,8,12,15
PC-LSQR	HL-SST	50	TX-7	1,2,4,8,12,15
PC-LSQR reg.	HL-SST	50	TX-7	1,2,4,8,12,15
BRUTUS	HL-SST	50	Strider	2,4,8,12

Tabelle 6.1: Berechnungen zur Beurteilung der Parallelisierung

6.1.2 Vergleich der Ansätze

Im Rahmen dieser Diplomarbeit wurden zwei Ansätze in Form von mehreren Programmen implementiert. Diese sollten bezüglich ihrer Geschwindigkeit und Effizienz untersucht und miteinander verglichen werden. Eine Übersicht der durchgeführten Berechnungen gibt Tabelle 6.2, die detailierten Ergebnisse finden sich in Abschnitt 6.5.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Rechner	CPU-Anzahl
BRUTUS	HL-SST	50	TX-7	1
GMAP	LL-SST	50	TX-7	1
LSQR	HL-SST	50	TX-7	1
PC-LSQR	HL-SST	50	TX-7	1
PC-LSQR reg.	HL-SST	50	TX-7	1

Tabelle 6.2: Berechnungen zum Vergleich der Ansätze

6.1.3 Vergleich unterschiedlicher Problemgrößen

Sowohl für den Brute-Force- als auch für den LSQR-Ansatz wurde getestet, wie sich die Problemgröße auf die Rechenzeit auswirkt. Dies geschah mit den in Tabelle 6.3 dargestellten Parametern. Die Ergebnisse sind in Abschnitt 6.6 erläutert.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Rechner	CPU-Anzahl
BRUTUS	HL-SST	20, 50, 70, 100, 120, 150, 200	TX-7	8
LSQR	HL-SST	20, 50, 70, 100	TX-7	8
PC-LSQR	HL-SST	20, 50, 70, 100	TX-7	8

Tabelle 6.3: Berechnungen zum Vergleich unterschiedlicher Problemgrößen

6.1.4 Vergleich unterschiedlicher Rechnersysteme

Es war für diese Diplomarbeit möglich, mehrere Rechnersysteme (beschrieben in Abschnitt 3.4) mit verschiedenen Architekturen zu verwenden. Diese sollten hinsichtlich ihrer Eignung für die verschiedenen Ansätze untersucht werden, was mit den in Tabelle 6.4 zusammengefassten Programmläufen geschah. Die Ergebnisse dazu finden sich in Abschnitt 6.7.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Rechner	CPU-Anzahl
BRUTUS	HL-SST	50	AzusA	15
BRUTUS	HL-SST	50	TX-7	8
BRUTUS	HL-SST	50	SX-6	12
BRUTUS	HL-SST	50	Strider	5
PC-LSQR	HL-SST	50	AzusA	15
PC-LSQR	HL-SST	50	TX-7	8
PC-LSQR	HL-SST	50	SX-6	12
PC-LSQR	HL-SST	50	Strider	5

Tabelle 6.4: Berechnungen zum Vergleich unterschiedlicher Rechnersysteme

6.1.5 Beurteilung der Gewichtsmatrix

Wie in Abschnitt 4.6 beschrieben wurde, wurde eine Methode entwickelt, um auch bei blockweiser Berechnung der Normalgleichungsmatrix $\mathbf{N}$ sämtliche Korrelationen zwischen den Blöcken berücksichtigen zu können. Zum Test dieser Funktionalität wurden die in Tabelle 6.5 dargestellten Berechnungen durchgeführt, die Ergebnisse werden in Abschnitt 6.8 diskutiert.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Rechner	CPU-Anzahl	Gewichtung
BRUTUS	HL-SST	100	TX-7	8	keine
BRUTUS	HL-SST	100	TX-7	8	ein Schritt
BRUTUS	HL-SST	100	TX-7	8	zwei Schritte

Tabelle 6.5: Berechnungen zur Beurteilung der Gewichtsmatrix

6.1.6 Untersuchung der Varianzkomponentenschätzung

Um die Kombination mehrerer Beobachtungstypen zu ermöglichen, wurde die Varianzkomponentenschätzung realisiert (siehe Abschnitt 4.7). Es sollte untersucht werden, ob die Monte Carlo-Varianzkomponentenschätzung die gleichen Ergebnisse liefert wie die strenge Varianzkomponentenschätzung, ob sich unterschiedliches Rauschen in den Varianzkomponenten äußert, und ob die Kombination mehrere Beobachtungstypen funktioniert. Die hierzu durchgeführten Berechnungen sind in Tabelle 6.6 zusammengefasst, die Ergebnisse dazu finden sich in Abschnitt 6.9).

Programm	Beobachtungen	Rauschniveau	$\mathbf{k_{max}}$
BRUTUS	HL-SST, SGG	-	100
BRUTUS	HL-SST	-	100
BRUTUS	HL-SST	5 / 5 / 30 mm	100
BRUTUS	HL-SST	30 / 30 / 5 mm	100
BRUTUS	HL-SST	30 / 30 / 30 mm	100
BRUTUS	HL-SST, SGG	-	70

Tabelle 6.6: Berechnungen zur Untersuchung der Varianzkomponentenschätzung

6.1.7 Untersuchung der Regularisierung

Es wurde in beiden Ansätzen die Möglichkeit zur Regularisierung geschaffen, um das Stabilisieren schlecht konditionierter Gleichungssysteme zu ermöglichen. Es war zu untersuchen, ob die Regularisierung funktioniert, und inwieweit die Regularisierung des LSQR-Algorithmus die Laufzeit beeinflusst. Dazu wurden die in Tabelle 6.7 aufgeführten Berechnungen durchgeführt. Die Ergebnisse dieser Berechnungen werden in den Abschnitten 6.10 und 6.11 erläutert.

Programm	Beobachtungen	$\mathbf{k_{max}}$	Anzahl Regularisierungsparameter
BRUTUS	SGG	200	-
BRUTUS	SGG	200	1
PC-LSQR	HL-SST	50	-
PC-LSQR reg.	HL-SST	50	1
PC-LSQR reg.	HL-SST	50	5

Tabelle 6.7: Berechnungen zur Untersuchung der Regularisierung

6.2 Verwendete Daten

Derzeit stehen GRACE Realdaten nur einem eingeschränlten Kreis von Nutzern und nur über einen Zeitraum von vier Monaten zur Verfügung. Der Satellit GOCE wird erst Ende 2006 seine operative Phase einnehmen. Deshalb wurde für diese Arbeit auf simulierte Daten zurückgegriffen, die aus dem EGM96-Schwerefeld generiert wurden. Das bietet den Vorteil, dass man die erzielten Ergebnisse direkt verifzieren kann, da aus der Berechnung wieder die EGM96-Koeffizienten resultieren müssen. Die verwendeten Orbitparameter sind in Tabelle 6.8 aufgeführt (im Fall GRACE gilt die mittlere Anomalie M für GRACE 1, GRACE 2 folgt in der gleichen Bahn mit 30 Sekunden Abstand). Eine detailierte Beschreibung dieses Themas findet sich u.a. in [GÖTZELMANN 2003].

Um den Einfluss des Messrauschens auf die Schwerefeldlösung (bzw. der Gewichtung auf die Rechenzeit) untersuchen zu können, wurden die simulierten Satellitenpositionen mit verschiedenen Rauschmodellen versehen.

Bahnparameter	GRACE	GOCE
a	6.778.000 m	6.628.000 m
e	0,001	0,001
i	89,5°	96,6°
Ω	0°	0°
ω	0°	0°
M	0°	0°

Tabelle 6.8: Startwerte der Orbit-Simulation

6.3 Gradvarianzen

Für die Beurteilung der Güte der Ergebnisse wurden die Gradvarianzen herangezogen. Diese berechnen sich laut Gleichung 6.1

$$\sigma_k = \sqrt{\sum_{l=0}^{k} \frac{(c_{k,l} - \widehat{c}_{k,l})^2 + (s_{k,l} - \widehat{s}_{k,l})^2}{2k+1}} \tag{6.1}$$

aus der Differenz zwischen den wahren Schwerefeldkoeffizienten $c_{k,l}$ und $s_{k,l}$ (also den Koeffizienten des Schwerefeldmodells, mit dem die Satellitenbahn simuliert wurde) und den berechneten Schwerefeldkoeffizienten $\widehat{c}_{k,l}$ und $\widehat{s}_{k,l}$.

6.4 Auswirkung der Parallelisierung

6.4.1 BRUTUS

Zur Bewertung der Auswirkung der Parallelisierung wurden Rechnungen mit einem maximalen Entwicklungsgrad k_{max}= 100 und 500.000 Datensätzen durchgeführt. Verwendet wurden nur SST-Beobachtungen. Als Rechner kam die NEC TX-7 zum Einsatz. Die erzielten Laufzeiten sind in Tabelle 6.9 aufgeführt und in Abbildung 6.1 grafisch dargestellt.

Anzahl CPUs	wall time	user time
1	638m45s	638m38s
2	300m2s	596m18s
4	149m6s	595m10s
8	76m25s	606m14s
12	53m26s	637m47s
15	47m56s	709m41s

Tabelle 6.9: Auswirkung der Parallelisierung von BRUTUS

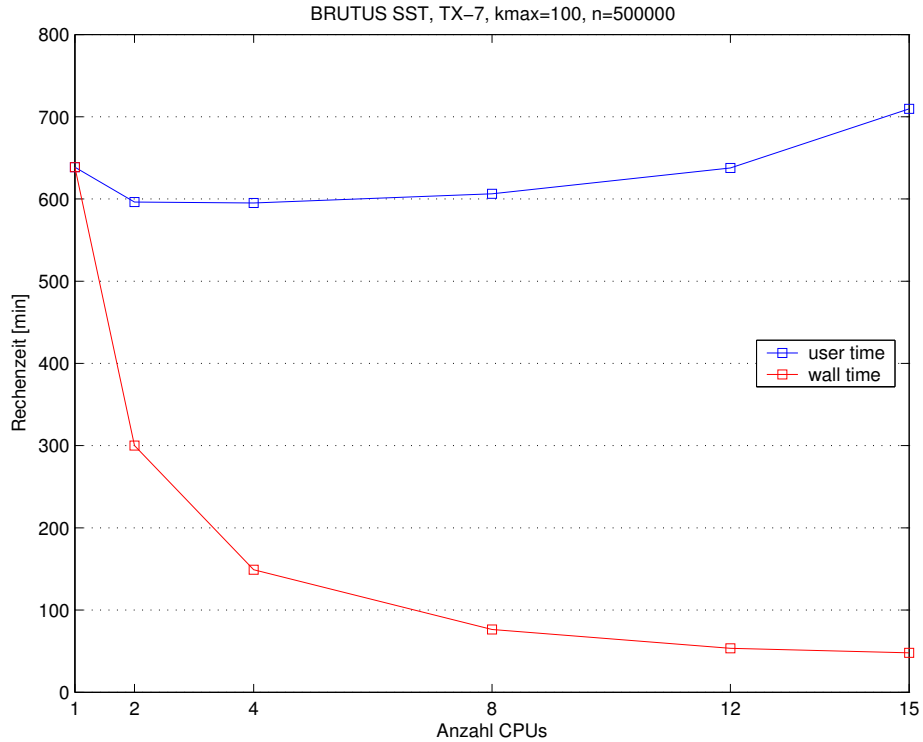


Abbildung 6.1: Auswirkung der Parallelisierung von BRUTUS

Die Ergebnisse entsprechen den Erwartungen bzw. übertreffen sie teilweise sogar. Eine Steigerung von einer auf zwei CPUs sorgt interessanterweise sogar für eine Beschleunigung um mehr als den Faktor zwei. Bis einschließlich acht CPUs ist die Steigerung praktisch linear - eine Verdoppelung der CPU-Anzahl sorgt für eine Halbierung der Rechenzeit (bzw. der wall time - die user time bleibt gleich).

Über acht CPUs ist die Steigerung nicht mehr linear. Das sieht man auch daran, dass die user time jetzt deutlich ansteigt. Diesen Sachverhalt findet man häufiger, weshalb die vorherrschende Meinung ist, dass OpenMP nur bis 8 CPUs optimal skaliert ist, und darüber hinaus weniger effizient wird.

Interessant ist auch, wie sich der Algorithmus auf einem Cluster verhält. Hierzu wurde mit unterschiedlicher Anzahl von Knoten Berechnungen auf dem Cray Strider-Cluster durchgeführt (mit k_{max} = 50, Tabelle 6.10). Auch hier skaliert der Algorithmus nahezu linear. Der Einfluss der MPI-Kommunikation auf die Rechenzeit ist in diesem Fall also klein.

Anzahl Knoten	wall time
1	50m33s
2	25m53s
4	13m3s
6	9m16s

Tabelle 6.10: Laufzeit in Abhängigkeit von der Knotenanzahl

6.4.2 GMAP

Zur Beurteilung der Güte der Parallelisierung von GMAP wurden Berechnungen mit k_{max} = 50 und 500.000 Beobachtungen (nur LL-SST) auf der TX-7 durchgeführt. Die Ergebnisse sind in Tabelle 6.11 und in Abbildung 6.2 dargestellt.

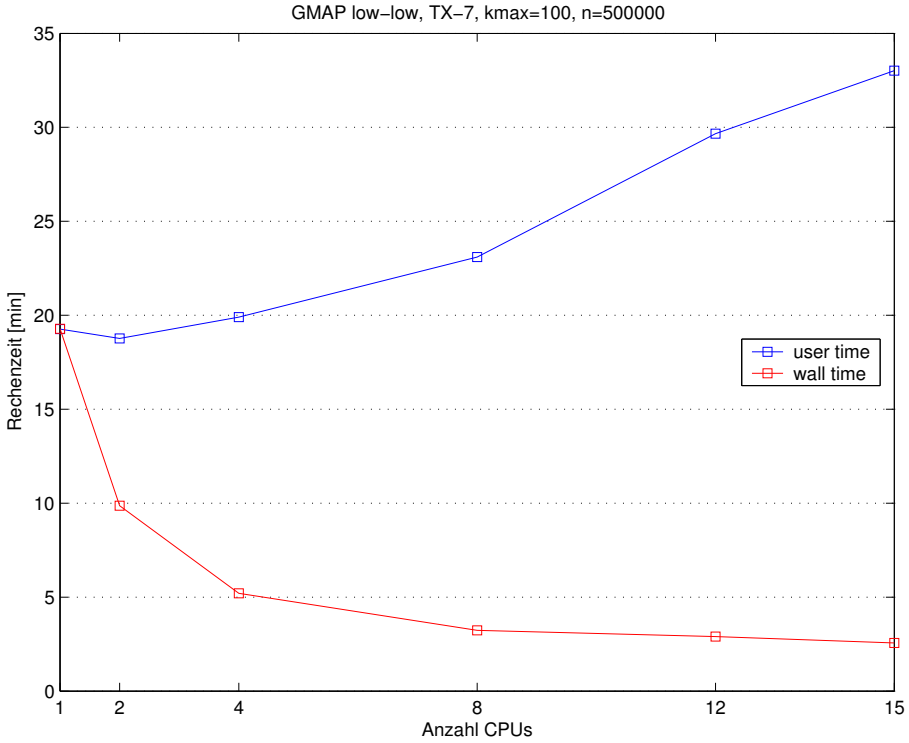


Abbildung 6.2: Auswirkung der Parallelisierung von GMAP

Anzahl CPUs	wall time	user time
1	19m17s	19m16s
2	9m52s	18m46s
4	5m12s	19m54s
8	3m14s	23m6s
12	2m54s	29m40s
15	2m34s	33m1s

Tabelle 6.11: Auswirkung der Parallelisierung von GMAP

Es tritt wie bei BRUTUS der Effekt auf, dass bei einer Berechnung mit 2 CPUs weniger user time benötigt wird als bei der Berechnung mit einer CPU. Darüber hinaus verhält sich GMAP scheinbar schlechter als BRUTUS und skaliert nicht linear. Zu bedenken ist dabei jedoch, dass mit BRUTUS k_{max} = 100 berechnet wurde, mit GMAP jedoch nur k_{max} = 50. In letzterem Fall ist der Rechenaufwand zu gering, um von einer größeren CPU-Anzahl deutlich zu profitieren. Dieser Effekt würde sich bei BRUTUS mit k_{max} = 50 oder geringer auch zeigen. Für größere Probleme ist davon auszugehen, dass GMAP in gleicher Weise skaliert wie BRUTUS, nämlich nahezu linear.

6.4.3 LSQR

Auch für LSQR wurden Rechnungen auf der TX-7 durchgeführt. Hier betrug der maximale Entwicklungsgrad $k_{max} = 50$, es wurden 500.000 Beobachtungen verwendet, für die Lösung wurden 170 Iterationen benötigt. Die resultierenden Laufzeiten sind in Tabelle 6.3 und in Abbildung 6.4 dargestellt.

Anzahl CPUs	wall time	user time
1	317m35s	317m32s
2	166m11s	328m29s
4	86m7s	333m9s
8	47m28s	350m31s
12	32m32s	337m56s
15	37m28s	341m48s

Abbildung 6.3: Auswirkung der Parallelisierung von LSQR

Der LSQR skaliert nicht exakt linear mit der Anzahl der Prozessoren, aber immer noch sehr gut. Die Verwendung von 15 CPUs bringt gegenüber 12 CPUs keine Beschleunigung, sondern eine Verlangsamung der Rechenzeit (der wall time). Die user time hingegen stieg nur unwesentlich an. Bei der Verwendung von 15 CPUs ist der Verwaltungsaufwand für das System relativ hoch, so dass das Starten und Beenden von Threads, die Verteilung der Schleifenduchläufe auf die Threads und deren Synchronisation einen Mehraufwand bedeutet, der die Berechnung verlangsamen kann, besonders dann, wenn der Verwaltungsaufwand im Vergleich zum Rechenaufwand sehr hoch ist.

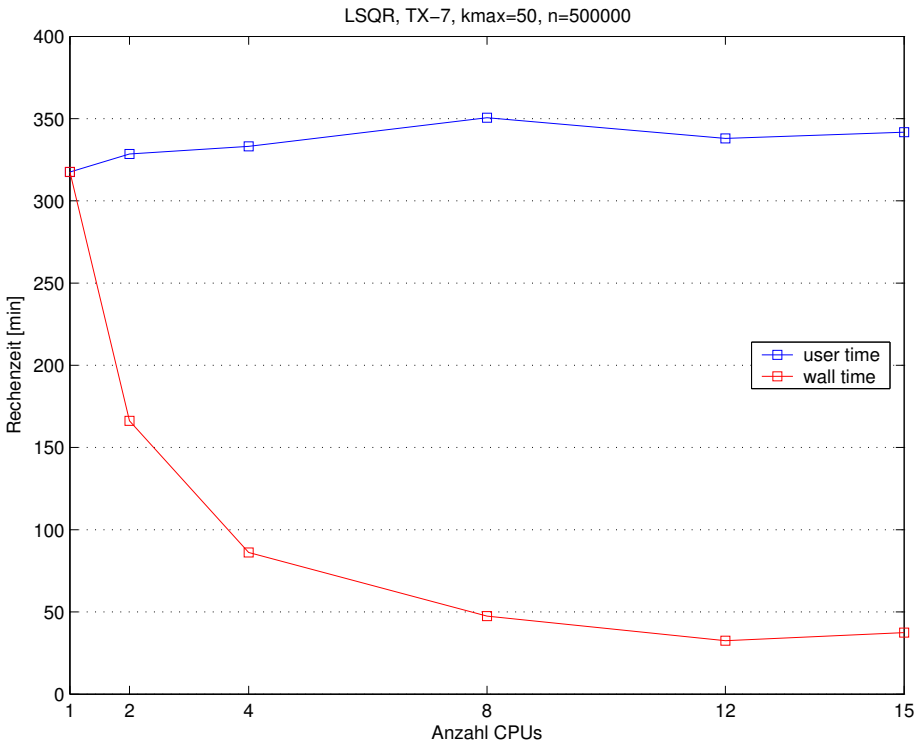


Abbildung 6.4: Auswirkung der Parallelisierung von LSQR

6.4.4 PC-LSQR

Für die Rechnungen mit PC-LSQR wurden die gleichen Rahmenbedingungen wie für LSQR gewählt. Zur Lösung des Systems wurden 12 Iterationen benötigt. Die entsprechenden Laufzeiten sind in Tabelle 6.5 und in Abbildung 6.6 aufgeführt.

Anzahl CPUs	wall time	user time
1	79m13s	79m10s
2	40m50s	80m31s
4	21m3s	82m41s
8	10m38s	83m20s
12	7m16s	84m5s
15	8m51s	85m41s

Abbildung 6.5: Auswirkung der Parallelisierung von PC-LSQR

PC-LSQR verhält sich hinsichtlich der Rechenzeit im wesentlichen wie LSQR. Das Programm skaliert etwas besser (nahezu linear bis zu 12 CPUs), zeigt aber auch den leichten Anstieg der Rechenzeit bei 15 CPUs gegenüber 12 CPUs.

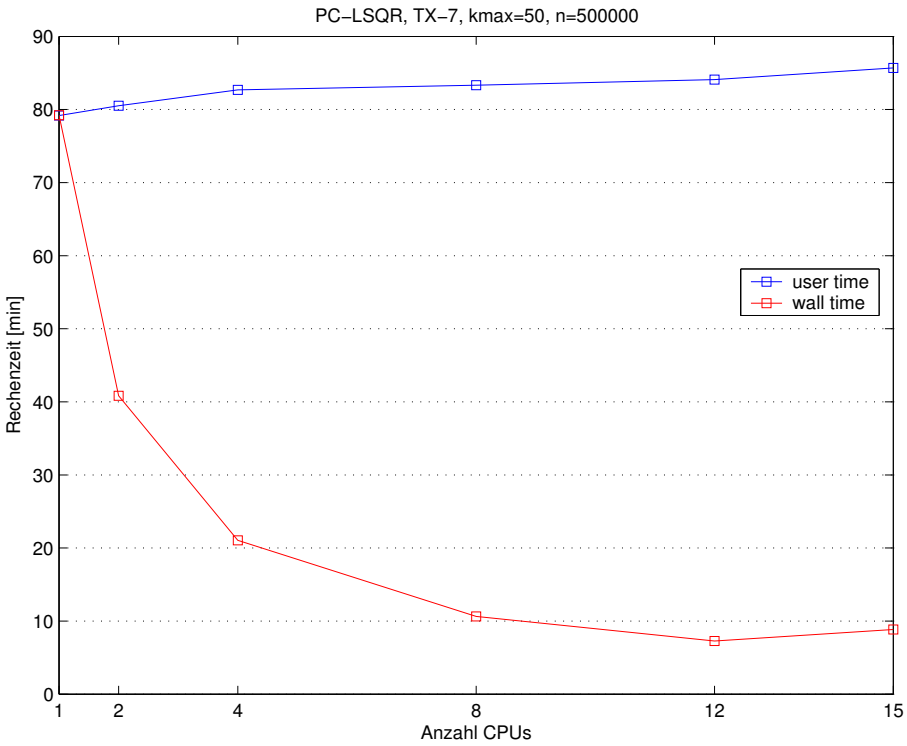


Abbildung 6.6: Auswirkung der Parallelisierung von PC-LSQR

6.4.5 PC-LSQR regularisiert

Auch die Rechnungen mit der regularisierten Variante des PC-LSQR wurden auf der TX-7 durchgeführt, wieder mit k_{max} = 50 und 500.000 Beobachtungen, woraus sich ebenfalls 12 Iterationen ergaben. Die Resultate finden sich in Tabelle 6.7 und in Abbildung 6.8. Ein Vergleich mit der nicht regularisierten Version, der auch die Laufzeitunterschiede erläutert, findet sich in Abschnitt 6.11.

Anzahl CPUs	wall time	user time
1	84m36s	84m35s
2	45m0s	86m37s
4	23m49s	87m52s
8	13m30s	89m51s
12	9m53s	90m4s
15	8m5s	91m9s

Abbildung 6.7: Auswirkung der Parallelisierung von PC-LSQR regularisiert

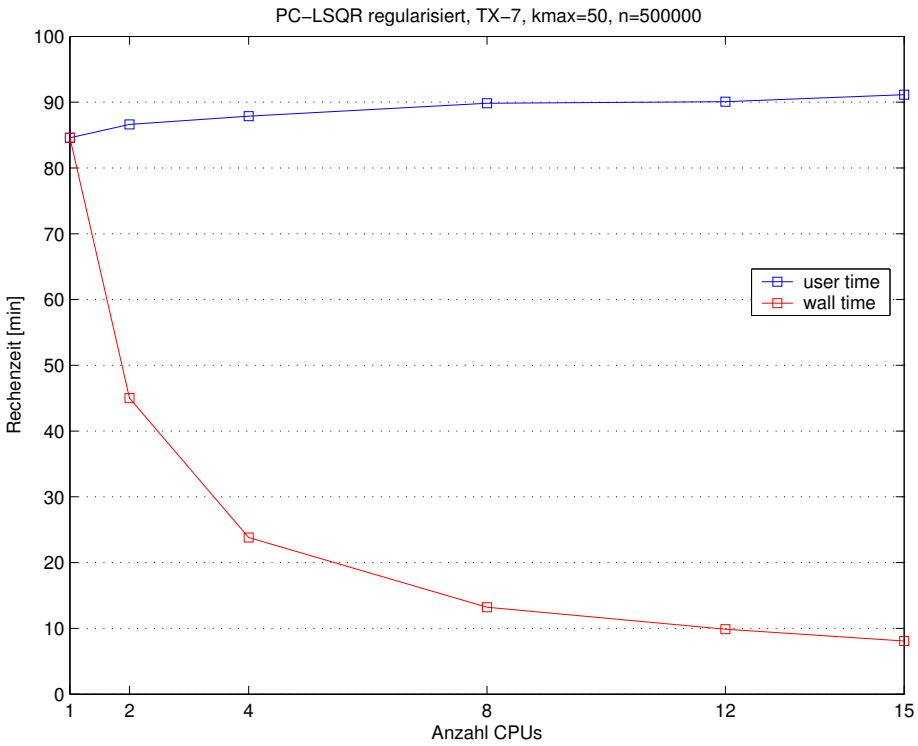


Abbildung 6.8: Auswirkung der Parallelisierung von PC-LSQR regularisiert

Das regularisierte PC-LSQR-Programm verhält sich erwartungsgemäß wie die nicht regularisierte Variante, mit einer Ausnahme: Bei Verwendung von 15 CPUs ergibt sich noch eine Beschleunigung der Berechnung gegenüber 12 CPUs. Dies rührt vermutlich daher, dass der Rechenaufwand pro Iteration etwas größer und das Verhältnis von Rechen- zu Kommunikationsaufwand damit besser ist.

6.5 Vergleich der Ansätze

Zum Vergleich der verschiedenen Programme und damit der verschiedenen Ansätze wurde jedes Programm auf der TX-7 ausgeführt. Es wurde jeweils nur eine CPU verwendet, die Performancemessung war angeschaltet (was gegenüber einer Berechnung ohne Performancemessung einen leichten Leistungsverlust bedeutet, da das System die Anzahl der ausgeführten Instruktionen und andere Parameter messen muss). Die Koeffizienten wurden bis Grad und Ordnung 50 geschätzt, es wurden 500.000 Datensätze verwendet. Hierbei ist jedoch zu bedenken, dass BRUTUS SST alle drei Satellitenkoordinaten verwendet, während LSQR, PC-LSQR und PC-LSQR regularisiert nur die radiale Komponente r verwenden. Auch GMAP low-low verwendet nur eine Beobachtung.

Die Ergebnisse sind in Tabelle 6.12 zusammengefasst und in Abbildung 6.9 grafisch dargestellt. GMAP low-low erzielt die geringste Laufzeit. Würde bei BRUTUS SST aber nur die radiale Komponente verwendet, würde sich hier die Laufzeit auf etwa ein Drittel reduzieren (gut 14 Minuten), was dann wieder schneller wäre als GMAP. Das deckt sich mit der Messung einer höheren Performance und damit auch Effizienz.

Das GMAP trotz praktisch gleichem Algorithmus eine geringere Performance erreicht als BRUTUS liegt an den aufwändigeren Berechnungen, die für das Besetzen der Designmatrix vonnöten sind. Es müssen für zwei Satelliten partielle Ableitungen und Legendresche Funktionen

Programm	Rechenzeit	Performance	Effizienz
BRUTUS SST	43m7s	3,6 GFLOPS	60%
GMAP low-low	19m17s	2,9 GFLOPS	48%
LSQR	317m35s	1,9 GFLOPS	32%
PC-LSQR	79m13s	1,7 GFLOPS	28%
PC-LSQR regularisiert	84m13s	1,7 GFLOPS	28%

Tabelle 6.12: Vergleich der Programme

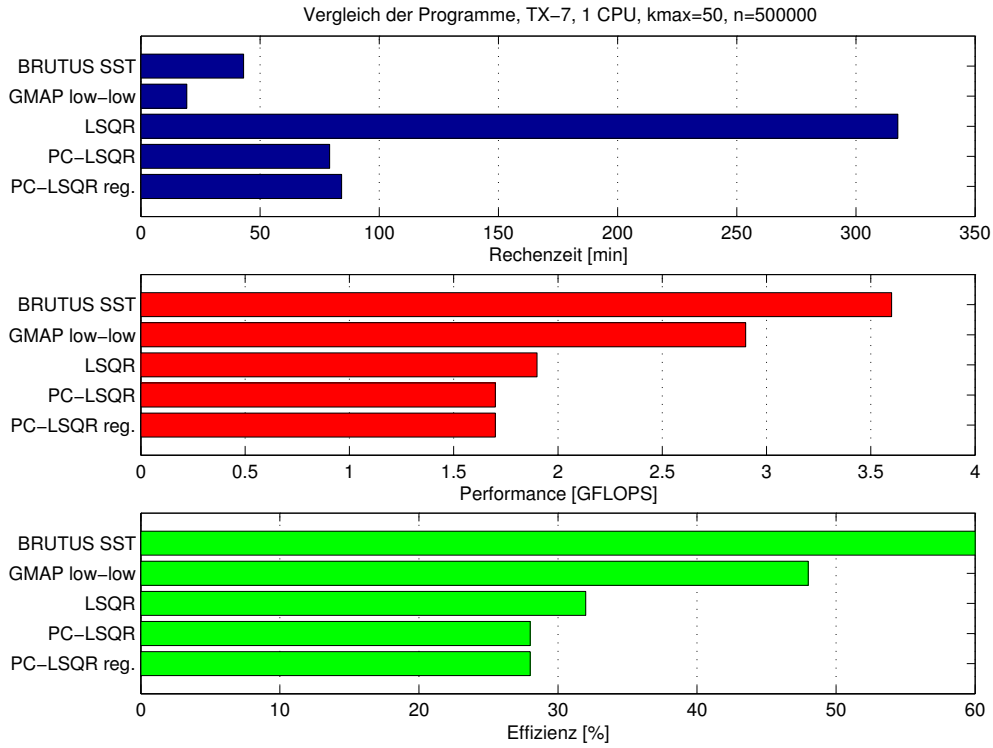


Abbildung 6.9: Vergleich der Programme

berechnet werden, und dies geschieht mit deutlich geringerer Geschwindigkeit als die Matrizenmultiplikation. Die resultierende Effizienz ist mit etwa 50 bis 60% für einen Skalarrechner sehr gut. Bei größeren Problemen oder angeschalteter Gewichtsmatrix (wenn praktisch die gesamte Rechenzeit nur für die Matrizenmultiplikation benötigt wird) steigt diese sogar weiter an. Im Fall k_{max} = 100 und Gewichtung in zwei Schritten wurde mit BRUTUS eine Performance von 4,6 GFLOPS erreicht, was einer Effizienz von 77% entspricht. Das System ist damit nahezu optimal ausgelastet.

Im Vergleich zu den beiden Brute-Force-Programmen fallen die iterativen Programme deutlich ab. Die Effizienz ist mit etwa 30% durchaus akzeptabel (es werden hier ja nur relativ kleine Vektoren verarbeitet), aber die Rechenzeit besonders des LSQR-Programms ist deutlich schlechter als die von BRUTUS und GMAP. PC-LSQR zeigt, dass die Vorkonditionierung einen deutlichen Vorteil bringt, da in diesem Fall die Rechenzeit um etwa den Faktor vier zurückgeht. Warum die regularisierte Variante des PC-LSQR eine etwas höhere Laufzeit benötigt wird in Abschnitt 6.11 erläutert.

6.6 Vergleich unterschiedlicher Problemgrößen

6.6.1 BRUTUS

Für BRUTUS wurde die Auswirkung unterschiedlicher Problemgrößen auf die Rechenzeit ermittelt. Die Berechnungen wurden auf der NEC TX-7 mit 8 CPUs durchgeführt, es wurden 500.000 Datensätze für die Beobachtungen verwendet. Die sich ergebenden Laufzeiten sind in Tabelle 6.13 und in Abbildung 6.10 zusammengefasst.

$\mathbf{k_{max}}$	wall time	user time	Lösungszeit
20	0m52s	5m26s	0,8s
50	6m58s	54m7s	1,7s
70	19m38s	155m30s	7,3s
100	77m53s	620m26s	30,0s
120	153m11s	1173m21s	142,7s
150	351m47s	2805m9s	301,9s
200	1119m32s	8685m32s	1303,6s

Tabelle 6.13: Auswirkung der Problemgröße auf die Rechenzeit, BRUTUS

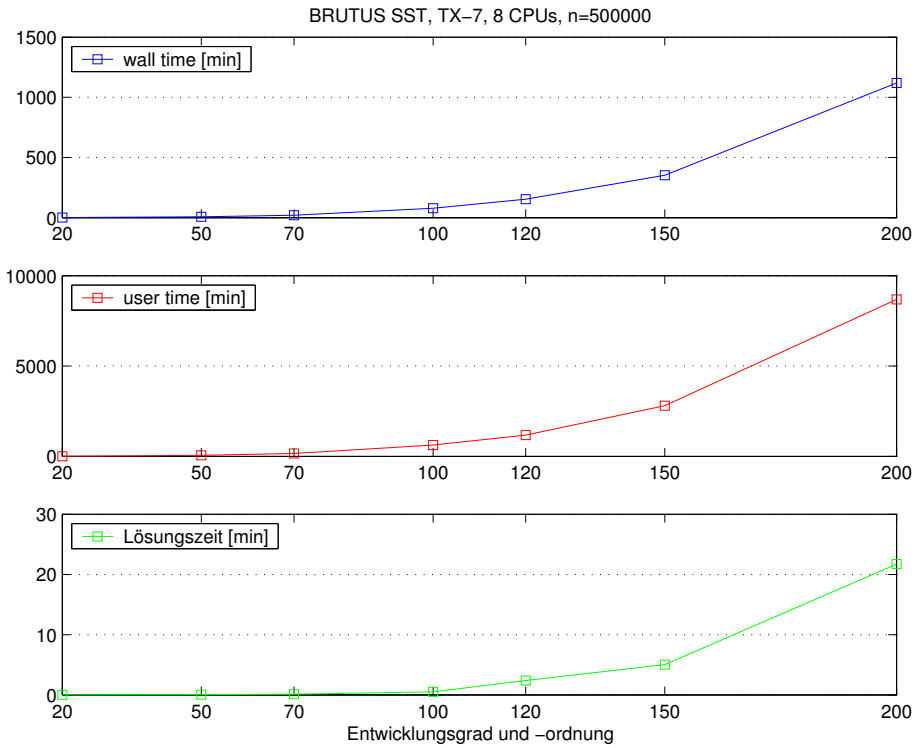


Abbildung 6.10: Auswirkung der Problemgröße auf die Rechenzeit, BRUTUS

Als maximale Problemgröße wurde eine Berechnung mit dem Entwicklungsgrad 200 durchgeführt. Größere Probleme waren leider nicht berechenbar, da die Anzahl der Elemente der Normalgleichungsmatrix $\mathbf{N}$ dann nicht mehr in einer 32 bit-Integer-Variable gespeichert werden kann.

Die Verwendung von 64 bit-Integern war nicht möglich, da die Intel MKL intern nur 32 bit-Integer verwendet.

Die Steigerung des Entwicklungsgrads sorgt für die erwartete Steigerung der Rechenzeit. Selbst bei k_{max} = 200 liegt diese mit knapp 19 Stunden aber immer noch in einem erträglichen Rahmen. Auch hier trägt die Rechenzeit zur Lösung des Systems mit gut 20 Minuten nur unwesentlich zur gesamten benötigten Rechenzeit bei.

6.6.2 LSQR

Auch für den LSQR-Algorithmus wurde der Einfluss unterschiedlicher Problemgrößen auf die Rechenzeit ermittelt. Hierzu wurden 500.000 Beobachtungen verwendet, als Rechner kam die NEC AzusA zum Einsatz, die Berechnung wurde parallel auf 12 Prozessoren durchgeführt. Die Ergebnisse sind in Tabelle 6.14 und in Abbildung 6.11 dargestellt.

$\mathbf{k_{max}}$	wall time	user time	Anzahl Iterationen
20	9m9s	81m56s	93
50	67m1s	684m11s	170
70	230m38s	2472m48s	336
100	2046m43s	23761m12s	1663

Tabelle 6.14: Auswirkung der Problemgröße auf die Rechenzeit, LSQR

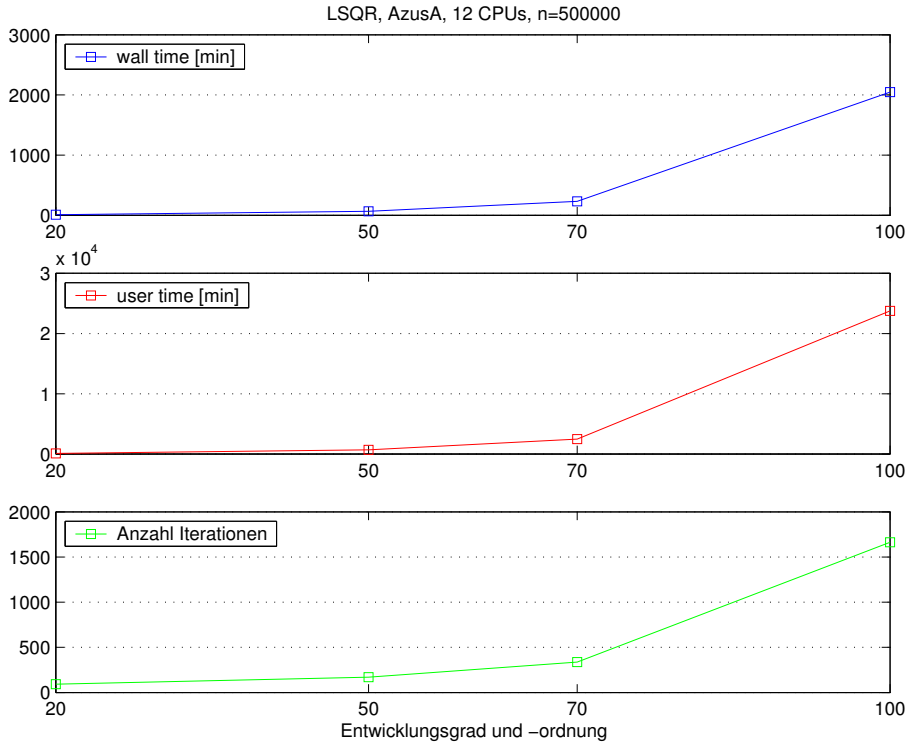


Abbildung 6.11: Auswirkung der Problemgröße auf die Rechenzeit, LSQR

Es ist deutlich erkennbar, dass mit steigender Problemgröße der Rechenaufwand extrem zunimmt. Daher war es nicht möglich, Problemgrößen über k_{max} = 100 zu berechnen, denn bei

dieser Problemgröße werden bereits 34 Stunden Rechenzeit benötigt. Bei Problemgrößen, bei denen die iterative Berechnung interessant wird (da der Speicherbedarf für den Brute Force-Ansatz ansteigt), versagt der LSQR-Ansatz also aufgrund des zu hohen Rechenaufwands.

6.6.3 PC-LSQR

Wie im vorherigen Abschnitt gezeigt wurde, verhält sich der LSQR-Algorithmus bei großen Problemen sehr ungünstig. Um abzuschätzen, wie sich der PC-LSQR Algorithmus verhält, wurden auch für diesen mehrere Rechnungen durchgeführt. Es kam wieder die AzusA mit 12 CPUs und 500.000 Beobachtungen zum Einsatz. Die Ergebnisse finden sich in Tabelle 6.15 und in Abbildung 6.12.

$\mathbf{k_{max}}$	**wall time**	**user time**	**Anzahl Iterationen**
20	3m21s	33m48s	11
50	17m46s	205m48s	12
70	48m30s	572m36s	22
100	735m53s	8512m5s	274

Tabelle 6.15: Auswirkung der Problemgröße auf die Rechenzeit, PC-LSQR

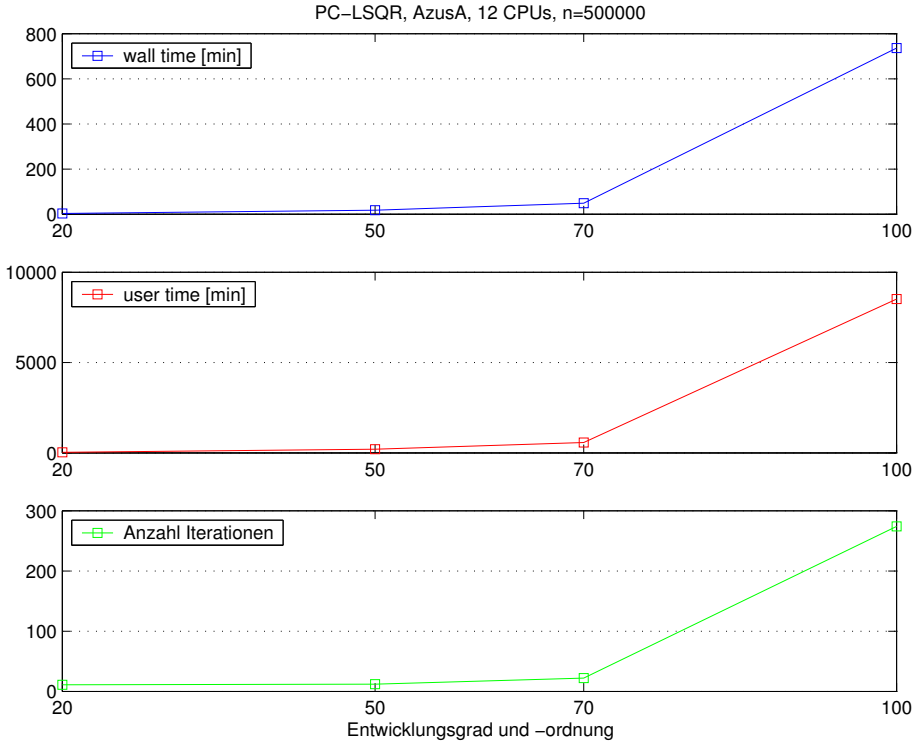


Abbildung 6.12: Auswirkung der Problemgröße auf die Rechenzeit, PC-LSQR

Bis einschließlich k_{max} = 70 hält sich die Rechenzeit und die Anzahl der benötigten Iterationen in vernünftigen Grenzen. Die benötigte Rechenzeit ist zwar etwas höher als beim Brute Force-Ansatz, aber doch deutlich geringer als beim LSQR-Algorithmus.

Aus dem Rahmen fällt hingegen die Berechnung bei k_{max} = 100. Die Berechnung benötigt plötzlich sehr viele Iterationen bis zur Konvergenz, dementsprechend steigt auch die Rechenzeit sehr stark an. Dieser Effekt lässt sich etwas mildern, wenn mehr Beobachtungen verwendet werden. Bei einer Million Beobachtungen werden nur noch 75 Iterationen benötigt, die Rechenzeit fällt dadurch auf etwas mehr als die Hälfte ab (nur etwa ein Viertel der Iterationen, von denen jede aber doppelt so lange dauert).

Bei k_{max} = 120 wird dieser Effekt noch stärker. Eine Berechnung mit 3 Millionen Beobachtungen musste nach einer Woche Rechnung auf 12 CPUs abgebrochen werden, da keine schnelle Konvergenz erkennbar war. Der PC-LSQR scheint also für die Problemgrößen, für die er aus Speichergründen gegenüber dem Brute Force-Ansatz interessant ist, nicht geeignet.

6.7 Vergleich unterschiedlicher Rechnersysteme

Für diese Diplomarbeit standen die in Abschnitt 3.4 beschriebenen Hochleistungsrechner, welche am Höchstleistungsrechenzentrum Stuttgart installiert sind, als Platformen zur Verfügung. Zum Vergleich dieser Rechner (und zum Test ihrer Eignung für die verschiedenen Ansätze) wurden auf allen Rechnern zwei Programmdurchläufe durchgeführt. Es wurden die Programme BRUTUS SST und PC-LSQR verwendet, jeweils mit einem maximalen Entwicklungsgrad von k_{max} = 50 und 500.000 Datensätzen. Die kleine Problemgröße wurde gewählt, um die Rechenzeit auf den kostenpflichtigen Rechnern SX-6 und Strider klein zu halten. Um die Vergleichbarkeit herzustellen, wurden alle Berechnungen bei etwa der gleichen theoretischen Spitzenleistung durchgeführt:

- Auf der NEC AzusA wurden 15 Prozessoren verwendet, mit einer Leistung von 15 · 3,2 = 48 GFLOPS.
- Auf der NEC TX-7 wurden 8 Prozessoren verwendet, mit einer Leistung von 8 · 6 = 48 GFLOPS.
- Auf der NEC SX-6 wurden 5 Prozessoren verwendet, mit einer Leistung von 5 · 9 = 45 GFLOPS.
- Auf dem Cray Strider wurden 6 Knoten verwendet, mit einer Leistung von 12 · 4 = 48 GFLOPS.

6.7.1 BRUTUS

Die Ergebnisse der Programmläufe mit BRUTUS sind in Tabelle 6.16 zusammengefasst. Die Laufzeiten sind in Abbildung 6.13 grafisch dargestellt, die erzielte Performance in Abbildung 6.14. Die user time auf Strider ist hochgerechnet, da hier der `time`-Befehl nur die wall time ermitteln kann (da `time` nur die Zeit auf dem Verwaltungsknoten nehmen kann, nicht auf den Rechenknoten). Die Ergebnisse sind auf GMAP übertragbar.

Rechner	wall time	user time	Performance	Effizienz
AzusA	12m43s	182m2s	13,7 GFLOPS	29%
TX-7	6m58s	54m7s	25,4 GFLOPS	53%
SX-6	10m22s	38m32s	16,6 GFLOPS	37%
Strider	9m16s	111m12s	18,6 GFLOPS	39%

Tabelle 6.16: Vergleich BRUTUS auf den unterschiedlichen Systemen

Als erstes fällt die Diskrepanz zwischen AzusA und TX-7 auf. Diese beiden Rechner sind sehr ähnlich aufgebaut, sie unterscheiden sich in erster Linie im eingesetzten Prozessortyp (AzusA:

Itanium 1, TX-7: Itanium 2). Der Itanium 2 ist jedoch ein wesentlich weiterentwickelter, effizienterer Prozessor. Er profitiert auch von seinem sehr großen L3-Cache von 6 MB (Prozessoren haben sehr schnelle Zwischenspeicher - genannt L1, L2 und L3-Cache - über die der Prozessor an das eigentliche RAM angebunden ist). Zusätzlich befindet sich die AzusA mit 15 Prozessoren in dem Bereich, in dem OpenMP an seine Grenzen gerät - ein Vergleich bei einer geringeren Spitzenleistung und damit einer geringeren Prozessoranzahl würde die AzusA vermutlich etwas besser aussehen lassen.

Die SX-6 enttäuscht auf den ersten Blick etwas, da sie über drei Minuten langsamer ist als die TX-7, und auch eine geringere Effizienz bringt. Bei diesen Ergebnissen ist zu bedenken, dass die SX-6 mit sehr langsamen Lesezeiten für die Daten auffiel - das Einlesen der 500.000 Datensätze dauerte etwa 200 Sekunden. Berücksichtigt man nur die tatsächliche Rechenzeit von sieben Minuten, erhält man eine Performance von 24,6 GFLOPS, was einer Effizienz von 55% entspricht. Erwarten würde man mehr - die Ursache für die Diskrepanz ist in der recht kleinen Problemgröße zu suchen.

Zur Verifizierung dieser Vermutung wurde noch eine Rechnung mit k_{max} = 100 auf der SX-6 durchgeführt. Aus dieser Rechnung resultierte eine Leistung von 7,9 GFLOPS für eine CPU, die Effizienz beträgt 88% und ist damit nahe am theoretischen Maximum. Es zeigt sich also, dass das Vektorsystem SX-6 für den Brute-Force-Ansatz sehr gut geeignet ist.

Das letzte getestete System ist der Opteron-Cluster Cray Strider. Er reiht sich in etwa zwischen AzusA und TX-7 ein. Die ermittelte Performance ist in Ordnung, da der Opteron einen wesentlich kleineren Cache als der Itanium 2 hat, und zusätzlicher Kommunikationsaufwand durch MPI entsteht.

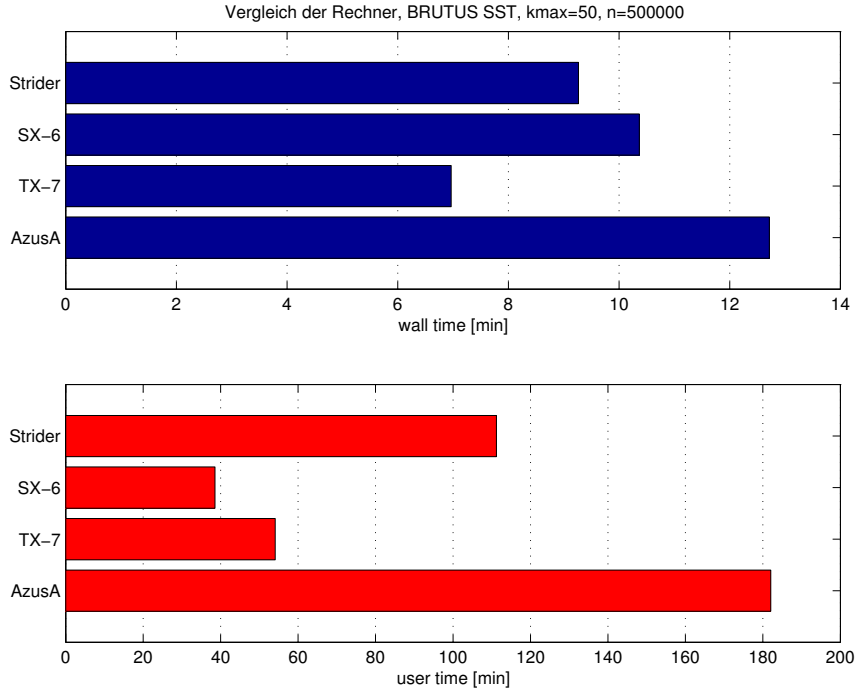


Abbildung 6.13: Rechenzeiten BRUTUS auf den unterschiedlichen Systemen

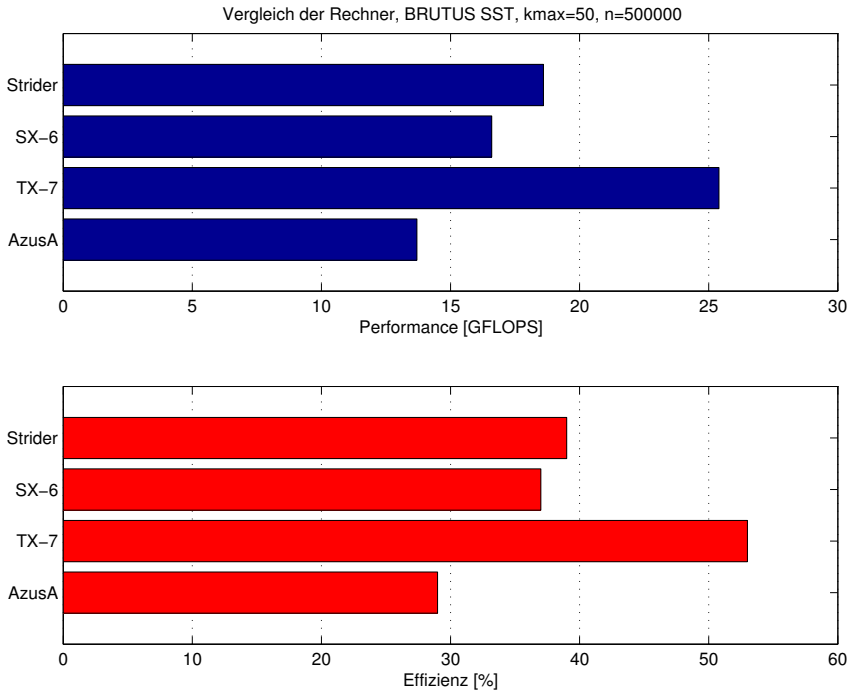


Abbildung 6.14: Performance von BRUTUS auf den unterschiedlichen Systemen

6.7.2 PC-LSQR

Die Ergebnisse der Berechnungen mit PC-LSQR auf den unterschiedlichen Systemen sind in Tabelle 6.17 und in den Abbildungen 6.15 und 6.16 zusammengefasst. Die user time auf Strider ist, wie schon im vorherigen Abschnitt erläutert, hochgerechnet. Die Ergebnisse sind auf den nicht vorkonditionierten LSQR und die regularisierte Variante übertragbar.

Rechner	wall time	user time	Performance	Effizienz
AzusA	15m5s	206m50s	8,7 GFLOPS	18%
TX-7	10m38s	83m20s	12,6 GFLOPS	26%
SX-6	69m34s	326m7s	1,1 GFLOPS	2%
Strider	11m44s	140m48s	11,4 GFLOPS	24%

Tabelle 6.17: Vergleich BRUTUS auf den unterschiedlichen Systemen

Der Vergleich von AzusA und TX-7 fällt diesmal etwas besser für die AzusA aus. Die TX-7 ist nun nicht mehr doppelt so schnell, sondern nur noch etwa um die Hälfte schneller. Da beim PC-LSQR nur verhältnismäßig kleine Vektoren verarbeitet werden, kann der Itanium 2 seine Vorteile (effizientes Ausführen mehrere Operationen, großer Cache) nicht ganz so gut ausspielen.

Die SX-6 erzielt ein denkbar schlechtes Ergebnis. Das Rechnen mit sehr kleinen Vektoren, verbunden mit einem großen Anteil nicht vektorisierbarer Berechnungen (Berechnen von Zeilen der Designmatrix) ist ein Szenario, für das Vektorrechner in keinster Weise geeignet sind. Die SX-6 erfüllt damit die Erwartungen, sie ist für den LSQR-Algorithmus nicht verwendbar.

Der Strider-Cluster ist nur wenig langsamer als die TX-7. Das Ergebnis ist auch deshalb erfreulich, weil die Kommunikationszeit für MPI unter einer Sekunde liegt und damit nicht relevant ist. Wie erhofft ist der PC-LSQR-Algorithmus also gut für Cluster geeignet.

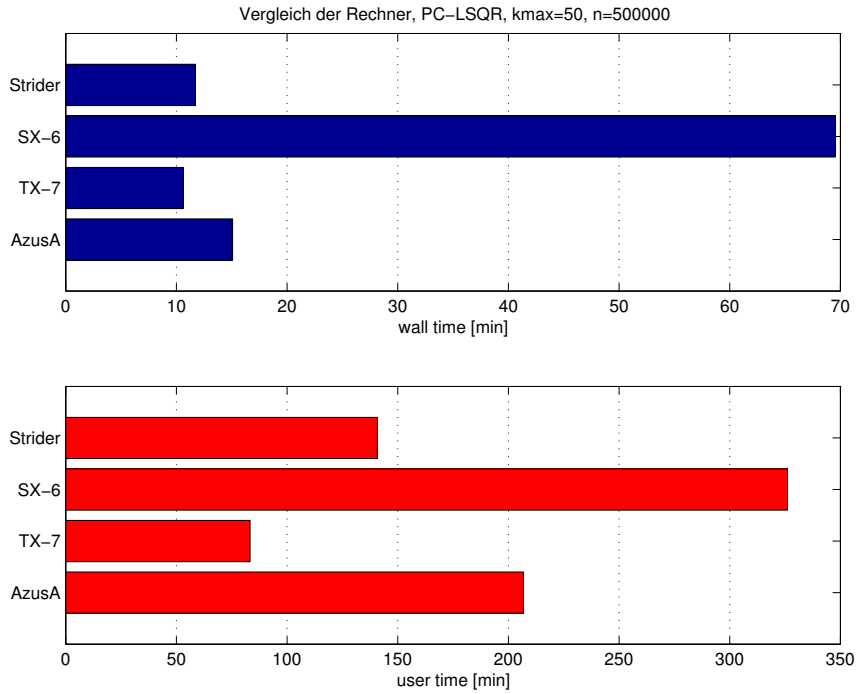


Abbildung 6.15: Rechenzeiten PC-LSQR auf den unterschiedlichen Systemen

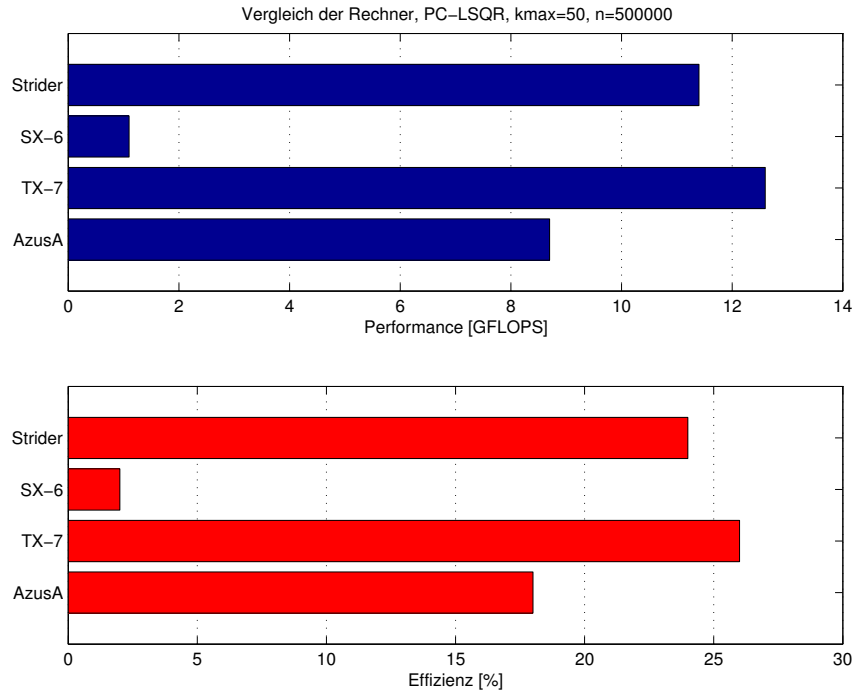


Abbildung 6.16: Performance von PC-LSQR auf den unterschiedlichen Systemen

6.8 Effekt der Gewichtsmatrix

6.8.1 Einfluss auf die Laufzeit

Um den Einfluss der Einführung einer Gewichtsmatrix auf die Laufzeit abschätzen zu können, wurde je eine Berechnung ohne Gewichtung, eine mit Gewichtung in nur einem Schritt (also unter Vernachlässigung der Korrelationen zwischen den Blöcken) und eine mit Gewichtung in zwei Schritten durchgeführt. Als Rechner kam die NEC TX-7 zum Einsatz. Es wurden 8 CPUs verwendet, der maximale Entwicklungsgrad betrug k_{max} = 100. Es wurden 500.000 Datensätze verwendet, die in Blöcke zu je 500 Beobachtungen aufgespalten wurden.

Die resultierenden Laufzeiten sind in Tabelle 6.18 dargestellt.

Art der Gewichtung	wall time	user time
keine	76m25s	606m14s
ein Schritt	154m41s	1192m20s
zwei Schritte	291m29s	2287m50s

Tabelle 6.18: Einfluss der Gewichtsmatrix auf die Laufzeit

Die Gewichtung in einem Schritt verdoppelt in etwa die Laufzeit gegenüber der Berechnung ohne Gewichtung. Dies resultiert aus dem erhöhten Rechenaufwand für die Berechnung $\mathbf{A}^T\mathbf{P}\mathbf{A}$ statt $\mathbf{A}^T\mathbf{A}$. Bei der Berechnung in zwei Schritten verdoppelt sich die Laufzeit erneut, da nun die gesamte Berechnung (bis auf einen Block) ein zweites Mal durchgeführt wird.

6.8.2 Einfluss auf die Lösung

Zur Überprüfung des Einflusses einer Gewichtsmatrix auf die Lösung wurden Simulationen mit Daten durchgeführt, die mit farbigem Rauschen behaftet waren. Getestet wurden sowohl verrauschte Positionen mit BRUTUS als auch Rauschen in den Beschleunigungen mit GMAP. Die Gewichtsmatrix wurde jeweils an das Rauschmodell ρ = 0,99 angepasst, im Fall von BRUTUS wurde also die Fehlerfortpflanzung für den Übergang von Positionen auf Beschleunigungen durchgeführt.

In beiden Fällen brachte die Einführung einer Gewichtsmatrix bei Berechnung in einem Schritt keine signifikante Verbesserung der Genauigkeit der Lösung. Die Gewichtung in zwei Schritten sorgte bei BRUTUS sogar für eine deutliche Verschlechterung der Ergebnisse, bei GMAP hatte sie wiederum keinen signifikanten Einfluss.

Die Ursache für das unterschiedliche Verhalten der beiden Programme ist in der unterschiedlichen Struktur der Gewichtsmatrix zu suchen. Dass die Gewichtung generell keinen Genauigkeitsvorteil bringt, enttäuscht etwas. Vielleicht liegen hier noch methodische Fehler vor, dieser Ansatz sollte auf jeden Fall weiter untersucht werden.

6.9 Varianzkomponentenschätzung

6.9.1 Vergleich von strenger VCE und MCVCE

Zum Vergleich von strenger und Monte Carlo-VCE wurden bei der Schätzung von Koeffizienten bis k_{max} =100 mit BRUTUS Varianzkomponenten über beide Varianten berechnet . Als Beobachtungen wurden je 500.000 Datensätze High-Low SST und SGG verwendet. Die resultierenden Redundanzanteile r_i und Varianzkomponenten s_i^2 sind in Tabelle 6.19 dargestellt.

Beobachtungsgruppe	$\mathbf{r_i}$ VCE	$\mathbf{r_i}$ MCVCE	$\mathbf{s_i^2}$ VCE	$\mathbf{s_i^2}$ MCVCE
λ	497427,8	497450,5	9,48e-20	9,48e-20
φ	497975,9	497452,3	1,53e-19	1,53e-19
r	495324,8	494812,8	1,83e-19	1,83e-19
t_{xx}	500000,0	500000,0	3,19e-34	3,19e-34
t_{yy}	500000,0	500000,0	2,43e-33	2,43e-33
t_{zz}	500000,0	500000,0	3,53e-33	3,53e-33

Tabelle 6.19: Vergleich von strenger VCE und MCVCE

Wie der Vergleich zeigt, sind die berechneten Redundanzanteile nicht identisch. Die relevanten Varianzkomponenten stimmen jedoch auf zwei Nachkommastellen überein. Damit ist der Beweis erbracht, dass die Monte Carlo-Varianzkomponentenschätzung Ergebnisse liefert, die denen der strengen Varianzkomponentenschätzung äquivalent sind.

Wie in Abschnitt 4.7.2 beschrieben wurde, gibt es zwei Gründe, die MCVCE zu verwenden: der deutlich geringere Speicherbedarf und der reduzierte Rechenaufwand. Der geringere Speicherbedarf ergibt sich dadurch, dass nicht für jede Beobachtungsgruppe eine eigene Normalgleichungsmatrix $\mathbf{N}_i$ berechnet werden muss. Im vorliegenden Fall reduziert sich der Speicherbedarf also auf etwa ein Siebtel (von etwa 5,6 GB auf 800 MB).

Der reduzierte Rechenaufwand resultiert aus dem Wegfallen der Multiplikation $\mathbf{N}_i = \mathbf{A}_i^T \mathbf{P}_i \mathbf{A}_i$. Im vorliegenden Fall reduziert sich die Rechenzeit (auf der TX-7, 8 CPUs) für die Varianzkomponentenschätzung von 19580,6 Sekunden für die strenge Varianzkomponentenschätzung auf 677,8 Sekunden für die MCVCE. Das ist eine Steigerung der Berechnungsgeschwindigkeit in etwa um den Faktor 30. Bei höheren Entwicklungsgraden dürfte dieser Vorteil der MCVCE gegenüber der VCE noch größer werden.

Die Ergebnisse zeigen, dass die Monte Carlo-Varianzkomponentenschätzung gegenüber der strengen Varianzkomponentenschätzung enorme Vorteile bringt. Es ist daher angebracht, nur die MCVCE zur Bestimmung der Varianzkomponenten zu verwenden.

6.9.2 Sensitivität gegenüber Rauschen

Im vorhergehenden Abschnitt wurde erläutert, dass die MCVCE im Vergleich zur VCE korrekte Ergebnisse liefert. Von Bedeutung ist aber auch und vor allem, ob die Varianzkomponentenschätzung überhaupt auf verschiedene Rausch- und damit Genauigkeitsniveaus in den Beobachtungen reagiert. Zu diesem Zweck wurden mit BRUTUS Koeffizienten bis Grad und Ordnung 100 geschätzt. Dabei kamen jeweils 500.000 High-Low SST Datensätze zum Einsatz, und zwar in vier Rauschniveaus: fehlerfrei sowie farbiges Rauschen der Größe 5 mm / 5 mm / 30 mm, 30 mm / 30 mm / 5 mm und 30 mm / 30 mm / 30 mm in den drei Positionskoordinaten x / y / z. Das Rauschen in x, y und z entspricht nach der Koordinatentransformation in etwa (aber nicht genau) einem entsprechenden Rauschen in λ, φ und r.

Die Ergebnisse der einzelnen Rechnungen sind in Tabelle 6.20 zusammengefasst. Es ist deutlich zu erkennen, dass mit größerem Rauschen die Gesamtvarianz ebenfalls größer wird. Bei

Beobachtungsgruppe	kein Rauschen	5 / 5 / 30 mm	30 / 30 / 5 mm	30 / 30 / 30 mm
λ	9,48e-20	3,31e-09	1,19e-07	1,19e-07
φ	1,53e-19	6,25e-08	6,07e-08	1,20e-07
r	1,83e-19	6,11e-08	6,23e-08	1,20e-07
Gesamtvarianz	1,44e-19	4,23e-08	8,08e-08	1,20e-07

Tabelle 6.20: Vergleich der Varianzkomponenten bei unterschiedlichem Rauschniveau

gleichem Rauschen in allen drei Komponenten ergeben sich auch gleiche Varianzkomponenten. Etwas schwieriger ist die Interpretation bei unterschiedlichen Rauschniveaus in den einzelnen Komponenten. So haben φ und r bei einem Rauschen von 5 mm eine größere Varianzkomponente als bei 30 mm, solange nicht alle Komponenten mit 30 mm Rauschen behaftet sind.

Dies liegt in zwei Ursachen begründet: einerseits entsprechen die drei Komponenten x, y und z eben nicht den drei Beobachtungsgruppen λ, φ und r. Andererseits beeinflussen sich alle Beobachtungsgruppen gegenseitig, da in den Verbesserungen $\mathbf{v}$ ja der Unbekanntenvektor $\widehat{\mathbf{x}}$ und damit der Einfluss aller Beobachtungen steckt. Die Ergebnisse lassen aber trotzdem erkennen, dass die Varianzkomponentenschätzung zuverlässige Werte liefert.

6.9.3 Kombination von High-Low SST und SGG

Zum Test der Funktionalität der Varianzkomponentenschätzung zur Kombination von High-Low SST und SGG (also der GOCE-Fall) wurden Berechnungen mit k_{max}= 70 und 500.000 Datensätzen durchgeführt. Es wurden sowohl für SST als auch für SGG fehlerfreie Daten verwendet. Zum Vergleich der Ergebnisse wurden die resultierenden Gradvarianzen grafisch dargestellt.

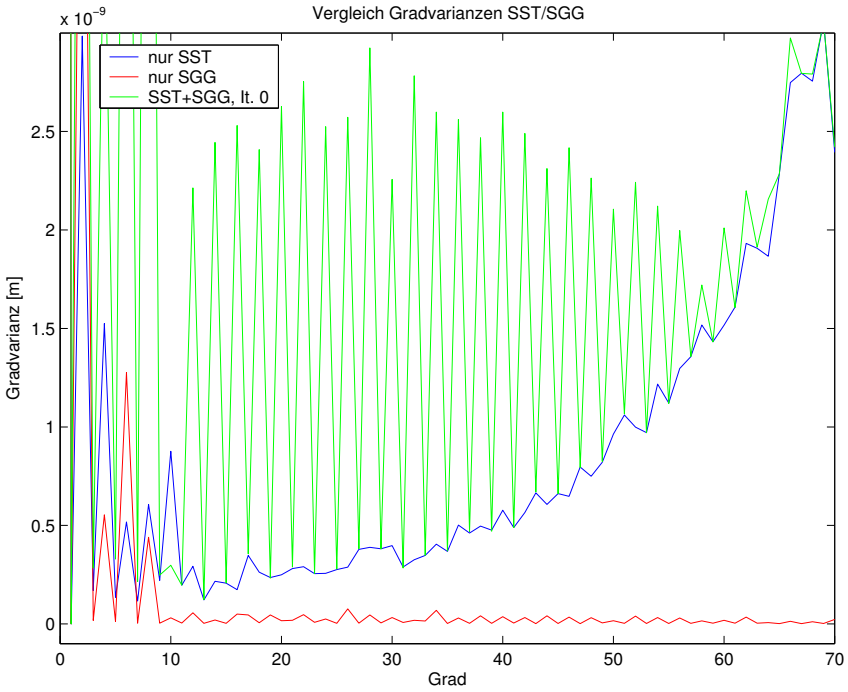


Abbildung 6.17: Vergleich der Gradvarianzen nach Iteration 0

Abbildung 6.17 zeigt die resultierenden Gradvarianzen bei der Verwendung nur der SST-Daten (blau), nur der SGG-Daten (rot) und der Kombination beider Beobachtungstypen ohne relative Gewichtung zueinander (grün). Die SGG-Ergebnisse sind genauer als die SST-Ergebnisse. In der Kombination sind die Ergebnisse jedoch noch schlechter als die SST-Ergebnisse.

Die Varianzkomponentenschätzung ermittelt, dass die SGG-Beobachtungen praktisch keinen Einfluss auf die Lösung haben (Redundanzanteile von 500.000). Ihre Varianzkomponenten sind in etwa um 10^{-16} kleiner als die der SST-Beobachtungen, weshalb dieser Wert als relatives Gewicht für die nächste Berechnung eingeführt wurden.

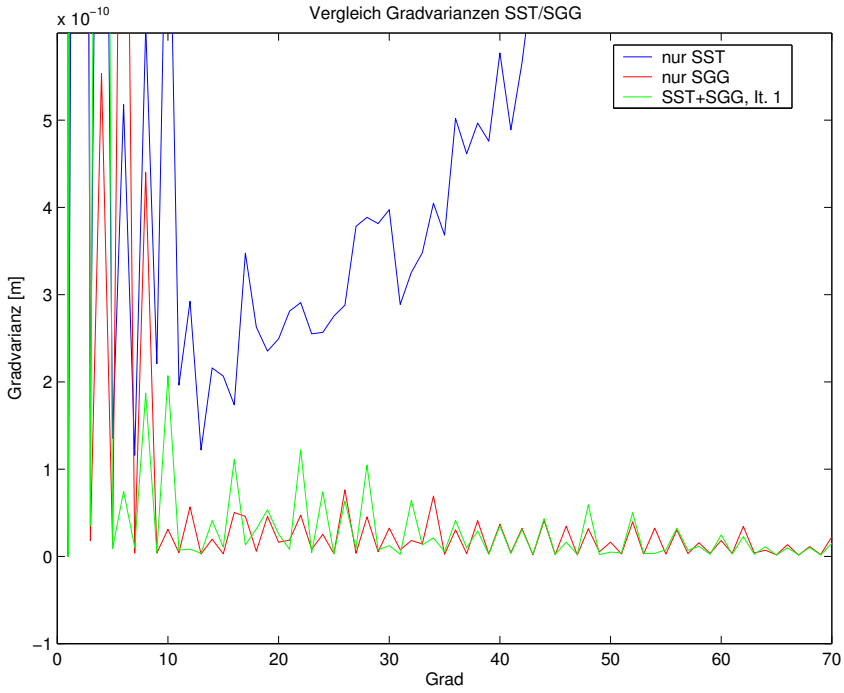


Abbildung 6.18: Vergleich der Gradvarianzen nach Iteration 1

Das Ergebnis des ersten Iterationsschritts, wieder im Vergleich zum SST-only und SGG-only-Ergebnis, zeigt ein wesentlich erfreulicheres Bild. Die Genauigkeit entspricht im wesentlichen der der SGG-Lösung, ist in einigen Koeffizienten sogar besser. Es ist dabei vor allem zu bedenken, dass es sich hierbei nur um simulierte Daten handelt, bei denen sowohl in den SST- als auch in den SGG-Beobachtungen der gleiche Einfluss des Schwerefeldes steckt. Dadurch wird hier der Anschein erweckt, dass die reine SGG-Lösung ja ausreicht. Bei realen Daten ist dies nicht der Fall, da SGG die langwelligen Anteile (also die niedrigen Entwicklungsgrade) nur schlecht erfassen kann. Hier wird die Varianzkomponentenschätzung dann unbedingt notwendig.

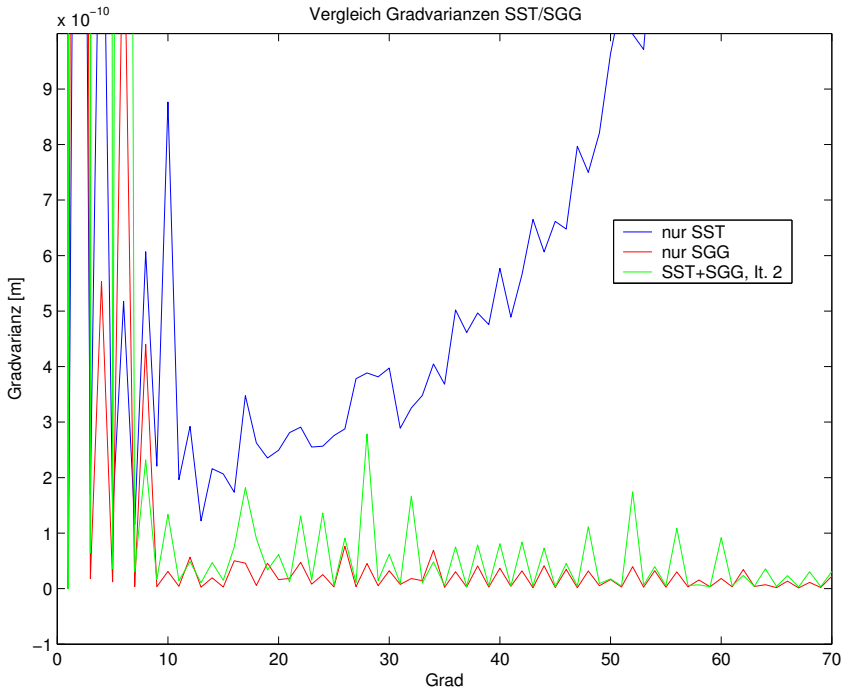


Abbildung 6.19: Vergleich der Gradvarianzen nach Iteration 2

Abbildung 6.19 zeigt, was passiert, wenn die Genauigkeit der SGG-Beobachtungen zu hoch angesetzt wird (relative Gewichtung 10^{-19}). Hier wird die Lösung wieder schlechter als im vorherigen Fall (Abbildung 6.18).

6.10 Regularisierung des Brute Force-Algorithmus

Wie im Abschnitt 4.9 erläutert wurde, kann es bei großen Gleichungssystemen zu Instabilitäten kommen, die das Ergebnis beträchtlich verschlechtern oder die Lösung des Systems sogar unmöglich machen.

Abbildung 6.20 zeigt ein Beispiel für einen solchen Fall. Es wurden Schwerefeldkoeffizienten bis Grad und Ordnung 200 bestimmt. Die Koeffizienten des zonalen Bandes weisen dabei große Fehler auf. Die Ursache liegt im sogenannten „Polar Gap“-Problem begründet. Die hier verwendeten simulierten Orbits haben die für GOCE vorgesehene Inklination von 96,6°. Dadurch weist die Bahn über den Polen große Lücken auf, die Koeffizienten des zonalen Bandes sind hierdurch schlecht bestimmt.

Zur Verminderung dieses Effektes wurde das Normalgleichungssystem bis Ordnung 16 unter Verwendung des EGM96-Feldes regularisiert. Abbildung 6.21 zeigt das Ergebnis nach der Regularisierung. Der auffällige Fehler im zonalen Band ist verschwunden. Die Genauigkeit der anderen Koeffizienten blieb von der Regularisierung unberührt.

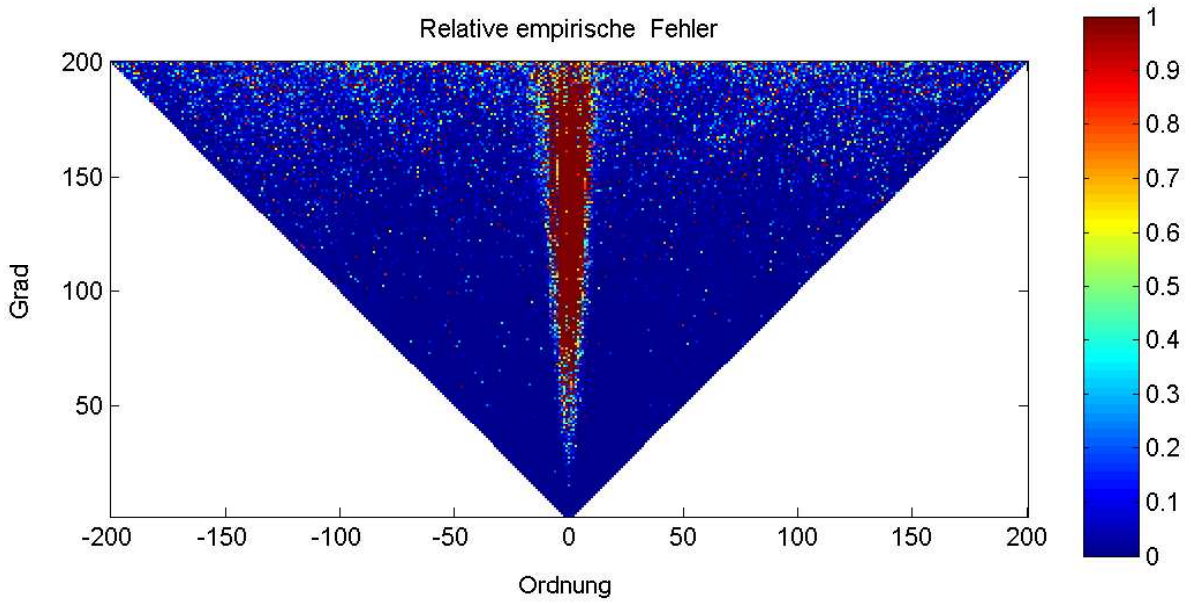


Abbildung 6.20: Relative empirische Fehler ohne Regularisierung

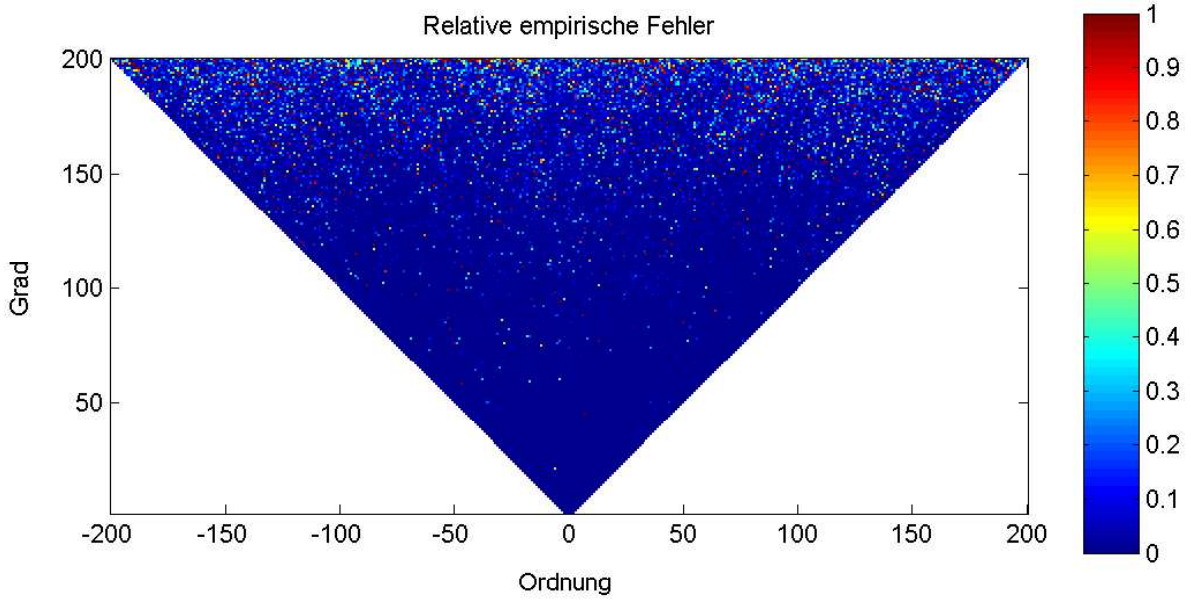


Abbildung 6.21: Relative empirische Fehler mit Regularisierung

6.11 Regularisierung des LSQR-Algorithmus

Wie in Abschnitt 5.7 erläutert wurde, besteht ein Vorteil des regularisierten LSQR-Algorithmus darin, dass die Berechnung unter Verwendung mehrerer Regularisierungsparmeter nur unwesentlich mehr Rechenzeit benötigt. Um dies zu verifizieren, wurden auf der NEC TX-7 (8 CPUs) mehrere Berechnung mit k_{max}= 50 und 500.000 Beobachtungen durchgeführt. Die Ergebnisse sind in Tabelle 6.21 aufgeführt.

Programm	Anzahl Regularisierungsparameter	wall time
PC-LSQR	-	10m38s
PC-LSQR regularisiert	1	13m13s
PC-LSQR regularisiert	5	13m30s

Tabelle 6.21: Vergleich PC-LSQR mit und ohne Regularisierung

Bei nur einem Regularisierungsparameter benötigt die regularisierte Variante etwas mehr Rechenzeit als die nicht regularisierte Version des PC-LSQR-Algorithmus. Dies liegt darin begründet, dass für das Abbruchkriterium das System mit zwei unterschiedlichen Beobachtungsvektoren gelöst werden muss (siehe 5.7.1).

Der Vergleich der Rechenzeiten für die Berechnung mit einem und mit fünf Regularisierungsparametern zeigt: die Rechenzeit steigt nur unwesentlich an.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Diplomarbeit war die Bereitstellung von Programmen zur Berechnung von Schwerefeldmodellen aus den Satellitenmissionen CHAMP, GRACE und GOCE. Die Programme sollten auf Hochleistungsrechnern implementiert und getestet werden. Es waren zwei verschiedene Ansätze zu implementieren. Zusätzlich sollten die Programme um Funktionalitäten wie Gewichtung, Varianzkomponentenschätzung und Regularisierung ergänzt werden.

Die folgenden Gebiete wurden bearbeitet:

- Die beiden Ansätze Brute Force und LSQR wurden auf verschiedenen Hochleistungsrechner-Architekturen implementiert, parallelisiert und hinsichtlich ihrer Performance optimiert. Die Parallelisierung erfolgte mit den beiden Standards OpenMP und MPI, um die Programme sowohl für Shared Memory- als auch für Distributed Memory-Systeme geeignet zu machen.
- Für den Brute-Force-Ansatz wurden Gewichtung, Varianzkomponentenschätzung und Regularisierung realisiert. Zur Gewichtung unter Berücksichtigung sämtlicher Korrelationen wurde eine Berechnung in zwei Schritten implementiert. Um den Bedarf an Rechenzeit und Speicher klein zu halten, wurde zusätzlich zur strengen Varianzkomponentenschätzung die Monte Carlo-Varianzkomponentenschätzung implementiert.
- Neben der einfachen Variante des iterativen LSQR-Algorithmus wurde eine Variante mit Vorkonditionierung (PC-LSQR) implementiert, um die Anzahl der benötigten Iterationen und damit den Rechenaufwand drastisch zu reduzieren. Zusätzlich wurde die Möglichkeit zur Regularisierung des Gleichungssystems auch beim LSQR-Algorithmus geschaffen.

Zur Überprüfung der durchgeführten Arbeiten wurden zahlreiche Berechnungen in den unterschiedlichen Konfigurationen durchgeführt. Aus den Berechnungen ließen sich folgende Ergebnisse ableiten:

- Die Parallelisierung beider Ansätze liefert beste Ergebnisse, die Programme skalieren bei ausreichender Problemgröße und im für OpenMP optimalen Bereich nahezu linear.
- Der Brute-Force-Ansatz ist hinsichtlich der Laufzeit schneller als der LSQR-Ansatz. Dies erkauft man sich jedoch durch einen höheren Speicherbedarf.
- Die Vorkonditionierung des LSQR-Algorithmus bringt drastische Laufzeitvorteile. Es ist also auf jeden Fall empfehlenswert, Vorkonditionierung anzuwenden.
- Bei großen Problemen benötigen LSQR und PC-LSQR sehr viele Iterationen und werden damit sehr langsam. Beim PC-LSQR-Algorithmus bringt die Verwendung einer größeren Beobachtungsanzahl teilweise Besserung.

- Beim Brute-Force-Ansatz und kleinen Problemen bringt der ccNUMA-Rechner NEC TX-7 die beste Performance. Bei größeren Problemen ist der Vektorrechner NEC SX-6 das deutlich schnellste System und erreicht eine Performance nahe dem theoretischen Maximum.
- Für das iterative LSQR-Verfahren ist die SX-6 ungeeignet. Die TX-7 bringt hier die beste Performance, dicht gefolgt vom Cluster Cray Opteron. Aufgrund des geringen Speicherbedarfs und der kleinen Datenmengen, die ausgetauscht werden müssen, ist der LSQR-Ansatz für Cluster sehr gut geeignet.
- Die gewählte Gewichtsmatrix bringt keine signifikante Verbesserung der Genauigkeit der Ergebnisse.
- Strenge Varianzkomponentenschätzung und MCVCE liefern äquivalente Ergebnisse. Die MCVCE bringt gewaltige Vorteile betreffend Speicherbedarf und Laufzeit mit sich, so konnte in einem Beispiel die Rechenzeit in etwa um den Faktor 30 reduziert werden.
- Es konnte gezeigt werden, dass die Kombination mehrerer Beobachtungstypen mittels Varianzkomponentenschätzung funktioniert. Hierfür ist ein iteratives Vorgehen nötig, welches jedoch schnell zum Ziel führt.
- Bei geeigneter Wahl des Regularisierungsparameters und der Regularisierungsmatrix kann die Regularisierung die Ergebnisse deutlich verbessern.
- Die Regularisierung des PC-LSQR-Algorithmus und die Berechnung mit mehreren Regularisierungsparametern bedeutet lediglich einen geringen zusätzlichen Rechenaufwand.

Insgesamt sind die Ergebnisse positiv zu sehen. Es stehen jetzt mehrere Programme mit großer Funktionalität zur Verfügung, die die Lösung größerer Probleme in geringer Zeit ermöglichen. Es ist die Möglichkeit gegeben, Daten aller drei Satellitenmissionen (CHAMP, GRACE und GOCE) auszuwerten.

Folgende Bereiche sollten noch weiter bearbeitet werden:

- Die Ergebnisse der Gewichtung sind nicht zufriedenstellend. Hier ist zu untersuchen, ob im gewählten Ansatz oder der gewählten Gewichtsmatrix Fehler liegen.
- Der PC-LSQR-Algorithmus enttäuscht hinsichtlich der erzielten Rechenzeiten und verursacht Schwierigkeiten besonders bei großen Problemen. Hier ist zu untersuchen, ob sich diese Schwierigkeiten überwinden lassen. Dies ist besonders deshalb wünschenswert, da dieser Algorithmus sehr geeignet für Cluster ist.
- Aufgrund von Beschränkungen in der verwendeten Bibliothek für lineare Algebra war es nicht möglich, Berechnungen mit einem Entwicklungsgrad von k_{kmax} > 200 durchzuführen. Bei Verfügbarkeit einer neuen Bibliothek sollte dies getestet und hinsichtlich der benötigten Laufzeit untersucht werden.
- Bei der gewählten Implementierung des Brute-Force-Ansatzes benötigt jeder Knoten eines Clusters eine eigene Kopie der Normalgleichungsmatrix. Mit existierenden Bibliotheken ist es möglich, diese über alle Knoten zu verteilen (z.B. PBLAS, ScaLAPACK). Eine Erweiterung der Programme um diese Funktionalität und die Untersuchung der Leistungsfähigkeit derselben war in dieser Diplomarbeit nicht mehr möglich, wäre aufgrund der weiten Verbreitung von Cluster-Systemen aber von großem Interesse.
- Im Jahr 2005 wird am HLRS ein sehr leistungsfähiges Vektorsystem in Dienst gestellt werden. Die bestehenden Brute-Force-Programme sollten dann daran angepasst werden, da nur dieses System die zu erwartenden Datenmengen der GOCE-Mission in einem vertretbaren zeitlichen Rahmen verarbeiten kann.

Literaturverzeichnis

[Ditmar, Klees 2002] Ditmar, P., Klees, R. (2002): A method to compute the Earth's gravity field from SGG/SST data to be acquired by the GOCE satellite. Delft University Press, Delft.

[Dowd 1998] Dowd, K., Severance, C. (1998): High Performance Computing, 2. Auflage. O'Reilly and Associates, Sebastopol. ISBN 1-56592-312-X

[ESA 2000] GO-RS-ESA-SY-0001(2): GOCE Mission Requirements Document, ESTEC, Noordwijk, Niederlande, April 2000.

[Flynn 1972] Flynn, M.J. (1972): Some computer organisations and their effectiveness. IEEE Trans Computers 21, 948-960.

[Golub, von Matt 1997] Golub, G., von Matt, U. (1997): Generalized cross-validation for large scale problems, revised version. Journal of Computational and Graphical Statistics, 6, 1 (1997).

[Götzelmann 2003] Götzelmann, M. (2003): Simulation von Satellitenbahnen unter Berücksichtigung von direkten Gezeiteneffekten. Studienarbeit, Geodätisches Institut, Universität Stuttgart.

[Kilmer, O'Leary 2001] Kilmer, M., O'Leary, D. (2001): Choosing regularization parameters in iterative methods for ill-posed problems. SIAM J. Matrix Anal. Appl. 22, No. 4 (2001), 1204-1221.

[Kusche 2003/1] Kusche, J. (2003): Noise variance estimation and optimal weight determination for GOCE gravity recovery. Advances in Geoscience (2003) 1, 81-85.

[Kusche 2003/2] Kusche, J. (2003): A Monte Carlo technique for weight estimation in satellite geodesy. Journal of Geodesy 76 (2003), 641-652.

[Kusche, Klees 2002] Kusche, J., Klees, R. (2002): Regularization of gravity field estimation from satellite gravity gradients. Journal of Geodesy 76 (2002), 359-368.

[OpenMP 2002] OpenMP C and C++ Application Program Interface, Version 2.0, 2002. `http://www.openmp.org`.

[Openshaw 1999] Openshaw, S., Turton, I. (1999): High Performance Computing and the Art of Parallel Programming, An introduction for geographers, social scientists and engineers. Routledge, London. ISBN 0-415-15692-0

[Paige, Saunders 1982/1] Paige, C., Saunders, M. (1982): LSQR: An algorithm for sparse linear equations and sparse least squares. ACM Transactions on Mathematical Software 8, No. 1 (1982), 43-71.

[Paige, Saunders 1982/2] Paige, C., Saunders, M. (1982): LSQR: Sparse linear equations and least squares problems. ACM Transactions on Mathematical Software 8, No. 2 (1982), 195-209.

[Reubelt, Austen 2003] Reubelt T., Austen G., Grafarend E. (2003): Harmonic Analysis of the Earth's Gravitational Field by Means of Semi-Continuous Ephemerides of a Low Earth Orbiting GPS-Tracked Satellite. Case Study: CHAMP. Journal of Geodesy 77 (2003), 257-278.

[Riepe 2004] Riepe, M. (2004): Sweetheart, what watch? Zeitmessung unter Unix/Linux und Windows. iX 3/2004, S. 102-105. Heise Zeitschriften Verlag, Hannover.

[Schönauer 2000] Schönauer, W. (2000): Scientific Supercomputing, Architecture and Use of Shared and Distributed Memory Parallel Computers. Eigenverlag. ISBN 3-00-005484-7

[Snir 1996] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996): MPI: The Complete Reference. The MIT Press, Cambridge. ISBN 0-262-69184-1