Ein Softwarekonzept zur hierarchischen Parallelisierung von stochastischen und deterministischen Inversionsproblemen auf modernen ccNUMA-Plattformen unter Nutzung automatischer Programmtransformation

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelgt von

Diplom-Informatiker
Andreas Wolf

aus Berlin-Lichtenberg

Berichter: Universitätsprofessor Christian H. Bischof, Ph. D.,
Universitätsprofessor Dr. rer. nat. Christoph Clauser

Tag der mündlichen Prüfung: 10. Juni 2011

Kurzfassung

Das hier vorgestellte Softwarekonzept befasst sich hauptsächlich mit der Unterstützung eines fortschreitenden Software-Entwicklungsprozesses. Dabei wurde der Bedarf nach Hochleistungs-Simulationssoftware genauso berücksichtigt wie eine überwiegende Nutzbarkeit der Software während des Entwicklungsprozesses. Die vorliegende Arbeit beschäftigt sich mit allen sich daraus ergebenden Anforderungen und deren Lösung durch verschiedene Software-Techniken und Strategien. Im Detail handelt es sich um eine schnelle Erweiterbarkeit des Simulationsprogramms, einer Unterstützung von stochastischen und deterministischen Verfahren zur Lösung von Inversionsproblemen und Unterstützung moderner ccNUMA-Rechnerarchitekturen. Besonders die deterministischen Verfahren sind auf die Berechnung von Ableitungen angewiesen. Dafür wird die Technik des automatischen Differenzierens eingesetzt, mit der effizienter Code zur Berechnung von Ableitungen auf Basis einer automatisierten Programm-Transformation erzeugt wird.

In dieser Arbeit wird einerseits aufgrund der komplexen Zusammenhänge eine Software-Technik angewendet und beschrieben, um die Programmcode-Transformationen wesentlich zu erleichtern. Andererseits werden verschiedene hierarchische Parallelisierungs-Strategien analysiert und verglichen, um eine effiziente Lösung für das Hochleistungsrechnen zu erzielen. Die beschriebenen Software-Techniken wurden in Verbindung mit einer mehrstufigen OpenMP-Parallelisierung an einem hydro-geothermalen Simulationsprogramm beispielhaft umgesetzt. Numerische Experimente belegen, dass die bevorzugte Parallelisierungs-Strategie effizient ist und die beispielhafte Anwendung der beschriebenen Software-Technik zeigt, dass sie praktikabel und robust ist.

Abstract

This thesis introduces a software concept to support proceeding software development processes. The concept considers not only the demand for high performance simulation software but also the maximal possible usability of the software during the development process. It addresses the resulting requirements and proposes different techniques and strategies to fulfill them. More precisely, the requirements are the fast extensibility of the simulation software, the support for stochastic as well as deterministic methods for solving inverse problems, and the support for ccNUMA capabilities of modern computer architectures. In particular any deterministic method makes necessary the computation of derivatives. These derivatives are computed by techniques of automatic differentiation. These techniques are based on an automated program transformation, generating efficient code for the computation of derivatives.

The new contributions of this thesis are as follows. On the one hand, the proposed software techniques alleviate the handling of the complex dependencies between the various requirements and considerably simplify the involved program transformations. On the other hand, different hierarchical parallelization strategies are introduced providing an efficient solution for inverse problems on high-performance computing platforms. The novel software techniques are illustrated in the context of a real-world hydro-geothermal simulation code involving a multilevel OpenMP parallelization. Numerical experiments indicate that the proposed parallelization strategy is efficient and that the new software techniques are feasible and robust.

Danksagung

Mit der Fertigstellung meiner Arbeit möchte ich mich bei all denjenigen bedanken, die mich während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik 12 der RWTH Aachen University über die ganzen Jahre begleitet und so viel unterstützt haben. Ganz besonders möchte ich mich hier als erstes bei apl. Prof. Dr.-Ing. H. Martin Bücker bedanken, der durch inspirierende wie auch kritische Fachdiskussionen meinen Werdegang bereichert hat. Im gleichen Maße möchte ich mich hiermit auch bei Dr. Volker Rath bedanken, denn beide haben mich in Hinblick auf meine fachliche, berufliche und persönliche Weiterentwicklung gefördert und durch ihre hilfsbereite Art meine wissenschaftlichen Forschungsarbeiten erleichtert. Die interdisziplinäre Arbeit wurde nicht zuletzt aufgrund ihrer beider Mitwirkung und auch Dank der ausgesprochen angenehmen und kollegialen Atmosphäre zu einer sehr glücklichen Zeit für mich.

Mein großer Dank gilt natürlich auch meinem Doktorvater Prof. Christian H. Bischof, Ph.D., für die vielen wertvollen Vorschläge und anregenden Diskussionen während meiner Promotionszeit. Durch ihn und den Dozenten seines Lehrstuhls wurde schon während meines Diplomstudiums mein Interesse für die wissenschaftliche Arbeit mit Hochleistungsrechnern geweckt und unterstützt. Darüber hinaus möchte ich mich bei Prof. Dr. rer. nat. Christoph Clauser für die bereitwillige Übernahme des Zweitgutachtens und die hilfreichen Anmerkungen zu meiner Dissertation gerne bedanken.

Meinen lieben Kollegen und lieben Kolleginnen möchte ich an dieser Stelle auch herzlich danken, dass diese Jahre meiner Assistententätigkeit in einem so kreativen und hilfsbereiten Umfeld stattfanden. Besonders möchte ich hier vom Lehrstuhl Oliver Fortmeier, Dr. Kathrin Fuchss Portela, Michael Lülfesmann, Dr. Monika Petera und Johannes Willkomm und den früheren Kollegen Dr. Emil Slusanschi, Dr. Arno Rasch und Dr. Andre Vehreschild danken. Hierbei möchte ich auch den lieben Kollegen Dieter an Mey, Christian Terboven, Dirk Schmidl und Sandra Wienke aus dem RZ, sowie Dr. Darius Mottaghy, Dr. Gabriele Marquart, PD Dr. Michael Kühn, Henrick Büsing und Christian Vogt aus dem Bereich der Geophysik herzlich danken. Nicht zuletzt bin ich auch Livi, meiner lieben Frau, für das große Verständnis und so manchen Verzicht während meiner Promotionszeit zu tiefsten Dank verpflichtet. Ohne ihre moralische Unterstützung wäre diese Arbeit für mich sicher sehr viel schwerer geworden.

Aachen, der 11. August 2011.

Inhaltsverzeichnis

1 Einleitung

In vielen Software-Projekten aus der Industrie und in den Wissenschaften gilt es oft, innerhalb definierter Zeiträume mit einem überschaubaren Aufwand die Projekt-Entwicklung voranzutreiben. Bei einer herausragenden Klasse von solchen Projekten handelt es sich um Anwendungen zur numerischen Simulation von physikalischen, chemischen oder biologischen Prozessen. Darauf basierend bestimmt man in der Regel unbekannte Parameter mit Hilfe von rechenintensiven stochastischen und deterministischen Methoden. Wesentlich für die Nutzung ist dementsprechend eine hohe Rechenleistung für die Anwendung und damit eine kurze Berechnungszeit auf den zur Verfügung stehenden Rechnersystemen. Unabhängig davon handelt es sich üblicherweise auch um einen immer fortschreitenden Prozess der Weiterentwicklung dieser Simulationssoftware. Diese nahezu kontinuierlich stattfindenden Erweiterungen und Verbesserungen dürfen dabei nicht die Zuverlässigkeit oder die generelle Nutzbarkeit der Anwendung beeinträchtigen.

Besonders im wissenschaftlichen Umfeld kommt es deshalb häufig vor, dass man sich aufgrund beschränkter Personalressourcen innerhalb einer Forschergruppe nicht unbedingt auf alle diese sehr facettenreichen Aspekte konzentrieren kann. Damit erscheint es notwendig, die einzelnen Aspekte besser zu trennen und damit unabhängig von einander bearbeiten zu können. Ein Weg ist es, auf programmtechnischer Ebene spezielle Softwaretechniken und Datenstrukturen einzusetzen. Diese beiden Softwareaspekte werden in der vorliegenden Arbeit in verschiedene Teilbereiche aufgeteilt und geeignete Lösungs-Konzepte vorgestellt. Das Gesamtlösungs-Konzept kann dann im industriellen Softwareentwicklungsprozess eingesetzt werden. Insbesondere wird es hier für konkrete Anforderungen bei der Entwicklung und Anwendung eines geologischen Simulationswerkzeugs beispielhaft durchgeführt.

Die nun vorliegende Arbeit trägt den sich so ergebenden Umständen im Zusammenhang mit einem hydro-geothermalen Simulationscode Rechnung. Bei dessen Entwicklung ergaben sich eine ganze Reihe von Besonderheiten, die in ihrer Gesamtheit eine Neuheit im Bereich der geophysikalischen Simulationswerkzeuge darstellen.

1. Die Simulationssoftware muss eine umfangreiche und schnell erweiterbare Berechnung verschiedenster geophysikalischer und gekoppelter Zustandsvariablen im porösen Medium ermöglichen.

2. Gleichzeitig mit der Anpassung und Weiterentwicklung des Simulationscodes für die Vorwärtsberechnung muss der dazugehörige Programmcode zur Behandlung von Inversionsproblemen synchron weiterentwickelt werden können.

3. Es sollen sowohl stochastische [1, 2, 3] als auch deterministische [4, 5, 6] Verfahren eingesetzt werden.

4. Einen Schwerpunkt bilden dabei die deterministischen Verfahren, wobei hier ein besonderes Gewicht auf die ableitungsbasierten Methoden und die Verwendung exakter Ableitungen gelegt wird.

5. Darüber hinaus sollen Ableitungen bezüglich möglichst vieler verschiedener Gesteinsparameter und auch zeitabhängiger Randwerte unterstützt werden, um so neuartige Untersuchungen im geowissenschaftlichen Umfeld zu ermöglichen.

Die Ableitungen zu Unterpunkt 4 sollen durch den Einsatz von Techniken des „automatischen Differenzierens“ (AD) berechnet werden. Die durch ein AD-Werkzeug [7, 8, 9] automatisierte Programmcode-Transformation erlaubt, neben der ursprünglichen Funktion auch die Ableitung nach einzelnen Parametern (einzelne Richtungsableitungen) exakt zu berechnen.

Das hier vorgestellte Gesamtlösungs-Konzept soll gleichzeitig auch hocheffiziente Lösungen in Hinsicht auf die Berechnungszeiten gewährleisten. Denn alle hier erwähnten Verfahren können von einer Parallelisierung profitieren oder sind sogar nur durch deren Einsatz sinnvoll anwendbar. Eine mehrstufige Parallelisierung umfasst sowohl die Lösung des Vorwärtsproblems (linearer [10, 11] und nicht-linearer Löser [12]) als auch die ableitungsbasierten und stochastischen Verfahren. Grundsätzlich muss dabei die Parallelisierung und die Software auf die Rechner-Architekturen abgestimmt sein.

Deshalb stellen die Trends bei modernen Rechner-Architekturen einen weiteren wichtigen Aspekt dar. Hier kommen standardmäßig immer leistungsfähigere Prozessoren mit einer Vielzahl integrierter Rechenkerne zum Einsatz, die oft in einem SMP-System (engl. „symmetric multi processor“) zusammen gefasst werden.

Aus der Sicht der Softwareentwicklung benötigt man einen gewissen Zusatzaufwand, um die Programme für die Nutzung der vielen Rechenkerne innerhalb eines SMP-Rechenknotens vorzubereiten (z.B. Parallelisierung mit OpenMP [13, 14]). Darüber hinaus muss für eine übergreifende Nutzung aller Rechenkerne des gesamten verteilten Rechner-Clusters eine im Vergleich dazu wesentlich aufwändigere Software-Parallelisierung vorgenommen werden (z.B. Parallelisierung mit MPI [15, 16]). Aus Entwicklersicht und im Sinne der Erweiterungsproduktivität ist die weniger aufwändige und überschaubarere Spracherweiterung zur Parallelisierung wünschenswert (z.B. OpenMP). Allerdings ist man dafür oft auf die entsprechend aufwändigere Rechner-Architektur z.B. in Form eines einzelnen großen SMP-Systems beschränkt.

Wegen der hohen Kosten für große SMP-Einzelsysteme versucht man deshalb, mehrere kleinere Rechenknoten in neuen Ansätzen „virtuell“ in ein großes SMP-System umzuwandeln, um die Vorteile der preiswerten Rechner-Cluster zu erhalten. Ein Vertreter dieser neuen Klasse von „vSMP“-Systemen stammt von der Firma ScaleMP™ [17], bei denen man einen solchen Verbund mit einem schnellen Netzwerk und einer Virtualisierungsschicht für den Programmierer als ein großes SMP-Einzelsystem darstellt. Damit liegen dann alle Vorteile durch weniger Kosten für das Rechnersystem und einem niedrigeren Parallelisierungsaufwand (für OpenMP) beisammen.

Es ist Ziel dieser Arbeit, dass die mehrstufige Parallelisierung sowohl geeignet sein muss für die derzeitig verfügbaren vSMP-Systeme als auch für die neuesten reinen SMP-Plattformen. Wichtig ist hier eine möglichst hohe „Erweiterungsproduktivität“, im Sinne schneller Erweiterungen, da von einer ständigen und überdies wichtigen Weiterentwicklung (zum Erreichen von 1. und 5. wie auch bei den Rechner-Architekturen) auszugehen ist.

1.1 Gesamtlösungs-Konzept

Ausgegangen wird von einem bereits parallelisierten Programmcode für eine einzelne Vorwärtsberechnung. Wobei die „ursprüngliche“ Parallelisierung nur die verteilte Berechnung einer Vorwärtssimulation adressiert und nicht die gleichzeitige Berechnung verschiedener Simulationen. Für die stochastischen Verfahren ist im Allgemeinen die Berechnung einer großen Zahl von Vorwärtssimulationen notwendig. Damit liegt es nahe, eine zusätzliche von der Vorwärtsberechnung abweichende Parallelisierung über die Anzahl der Simulationen einzuführen. Für die ableitungsbasierten Methoden zur Lösung von Inversionsproblemen kann man analoge Überlegungen anstellen, nämlich eine von der ursprünglichen Parallelisierung unabhängige über der Menge aller benötigten Richtungsableitungen.

Damit wird klar, dass zusätzlich und unabhängig zur ursprünglichen Parallelisierung hierarchisch auch über die Anzahl der Richtungsableitungen oder Vorwärtssimulationen noch eine „äußere“ Parallelisierungsebene aufgespannt werden kann. In Kombination kann somit die allgemeine Parallelisierbarkeit erhöht werden. Das ist immer dann von Vorteil, wenn eine einzelne Parallelisierungsebene in der maximalen Anzahl der verwendbaren Rechenkerne beschränkt ist. Für die ursprüngliche Parallelisierung könnte die Limitierung in bestimmten Modelleigenschaften, wie einer zu groben Diskretisierung mit zu wenig unabhängigen Berechnungsblöcken, liegen. Dagegen ist die äußere Parallelisierung natur-

gemäß immer durch die Anzahl der benötigten Richtungsableitungen oder Vorwärtssimulationen beschränkt.

Die gleichzeitig zu erfüllenden Eigenschaften wie diese mehrstufige Parallelisierung, eine hohe Erweiterungsproduktivität, hohe Rechenleistung auf modernen Rechner-Architekturen und die Nutzung exakter Ableitungen (mit Hilfe von Programmcode-Transformation) bilden die wichtigen „allgemeinen Vergleichskriterien“ und unterscheiden die in dieser Arbeit vorgestellten Konzeptlösungen von denen herkömmlicher Konzeptfindungs-Prozesse. Sie sind damit ein Alleinstellungsmerkmal der vorliegenden Arbeit. Konkret umfasst das Gesamtlösungs-Konzept die folgenden Teilbereiche:

A) Da alle Zielsetzungen und allgemeinen Vergleichskriterien untereinander sehr komplexe Abhängigkeiten bedingen, ist es auch ein Anspruch dieser Forschungsarbeit, die verschiedenen Facetten im Detail zu untersuchen und darzustellen.

B) Ein großer Teilbereich zur Lösung der Parallelisierungs-Problematik im Sinne der Erweiterungsproduktivität ist eine Softwaretechnik für die Umgehung einiger problembehafteter Abhängigkeiten und einem weitestgehend automatisierten Übergang zwischen Änderungen am Simulationscode (Weiterentwicklung), dem AD-Werkzeug und der Parallelisierung. Im Wesentlichen handelt es sich hierbei um eine systematische Ausblendung der Parallelisierungsanweisungen gegenüber dem AD-Werkzeug. Wichtigste Maßnahmen sind einmalige strukturelle Umstellungen und zusätzliche leicht automatisierbare Ersetzungen.

C) An dieser Stelle bleibt die Frage nach der zu verwendenden Parallelisierungsstrategie für die Ableitungsberechnungen offen. Da es hierfür eine große Anzahl von Variationen gibt, befasst sich der zweite große Teilbereich mit dieser Fragestellung. Die Ausführung dieser ganzen Reihe von systematischen Verbesserungen ist wichtig, weil die jeweiligen Teilerkenntnisse im Rahmen der vorliegenden Forschungsarbeit zu den letztendlichen Verbesserungen und Bewertungen führen. Dabei unterscheiden sich die Strategien nicht nur in der Art und Weise, wie und womit sie die Rechenarbeit aufteilen, sondern auch in der Variation der zugrunde liegenden Datenstrukturen.

Für die stochastischen Methoden sind die Möglichkeiten zur Parallelisierung über der Anzahl der Vorwärtssimulationen eingeschränkter, aber für den Vergleich mit den Richtungsableitungen notwendig.

D) Den letzten Teil stellt ein Vergleich bezüglich der Eignung für moderne SMP und vSMP-Systeme dar. Ziel ist es hier, die Anforderungen der modernen Rechner-Architekturen für eine hohe Rechenleistung mit einzubeziehen. Deswegen wird durch Vergleich der zuvor definierten „allgemeinen Vergleichskriterien“ die finale Parallelisierungsstrategie ausgewiesen und durch Vergleichsmessungen untermauert. Ein solch umfassender Vergleich der verschiedenen Parallelisierungsstrategien liegt in dem hier betrachteten Anwendungsbereich bisher nicht vor. Neu ist damit auch das Ergebnis der endgültigen Strategie in Hinblick auf alle hier definierten Anforderungen und Zielsetzungen.

Für die Ableitungsparallelisierung wird außerdem analysiert, ob bei deren Umsetzung Zusatzanforderungen an das AD-Werkzeug oder andere Werkzeuge anfallen würden. Im Sinne der Erweiterungsproduktivität wäre dies zu vermeiden. Außerdem ist für die Auswahl einer finalen Strategie auch wichtig, ob es Gemeinsamkeiten zwischen der Parallelisierung für die Richtungsableitungen und den stochastischen Vorwärtssimulationen gibt, denn dies kann auch den Parallelisierungsaufwand klein halten. Gerade diese beiden Kriterien können als wichtige Neuheiten für die Auswahl in solchen Untersuchungen ansehen werden.

Nicht Ziel dieser Arbeit ist die Erstellung eines verallgemeinerten Entwurfsmusters (engl. „design patterns“ [18, 19]). Diese Einschränkung ergibt sich einerseits daraus, dass

sowohl die Parallelisierung als auch AD nicht durch eine Objekt-orientierte Beschreibung erfassbar ist. Andererseits wird in der vorliegenden Arbeit gerade auf die sprachtechnischen Eigenheiten und deren Umsetzung mit OpenMP großer Wert gelegt, sodass diese Spezialisierung kaum für ein global verallgemeinertes Konzept dienlich sein kann. Da andere Spracherweiterungen zur Parallelisierung wie z.B. MPI sich erheblich von OpenMP unterscheiden, würde ein entsprechender Konzeptentwurf, selbst wenn unter den selben Gesichtspunkten betrachtet, eine vollkommen andere Konzeptlösung erfordern. Deswegen wird in dieser Arbeit auch auf eine Konzept-Erweiterung in diese Richtung verzichtet.

1.2 Gliederung der vorliegenden Arbeit

Die hier vorliegende Arbeit gliedert sich in sechs Bereiche. Das direkt anschließende Grundlagenkapitel 2 gibt als erstes eine Einführung in die hydro-geothermale Simulation und die darauf aufbauenden inversen Problemstellungen. Danach folgen die Projektanforderungen an die Simulationssoftware mit den resultierenden Konzeptanforderungen, die aus den geowissenschaftlichen Fragestellungen entstehen. Weiterhin wird ein kurzer Vergleich zu einem Stellvertreter von Simulations-Projekten mit vergleichbaren geowissenschaftlichen Zielsetzungen gezogen. Abschließend werden die Grundlagen der Rechner-Architekturen und die dafür nötigen Spracherweiterungen zur Parallelisierung besprochen.

Als nächstes folgen in Kapitel 3 die zum Gesamtlösungs-Konzept gehörenden Vorbetrachtungen und Grundzusammenhänge für die projektgetriebene Softwareentwicklung (Teilbereich A). Begonnen wird hier als erstes mit einer Abgrenzung bezüglich anderer Methoden und Formen der allgemeinen schnellen Softwareentwicklung. Danach werden die einzelnen elementar definierten Grundanforderungen und deren komplexe Abhängigkeiten erläutert. Diese gliedern sich in vier Unterkapitel bezüglich der Simulationssoftware, der AD-Programmcode-Transformation, der Parallelisierung und der Erweiterungsproduktivität.

In dem Kapitel 4 wird die Softwaretechnik „OpenMP-hiding“ vorgestellt (Teilbereich B). Dabei wird nacheinander erläutert, wie der parallelisierungs-spezifische Automatismus ausgenutzt, die originale Parallelisierung erhalten und später auf die Ableitungsanweisungen erweitert wird. Abgeschlossen wird das Teilkonzept mit wichtigen Überlegungen zur Aufwandsminimierung und Umgehung möglicher zusätzlicher Probleme.

Die anschließende Besprechung der Parallelisierungsstrategien (Teilbereich C und D) erstreckt sich über die beiden Kapitel 5 und 6. In Kapitel 5 werden die grundlegenden Parallelisierungsstrategien diskutiert. Es gliedert sich grob selbst in die drei Parallelisierungsaspekte: Vorwärtssimulation, AD-generierter Transformationscode und stochastische Versuche auf. Die letzten beiden werden jeweils mit einer Bewertung und ersten synthetischen Vergleichstests abgeschlossen. Die wesentlichen Parallelisierungsstrategien befinden sich dabei in dem Unterkapitel über den AD-generierten Transformationscode.

Entsprechend widmet sich das zugehörige Kapitel 6 ausführlich dem Zusammenspiel zwischen den mehrstufigen Parallelisierungen, den modernen Rechner-Architekturen und realen Anwendungsszenarien. Hier werden zuerst die speziellen Anpassungen bezüglich der SMP- und vSMP-Systeme besprochen und im Anschluss detailliert verschiedene Vergleichstests diskutiert. Zuletzt wird ein Unterkapitel mit einem Ausblick auf die Übertragbarkeit in andere Projekte gegeben und resultierende zukünftige Forschungsarbeiten umrissen.

Abgeschlossen wird die hier vorliegende Arbeit mit den Schlussbemerkungen in Kapitel 7. Danach folgen das Literaturverzeichnis, verschiedene Beispiele und Zusatzinformation im Anhang.

2 Grundlagen - Simulation, Software und Rechner-Architekturen

2.1 Die hydro-geothermale Anwendungssoftware - Vorwärtssimulation

Bei dem zu Grunde liegenden Simulationscode handelt es sich um eine Neuentwicklung (Suite-Projekt) basierend auf der Vorgängersoftware SHEMAT [20] mit einer neuen Zielausrichtung für inverse Problemstellungen, Unsicherheitsanalyse und der Eignung für das Hochleistungsrechnen.

Entsprechend haben sowohl das ursprüngliche SHEMAT als auch die Neuentwicklung einen vergleichbaren Grundstock von Algorithmen und Formeln zur Vorwärtsberechnung eines Systems von gekoppelten zeitabhängigen partiellen Differentialgleichungen (kurz PDGL [21, 22, 23]). Die in der Suite enthaltenen einzelnen PDGL stehen für die Berechnung des Grundwasserflusses, des Wärmetransports, des Transports von gelösten Substanzen bei hohen Temperaturen oder auch der Verteilung des elektrischen Potentials. Allerdings sind für die Suite die chemischen Reaktionen und die damit verbundenen Rückkopplungen noch nicht implementiert. Davon unabhängig werden die chemischen Transportgleichungen ganz analog zu dem Wärmetransport behandelt und aufgestellt.

Beispielhaft wird der Kopplungsprozess und die prinzipielle Methodik für die Vorwärtssimulation an den partiellen Differentialgleichungen für den Grundwasserfluss (1) und dem Wärmetransport (2) erläutert.

$$\nabla \cdot \left[\frac{\rho_f \, g}{\mu_f} \mathbf{k} \cdot \nabla h \right] + Q = S \, \frac{\partial h}{\partial t} \tag{1}$$

Über die Gleichung (1) wird die zu einem bestimmten Druck entsprechende Wasserstandshöhe h berechnet. Sie wird in der Einheit $[m]$ angegeben und auch als hydraulisches Potential bezeichnet. Dazu werden auf der linken Seite die Quellterme Q in $[m^3\, s^{-1}]$ und die hydraulische Permeabilität $\mathbf{k}$ $[m^2]$ mit einbezogen. Außerdem benötigt man hier noch die Schwerebeschleunigung g $[m\, s^{-2}]$, Dichte ρ_f $[kg\, m^{-3}]$ und die dynamische Viskosität μ_f $[Pa\, s]$ des Porenwassers. Auf der rechten Seite wird die Ableitung nach der Zeit t $[s]$ mit dem spezifischen Speicherkoeffizienten S $[m^{-1}]$ multipliziert.

Die Temperatur T $[°C]$ wird in der Wärmetransportgleichung (2) auf der linken Seite über einen konduktiven Term und die Wärmeproduktionsrate A $[W\, m^{-3}]$ (ähnlich zu dem Quellterm Q des Grundwasserflusses) definiert. Auf der rechten Seite steht ein advektiver Wärmetransportterm, wobei $(\rho c)_e$ $[J\, m^{-3}\, K^{-1}]$ für die effektive volumetrische Wärmekapazität des gesättigten Mediums steht.

$$\nabla \cdot [\lambda_e \nabla T] - (\rho c)_f \, \mathbf{v} \cdot \nabla T + A = (\rho c)_e \frac{\partial T}{\partial t} \tag{2}$$

In den Termen der linken Seite fällt auf, dass neben der effektiven Wärmeleitfähigkeit λ_e $[W\, m^{-1}\, K^{-1}]$ und der thermischen Kapazität $(\rho c)_f$ $[J\, kg^{-1}\, K^{-1}]$ der Flüssigkeit, die Temperatur auch von der Darcy-Geschwindigkeit

$$\mathbf{v} = -\frac{\rho_f \, g}{\mu_f} \mathbf{k} \cdot \nabla h \tag{3}$$

in $[m\, s^{-1}]$ abhängt.

Die in den Koeffizienten der Differentialgleichungen vorkommenden Gesteins- und Flüssigkeitseigenschaften (z.B. ρ_f und μ_f) hängen dabei signifikant von der Temperatur und dem Porenwasserdruck ab. Der Porenwasserdruck basiert dabei auf der Verteilung des hydraulischen Potentials und der Tiefeninformation, siehe dazu die Definition von Marsily [24]. Auf der anderen Seite wird in der Gleichung (3) auch gezeigt, dass die Darcy-Geschwindigkeit $\mathbf{v}$ auch direkt vom hydraulischen Potential h abhängig ist. Diese direkten und indirekten Abhängigkeiten (für T über $\mathbf{v}$ von h) sind nur ein Beispiel für die vielen Kopplungen zwischen den partiellen Differentialgleichungen.

Für alle weiteren Betrachtungen sind die bis zu dieser Stelle gemachten groben Ausführungen ausreichend. Weiterführende Details zu den hier schon aufgeführten und anderen physikalischen Prozessen (wie Transport), deren PDGL, gegenseitigen Abhängigkeiten (physikalische Kopplungen) und der angewendeten Methodik kann man in [20] nachlesen.

Darüber hinaus werden hier alle PDGL mit einem gebräuchlichen „zell-zentrierten Finite-Differenzen“-Verfahren diskretisiert. Unter Einbeziehung der Randwerte führt das jeweils zu diskreten Versionen von gekoppelten linearen Gleichungssystemen. Als Beispiel kann man für das hydraulische Potential h aus (1) das in (4) angedeutete lineare Ersatzgleichungssystem erstellen.

$$M_{(T,h)} \cdot h = b_{(T,h)} \tag{4}$$

Hierbei hängen sowohl die Systemmatrix $M_{(T,h)}$ als auch die rechte Seite $b_{(T,h)}$ jeweils von der alten Lösung der Temperatur T und des hydraulischen Potentials h des letzten Zeitschrittes ab. Das vollständige nichtlinear gekoppelte System (von linearen Ersatzgleichungssystemen) kann dann durch einfache alternierende Fixpunktiterationen [12] (in der Literatur auch als „Picard-Iteration“ bezeichnet) gelöst werden.

Der gesamte Berechnungsprozess der Vorwärtssimulation ist schematisch in Abbildung 1 dargestellt. Man sieht hier als erstes die äußerste Iteration über die einzelnen zu

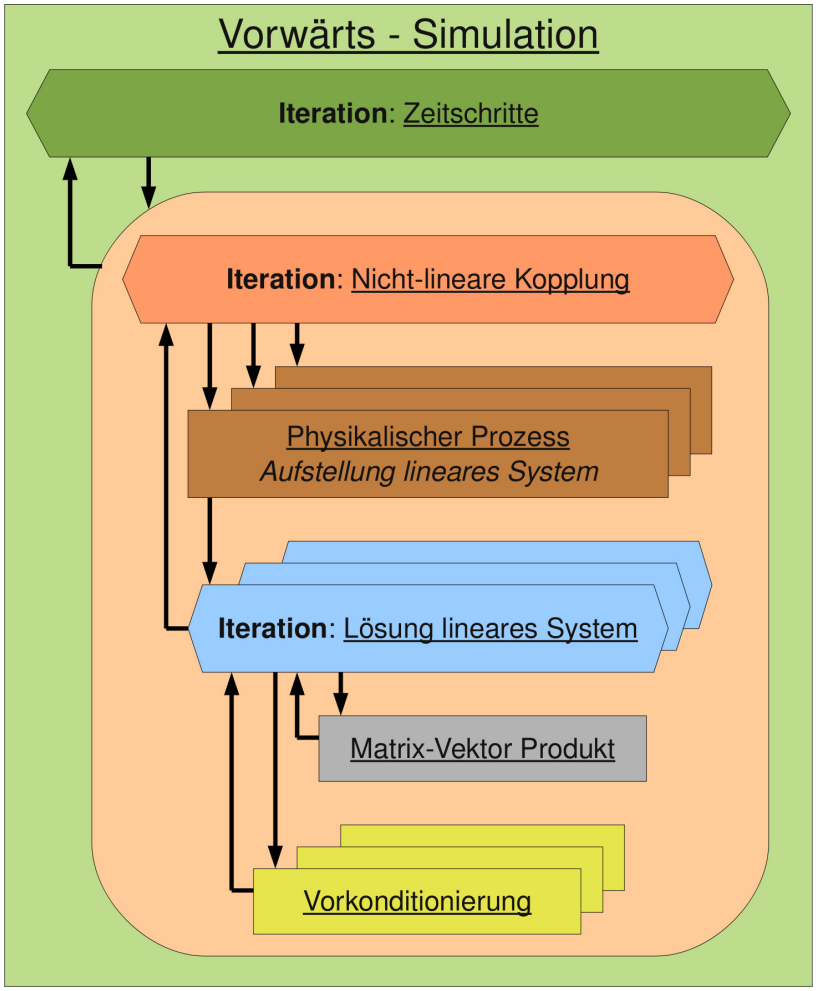


Abbildung 1: Schematischer Ablauf der Vorwärtssimulation.

simulierenden Zeitschritte (grün). Für jeden Zeitschritt wird die oben erwähnte Fixpunktiteration zur Lösung des gekoppelten Systems durchgeführt (rot, orange). Entsprechend wird in jeder einzelnen Iteration für jeden physikalischen Prozess bzw. seiner PDGL ein lineares Ersatzgleichungssystem aufgestellt (braun). Ausgenommen ist hier das elektrische Potential, welches nur einmalig am Ende eines Zeitschrittes berechnet werden muss.

Darauf hin werden jeweils die aufgestellten Systeme an die linearen Gleichungslöser zur endgültigen Berechnung der Zustandsgrößen, wie h oder T, übergeben (blau). Einige dieser Gleichungslöser [10, 11] benötigen externe Aufrufe in Form von Matrix-Vektor-Produkten $M \cdot x$ (grau) oder können verschiedene Vorkonditionierer [25, 26] (gelb) einbinden.

Die mehrfach hinter einander angedeuteten Objektkästen (blaue, braune, gelbe) stehen für eine bestimmte Flexibilität und Austauschbarkeit auf der jeweiligen Abstraktionsebene. Für die physikalischen Prozesse heißt das beispielsweise, dass nur eine oder nur bestimmte Zustandsgrößen berechnet werden. Auf der anderen Seite kann man bei den linearen Gleichungssystemlösern entscheiden, ob man nicht für ein sehr kleines Rechenmodell einen exakt rechnenden direkten Löser (z.B. aus LAPACK [27]) verwenden möchte und für ein großes System einen iterativen (wie BiCGStab [11]). Ursprünglich war auch für die Lösung der nicht-linearen Kopplung (rot, orange) eine flexible Schnittstelle vorgesehen. Zwei zwischenzeitlich implementierte Alternativen basierend auf Newton-Krylov [28, 29] (eine unter Verwendung des NITSOL-Paketes [30]) werden aber nicht mehr verwendet, weil unter anderem kein geeigneter Matrix-freier Vorkonditionierer zur Verfügung steht.

2.2 Von der Vorwärtssimulation zur inversen Problemstellung

Bei der Untersuchung des geologischen Untergrunds ist man auf begrenzte experimentelle Methoden angewiesen, die sich in Hinsicht auf gemessenen Größen (wie dem Wasserdruck oder der Temperatur), von erschließbaren Gesteinseigenschaften (z.B. die Permeabilität $\mathbf{k}$ oder die Wärmeleitfähigkeit λ) oder von impliziten Annahmen stark unterscheiden. Diese unterschiedlichen experimentellen Beobachtungen („Messdaten“) werden oft nur von wenigen Eigenschaften oder Prozessen bestimmt, so dass damit auch nur eingeschränkte Aussagen möglich sind. Ein einfaches Beispiel ist die Temperaturverteilung T, die im einfachsten Fall hauptsächlich von der räumlichen Verteilung der Wärmeleitfähigkeit vereinzelter Gesteinsschichten abhängen kann. In Kurzform lässt sich das als eine Funktion F wie in (5) schreiben.

$$T := F(\lambda) \tag{5}$$

Die Erfassung von mehr Messdaten eines Typs (z.B. nur die Temperatur) kann zwar unter günstigen Umständen zu verbesserten oder verfeinerten Vorstellungen führen, oft jedoch ist dies aus praktischen Gründen nicht durchführbar (Kosten im weitesten Sinn), oder aus prinzipiellen Erwägungen nicht möglich (physikalisches Messprinzip).

Die Kombination von unterschiedlichen experimentellen Methoden kann eine Lösung für das zuletzt genannte Problem sein. Dabei kann es sich beispielsweise um kombinierte Messungen der Temperatur und indirekt auch von dem hydraulischen Potential h handeln. Diese Abhängigkeiten als Funktion ausgedrückt ergibt

$$h, T := F(\mathbf{k}, \lambda). \tag{6}$$

Das übliche Vorgehen bei der quantitativen Interpretation experimenteller Daten ist wie folgt. Für die relevanten physikalischen, chemischen oder biologischen Prozesse werden mathematische Modelle aufgestellt, die das Systemverhalten beschreiben. Dies sind die schon erwähnten Systeme von gewöhnlichen oder partiellen Differentialgleichungen. Durch eine numerische Simulation des zu untersuchenden Systems (Lösung des Vorwärtsproblems bzw. Auswertung von F) werden „synthetische“ Beobachtungen (h, T) erzeugt, welche dann nach zuvor definierten Kriterien mit den „wirklichen“ Beobachtungen ($\hat{h}$ und $\hat{T}$) verglichen werden. Bei einer ausreichenden Übereinstimmung versucht man dann, ausgehend von den gemessenen Daten, Rückschlüsse auf die eigentlichen „realen“ Modellparameter (von z.B. $\mathbf{k}$ und λ) zu ziehen.

Hier sind Verfahren vorteilhaft, welche zufällig oder systematisch die Modellparameter (z.B. Gesteinseigenschaften oder ihre räumliche Verteilung) variieren, bis eine optimale Anpassung erreicht ist. Alle diese Methoden können als ein zur Lösung des Vorwärtspro-

blems zugehöriges inverses Problem F^{-1} (Parameterschätzung) im weiteren Sinne aufgefasst werden. Die Umkehrfunktion von F kann man einerseits wie

$$\mathbf{k}, \lambda := F^{-1}(h, T) \tag{7}$$

schreiben und mit dem Einsetzen der Beobachtungswerte als $\hat{\mathbf{k}}, \hat{\lambda} := F^{-1}(\hat{h}, \hat{T})$ umformulieren. Mit der entsprechenden Methodik für die Berechnung von F^{-1} lassen sich dann leichter Fragestellungen nach guten Werten $(\hat{\mathbf{k}}, \hat{\lambda})$ für die Modellparameter $(\mathbf{k}, \lambda)$ zu gegebenen Messwerten $(\hat{h}, \hat{T})$ beantworten.

Bei dieser sehr vereinfachten Sicht wird allerdings nicht berücksichtigt, dass es sich bei den inversen Problemstellungen im Allgemeinen um so genannte „schlecht gestellte“ Probleme handelt (engl. „ill-posed problems“, siehe [31, 32, 33, 34, 35, 36]), also nicht immer gesichert ist, dass eine eindeutige Lösung existiert. Während das Vorwärtsproblem meist durch eine gut gestellte Modell-Simulation $F(\mathbf{k}, \lambda)$ gelöst werden kann, können die teilweise diskreten Messwerte $(\hat{h}, \hat{T})$ für die Auswertung des inversen F^{-1} zum Hindernis werden. So können beispielsweise zu wenige oder widersprüchliche Messwerte (wegen Messfehlern) dazu führen, dass es keine (oder eine nicht eindeutige) Lösung im numerischen Sinne für $F^{-1}(\hat{h}, \hat{T})$ gibt. Darüber hinaus können besondere physikalische Modell-Strukturen dazu führen, dass die Lösung von F nicht stetig von den Eingabeparametern (z.B. $\mathbf{k}, \lambda$) abhängt. Abhilfe kann bei solchen (schlecht gestellten) Problemen eine Umformulierung mittels Regularisierung (siehe dazu [37, 38, 5]) schaffen. Dazu ist es hilfreich und bei einigen Methoden notwendig, so genannte „a priori“-Vorinformation über die zu schätzenden Modellparameter (bzw. deren angestrebten Fehler) einzubringen.

Für das dieser Arbeit zugrunde liegende Simulationswerkzeug bestehen die Methoden zur Lösung der inversen Problemstellung aus Formulierungen einer qualitativen Gütefunktion und verschiedenen Optimierungsalgorithmen. Die Gütefunktion soll hier lediglich unter Berücksichtigung der a priori Information die Übereinstimmung der synthetischen mit den gemessenen Daten (bei Beachtung der Messfehler) in mathematischer Form ausdrücken. Für die in dieser Arbeit vorgestellten Anwendungen werden deterministische Varianten des bayesschen Ansatzes zur Optimierung verwendet [39, 40, 41].

Sowohl die Gütefunktion als auch der konkrete Optimierungsalgorithmus müssen aufeinander abgestimmt sein, deshalb werden sie hier unter dem Begriff „Optimierungsfunktion“ zusammengefasst. Bei den Optimierungsfunktionen für das zugrunde liegende Projekt handelt es sich um verschiedene Formulierungen eines Minimierungsproblems [42]. Hauptsächlich wird eine bayessche Formulierung mit einer Minimierung durch ein Gauß-Newton-Verfahren und anschließender Schrittweitenbestimmung (engl. „line-search“) verwendet [2]. Zusätzlich wurden dessen Daten- und Parameterraum Formulierungen nach [43], direkte Minimierungen mittels „nicht-linearer konjugierter Gradienten“-Verfahren (siehe NLCG in [44]) und Quasi-Newton-Techniken basierend auf der LBFGS-Softwarebibliothek [45, 46] implementiert.

Den ganzen Optimierungsprozess zur Beantwortung der inversen Fragestellung (F^{-1}) kann man aufbauend auf dem Schema der Vorwärtssimulation (Abbildung 1) wie in Abbildung 2 gezeigt schematisch darstellen. Dieses Schema zeigt beispielhaft die Parameterschätzung für den deterministischen Optimierungsprozess. Es beginnt mit der äußeren Iterationsschleife (**Iteration**: Parameterschätzung - obere dunkel-graue Box) über die Anzahl der Anpassungsversuche für die Modellparameter (z.B. $\mathbf{k}, \lambda$). Für jeden Anpassungsversuch (Inversionsiteration) müssen zuerst alle Richtungsableitungen (**Iteration**: Jacobi-Matrix - gelbe Box) für alle Zustandsvariablen (z.B. h, T) nach den verschiedenen Modellparametern berechnet werden, also jeweils eine Iteration für $\partial(h, T)/\partial\mathbf{k}$ und eine für $\partial(h, T)/\partial\lambda$. Das kann man entweder direkt durch den Aufruf der Ableitungsfunktion der Vorwärtssimulation bewerkstelligen oder mit einem zweistufigen Verfahren.

Letzteres ist eine für die Ableitungsberechnung wichtige Besonderheit und wird in der Literatur (z.B. [47, 48, 49]) auch als „verspäteter Einsatz“ bezeichnet (im engl. unter „delayed propagation of derivatives“). Man kann die Technik in Abbildung 2 daran erkennen, dass innerhalb der Zeitschrittschleife (grüne Box) zuerst die originale nicht-lineare

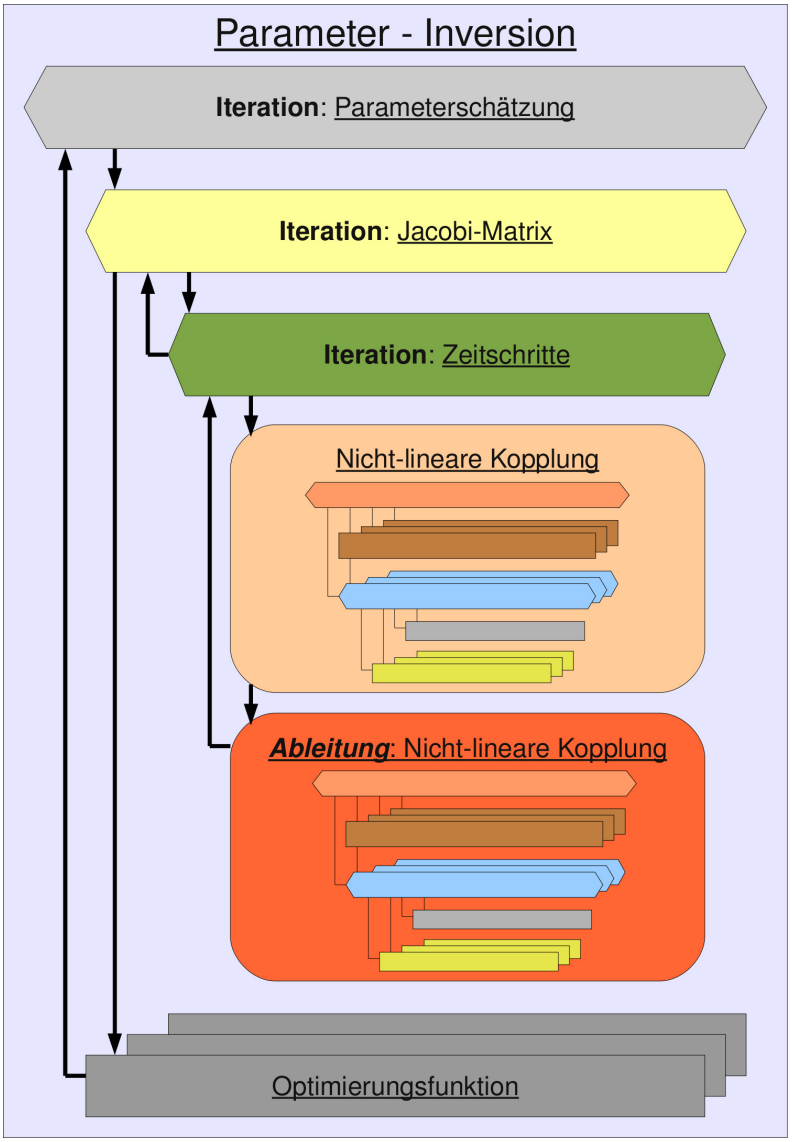


Abbildung 2: Schematischer Ablauf für die Parameterschätzung (Inversion).

Kopplung (originale Modell-Simulation - Hintergrund in orange) ausgeführt wird. Erst im Anschluss, wenn dort der Fixpunkt erreicht wurde, wird dann die eigentliche Ableitungsberechnung (dunkel-roter Hintergrund) „verspätet“ angestoßen.

Hat man die Jacobi-Matrix durch die mehrfachen kompletten Durchläufe der Zeitschrittschleife vollständig berechnet, kann man sie an eine der Optimierungsfunktionen (untere dunkel-graue Boxen) übergeben und eine neue Schätzung der Parameter $\mathbf{k}, \lambda$ (Anpassung) vornehmen. Danach beginnt dann die nächste Inversionsiteration (oben) mit dem Ziel weitere Verbesserungen zu erreichen.

Alternative Ansätze sind stochastische Verfahren, wie z.B. die geostatistische Simulation bis hin zur Parameterschätzung mit Monte Carlo (z.B. sequentieller Simulation [50, 51]) oder Ensemble Kalman Filter Methoden (EnKF), die methodisch auf der Berechnung unterschiedlich gestörter Versuche (Vorwärtssimulationen) beruhen. Aus technischer Sicht ergeben sich die wesentlichen Unterschiede gegenüber den deterministischen Methoden in der Verwendung anderer Optimierungsalgorithmen und der Berechnung vieler Vorwärtssimulationen statt einzelner Richtungsableitungen.

2.3 Profil der Projektanforderungen

Die Simulation geowissenschaftlicher Prozesse durch Systeme von partiellen Differentialgleichungen stellt folgende Herausforderungen.

1. Es liegt hier meist eine Multiphysik-Simulation vor, wobei die Aufstellung des Systems über alle PDGL und die Parametrisierung des inversen Problems nicht trivial ist, da hier problemangepasste Entscheidungen bezüglich der zu berücksichtigenden Prozesse und der physikalischen Modelle getroffen werden müssen. Diese bestimmen die Natur und Komplexität des zu lösenden Systems.

2. Für die numerische Lösung ist eine Diskretisierung der PDGL erforderlich. Die Anforderungen an räumlicher oder zeitlicher Auflösung einerseits und numerischer Genauigkeit und Stabilität andererseits müssen in einem vernünftigen Verhältnis zum Aufwand (Rechenzeit und Speicherverbrauch) stehen, da oft große und komplexe Modelle notwendig sind.

3. Inverse Probleme erfordern unabhängig von der verwendeten Methode einen um mehrere Größenordnungen höheren Aufwand, da hier entweder eine große Zahl von Vorwärtsberechnungen erfolgen muss (z.B. bei stochastischen Methoden), oder die aufwändige Berechnung von Ableitungsinformationen (z.B. für deterministische Methoden). Daraus resultiert, dass ein hoher Aufwand für die systematische Modellveränderung nötig ist.

Diese Arbeit trägt den sich so ergebenden Umständen im Zusammenhang mit einem hydro-geothermalen Simulationscode Rechnung. Ausgehend von einem bestehenden reinen Vorwärtssimulationscode (SHEMAT [20]) wurde durch Neuentwicklung, Umformulierung und Erweiterung ein umfassenderes Werkzeug (Suite) zur Handhabung der entsprechenden inversen Probleme entwickelt. Es wird in dieser Arbeit oft auch als „Suite-Projekt/Software“ oder „Simulations-Suite“ bezeichnet. Insgesamt ergaben sich eine ganze Reihe von Besonderheiten, die in ihrer Gesamtheit eine Neuheit im Bereich der geophysikalischen Simulationswerkzeuge darstellt:

1. Die Programmsuite macht eine umfangreiche und schnell erweiterbare Berechnung verschiedenster geophysikalischer Zustandsvariablen im porösen Medium möglich. Dazu gehören der durch Druckunterschiede hervorgerufene Fluidfluss, der Wärme- und Massentransport durch diesen Fluss, sowie das durch die Fluidbewegung hervorgerufene elektrokinetische Potential.

2. Alle diese Prozesse werden gekoppelt betrachtet, da die Gesteinseigenschaften, die sich in den Koeffizienten der untersuchten partiellen Differentialgleichungen wiederfinden, von den Zustandsvariablen abhängen.

3. Gleichzeitig mit der Anpassung und Weiterentwicklung des Simulationscodes für die Vorwärtsberechnung wird der dazugehörige Programmcode für die inverse Problembehandlung synchron weiterentwickelt.

4. Es werden verschiedene auf dem identischen Vorwärtscode beruhende Optimierungsmethoden eingesetzt, die eine Parameterschätzung (Inversion) bezüglich nahezu aller vorkommenden Gesteinseigenschaften ermöglichen. Dies sind sowohl stochastische [52, 53, 3, 54] als auch deterministische [40, 41, 39] Verfahren. Bei letzteren wurde besonderes Gewicht auf die ableitungsbasierten Methoden und die Verwendung exakter Ableitungen gelegt.

Der letzten beiden Punkte 3. und 4. wurden durch den Einsatz von Techniken des „automatischen Differenzierens“ (AD) [55, 47, 56] erreicht, die durch eine automatisierte Programmcode Transformation erlauben, neben der ursprünglichen Funktion auch die Ableitung nach einzelnen Parametern exakt zu berechnen. Dies ist bei der Lösung von inversen

Problemen (z.B. in [57]) erforderlich, kann jedoch auch bei der Implementation effizienter Vorwärtsalgorithmen (Newton basiert [28, 29]) hilfreich sein. Im Fall der inversen Probleme wurde gezeigt [58, 59, 60, 61], dass sich Vorteile bei der Parameterschätzung gegenüber der Verwendung von numerisch bestimmten Ableitungen (z.B. durch Differenzenquotienten) ergeben. Dies ist zu erwarten, da kein Suchprozess für eine optimale Bestimmung der Schrittweite mehr notwendig ist und oft auch weniger Inversionsiterationen (Iterationen der Parameterschätzung) erforderlich sind.

Da bei der Parameterschätzung viel Wert auf möglichst viele verschiedene Gesteinsparameter und zeitabhängiger Randwerte gelegt wird, ist auch hier eine erhöhte Flexibilität, über die Punkte 1. und 2. hinaus, gefordert. Wie in dem Einleitungskapitel 1 schon erläutert wurde, profitieren die erwähnten Verfahren nicht nur von einer Parallelisierung, sondern die Anforderungen sind teilweise erst dadurch mit einem angemessenen Berechnungszeitraum erfüllbar. Da die Parallelisierung sowohl die Lösung des Vorwärtsproblems (linearer und nicht-linearer Löser) als auch die ableitungsbasierten und stochastischen Methoden umfassen soll, wurde ein mehrstufiges (geschachteltes) Parallelisierungskonzept angestrebt.

2.3.1 Aus den Projektanforderungen resultierende Konzeptanforderungen

Für die zu erarbeitende Simulations-Suite wurde aufgrund des Vorgängers SHEMAT als Hauptimplementationssprache Fortran77/95 [62, 63] und zur Parallelisierung wegen der algorithmischen Flexibilität und des geringeren Implementationsaufwands die Spracherweiterung OpenMP festgelegt. Hieraus ergaben sich im Zusammenhang mit AD verschiedene Teilproblemstellungen.

Die hier vorliegende Arbeit entwickelt nun neue generalisierte Lösungsansätze der Teilproblematiken in Hinblick auf ein Gesamtlösungs-Konzept. Im Einzelnen ergeben sich drei wichtige Teilproblemstellungen:

1. Festlegung der Schnittstellen und Charakteristika aller Funktionen bzw. deren Erweiterungen, die für die Lösung des Vorwärtsproblems erforderlich sind, der konzeptionellen Verträglichkeit mit einer möglichen AD-Anwendung sicher stellt oder die übergeordnete Parallelisierungsstufe der Inversionsverfahren erlaubt.

2. Entwicklung von Parallelisierungsansätzen, die bei der Berechnung von Richtungsableitungen speziell für mit durch AD generierte Programmcodes verwendbar sind, unter Berücksichtigung einer geringen Vor- und Nachbearbeitung. Denn AD-Aktualisierungen werden häufig notwendig, sobald Funktionsmodifikationen am Vorwärtsproblem erzeugt wurden.

3. Festlegung einer effizienten Parallelisierungsstrategie, die möglichst unabhängig von den geowissenschaftlichen Anforderungen implementiert werden kann und gleichzeitig aktuelle SMP- und vSMP-Rechner unterstützt, um realistische Probleme mit vertretbaren Zeitaufwand lösen zu können.

Die beiden gleichzeitig zu erfüllenden Eigenschaften einer hohen Erweiterungsproduktivität und hoher Rechenleistung auf modernen Rechner-Architekturen (siehe Kapitel 2.5.3) unterscheidet die in dieser Arbeit vorgestellten Konzeptlösungen von denen herkömmlicher Konzeptfindungsprozesse, welche im numerischen Umfeld typischerweise nur die Rechenleistung adressiert.

2.4 Abgrenzung und Vergleich zu anderer Simulationssoftware

Ein Vergleich mit anderen Projekten aus dem geowissenschaftlichen Umfeld veranschaulicht die wichtigsten Neuerungen und die grundsätzlichen Unterschiede bzw. Vorteile durch die in den späteren Kapiteln beschriebenen Techniken.

Stellvertretend für die große Vielfalt an hydro-geothermalen Simulationswerkzeugen soll das TOUGH2-Projekt [64, 65] betrachtet werden. Es eignet sich für den nachfolgenden Vergleich besonders gut, weil es einerseits sehr ausgereift ist und andererseits viele Aspekte der den geowissenschaftlichen Projektzielen entsprechenden multiphysikalischen Simulation beherrscht. Es gibt davon abweichend physikalische Aspekte und numerische Algorithmen, die ausschließlich entweder TOUGH2 oder die hier zugrunde liegende Simulations-Suite bewältigen. Das ist den jeweiligen unterschiedlichen wissenschaftlichen Zielsetzungen geschuldet und für die nachfolgende Erläuterungen unerheblich.

Das TOUGH2-Projekt ist ein gutes Beispiel, für die typischen Schwierigkeiten bei der Entwicklung von Hochleistungs-Simulationswerkzeugen. Denn dabei handelt es sich oft um ein im Nachhinein (mit neuen Anforderungen) wachsenden Entwicklungsprozess. Das kann zur Folge haben, dass der Simulationscode nur mit sehr großem Aufwand an neue Anforderungen angepasst werden kann oder separate Entwicklungszweige notwendig macht, die nicht alle Anforderungen zugleich (oder optimal) erfüllen.

Eine oft nachträglich auftretende Anforderung ist beispielsweise die Parallelisierung und sie führte zu dem mit Hilfe von MPI [15] entwickelten TOUGH2-MP [66] Programmcode. Bei einem anderen Entwicklungszweig handelt es sich um die zur Lösung von Inversionsproblemen nötige Erweiterung ITOUGH2 [67, 68]. Diese Erweiterung lässt sich logisch leicht von dem Vorwärtsproblem trennen und ist so aufgebaut, dass nach Bedarf der Vorwärtscode TOUGH2 aufgerufen werden kann.

Im Wesentlichen ist das hier möglich, weil ITOUGH2 für die deterministischen Verfahren, zur Parameterschätzung, mehrere Ableitungen über die Berechnung von Differenzenquotienten vornehmen kann. Somit werden dann bei der deterministischen wie auch für die stochastische Lösungsfindung mehrere Auswertungen von Vorwärtsproblemen (TOUGH2-Aufrufe) benötigt. Im Gegensatz dazu soll in der dieser Arbeit zugrunde liegenden Simulations-Suite bei den deterministischen Inversionsverfahren auf exakte Ableitungen mit Hilfe von AD-Techniken zurückgegriffen werden können. Dieser numerische Vorteil (belegt in [58, 59, 60, 61]) ist einer der *wichtigen* Unterschiede gegenüber den herkömmlichen geowissenschaftlichen Simulationswerkzeugen und war zum Zeitpunkt der Fertigstellung (siehe dazu [69]) ein Vorreiter in diesem Umfeld.

Um den erhöhten Berechnungsaufwand bei einer Parameterschätzung gerecht zu werden, kann man auch hier damit beginnen, eine zusätzliche Parallelisierung aufzubauen. Für unser Beispiel wurde das mit Hilfe von PVM [70] als gesonderter Entwicklungszweig ITOUGH2-PVM [71] verwirklicht. Während in TOUGH2-MP ein einzelnes Vorwärtsproblem selbst parallel berechnet werden kann, handelt es sich bei ITOUGH2-PVM um eine Parallelisierung zur gleichzeitigen Berechnung mehrerer unabhängiger (serieller) Vorwärtsprobleme. Aus vielerlei Gründen, die später im Kapitel 5.2.4 erläutert werden, ist eine Kombination der Parallelisierungen bei Berechnungen mit ITOUGH2-PVM und den daraus resultierenden TOUGH2-Aufrufen in Form von TOUGH2-MP wünschenswert.

Handelt es sich, wie in diesem Beispiel angedeutet, um sehr unterschiedliche Arten von Parallelisierungen (MPI und PVM), kann eine kombinierende Implementation sehr aufwändig oder in dem gesteckten Rahmen sogar unerfüllbar sein. In der speziellen Literatur zu ITOUGH2-PVM [71] und TOUGH2-MP [66] findet man keinen eindeutigen Hinweis, ob hier eine Kombination der Parallelisierungen möglich ist. Allerdings weist die Internetseite des Projekts [72] mit „PEST/JUPITER“-Protokoll auf eine Erweiterung hin, die unter anderem genau so eine kombinierte Parallelisierung unterstützt.

Festzuhalten ist, dass dieser schrittweise Ansatz zu vielen Entwicklungszweigen und dadurch zu Wartbarkeits- und Installations-Problemen führen kann. Im Gegensatz dazu wurde für die Simulations-Suite ein ganzheitliches Konzept entwickelt, welches Inversion unter Nutzung von Parallelisierung, und Code-Wartbarkeit sowie Erweiterungsproduktivität realisiert.

Ein *wichtiger* Hauptunterschied ist die Verwendung einer einzelnen Art der Parallelisierung (OpenMP) und deren Fähigkeit, die wünschenswerte Kombination der verschiedenen

Parallelisierungsansprüche in einer natürlichen Art und Weise zu ermöglichen. Das verbessert insgesamt die Eignung für das Hochleistungsrechnen, den Implementationsaufwand für die Parallelisierungen und damit die spätere Wartbarkeit und Erweiterbarkeit.

2.5 Rechner-Architekturen

Dieses Kapitel widmet sich den heute wichtigsten Rechner-Architekturen. Es geht dabei nicht um eine vollständige Aufzählung, sondern um Rechnersysteme, die grob in Systeme mit „gemeinsamen Speicher“ und „verteilten Speicher“ unterteilt werden können. Sie stellen bezüglich einer in dieser Arbeit sehr wichtigen Eigenschaft, der Speicherbandbreite, die aussage-kräftigste Klassifizierung dar.

2.5.1 „shared memory“-Architektur

Mittlerweile enthalten aktuelle Prozessoren eine Vielzahl von Rechenkernen (engl. „cores“). Da sich alle Rechenkerne den Hauptspeicher teilen („gemeinsamer Speicher“), verwendet man auch den englischen Fachbegriff „shared memory“. Der Programmieraufwand für eine Parallelisierung ist auf Grund des einfachen Speichermodells vergleichsweise gering.

Nachfolgende Abbildung 3 soll dies anhand eines einfachen Systems mit einem blaumarkierten Multikern-Prozessor und seinem, für alle Rechenkerne (Kern1 bis Kern4) gleich-

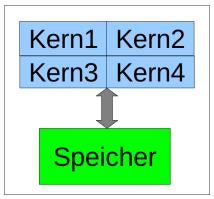


Abbildung 3: Einfaches Symmetrisches Multiprozessorsystem (SMP) mit einem 4Kern-Prozessor (engl. „quad-core“), der auf einen „gemeinsamen“ Hauptspeicher zugreift.

berechtigten, grün-markierten Hauptspeicher (kurz „Speicher“) veranschaulichen. Unter „gleichberechtigt“ ist zu verstehen, dass alle Rechenkerne einen gleich schnellen Zugriff auf den Hauptspeicher haben, weshalb ein solches System auch mit „Einheitlichen Speicherzugriff“ (engl. „uniform memory access“, kurz UMA) oder als „Symmetrisches Multiprozessorsystem“ (kurz SMP, engl. „symmetric multi processor“) bezeichnet wird.

Daraus resultiert, dass sich die Speicherbandbreite mehr oder weniger auf alle Rechenkerne aufteilt. Die Speicherbandbreite bezeichnet die Menge an Daten, die pro Zeiteinheit in den Rechenkern zur Abarbeitung geladen werden kann. Im Allgemeinen kann ein SMP-System auch aus mehreren Einzel- oder Multikern-Prozessoren bestehen.

„ccNUMA“-Charakteristika

Wenn es in einem SMP-System viele Rechenkerne oder Prozessoren gibt, dann stehen sie letztendlich bei dem Zugriff auf den „gemeinsamen“ Hauptspeicher in Konkurrenz zu einander. Das kann zu außerordentlichen Verzögerungen beim Datenzugriff führen. Um dies zu umgehen, gibt es die so genannten „ccNUMA“-Systeme (engl. „cache-coherent Non-Uniform Memory Architecture“), siehe Abbildung 4. Man kann in dem Schema sehen, dass jeder der beiden 4Kern-Prozessoren seinen eigenen direkt angebundenen Speicher hat. Darüber hinaus können in diesem Beispiel die beiden Prozessoren über einen zusätzlichen rot-markierten internen Datenbus kommunizieren und auf Daten in dem anderen Speicher zugreifen.

Dies bedingt zwei wichtige Besonderheiten bei dem Umgang mit ccNUMA-Systemen. 1. Es handelt sich hierbei noch immer um ein SMP-System mit „gemeinsam“ nutzbaren Speicher, da ein Rechenkern von dem linken Prozessor über den Datenbus auch auf den Speicher des rechten Prozessors voll zugreifen kann. Aus der Sicht des Programmierers ist

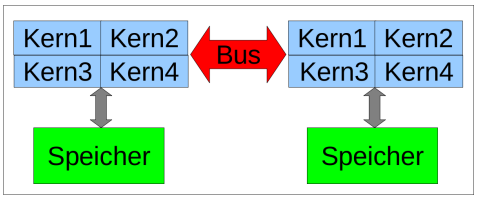


Abbildung 4: Einfaches „cache-coherent Non-Uniform Memory Architecture“-System (ccNUMA) bestehend aus einem SMP-Rechenknoten mit zwei 4Kern-Prozessoren.

es hier nicht wahrnehmbar, ob die Daten von dem eigenen direkt angebunden Speicher geholt wurden oder von dem des Nachbarprozessors. 2. Obwohl der Datenzugriff transparent erfolgt, dauert der Zugriff auf den Nachbarspeicher deutlich länger. Das hat zur Folge, dass man für eine gute Rechenleistung darauf achten muss, die Daten für einen Rechenkern möglichst aus dem an den Prozessor direkt angebunden Hauptspeicher zu laden.

Darüber hinaus gibt es größere (höhergradige) ccNUMA-Systeme mit einer ganzen Hierarchie von Datenbussen, bei deren Zugriff man viele unterschiedliche Laufzeiten beachten muss. Allerdings ist es bei einer günstigen Platzierung der Daten denkbar, dass man auch von einer erhöhten Gesamtspeicherbandbreite profitieren kann. Denn mit jedem Prozessor und seinem direkt angebundenen Speicher kann im Idealfall die Gesamtspeicherbandbreite mit der Anzahl der Prozessoren (nicht Rechenkerne) skalieren.

Daraus folgt, dass die Datenplatzierung für alle ccNUMA-Systeme eine entscheidende Rolle spielt, wenn man von der zur Verfügung stehenden erhöhten Gesamtspeicherbandbreite profitieren möchte.

2.5.2 „distributed memory“-Architektur

Im Gegensatz dazu besteht bei einem System mit „verteiltem Speicher“ (engl. „distributed memory“) nicht die Möglichkeit, transparent auf den Speicher der anderen Prozessoren zuzugreifen. Im einfachsten Fall könnte ein solches System aus mehreren kleinen abgeschlossenen Rechenknoten bestehen, die über ein externes Netzwerk verbunden sind. Sobald man aus einem Knoten heraus auf die Daten eines anderen Rechenknotens zugreifen möchte, muss man hier explizit über das Netzwerk kommunizieren und die Daten austauschen, was den Programmieraufwand deutlich erhöht. Allerdings steht dem eine hohe theoretische Speicherbandbreite gegenüber, die mit der Anzahl der Rechenknoten skaliert.

Moderne Cluster - Netzwerkverbund von SMP-Knoten

Da die Kommunikation über das Netzwerk langsam sein kann, versucht man die Anzahl der über das Netzwerk kommunizierenden Rechenknoten klein zu halten. Deshalb bestehen heutzutage viele moderne Rechencluster aus einem Netzwerkverbund von einzelnen preiswerten kleineren SMP-Systemen.

Die effektive Programmierung erfordert dann allerdings die gleichzeitige Anwendung verschiedener Parallelisierungs-Paradigmen einmal für „verteilten“ und einmal für „gemeinsamen“ Speicher, was die Software-Komplexität stark erhöht.

2.5.3 vSMP-Architektur, Besonderheiten bei ScaleMP

Eine Alternative ist die Implementation einer Virtualisierungsschicht [73, 74] zur Abbildung einer Architektur mit „gemeinsamen Speicher“ über einem Netzwerkverbund von SMP-Konten, verschiedene Ansätze siehe [75, 76, 77]. Eine kommerziell-verfügbare Lösung wurde von der Firma ScaleMP [17] entwickelt (der vSMP™ - „versatile symmetric multiprocessor“ - „vielseitiger symmetrischer Multiprozessor“).

Das Schema in Abbildung 5 skizziert vSMP an einem Beispiel mit zwei SMP-Knoten.

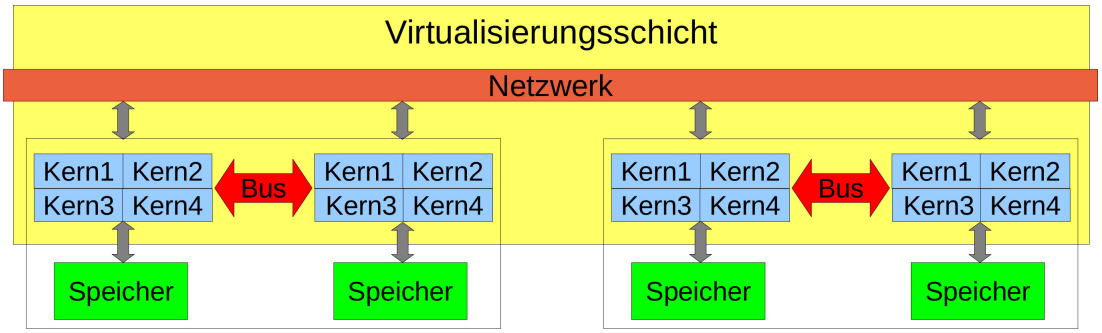


Abbildung 5: Höhergradiges ccNUMA-System in Form eines „vielseitigen symmetrischen Multiprozessor“-System (vSMP) mit zwei SMP-Rechenknoten (je zwei 4Kern-Prozessoren).

Aus Sicht des Programmierers gibt es nun ein großes (virtuell) zusammen hängendes SMP-System mit insgesamt vier 4Kern-Prozessoren. Sollte hier ein Rechenkern im linken Rechenknoten Daten aus dem rechten benötigen, dann sorgt die hell-gelb-markierte Virtualisierungsschicht für die transparente Kommunikation über das Netzwerk (also zwischen den Rechenknoten). Dadurch ist der Programmieraufwand genauso gering, wie bei einem echten SMP-System.

Da der Datenaustausch über das Netzwerk im Allgemeinen noch langsamer ist als zwischen zwei Prozessoren innerhalb eines Rechenknotens, kommt praktisch eine Art ccNUMA-Stufe hinzu. Sind schon die einzelnen Rechenknoten selbst kleine ccNUMA-Systeme, baut sich eine Art ccNUMA-Hierarchie höheren Grades auf. Das soll heißen, dass man hier mehr denn je auf eine günstige Datenplatzierung achten muss.

Abgesehen von dieser erhöhten ccNUMA-Eigenschaft hat man insgesamt aber ein kostengünstiges und potentiell sehr leistungsfähiges Rechnersystem mit geringem Programmieraufwand. Das macht diese zukunftsweisende Architektur so interessant für diese Arbeit und für das Hochleistungsrechnen allgemein.

2.6 Spracherweiterungen zur Parallelisierung: OpenMP vs. MPI

Bei der Parallelisierung von Programmcodes kann man auf eine ganze Reihe verschiedener Ansätze zur Parallelisierung zurückgreifen. Für die zuvor beschriebenen Architektur-Klassen verwendet man oft Spracherweiterungen, wie MPI [15, 16] oder OpenMP [13, 14]. Dabei ist der MPI-Standard der flexiblere, weil er sich sowohl auf Systemen mit „gemeinsamen“ und „verteilten“ Speicher einsetzen lässt. Allerdings bedingt dies oft einen nicht unerheblichen Programmieraufwand. Dagegen geht eine OpenMP-Parallelisierung im Allgemeinen mit einem viel geringeren Gesamtaufwand einher. Das Einsatzgebiet beschränkt sich für OpenMP aber auf Systeme mit „gemeinsamen Speicher“. Letztendlich sind der Programmieraufwand und die Ausweitung der Nutzbarkeit auf vSMP-Systeme die Hauptgründe für den ausschließlichen Fokus auf OpenMP in dieser Arbeit.

Während bei MPI die Datenobjekte für den Rechenprozess meist als lokal angesehen werden, liegt bei OpenMP ein komplexeres Speichermodell vor. Hier wird ein so genannter Datensichtbarkeitsbereich (engl. „data scoping“) definiert, welcher eine generelle Unterscheidung zwischen „lokalen“ (bzw. geschützten, Fachwort engl. *private*) und „geteilten“ (Fachwort engl. *shared*) Datenobjekten vornimmt. Für einen Rechenprozess ist der Zugriff auf eine Variable möglich, wenn sich diese in seinem „Sichtbarkeitsbereich“ befindet. Hierbei heißt „lokal“, dass nur der Rechenprozess bzw. OpenMP-Unterprozess (engl. „OpenMP thread“) selbst Zugriff auf sein Datenobjekt hat. Im „geteilten“ Fall (*shared*-Datenattribut) dürfen mehrere Unterprozesse gleichzeitig und direkt auf das selbe Datenobjekt zugreifen. Zur Vereinfachung definiert der OpenMP-Standard eine Reihe von Voreinstellungen („Datensichtbarkeitsbereich-Automatismus“). Die voreingestellten

Datensichtbarkeitsbereichs-Attribute der Datenobjekte können aber fast immer explizit umdefiniert werden, um sie besser an den Algorithmus anzupassen.

Ein völlig anderer Aspekt bei dem Umgang mit Parallelität ist der Umstand, dass das Betriebssystem im Allgemeinen selbst entscheidet, welcher Unterprozess wann auf welchem Prozessorkern ausgeführt wird. Diese Zugehörigkeit von Prozessorkern und Unterprozess ist normalerweise nicht fest und kann mehrmals wechseln. Wie schon zuvor erwähnt wurde, ist es für ccNUMA-Systeme aber wichtig, die Affinität von Unterprozessen und den von ihnen benötigten Daten sicherzustellen. Das kann durch eine so genannte automatische „Seitenmigration“ (Datenobjekte wechseln automatisch den Speicherplatz) oder durch eine feste Zuweisung des Rechenkerns gewährleistet werden.

Deshalb wurde für diese Arbeit bei allen Messungen implizit oder explizit erzwungen, dass jeder Unterprozess einem bestimmten Prozessorkern während der gesamten Laufzeit fest zugeordnet wird. Diese Zugehörigkeit wechselt nicht und sorgt damit dafür, dass prozessornah platzierte Datenobjekte eines Unterprozesses während der gesamten Laufzeit weitestgehend prozessornah bleiben. Deshalb wird im Weiteren bezüglich der Datenlokalität mehr oder weniger von einer Gleichstellung der beiden Begriffe Unterprozess und Prozessorkern ausgegangen. Belege und weiterführende Untersuchungen für die Nützlichkeit einer festen Zuordnung von Prozessorkern und Unterprozess findet man in [78].

Wichtig ist hierbei, dass alle in der vorliegenden Arbeit ausgeführten Erläuterungen zu dem Speichermodell, dem Datensichtbarkeitsbereich-Automatismus und dem Umgang mit den OpenMP-Direktiven und Klauseln sich auf den OpenMP-Standard [79] in Version 2.5 beziehen. Diese Einschränkung bedeutet nicht notwendigerweise, dass sich die hier gemachten Ausführungen nicht auch auf neuere Versionen übertragen lassen. Es legt vielmehr einen stabilen Zwischenstand zur Anwendbarkeit des vorliegenden Gesamtlösungs-Konzeptes fest, der weitestgehend mit ausgereiften Compiler-Implementationen verifiziert werden konnte.

3 Projektgetriebene Softwareentwicklung

Für die vorliegende Arbeit zeigt die Abbildung 6 eine Übersicht für eine Reihe von Grundanforderungen. Jede dieser Anforderungen hat bestimmte kennzeichnende Merkmale. Diese Merkmale bedingen wiederum bestimmte Abhängigkeiten und Beziehungen zwischen den einzelnen Grundanforderungen. Zum besseren Verständnis werden die farbigen Ob-

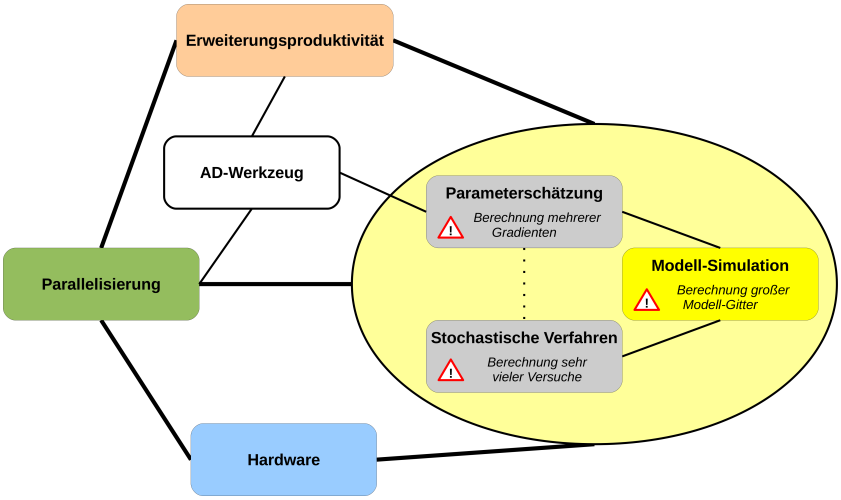


Abbildung 6: Übersicht der Grundanforderungen (farbige Objektkästen) und deren Beziehungen (schwarze Verbindungslinien).

jektkästen der Grundanforderungen mit den dazugehörenden Abhängigkeiten (schwarze Verbindungslinien) als Einführung verschiedener Abschnitte wiederholt dargestellt. Die Gesamtübersicht macht außerdem deutlich, dass eine schnelle Erweiterbarkeit und damit eine hohe Erweiterungsproduktivität für das zugrunde liegende Softwareprojekt einen hohen Stellenwert einnimmt. Deswegen ist als erstes eine klare Abgrenzung zu anderen Konzepten der Softwareentwicklung wichtig und wird direkt im nachfolgenden Kapitel 3.1 ausgeführt.

Die Abbildung 6 deutet auch die komplexen Zusammenhänge zwischen den einzelnen Grundanforderungen an. Diese Abhängigkeiten wurden im Rahmen der Forschungsarbeit herausgearbeitet und umfassend untersucht. Deswegen ist die ab Kapitel 3.2 aufgeführte detaillierte Zusammenstellung ein wichtiger Bestandteil der vorliegenden Forschungsarbeit.

3.1 Abgrenzung zu Agilen Methoden, Extreme Programming und Rapid Prototyping

Eine der wichtigsten Anforderungen ist eine hohe Erweiterungsproduktivität mit hoher Qualität. Für den stark projektgetriebenen Entwicklungsprozess bedeutet das, dass über einen längeren Zeitraum ständig neue Teilprojektziele erreicht werden müssen. Ein Teilprojektziel gilt als erreicht, wenn zu einem bestimmten Zeitpunkt ein definierter Funktionsumfang fertig gestellt, getestet und für die wissenschaftliche Forschung verwendet werden konnte.

Neben dem wissenschaftlichen Anspruch steht bei solchen Prozessen oft das Lösen der Programmieraufgabe im Vordergrund der Softwareentwicklung und weniger ein formalisiertes Vorgehen. Im Gegensatz zu dieser aus dem „Extreme Programming“ [80, 81] bekannten Vorgehensweise muss für das Suite-Projekt eine gewisse Qualität erhalten bleiben. Demnach muss der neue Funktionsumfang reibungslos in den Rahmen der bisherigen Software hineinpassen, ohne Einschränkungen bezüglich der bisherigen Funktionalität zu besitzen oder solche zu verursachen.

Die vorliegende Arbeit konzentriert sich deshalb auf Softwaretechniken, die solche projektgetriebenen Entwicklungsarbeiten programmiertechnisch stark erleichtern und beschleunigen. Das hält einen wichtigen Bereich des Softwareentwicklungsprozesses flexibel und schlank. Ausgehend von einer notwendigen Agilität (Beweglichkeit in der Softwareentwicklung) und den Vorteilen des „Extreme Programming“, werden wichtige „agile Prinzipien“ (manchmal auch als „Agile Methode“ bezeichnet) eingebunden und genutzt. Details zur „Agilen Softwareentwicklung“ kann man in [82, 83] nachlesen.

Ein wichtiger Unterschied besteht aber darin, dass in der vorliegenden Arbeit nicht versucht wird, die reine Entwurfsphase auf das absolutes Mindestmaß zu reduzieren, sondern die hier beschriebenen Softwaretechniken einzusetzen. Man erhält dafür den Vorteil, dass man viel schneller und reibungsloser ausführbare Software mit dem im Teilprojekt definierten wissenschaftlichen Funktionsumfang erstellen kann. Im Detail würden sonst einige der Grundanforderungen, wie z.B. die Parallelisierung, immer einen gewissen Zusatzaufwand in der Entwicklungsarbeit verursachen.

Auf Grund der hier eingesetzten Softwarekonzepte und Techniken kann dieser für die weiteren Projektphasen fast vollständig entfallen. Die obersten vier Ziele bei der Software-Entwicklung bzw. Erweiterung definieren sich hier dementsprechend in Anlehnung an einige der „agilen Prinzipien“ (siehe [82, 83]).

- Vorhandene Datenobjekte (oder Programmroutinen) müssen mehrfach verwenden werden können *(z.B. temporäre Hilfsfelder)*, um den Speicherbedarf (oder die Programmwartung) zu reduzieren. Die Wiederverwendung hat gegebenenfalls auch Vorrang gegenüber einer nur leicht modifizierten Programmkopie *(z.B. die Hilfsfelder auf maximalen Anwendungsfall ausdehnen)*. In einem solchen Fall wird dann der schon vorhandene Programmabschnitt eher um diese neue Fähigkeit erweitert.

- Zweckmäßigkeit muss im Allgemeinen gegeben sein, darf aber zu Versuchszwecken im Rahmen der wissenschaftlichen Forschung außer Acht gelassen werden *(z.B. Test einer Newton-Methode als alternativen nicht-linearen Systemlöser)*. Dabei kann es sich um kleine Speziallösungen handeln, die zum schnellen Erreichen des Projektziels notwendig sind, aber langfristig entfernt oder allgemeiner umgesetzt werden müssen.

- Die gewünschte Flexibilität in der algorithmischen Erweiterbarkeit bedingt eine an Schnittstellen orientierte Programmierung *(z.B. für die linearen Gleichungslöser)*. Diese möglichst stabilen Schnittstellen können aber durchaus vorübergehend erweitert werden, um schneller das Projektziel zu erreichen *(z.B. um einen linearen Gleichungslöser mit alternativer Datenstruktur zu testen)*.

- Nicht zuletzt ist anzustreben, die einfachste mögliche Lösung eines Problems zu wählen, solange keine wichtigen Grundanforderungen gefährdet werden.

Darüber hinaus konnte man in dem dieser Arbeit zugrunde liegendem Softwareprojekt davon ausgehen, dass zumindest die Funktionsklassen und Gruppierungen im Wesentlichen von Anfang an fest standen, nicht aber deren Umfang oder andere Einzelheiten zu den zu verwendenden Algorithmen selbst.

Rapid Prototyping

Auch für die Softwareentwicklung gibt es den Begriff des „Rapid Prototyping“ (RPT). Allerdings fasst man darunter oft eine ganze Reihe von Werkzeugen (im weitesten Sinne) zusammen, die eine große Menge an Funktionalität vereinbaren und erleichtert nutzbar machen.

Auf diesem Weg kann man oft die grundlegende Funktionalität für ein neues Projekt vortesten, und damit eine gute Grundlage für die eigentliche Softwareentwicklung schaffen. Dabei bietet die starke Spezialisierung der meisten RPT-Werkzeuge nicht nur Vorteile, sie erschweren oft auch Optimierungen oder Parallelisierung (soweit nicht direkt vorgesehen).

Denn bei ihnen steht üblicherweise die schnelle Entwicklung im Vordergrund oder ein großer Funktionsumfang und nicht unbedingt der Einsatz im Hochleistungsrechnen.

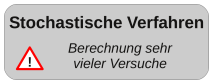
Ein anderes Problem besteht darin, dass der Entwickler einer neuen Anwendung im stärkeren Maße von dem verwendeten Werkzeug abhängt. Angenommen, man setzt beispielsweise MATLAB™ [84] als RPT-Werkzeug für ein neues Projekt ein. Dann kann man bei dem Umgang mit „dichtbesetzten“ Matrizen teilweise schon auf Parallelisierung zurückgreifen, um die anfallende Rechenarbeit zeitlich zu verkürzen. Für „schwachbesetzte“ Matrizen (Fachwort engl. „sparse matrices“) gibt es diese Fähigkeit aber noch nicht in gleicher Weise. Das heißt für den Anwendungsentwickler, dass er oft erst auf die entsprechende Unterstützung seitens des RPT-Werkzeugs bzw. dessen Hersteller warten muss.

Damit wird auch klar, dass die für diese Arbeit vorgegebene Zielausrichtung nicht vereinbar wäre mit dem typischen „Rapid Prototyping“-Ansatz. Die potentiellen zusätzlichen Abhängigkeiten mit dem zu verwendenden RPT-Werkzeug würde alle unabhängigen Optimierungen und Anpassungen für spezielle Rechner-Architekturen oder die Parallelisierungen erschweren und könnten die Vorteile dadurch sogar zunichte machen.

3.2 Grundanforderungen der Simulationssoftware

Für die numerische Simulation eines physikalischen Vorgangs (Vorwärtssimulation), muss im Allgemeinen das zugrunde liegende physikalische Modell spezifiziert sein. Im Folgenden bezeichnet die Modell-Simulation die Gesamtheit aller Algorithmen und Funktionen zur vollständigen numerischen Berechnung des physikalischen Modells bzw. Vorgangs. Eine konkrete Modell-Konfiguration legt alle physikalischen und numerischen Eigenschaften für die eigentliche Simulation fest. Darunter zählt zu den numerischen Eigenschaften die Art und Genauigkeit der Diskretisierung. Unabhängig von den anderen numerischen und physikalischen Eigenschaften, die die Simulationssoftware bewältigen können muss, bedingt der Grad der Diskretisierung direkt bestimmte Anforderungen an das Rechnersystem. Ein besonders fein diskretisiertes Modell mit hoher Auflösung (viele Gitterpunkte), benötigt unter anderem entsprechend viel Datenplatz für deren interne Repräsentation auf dem Rechnersystem. Daraus folgt auch, dass mit der Anzahl der repräsentierten und zu berechnenden Gitterpunkte auch deren Gesamtberechnungsaufwand entsprechend hoch ist.

Merkmale: *hoher Speicherverbrauch* und *hoher Berechnungsaufwand* entsprechend der Modell-Diskretisierung, algorithmische Flexibilität in der Berechnung *unterschiedlicher physikalischer Eigenschaften*

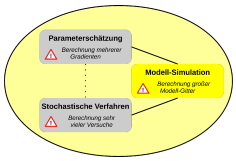


Stochastische Verfahren [1, 2, 3] können angewendet werden bei Optimierungsverfahren zur Parameterschätzung, genauso wie bei Unsicherheitsanalysen. In beiden Fällen gehört dazu üblicherweise eine vielfache Berechnung von Modell-Simulationen. Diese unterscheiden sich dann anhand einer modifizierten Modell-Konfiguration mit kleinen Abweichungen bzw. Störungen bezüglich bestimmter physikalischer Eigenschaften. Deren jeweilige Berechnung wird im Folgenden als Versuch bezeichnet. Die Anwendung einer stochastischen Methode kann man prinzipiell in drei Prozessabschnitte aufteilen: 1. die initiale Generierung der einzelnen Modell-Konfigurationen, 2. die Berechnung dieser unterschiedlichen Versuche und 3. die Analyse und Weiterverarbeitung der Berechnungsresultate. Darüber hinaus kann es sich bei dem Verfahren um einen iterativen Prozess handeln, der abhängig von dem jeweiligen Analyse-Ergebnis weitere Durchläufe der Prozesskette von 1. bis 3. anstoßen kann. Kritisch ist meist der zweite Prozessabschnitt, da die numerische Simulation aller generierten Versuche einerseits den Berechnungsaufwand insgesamt erhöht und andererseits sich damit auch der Speicherplatzbedarf erhöhen kann.

Merkmale: *erhöhter Speicherverbrauch* und *erhöhter Berechnungsaufwand* entsprechend

der Anzahl der Versuche, algorithmische Flexibilität zur Verwendung *verschiedener stochastischer Verfahren und Algorithmen*

Neben den stochastischen Methoden gibt es die ableitungsbasierten Optimierungsverfahren für die Parameterschätzung. Obwohl sie unter stark vereinfachten Annahmen aus stochastischen Grundprinzipien hergeleitet werden können [40, 41], sind sie eher deterministischer Natur. Sie verlangen im Allgemeinen die Berechnung von Ableitungen für die so genannten „aktiven“ Parameter. Die Verwendung der Gradienten kann entweder durch explizites Berechnen oder indirekt vorgenommen werden. Im letzteren Fall werden nicht die Gradienten selbst, sondern bestimmte Linearkombinationen dieser Ableitungen berechnet. In der vorliegenden Arbeit wird im Gegensatz dazu von Optimierungsverfahren mit der Nutzung von explizit berechneten Gradienten ausgegangen, wobei die Ableitungen als Jacobi-Matrix gesammelt werden. Das ermöglicht unter anderem eine effiziente Weiterverarbeitung der Jacobi-Informationen für numerische Unsicherheitsanalysen in Form von Matrizen zur Darstellung der Auflösung und Kovarianz. Die vollständige Bestimmung der Jacobi-Matrix und damit aller benötigter Ableitungen führt deshalb auch zu einem erhöhten Berechnungsaufwand und mitunter auch zu einem erhöhten Speicherplatzverbrauch.
Merkmale: *erhöhter Speicherverbrauch* und *erhöhter Berechnungsaufwand* entsprechend der Anzahl der Gradienten, algorithmische Flexibilität zur Verwendung *verschiedener ableitungsbasierter Optimierungsverfahren und Algorithmen*



Die Gesamtheit aller Programmteile für die Modell-Simulation, den stochastischen Verfahren und der Parameterschätzung bilden ein umfangreiches wissenschaftliches „Programmpaket“ zur numerischen Simulation, Optimierung und Unsicherheitsanalyse. Für die Entwicklung und damit Programmierung eines solchen Programmpaketes sind die Beziehungen der einzelnen Programmteile untereinander von entscheidender Bedeutung. Es lassen sich bisher folgende Abhängigkeiten formulieren und als Dreiecksbeziehung zusammenfassen.

- Die stochastischen Verfahren können eine nahezu unveränderte Auswertung der Modell-Simulation nutzen. Ein eigener Modell-Simulationscode für die Berechnung der stochastischen Versuche ist unnötig, solange über eine vereinheitliche Schnittstelle direkt auf den Programmteil der Modell-Simulation zugegriffen werden kann. Wichtig ist hier nur die Möglichkeit mit unterschiedlichen Modell-Konfigurationen starten zu können und über eine entsprechende Schnittstelle die Berechnungsresultate übernehmen zu können. Damit besteht der Programmteil für die stochastischen Verfahren, ausschließlich aus Algorithmen-Sammlungen zur systematischen Störung der Modell-Konfigurationen und zur Analyse und Weiterverarbeitung der Resultate aus der Modell-Simulation. Die Abhängigkeit mit der Modell-Simulation definiert sich hier also über die *Schnittstellen-Spezifikation*.

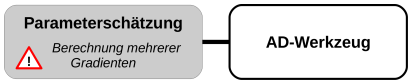
- Die Berechnung von Ableitungen bezüglich verschiedener Parameter hängt numerisch stark von der Funktionalität der Modell-Simulation ab. Sie wird im Folgenden auch als „originale Funktion“ bezeichnet. Nach einer Modifikation bzw. Erweiterung der originalen Funktion muss gewährleistet werden, dass auch die Ableitungsfunktion für die Berechnung der Gradienten neu angepasst wird. Darüber hinaus benötigt man eine einheitliche Schnittstelle zum Aufruf der Ableitungsberechnung und eine Sammlung mit Optimierungsalgorithmen, die die berechneten Jacobi-Informationen weiter verarbeiten kann. Eine Abhängigkeit drückt sich hier in einer möglichst „rei-

bungslosen Anpassung“ der Gradientenberechnung nach wichtigen Änderungen an der Funktionalität der Modell-Simulation aus.

- Sowohl die stochastischen Verfahren als auch die ableitungsbasierte Parameter-Inversion benötigt im Allgemeinen Zugriffsfunktionen auf einzelne physikalische Parameter. Das ist hilfreich, um sie einerseits stochastisch zu stören und andererseits um die für die Invertierung aktiven Parameter aktualisieren zu können. Idealerweise bilden diese Zugriffsfunktionen eine Abhängigkeit, da sie dann einmalig für beide Zwecke zur Verfügung stehen können. Darüber hinaus erzeugen Änderungen, die man für die stochastischen Verfahren an dem Programmteil der Modell-Simulation vornimmt, meistens auch Anpassungen an der Berechnungsfunktion für die Gradienten und umgekehrt. Daraus folgt insgesamt eine direkte Abhängigkeit zwischen den stochastischen Verfahren und der Parameterschätzung.

Im Weiteren wird diese Dreiecksbeziehung zusammengefasst als Programmpaket oder einfach Simulationssoftware bezeichnet, da die meisten anderen Abhängigkeiten sich oft auf alle drei Einzelkomponenten beziehen lassen.

3.3 Abhängigkeiten durch AD-Programmcode-Transformation



Das explizite Berechnen von Ableitungen kann mit numerischen Differenzenquotienten oder statt dessen auch mit Hilfe der Techniken des „Automatischen Differenzierens“ erfolgen. Bei letzterem werden Ableitungen berechnende Funktionen erzeugt, die effizient und numerisch exakt sind.

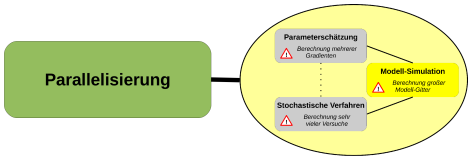
Die wichtigsten AD-Techniken beruhen auf einer Programmcode-Transformation (bzw. „Quellcode zu Quellcode“, Fachbegriff engl. „source to source“) oder dem Ersetzen (Überladen) der Rechenoperatoren, das im Englischen auch als „operator overloading“ bezeichnet wird. Hierbei ist zu beachten, dass für Fortran das „Ersetzen der Rechenoperatoren“ erst ab Fortran90/95 wirklich möglich ist. Das ursprüngliche Softwareprojekt (SHEMAT) basierte dagegen weitestgehend auf Fortran77 und bildete zum Teil die anfängliche Codebasis für das neue Suite-Projekt. Daraus ergab sich eine Notwendigkeit, AD über eine Programmcode-Transformation durchzuführen. Darüber hinaus stellt die Programmcode-Transformation auch gleichzeitig den allgemeineren Fall bezüglich auftretender Probleme in diesem Projekt dar.

Deswegen wird für die weiteren Betrachtungen davon ausgegangen, dass man eine Programmcode-Transformation mit einem geeigneten AD-Werkzeug (z.B. Adifor [7, 85], TAF [8, 86] oder Tapenade [9]) durchführt. Dabei wird aus dem Programmcode der Modell-Simulation ein neuer zusätzlicher AD-Programmcode erzeugt, der neben der originalen Funktion nun auch die Berechnung einer Ableitung für einen spezifizierten Parameter ermöglicht.

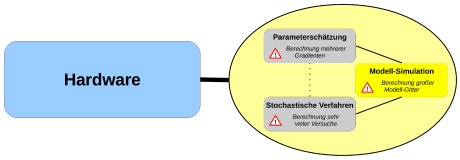
Die Verträglichkeit des AD-Werkzeugs mit dem reinen Sprachstandard der Programmiersprache des Modell-Simulationscodes bedingt dabei eigentlich eine eigene Abhängigkeit. Diese Abhängigkeit wird im Weiteren vernachlässigt, weil von einem für diese Programmiersprache geeignetem AD-Werkzeug ausgegangen wird. Andererseits muss der erzeugte AD-Programmcode über weitestgehend stabile Schnittstellen mit der Parameterschätzung verfügen. Ohne diese Forderung könnte eine etwaige Anpassung, die nach einer Änderung in der Modell-Simulation nötig wäre, auch eine Änderung an der Schnittstelle erfordern.

Merkmale: *Programmcode-Transformierbarkeit* durch das AD-Werkzeug, ausreichend flexible bzw. *stabile Schnittstellen*

3.4 Abhängigkeiten durch die Parallelisierung



Eine wichtige Ressource für die Anwendung des Programmpaketes ist die zu erwartende hohe bzw. erhöhte Rechenzeit. Diese soll durch eine gleichzeitige Kombination zweier Parallelisierungsstufen angegangen werden. Dazu bringt man im Allgemeinen bestimmte Programmbefehle und Informationen in den Simulationscode ein, welche nach Möglichkeit die Flexibilität in der algorithmischen Vielfalt (potentiellen Erweiterungen) nicht erschweren. Davon abgesehen müssen für eine erfolgreiche Kombination beide Parallelisierungsstufen unabhängig von einander sein, d.h. Änderungen an dem Parallelisierungscode für die eine Stufe dürfen den Code und die Funktionsweise der Anderen nicht beeinflussen.
Merkmal: *zwei von einander unabhängige Parallelisierungsstufen*

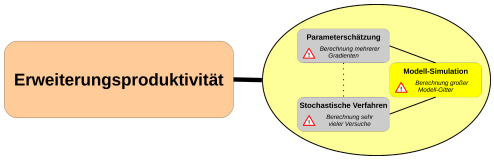


Im Sinne des Hochleistungsrechnens gibt es aus der Sicht der Simulationssoftware an das Rechnersystem zwei Anforderungen: genügend Rechenleistung und ausreichend Speicherplatz. Details zu den wichtigen Rechner-Architekturen findet sich in Kapitel 2.5. Eine Neuheit auf diesem Gebiet stellt eine so genannte „vSMP“-Architektur (siehe Kapitel 2.5.3) dar, die die besten Eigenschaften, wie Parallelisierbarkeit und günstiger Preis vereinigt. Für eine hohe Rechenleistung ist die „ccNUMA“-Eigenschaft (Kapitel 2.5.1), die ein solches vSMP-System mit vielen Anderen gemein hat, ein wichtiges Merkmal. Um diese Fähigkeit auszunutzen, muss die zugrunde liegende Software (und eingesetzte Softwaretechnik) bestimmte Anforderungen im Umgang mit den Datenstrukturen erfüllen. Im Einzelnen beeinflusst dabei die Rechner-Architektur die Fragestellung nach der Verwendung von globalen oder lokalen Daten, welche Daten von welchem Prozessorkern behandelt werden und auf welchem Teil des Rechnersystems die Daten platziert werden.
Merkmal: Ausnutzung der *„ccNUMA“-Architektur*

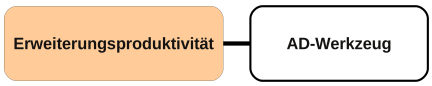
Einerseits kann man die Unterstützung der ccNUMA-Eigenschaften als Teil der Frage nach den Datenstrukturen und damit allgemein als Aspekt der Simulationssoftware sehen. Andererseits ist sie wegen der Abhängigkeit zu den Rechenkernen nicht von der Parallelisierung zu trennen. Lässt man dies hier einmal außen vor, verbleiben noch die Abhängigkeiten, die sich auf eine unterschiedliche Ressourcennutzung beziehen.

So ist es beispielsweise ein Unterschied, ob mehrere Rechenkerne (zur Leistungssteigerung) redundante Berechnungen ausführen oder ob diese Berechnung von nur einem Rechenkern erledigt wird. Im ersten Fall ergibt sich mehr Rechenarbeit und gegebenenfalls auch ein höherer Speicherbedarf. Bei dem Letzten kann die notwendig gewordene Datensynchronisation mit den anderen Kernen die Rechenleistung negativ beeinflussen. Es gilt also verschiedene Parallelisierungsstrategien hinsichtlich bestimmter Einschränkungen durch die Rechnersysteme abzuwägen.
Merkmale: sparsamer oder zumindest *flexibler Speicherplatzverbrauch*, leistungsfähige bzw. *effiziente Rechenarbeitsverteilung*

3.5 Übergreifende Abhängigkeiten durch die Erweiterungsproduktivität



Die schon erwähnte Erweiterungsproduktivität hat einen nahezu universellen Einfluss auf fast alle anderen Grundanforderungen, da die Forderung nach einer effizienten Entwicklungsarbeit im Wesentlichen eine sehr schnelle Handhabung in jeglicher Hinsicht notwendig macht. Die bei den Merkmalen geforderte Flexibilität nach algorithmischer Erweiterbarkeit und Vielfalt bedingt deshalb zusätzliche stabile Schnittstellen und eine entsprechend komplexe Funktionsmodularisierung auf Programmebene. Das ist eine bekannte Technik und wird an dieser Stelle nur der Vollständigkeit halber aufgeführt, da sie schon als gegeben betrachtet wird. Wichtiger sind statt dessen die Fernwirkungen auf die anderen Grundanforderungen und Beziehungen, die in den nachfolgenden Abschnitten erläutert werden.



Eine Maßnahme zur Verbesserung der Erweiterungsproduktivität ist eine Vereinfachung des Anpassungsprozesses nach einer Modifikation der originalen Funktion (Erweiterung an der Modell-Simulation). Dies kann durch eine Reduzierung der Anpassung auf nur nötige Programmabschnitte erreicht werden. Man lagert dazu alle zu übergehenden Programmabschnitte in einzelne Routinen aus. Im Detail handelt es sich dabei um fast alle Ein-/Ausgaben und solche Funktionen und Programmabschnitte, die nicht von den aktiven Parametern abhängen. Daraus folgt eine tiefer gehende zusätzliche Funktionsmodularisierung, die sich an der Abhängigkeit von den aktiven Ableitungsparametern orientiert. Erst damit ist man in der Lage, dem AD-Werkzeug nur noch diese unbedingt nötigen Programmabschnitte vorzulegen. Der Vorteil liegt in einer schnelleren und reibungsloseren Anpassung der Ableitungsfunktion.
Merkmal: erweiterte *ableitungs-spezifische Modularisierung*

Normalerweise erfordert die Parallelisierung von Programmen zusätzlichen Code. Jede Änderung oder Erweiterung an der allgemeinen Funktionalität führt deshalb potentiell auch zu einem Anpassungsaufwand bezüglich der Parallelisierung. Um diese zeitraubende und anspruchsvolle Problematik zu minimieren, wird oft mit einer geeigneten Abstraktion oder Modularisierung gearbeitet. Dadurch sollen die Parallelisierungsabschnitte im Allgemeinen gebündelt, übersichtlich, klein und damit handhabbarer gehalten werden. In der vorliegenden Arbeit wird eine darüber hinaus gehende Softwaretechnik beschrieben, die zusätzlich gezielt bestimmte parallelisierungs-spezifische Automatismen so verstärkt, dass Einsparungen erzielt werden können. Für OpenMP beziehen sich die Automatismen auf den so genannten „Datensichtbarkeitsbereich“ (mehr in Kapitel 4.2). Zusammen mit der Modularisierung kann der Anpassungsaufwand drastisch verkürzt und damit die Erweiterungsproduktivität verbessert werden.
Merkmale: erweiterte *parallelisierungs-spezifische Modularisierung*, Nutzung der *Datensichtbarkeitsbereich-Automatismen*

Die nachfolgend beschriebene (letzte) Abhängigkeit bezieht sich indirekt auch auf die Erweiterungsproduktivität, weshalb sie hier im selben Abschnitt aufgeführt ist.

Die von den AD-Werkzeugen beherrschten Programmiersprachen umfassen oft nur wenige Spracherweiterungen. So werden die zur Parallelisierung nötigen Spracherweiterungen meist nicht flexibel genug oder nur teilweise unterstützt. Das

kann zu falschen Ableitungsfunktionen oder zur Zerstörung der ursprünglichen Parallelisierung führen. Deshalb ist bei der Anwendung von Werkzeugen mit Codetransformation oft ein erheblicher Vorbereitungs- oder Nachbearbeitungsaufwand für jede Aktualisierung nötig.

Möchte man diesen ständigen Zusatzaufwand vermeiden, muss man entweder ein geeignetes Werkzeug finden oder das gegebene anpassen. Eine gänzlich andere Strategie wird in dieser Arbeit vorgestellt. Sie versteckt gewissermaßen die Parallelisierung vor dem AD-Werkzeug. Das erfordert ein paar *einmalige* Vorbereitungen am Vorwärtssimulationscode und gegebenenfalls zusätzlich noch einige wenige leicht *automatisierbare* Schritte. Letztendlich handelt es sich hierbei um eine Erweiterung des schon in der letzten Abhängigkeit beschrieben Zusammenspiels von Modularisierung und Automatismennutzung.
Merkmale: erweiterte *parallelisierungs-spezifische Modularisierung*, Nutzung der *Datensichtbarkeitsbereich-Automatismen*

Algorithmische Flexibilität und stabile Schnittstellen

Die im Rahmen der Weiterentwicklung der Simulationssoftware geforderte Flexibilität bezüglich

- unterschiedlicher physikalischer Eigenschaften,
- verschiedener stochastischer Verfahren oder
- verschiedener ableitungsbasierten Optimierungsverfahren

wird im Folgenden einfach als Forderung nach „Algorithmen-Modularisierung“ zusammengefasst. Diese Algorithmen-Modularisierung wird gemeinsam mit allen nötigen stabilen Schnittstellen-Definitionen als ausreichend gegeben betrachtet. Die dafür verwendete Softwaretechnik oder deren Entwicklung ist nicht Bestandteil dieser Arbeit, da die Vorgehensweisen als hinreichend untersucht und bekannt gelten. Wichtig für alle weitergehenden Betrachtungen sind die daraus resultierenden Algorithmen-Sammlungen, die jeweils als eine Menge von unterschiedlichen und gekapselten Funktionsbausteinen angenommen wird. Das führt zu der Notwendigkeit, alle Funktionsbausteine möglichst unabhängig voneinander und ohne Nebeneffekte bezüglich der Grundanforderungen implementieren zu können. Die nachfolgend beschriebenen Softwaretechniken sollen diese Unabhängigkeit weitestgehend gewährleisten.

4 Teilkonzept - Softwaretechnik: „OpenMP-hiding“

Bei der hier vorgestellten „OpenMP-hiding“-Technik handelt es sich im Wesentlichen um eine Ausblendung (engl. „hiding“, im Sinne von verstecken) der Parallelisierungsanweisungen für alle notwendigen Programmteile. Die Erweiterungsproduktivität hat dabei Einfluss auf die Auswahl der notwendigen Programmteile. Eine Übersicht aller damit lösbaren Abhängigkeiten und Anforderungen zeigt die Abbildung 7. Weiter unten werden dann die

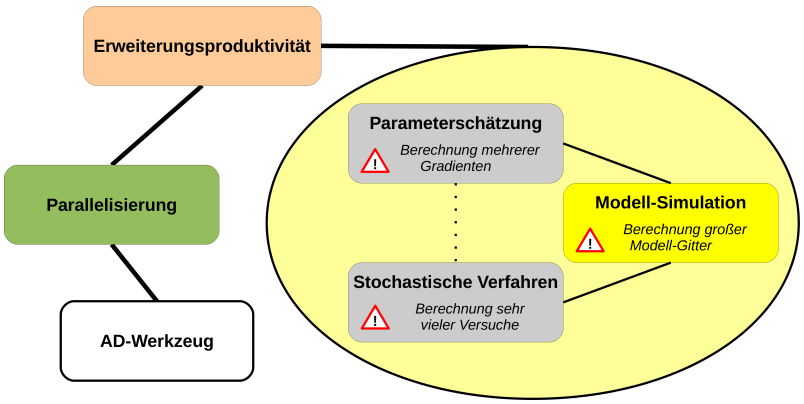


Abbildung 7: Grundanforderungen (farbige Objektkästen) und Abhängigkeiten (schwarze Verbindungslinien) die durch den Einsatz der „OpenMP-hiding“-Softwaretechnik gelöst werden.

einzelnen Abhängigkeiten und Zusammenhänge näher erläutert. Zuvor muss aber für ein besseres Verständnis die Aufteilung der Parallelisierung genauer beschrieben werden.

Geht man von dem Programmcode der originalen Modell-Simulation aus und generiert den zugehörigen Ableitungscode durch ein Transformations-Werkzeug, ergeben sich für das mehrstufige Parallelisierungskonzept insgesamt drei Forderungen. Diese Forderungen gliedern sich in:

1. Erhaltung der originalen Parallelisierung für die verbliebenen originalen Programmanweisungen im generierten Ableitungscode,

2. Erweiterung der originalen Parallelisierung auf die neuen Programmanweisungen zur Ableitungsberechnung und

3. Erweiterung um eine zusätzliche Parallelisierung für die gleichzeitige mehrfache Ableitungsberechnung.

Die in diesem Kapitel vorgestellte Softwaretechnik beschäftigt sich mit einer Lösung zur Erfüllung der ersten beiden Forderungen. Gleichzeitig darf die zusätzliche Parallelisierungsstufe der 3. Forderung (Lösung in Kapitel 5) durch den Einsatz dieser Technik nicht beeinträchtigt werden.

Im Allgemeinen geht man von einer gemeinsamen Parallelisierung für die beiden Forderungen 1. und 2. aus, allerdings wird in diesem und den nachfolgenden Unterkapiteln einzeln auf den jeweiligen Parallelisierungsteil eingegangen. Diese vorerst getrennte Betrachtung ist notwendig, da es sich um aufeinander aufbauende Einzelmaßnahmen und Richtlinien mit unterschiedlichen Zielsetzungen handelt.

Wegen des hohen Stellenwerts der Erweiterungsproduktivität ist es nötig, alle anfallenden Vor- und Nachbearbeitungs-Maßnahmen (für das Zusammenspiel von Parallelisierung und AD-Werkzeug) entweder automatisch oder durch einmalige manuelle Umstrukturierungen minimal zu halten. Die „OpenMP-hiding“-Technik erfordert dazu

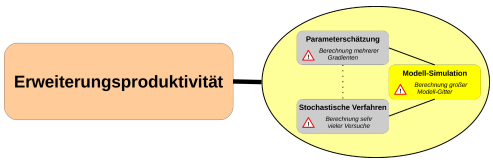
- die Einhaltung bestimmter struktureller Richtlinien während der Softwareentwicklung und erlaubt so

- einen automatisierten Übergang von dem originalen Programmcode zum AD-Werkzeug und zurück zum transformierten Ableitungscode.

Dies ergibt eine schnelle Aktualisierung des Ableitungscodes nach Änderungen am originalen Programmcode, sowie eine umfassende und effiziente Erweiterung der Parallelisierung des Ableitungscodes.

Wie schon zuvor erwähnt wurde, bezieht sich die hier zu unterstützende Parallelisierung auf den OpenMP-Standard bis Version 2.5, wobei die hier vorgestellte Softwaretechnik den Umgang mit den wichtigsten OpenMP-Direktiven und Klauseln systematisch beschreibt. Die auch als OpenMP-Konstrukte bezeichneten Direktiven können im Programmcode separat auftreten und definieren die unterschiedlichen Berechnungsblöcke, regeln die Art und Weise der Arbeitsverteilung oder spezifizieren verschiedene Arten der Synchronisation. Eine Synchronisation kann beispielsweise das Warten auf andere Unterprozesse, eine (nicht parallele) nacheinander erfolgende Abarbeitung eines Programmabschnitts durch einzeln laufende Unterprozesse oder die Herstellung einer einheitlichen konsistenten Datensicht (durch Austausch über den Hauptspeicher) der Unterprozesse bedeuten. Dazugehörige Nebenbestimmungen in Form von Klauseln stehen immer unmittelbar nach bestimmten Direktiven. Die Mehrheit der Klauseln spezifiziert unter anderem Sichtbarkeitsbereichs-Attribute von Datenobjekten oder verschiedene Synchronisationen (auch globale Rechenoperationen). Eine Übersicht mit entsprechenden Kurzbeschreibungen findet man im Anhang A in Tabelle 2 (Direktiven) und Tabelle 3 (Klauseln). Darüber hinaus gibt es im Standard 2.5 noch weitere Klauseln, wie beispielsweise *schedule*, *nowait* oder *num_threads*, die aufgrund vernachlässigbarer Nebeneffekte nicht explizit in dieser Arbeit besprochen werden.

Im Einzelnen handelt es sich bei dem hier vorgestellten Lösungskonzept um vier aufeinander zugeschnittene Maßnahmen. Mit der ersten wird so viel wie möglich in einen Automatismus (für den „Datensichtbarkeitsbereich“, engl. „data scoping“) hineingesteckt, wodurch ein erster Teil von OpenMP-Klauseln überflüssig wird und entfallen kann. Verstärkt wird dieser Effekt durch ein zusätzliches Verschieben der verbliebenen OpenMP-Direktiven und Klauseln in andere Programmregionen oder neue Zwischenroutinen („parallelisierungs-spezifische Modularisierung“). Die dritte Maßnahme bezieht sich auf strukturelle Änderungen, mit deren Hilfe zusätzliche problematische Übergänge geregelt werden können (Kapitel 4.4.1). Als letztes müssen für eine korrekte Ableitungsberechnung einige wenige OpenMP-Konstrukte bzw. Klauseln umgangen werden, indem man die zugrunde liegenden Rechenoperationen explizit ausformuliert (Kapitel 4.5). Zusammen sorgt das in den für die Algorithmen-Modularisierung wichtigen Programmteilen für ein Minimum an Parallelisierungsspezifikationen.



Damit ergibt sich direkt eine Verbesserung der Erweiterungsproduktivität bezüglich der Simulationssoftware an sich. Ausgangspunkt ist die geforderte Flexibilität durch „Algorithmen-Modularisierung“ mit den flexiblen Schnittstellen. Wenn die notwendig flexiblen Programmabschnitte von expliziten Parallelisierungs-Anweisungen nahezu frei gehalten werden, dann benötigt der Großteil aller Erweiterungen entsprechend wenig Anpassungen bezüglich der Parallelisierung und ist somit wesentlich einfacher und schneller zu realisieren.

Weiterhin ist die Parallelisierung selbst entsprechend den Einschränkungen der Richtlinien nach vereinfacht und damit insgesamt handhabbarer

geworden. Damit kann auch die Erweiterungsproduktivität in Hinblick auf Anpassungen der Parallelisierungs-Strategie für zukünftige Rechner-Architekturen gewährleistet werden. Das entspricht nicht unbedingt einer kompletten Lösung bzw. einer Befreiung von der manuellen Spezifikation der Parallelisierung, kann aber auch eine Verbesserung im Sinne der Erweiterungsproduktivität sein.

Alle anfallenden Vor- und Nachbearbeitungsschritte, die nicht mit einer einmaligen Codeumstellung (der Vorwärtssimulation) gelöst werden können, sind so vereinfacht, dass sie mit kleinen Werkzeugen leicht automatisiert werden können. Außerdem gilt: Über den Erhalt der Parallelisierung für die originalen Modell-Simulation hinaus soll auf dem generierten Ableitungscode auch sichergestellt sein, dass die Ableitungsberechnung selbst von einer Parallelisierung profitiert. Dafür ist es zusätzlich nötig, einige OpenMP-Direktiven und Klauseln direkt durch ihre explizite Ausformulierung zu ersetzten. Das betrifft im Wesentlichen bestimmte Rechenoperationen, die dort implizit erzeugt werden. Meist ist dies mit nur einem geringen und einmaligen Zusatzaufwand realisierbar.

4.1 Flexibilität für Datensichtbarkeitsbereichs-Attribute

Grundlegend möchte man bei der Verwendung einer Spracherweiterung (zur Parallelisierung) so wenig wie möglich Einschränkungen unterliegen. Für OpenMP folgt daraus, dass höchste Flexibilität beim Umgang mit den Daten notwendig ist. Insbesondere betrifft dies die Datensichtbarkeitsbereichs-Attribute aller Variablen. Der Datensichtbarkeitsbereich wurde bereits in Kapitel 2.6 eingeführt und definiert die jeweiligen Zugriffs-Rechte der Unterprozesse auf die Datenobjekte.

Im einfachsten Fall gibt es nur eine parallele Region (kurz **PR**) und keine der Variablen wechselt sein Datensichtbarkeitsbereichs-Attribut. Dabei umfasst die parallele Region einen bestimmten Programmbereich, für den eine (neue) Gruppe von Unterprozessen („OpenMP-Threads“) gestartet wird. Diese Gruppe kann dann innerhalb des eingeschlossenen Bereichs die Rechenarbeit aufteilen oder redundant ausführen. Für die Kurzdarstellung im Weiteren wird eine einzelne parallele Region mit runden Klammern und dem Datensichtbarkeitsbereichs-Attribut *private* oder *shared* versehen (hier werden die originalen Attributs-Namen aus dem Englischen verwendet, um ein erleichtertes Verständnis bei der Verwendung innerhalb von Programmbeispielen zu erreichen). Das ergibt dann beispielsweise „(*private* in **PR**)“.

Im Allgemeinen hat man dagegen mehrere parallele Regionen mit vielen temporären Änderungen der Attribute innerhalb und zwischen den einzelnen Regionen. Solche Änderungen sind normalerweise nur an bestimmten Bereichsübergängen möglich. Dabei ergeben sich häufig Probleme an den Übergängen (Kurzdarstellung mit $\rightarrow$) in eine parallele Region oder in ein Arbeitsverteilungs-Konstrukt (engl. „work-sharing construct“).

Ein „Arbeitsverteilungs-Konstrukt“ (kurz **AK**) definiert innerhalb einer parallelen Region einen festgelegten Teilbereich, in dem unterschiedliche Berechnungsblöcke systematisch auf unterschiedliche Unterprozesse aufgeteilt werden. Sie werden im Folgenden auch als (OpenMP spezifische) Anweisungen zur Arbeitsverteilung bezeichnet. Ein einfaches Beispiel kann eine beliebige Programmschleife von 1 bis 10 sein. Der dazu gehörende Schleifenkörper entspricht dann den zehn verschiedenen Berechnungsblöcken (einer für jeden Durchlauf). Eingegrenzt wird so ein Arbeitsverteilungs-Konstrukt (für eine Programmschleife) in der Kurzdarstellung mit eckigen Klammern analog zur parallelen Region, z.B. „[*private* in **AK**]“.

Ein Arbeitsverteilungs-Konstrukt befindet sich immer innerhalb einer parallelen Region. Eine parallele Region dagegen kann alleine (außerhalb) oder innerhalb einer anderen parallelen Region **PR** stehen. Deshalb wird für eine bessere Übersicht noch das Schlüsselwort „äußere“ oder „innere“ mit eingefügt. Das erstere kennzeichnet einen äußeren großen

parallelen Programmbereich - z.B. „(*private* in äußerer **PR**)“, das Zweite einen inneren verstärkt parallelisierten Teilbereich - z.B. „(*private* in innerer **PR**)“. Für zwei geschachtelte parallele Regionen erhält man so

„(*private* in äußerer **PR** → (*private* in innerer **PR**))“.

Zur Vereinfachung betrachtet man im Folgenden nur den Übergang bzw. die Definition der Datensichtbarkeitsbereichs-Attribute einer einzelnen Variable. Diese Attribute regeln den Zugriff der einzelnen Unterprozesse auf die Daten, hierfür muss für alle Datenvariablen eindeutig festgelegt sein, ob die OpenMP-Unterprozesse gemeinsam darauf zugreifen können (*shared*), oder jeder Unterprozess auf seiner eigenen Kopie (mit *private*-Attribut) arbeiten soll. Daraus ergeben sich für eine einzelne Variable die folgen Übergangs-Szenarien in Tabelle 1.

$\ddot{U}_{pPsP}$	(*private* in äußerer **PR** → (*shared* in innerer **PR**))
$\ddot{U}_{pPpP}$	(*private* in äußerer **PR** → (*private* in innerer **PR**))
$\ddot{U}_{sPsP}$	(*shared* in äußerer **PR** → (*shared* in innerer **PR**))
$\ddot{U}_{sPpP}$	(*shared* in äußerer **PR** → (*private* in innerer **PR**))
$\ddot{U}_{pPsA}$	(*private* in äußerer **PR** → [*shared* in innerem **AK**]) Nicht OpenMP konform!
$\ddot{U}_{pPpA}$	(*private* in äußerer **PR** → [*private* in innerem **AK**])
$\ddot{U}_{sPsA}$	(*shared* in äußerer **PR** → [*shared* in innerem **AK**])
$\ddot{U}_{sPpA}$	(*shared* in äußerer **PR** → [*private* in innerem **AK**])
$\ddot{U}_{pspP}$	(*private* in **PR**) → (*shared* in **PR**) → (*private* in **PR**) → ...

Tabelle 1: Übergangs-Szenarien der Datensichtbarkeitsbereichs-Attribute einer Variable zwischen parallelen Regionen (**PR**) und Arbeitsverteilungs-Konstrukten (**AK**).

Die ersten vier Varianten ($\ddot{U}_{pPsP}$, $\ddot{U}_{pPpP}$, $\ddot{U}_{sPsP}$ und $\ddot{U}_{sPpP}$) behandeln den allgemeinen Übergang, wenn innerhalb einer parallelen Region eine weitere geöffnet wird (geschachtelte Parallelisierung). Dabei wird eine neue Gruppe von Unterprozessen erzeugt und je nachdem von der Variable eine neue Kopie für jeden Unterprozess angelegt oder der gemeinsame Zugriff der gesamten Gruppe erlaubt.

Bei den nächsten vier Szenarien ($\ddot{U}_{pPsA}$, $\ddot{U}_{pPpA}$, $\ddot{U}_{sPsA}$ und $\ddot{U}_{sPpA}$) handelt es sich um den Übergang der Variablen in einen Bereich, der durch eine OpenMP-Arbeitsverteilungs-Direktive parallelisiert ist. Man eröffnet hier keine neue Gruppe von Unterprozessen, kann aber trotzdem entscheiden, wie der Variablenzugriff für die derzeitige Unterprozess-Gruppe innerhalb dieses neuen Bereiches sein soll. Hierbei ist speziell der Übergang $\ddot{U}_{pPsA}$ von *private* zu *shared* nach dem OpenMP-Standard unzulässig, da es als zu aufwendig angesehen wird, im Nachhinein mehrere Kopien konsistent zu einem einzelnen gemeinsam nutzbaren Datenobjekt zusammen zu fassen.

Das letzte Szenario $\ddot{U}_{pspP}$ zeigt einen beliebigen nicht-geschachtelten Übergang der Variablen von einer abgeschlossenen parallelen Region in die Nächste (auf gleicher Ebene), wobei sie einmal *private* und ein anderes Mal *shared* sein soll (unabhängig von der Reihenfolge).

Es ist vorteilhaft, wenn man sich auf bestimmte Übergangs-Szenarien einschränken kann. Für solchen Programmcode verhält sich eine OpenMP-Parallelisierung stabiler bzw. robuster gegenüber einer möglichen Programmcode-Transformation. Erfahrungen durch die Implementation an dem dieser Arbeit zugrunde liegenden Simulationswerkzeug bestätigten dies. In Kapitel 4.4.4 wird dazu ein detailliertes Anschauungsbeispiel ausgeführt, bei dem sich ein so genannter „kritischer Datenwettlauf“ (engl. „Data-Race“) und falsche Initialisierungen in dem generierten Ableitungscode vermeiden lassen.

Es ist Bestandteil der hier vorgestellten Strategie, eine bestimmte Teilmenge der Übergangs-Szenarien aus den erwähnten Gründen zu bevorzugen (Vorzugsvarianten). Für die restlichen Möglichkeiten muss man dann lediglich sicherstellen (will man sie nicht verbieten), dass sie effizient und möglichst ohne nachteilige Nebeneffekte realisierbar sind (ausgenommen $\ddot{U}_{pPsA}$).

4.1.1 Vorzugsvarianten

Die eigentliche Bevorzugung entsteht durch eine zielgerichtete Ausnutzung der Voreinstellungen (bezüglich der Datensichtbarkeitsbereichs-Attribute) die im OpenMP-Standard definiert sind und hier zu Richtlinien in Form von Vorzugsvarianten führt. Im Detail erklären sich die Vorzugsvarianten (gekennzeichnet durch $\oplus$) und die zu umgehenden Varianten (gekennzeichnet durch $\ominus$) für die acht gültigen Übergangsszenarien (aus Tabelle 1) wie folgt.

- $\oplus$ $\ddot{U}_{pPsP}$, $\ddot{U}_{sPsP}$: Es erwies sich als besonders hilfreich, den Übergang in eine neue parallele Region nach $\ddot{U}_{pPsP}$ (Wechsel von *private* $\rightarrow$ *shared*) und $\ddot{U}_{sPsP}$ (Durchreichen des *shared*-Attributes) zu bevorzugen. Denn bei diesen Übergängen müssen keine zusätzlichen lokalen Kopien (*private*) der bisherigen Datenobjekte angelegt werden (speichereffizient). Hierbei wird praktisch der Zugriff auf die Variablenobjekte beibehalten und führt damit auch zu weniger Problemen bei einer AD-Programmcode-Transformation.

- $\ominus$ $\ddot{U}_{pPpP}$, $\ddot{U}_{sPpP}$: Andererseits lassen sich sowohl $\ddot{U}_{pPpP}$ als auch $\ddot{U}_{sPpP}$, also das zusätzliche Anlegen einer lokalen Kopie (*private*) für die neue Region, leicht durch das explizite Anlegen einer neuen lokalen Variable nachholen (ohne OpenMP-Direktive). Da beide eine manuelle (wenn auch einmalige) Umstellung benötigen, sollen sie hier nicht bevorzugt werden.

- $\oplus$ $\ddot{U}_{pPpA}$, $\ddot{U}_{sPsA}$: Für den Übergang in einen Arbeitsverteilungsbereich (innerhalb eines Arbeitsverteilungs-Konstruktes) ändert sich der zu bevorzugenden Wechsel, um auch hier den Zugriff auf eine Variable beizubehalten. Variablen sollten innerhalb des Bereichs des Arbeitsverteilungs-Konstrukts so genutzt werden, wie in der umgebenden parallelen Region, die bevorzugten Übergänge sind jetzt also $\ddot{U}_{pPpA}$ und $\ddot{U}_{sPsA}$.

- $\ominus$ $\ddot{U}_{sPpA}$: Analog dazu lässt sich die letzte gültige „Arbeitsverteilungs-Variante“ $\ddot{U}_{sPpA}$ ebenfalls wie $\ddot{U}_{pPpP}$ und $\ddot{U}_{sPpP}$ mit einer neu anzulegenden lokalen Variable leicht realisieren, wird aber wegen der nötigen Codeumstellung ebenfalls nicht bevorzugt.

- $\ominus$ $\ddot{U}_{pspP}$: Die letzte Variante $\ddot{U}_{pspP}$ stellt auch einen nicht zu bevorzugenden Fall dar. Denn man könnte mehrere parallele Regionen auf einer Ebene auch wie viele Arbeitsverteilungs-Konstrukte innerhalb einer parallelen Region auffassen. Der Vergleich stimmt nicht ganz, kann aber so interpretiert werden, dass kontinuierliche Übergänge, also nur „geteilt“ oder nur „lokal“, zu bevorzugen sind. Entsprechend sind alle wechselhaften Übergänge von „geteilt“ zu „lokal“ (oder anders herum) problematisch und müssen wieder durch das Anlegen neuer lokaler Variablen (für die Bereiche mit *private*-Attribut) realisiert werden.

Durch die Nutzung dieses Automatismus können dann bestimmte OpenMP-Klauseln weggelassen werden. Alles in Allem widmen sich die nachfolgenden zwei Unterkapitel über den „Datensichtbarkeitsbereich-Automatismus“ und der parallelisierungs-spezifischen Modularisierung dem Einrichten der Vorzugsvarianten ($\ddot{U}_{pPsP}$, $\ddot{U}_{sPsP}$, $\ddot{U}_{pPpA}$ und $\ddot{U}_{sPsA}$). Diese ersten Maßnahmen beschreiben die Richtlinien am Definitionsbereich der Datenstrukturen. Das resultierende Entfernen der ersten OpenMP-Klauseln gestaltet sich hier vergleichsweise unproblematisch, da noch keine Programmcode-Transformation (durch ein AD-Werkzeug) betrachtet wird.

Danach werden die von den Vorzugsvarianten abweichenden Möglichkeiten ($\ddot{U}_{pPpP}$, $\ddot{U}_{sPpP}$, $\ddot{U}_{sPpA}$ und $\ddot{U}_{pspP}$) im Kapitel 4.4.1 betrachtet und deren Handhabung erläutert. Hier muss im Gegensatz zu vorher schon auf eine mögliche Weiterverarbeitung in einem AD-Werkzeug eingegangen werden. Allerdings bezieht sich dieses und die ersten zwei Unterkapitel immer noch nur auf die korrekte Berechnung der originalen Modell-Simulation und haben dementsprechend noch wenig mit der Berechnung von Ableitungen zu tun. Dabei erleichtern sie, wie man später sieht, die Parallelisierung der Ableitungsberechnung enorm und machen insgesamt die Weiterverarbeitung in einem AD-Werkzeug einfacher.

Unabhängig von der Erhaltung der Flexibilität bei den Datensichtbarkeitsbereichs-Attributen, gibt es noch zwei Teilprobleme, die auch innerhalb dieser Softwaretechnik gelöst werden müssen. Bei dem Ersten handelt es sich um Verschiebungen von Programmanweisungen, wie sie bei bestimmten Optimierungen auftreten können. Das Kapitel 4.4.1 zeigt unter anderem eine gewisse Verträglichkeit damit auf, geht aber für alles Weitere von nur lokalen Verschiebungen aus, die unproblematisch sind.

Das zweite Teilproblem entsteht bei zusätzlichen Rechenoperationen und Synchronisationen auf einer eingeschränkten Menge von Variablen, die durch bestimmte OpenMP-Direktiven impliziert werden. Abgesehen von der Problematik mit den Verschiebungen, betrifft diese Problemklasse im Wesentlichen nur die parallelisierte Berechnung der Ableitungsanweisungen. Hiermit befasst sich ausführlich das Kapitel 4.5 und beschreibt, wie dieses Teilproblem umgangen werden kann.

4.2 Datensichtbarkeitsbereich-Automatismus

In diesem Kapitel liegt das Hauptaugenmerk auf den Datensichtbarkeitsbereichs-Attributen, da sie die Grundlage für alle nachfolgenden Betrachtungen bildet. Dazu muss als erstes eine Zerlegung des Begriffs „OpenMP-Parallelisierung“ erfolgen, der im Wesentlichen aus: (1) Anweisungen für die „Arbeitsverteilung“ auf die einzelnen Unterprozesse, (2) den „OpenMP abhängigen Rechenoperationen“, (3) den Synchronisationen und (4) der Spezifikation der Datensichtbarkeitsbereichs-Attribute, besteht.

Mit Anweisungen für die „Arbeitsverteilung“ (1) sind hier ausschließlich Direktiven gemeint, die an sich keine Rechenoperationen (2) oder Synchronisationen (3) beinhalten und hauptsächlich die Verteilung der Rechenarbeit auf die Unterprozesse regeln. Hierbei werden implizite Synchronisationen, wie das Warten auf alle Unterprozesse am Ende eines OpenMP-*do*-Konstrukts, ignoriert. Bei den Datensichtbarkeitsbereichs-Attributen (4) ist zu beachten, dass Voreinstellungen implizit angenommen werden oder explizit definiert sind, die im Folgenden auch als „Datensichtbarkeitsbereich-Automatismus“ bezeichnet werden.

Damit sind nicht die „autoscoping“-Fähigkeiten einiger Compiler gemeint, die während der Codeübersetzung automatisch solche Datenattribute durch aufwändige Analysetechniken hinzufügen können (siehe dazu [87]).

Hier geht es in erster Linie um die durch den OpenMP-Standard definierte Voreinstellungen. Es nimmt dem Entwickler allerdings nicht die Entscheidung ab, ob eine Variable für „geteilten“ oder „lokalen“ Zugriff ausgelegt sein muss. Vielmehr gibt es eine Reihe von impliziten Annahmen. Im Folgenden werden drei Gruppen unterschieden. Die erste bezeichnet mit „global übergeben“ alle Variablen die über ihre globale Deklaration ihre Werte an Unterroutinen weitergeben. Andere Variablen heißen „intern deklariert“, wenn diese Variablen nicht global und ausschließlich innerhalb einer parallelen Region deklariert und angelegt werden. Als letzte Gruppe werden alle übrigen Variablen mit „lokal übergeben“ zusammengefasst.

Bei den lokal übergebenen Variablen kann es sich auch um eigentlich global deklarierte Datenobjekte handeln. Das kann der Fall sein, wenn die globale Variable als Funktionsparameter von außerhalb einer parallelen Region in den inneren Bereich übergeben werden. Alle Variablen, die schon vor dem Übergang in die parallele Region deklariert waren und übergeben werden, können der Standard-Voreinstellung entsprechend, entwe-

der als „geteilt“ (*shared*) oder „lokal“ (*private*) angenommen werden. Ein Beispiel hierfür ist die Variable *dauto* im ersten Programmauszug Abbildung 8. In dieser Abbildung gibt

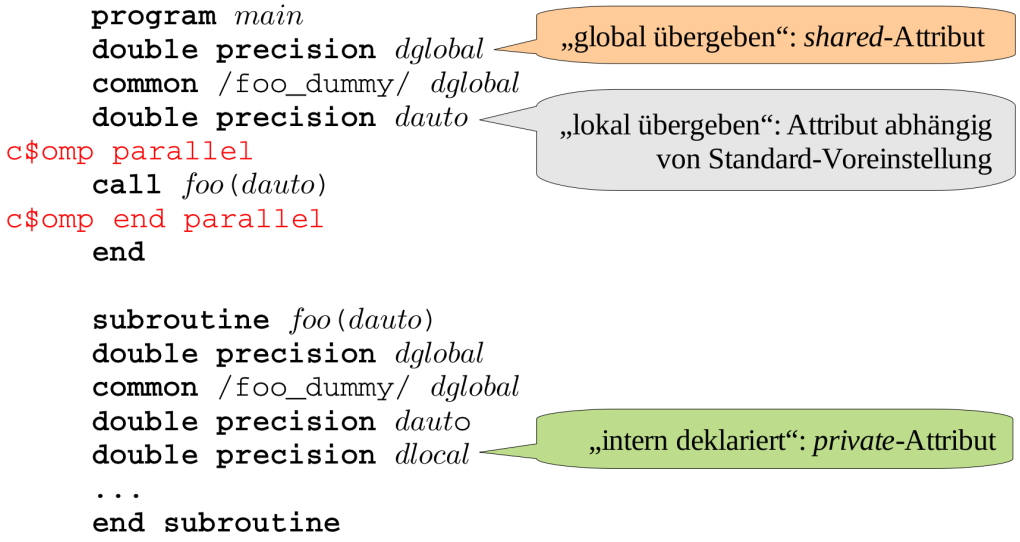


Abbildung 8: Beispielcode für drei Variablen mit unterschiedlichem Datensichtbarkeitsbereich.

es ganz allgemein einen *main*-Programmteil mit einer global deklarierten Variable *dglobal* und einer lokalen Variable *dauto*. Außerdem gibt es dort eine parallele Region, in der eine Unterroutine *foo* aufgerufen wird. In dieser wiederum wird die im *main*-Teil deklarierte Variable *dauto* über die Parameterliste von *foo* durchgereicht und durch eine zusätzliche lokale Variable *dlocal* innerhalb der Unterfunktion *foo* ergänzt. Weiterhin enthält *foo* auch die globale Deklaration von *dglobal*. Diese global übergebene Variable wird laut OpenMP-Standard erst einmal für einen „geteilten“ Zugriff (*shared*-Attribut) angelegt. Für alle innerhalb einer parallelen Region deklarierten Variablen (z.B. *dlocal*) ist dagegen das *private*-Attribut vorgesehen. Von dieser Standardannahme abweichend können die Datensichtbarkeitsbereichs-Attribute immer auch explizit umdefiniert werden (vergleiche dazu Kapitel 2.6). Da die Variable *dauto* als Funktionsparameter direkt und damit lokal über die Parallelisierungsgrenze übergeben wird, ist sie hier ein Beispiel für die voreingestellten Attributseigenschaften (entsprechend der Standard-Voreinstellung).

Allerdings muss beachtet werden, dass nur diese lokal übergebenen Variablen für unterschiedliche nacheinander stehende parallele Regionen auch unterschiedliche Datensichtbarkeitsbereich-Attribute aufweisen dürfen. Dagegen haben die global übergebenen Variablen einheitlich für das gesamte Programm ausschließlich einen nur geteilten Zugriff (*shared*-Attribut) oder sind nur lokal (durch Spezifikation mit *threadprivate*). Hier ist kein Wechsel zwischen unterschiedlichen parallelen Regionen möglich. Für die intern deklarierten Variablen (wie *dlocal*) darf das Datenattribut nur gegenüber einer in dieser Region enthaltenen weiteren parallelen Region (geschachtelte Parallelisierung) wechseln, da sie bezüglich dieser inneren Parallelisierung als außerhalb angelegt gilt. In so einem Fall wirkt die Variable gegenüber der inneren Region wie „lokal übergeben“.

Das zweite Programmbeispiel in Abbildung 9 zeigt einen solchen inneren Übergang. Dort ist die Variable *dlocal* innerhalb von *foo* erst einmal lokal (grüner Bereich), da sie innerhalb der äußeren parallelen Region (oberer roter Bereich) deklariert und angelegt wird. Innerhalb der zweiten inneren Region (unterer roter Bereich) hat man dagegen die freie Auswahl. Hier kann *dlocal* sowohl für einen „geteilten“ Zugriff vorgesehen oder auch „lokal“ angelegt sein (grauer Bereich).

Für alle weiteren Erklärungen wird nun ein einfaches Programmbeispiel (Abbildung 10) betrachtet. Es enthält wie bisher einen *main*-Teil, in dem eine Unterfunktion *foo* aufgerufen wird. Die Routine *foo* enthält eine einfache Schleife für die Berechnung der Variablen y und eine angedeutete Weiterverarbeitung außerhalb dieser Schleife („use vector [y]“).

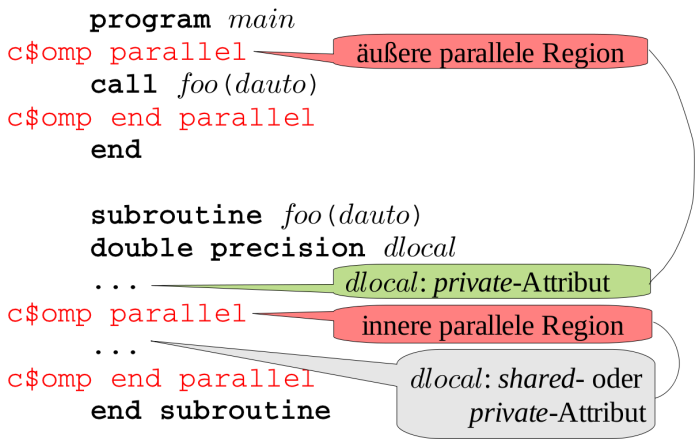


Abbildung 9: Geschachtelte Parallelisierung mit Datensichtbarkeitsbereich-Wechselmöglichkeit für die Variable *dlocal*.

```
! main routine
...
double precision x(100)
common /dummy/ x
call foo(100)
...

! original function: [y] = [x]*[x]
subroutine foo(N)
integer i, N
double precision x(100), y(100), tmp
common /dummy/ x
do i = 1, N
   tmp = x(i)
   y(i) = tmp * tmp
end do
... ! use vector [y]
end subroutine
```

Abbildung 10: Beispiel-Routine ohne Parallelisierung, bestehend aus dem *main*-Programmteil mit dem eigentlichen Funktionsaufruf der Unterfunktion *foo*.

Ausgehend von diesem Grundbeispiel soll nun die Berechnung von y in der Schleife auf mehrere OpenMP-Unterprozesse aufgeteilt, also parallelisiert werden.

Dazu wird als erstes und unmittelbar über der Rechenschleife eine parallele Region eröffnet und anschließend mit der OpenMP-Direktive *do* die Arbeit verteilt (siehe Abbildung 11(a)). Vorgabe und damit wichtig ist hierbei, dass innerhalb dieser Rechenschleife der Vektor y nur einmal existiert und die einzelnen Unterprozesse bestimmte Teile davon berechnen, ohne sich davon eine eigene Kopie anzulegen. Dadurch wird es nötig, eine eigene Kopie der Zwischenvariable *tmp* für jeden beteiligten Unterprozess anzulegen. Für ein besseres Verständnis aller Möglichkeiten enthält die Abbildung 11 drei Varianten für unterschiedliche Standard-Voreinstellungen.

Im Beispiel der Abbildung 11(a) wird eine explizite Deklaration durch die Verwendung der Klausel „*default(**none**)*“ erzwungen. In diesem Fall müssen alle innerhalb der parallelen Region zu verwendenden Variablen eindeutig einem Datensichtbarkeitsbereichs-Attribut zugewiesen werden (*private(i, tmp) shared(N, x, y)*). Als zweites befindet sich in Abbildung 11(b) die Definition der Routine *foo* mit *private* als *default*-Voreinstellung. Damit würden fast alle Variablen „lokal“ deklariert, insbesondere auch die y-Variable. Da man y nach Vorgabe aber nicht als lokale Kopie verwenden soll, muss man explizit durch

```
(a)     subroutine foo(N)
        double precision x(100)
        double precision y(100)
        double precision tmp
        common /dummy/ x
c$omp   parallel default(none)
c$omp&  private(i, tmp)
c$omp&  shared(N, x, y)
c$omp   do
        do i = 1, N
            tmp = x(i)
            y(i) = tmp * tmp
        end do
c$omp   end do
        ... ! use vector [y]
c$omp   end parallel
        end subroutine
```

```
(b)     subroutine foo(N)
        double precision x(100)
        double precision y(100)
        double precision tmp
        common /dummy/ x
c$omp   parallel default(private)
c$omp&  shared(N, x, y)
c$omp   do
        do i = 1, N
            tmp = x(i)
            y(i) = tmp * tmp
        end do
c$omp   end do
        ... ! use vector [y]
c$omp   end parallel
        end subroutine
```

```
(c)     subroutine foo(N)
        double precision x(100)
        double precision y(100)
        double precision tmp
        common /dummy/ x
c$omp   parallel default(shared)
c$omp&  private(i, tmp)
c$omp   do
        do i = 1, N
            tmp = x(i)
            y(i) = tmp * tmp
        end do
c$omp   end do
        ... ! use vector [y]
c$omp   end parallel
        end subroutine
```

Abbildung 11: Parallelisierung der Routine *foo*: `(a)` mit erzwungener expliziten Deklaration der Datensichtbarkeitsbereich-Attribute, `(b)` mit *private*-Attribut als Voreinstellung und `(c)` mit *shared*-Attribut als Voreinstellung.

die Angabe von *shared(y)* die *default*-Deklaration überschreiben. Als letzte Variante (Abbildung 11`(c)`) sieht man die analoge Umsetzung mit *shared* als *default*-Voreinstellung. Entsprechend würde jetzt y als *shared* umgesetzt, die Variable *tmp* allerdings auch. Deshalb muss *tmp* explizit mit *private(tmp)* der *shared*-Voreinstellung (siehe *default*-Klausel) entgegen definiert werden.

4.2.1 Ersetzung der expliziten *shared*-Spezifikation

Ziel soll es nun sein, sich der expliziten Spezifizierung des Datensichtbarkeitsbereichs-Attributs zu entledigen. Dazu kann man in einem ersten Schritt die parallele Region aus *foo* heraus nach außen ziehen. Beispielhaft wird dies in der Abbildung 12 versucht. Allerdings muss dazu der Vektor y explizit mit *shared* spezifiziert werden. Sonst würde y nur lokal angelegt werden, da er innerhalb der parallelen Region deklariert ist. Hinzu kommt, dass man diese *shared*-Deklaration erst in der Routine *foo* vornehmen kann, weil erst dort der Vektor deklariert ist. Das ist auf die hier gezeigte Weise aber nicht möglich, denn es verstößt gegen den vorausgesetzten OpenMP-Standard. Deshalb muss man entweder den Vektor y in den globalen Speicher- bzw. Variablenbereich verschieben oder dessen Deklaration aus der parallelen Region mit hinaus verschieben. Letzteres wird in der Variante der Abbildung 13 gezeigt, dort wird y genauso wie die parallele Region in den *main*-Teil verschoben und entsprechend als zusätzlicher Funktionsparameter y an die Unterroutine *foo* übergeben.

Allgemein möchte man solche zusätzlichen Parameterübergaben vermeiden, da sie einen beträchtlichen Nachbearbeitungsaufwand erforderlich machen können. Stattdessen kann man y auch in den globalen Speicherbereich verschieben. Im Folgenden werden drei Möglichkeiten gezeigt. Dazu beginnt die Abbildung 14 mit der „MODULE“-Version. Dabei wird die Deklaration der Variable y einfach in ein eigenes nur für *foo* verfügbares „Fortran-MODULE“ („*foo_dummy_shared*“) verschoben. Die in einem „Fortran-MODULE“ deklarierten Variablen sind zwar nicht uneingeschränkt global, liegen aber im globalen Speicherbereich, weshalb sie entsprechend den Richtlinien für globale Variablen als *shared* voreingestellt werden. Dasselbe gilt für die anderen beiden Varianten. Die zweite Version in Abbildung 15 verschiebt durch einen *foo*-eigenen „COMMON“-Block das y in den globalen Bereich. Die letzte Lösung (Abbildung 16) nutzt dazu das „SAVE“-Attribut.

Alle vier Varianten (Abbildungen 13 bis 16) haben gemein, dass man innerhalb der Un-

```
      ! main routine
      ...
      double precision x(100)
      common /dummy/ x
c$omp parallel
c$omp& shared(x)
      call foo(100)
c$omp end parallel
      ...

      subroutine foo(N)
      double precision x(100), y(100), tmp
      common /dummy/ x
c$omp& shared(y) ! Nicht OpenMP konform !
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 12: Verschiebung der parallelen Region in den *main*-Programmteil, wobei die grau markierte *shared(y)*-Klausel zur *parallel*-Direktive gehört und damit an dieser Stelle nicht nachträglich definiert werden darf - nicht OpenMP konform. Andererseits kann diese *shared*-Klausel auch nicht zu der nachfolgenden *do*-Direktive gehören, da es sich dann um den nicht-erlaubten Übergang $\ddot{U}_{pPsA}$ aus Tabelle 1 handeln würde.

```
      ! main routine
      ...
      double precision x(100), y(100)
      common /dummy/ x
c$omp parallel
c$omp& shared(x, y)
      call foo(100, y)
c$omp end parallel
      ...

      subroutine foo(N, y)
      double precision x(100), y(100), tmp
      common /dummy/ x
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 13: Verschiebung der parallelen Region in den *main*-Programmteil - Lösung durch zusätzlichen Übergabeparameter

terroutine *foo* keine Datensichtbarkeitsbereichs-Attribute mehr spezifizieren musste. Das gilt nicht allgemein, da man für einige Spezialfälle, wie z.B. bei „Arbeitsverteilungs“-Konstrukten, vorübergehend „lokale“ Kopien von „geteilten“ Variablen anlegen und ver-

```
      module foo_dummy_shared
      double precision y(100)
      end module

      subroutine foo(N)
      use foo_dummy_shared
      double precision x(100), tmp
      common /dummy/ x
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 14: Variante mit nachträglichem *shared* Datensichtbarkeitsbereich der Variablen **y** - mit Hilfe einer zusätzlichen „Fortran-MODULE“-Deklaration.

```
      subroutine foo(N)
      double precision x(100), y(100)
      double precision tmp
      common /dummy/ x
      common /foo_dummy_shared/ y
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 15: Variante mit nachträglichem *shared* Datensichtbarkeitsbereich der Variablen **y** - mit einem „COMMON“-Block.

```
      subroutine foo(N)
      double precision x(100), y(100)
      double precision tmp
      common /dummy/ x
      save y
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 16: Variante mit nachträglichem *shared* Datensichtbarkeitsbereich der Variablen **y** - mit einer „SAVE“-Direktive.

wenden kann. Innerhalb des OpenMP-Standards ist das allerdings explizit erlaubt und damit vorgesehen. Dennoch kann dies bei der Weiterverarbeitung während einer automatischen Programmcode-Transformation zu erheblichen Problemen führen. Wann dies auf-

tritt sowie Strategien zur Umgehung dieser Probleme werden ausführlich in Kapitel 4.4.1 und 4.5 beschrieben.

Insbesondere für AD-Werkzeuge mit Codetransformation muss man im Nachfolgenden von einem vollständig von expliziten Datensichtbarkeitsbereichs-Attributen bereinigten Programmcode ausgehen können. Erlaubt sind dann nur noch Direktiven zur Arbeitsverteilung und damit keine impliziten Synchronisationen oder OpenMP-Rechenoperationen.

Für alle Variablenobjekte, auf die „geteilt“ zugegriffen werden soll (*shared*-Attribut), fasst das nachfolgende Maßnahmenpaket 4.1 die möglichen Umstrukturierungen zusammen, um die expliziten Attributsdeklarationen überflüssig werden zu lassen.

Maßnahmen 4.1 *Ersatz von* **shared***-Spezifikationen*

- *Als neue Übergabeparameter einführen und damit von außen „lokal übergeben“*
- *Verschiebung in den globalen Speicherbereich durch Überführung in ein neues „Fortran-MODULE“, „COMMON“-Block oder mit einer zusätzlichen „SAVE“-Deklaration*

Anmerkung: Alle in dieser Arbeit beschriebenen Maßnahmen und Richtlinien können Programmumstellungen verursachen, die die Rechenleistung bzw. Effizienz beeinflussen. Wenn keine zusätzlichen Angaben aufgeführt sind, wird davon ausgegangen, dass gegebenenfalls auftretende Nachteile für den typischen realen Programmcode vernachlässigt werden können. Insbesondere entspricht das den gemachten Erfahrungen für das dieser Arbeit zugrunde liegende Simulationswerkzeug, da keine im Suite-Projekt umgesetzten Maßnahmen zu beachtenswerten Verlusten bei der Rechenleistung führten.

4.2.2 Ersetzung der expliziten *private*-Spezifikation

Für eine allgemeine Lösung benötigt man die volle Kontrolle, welche Variablen „geteilten“ Zugriff haben oder „lokal“ sein sollen. Deshalb wird nun erläutert, wie mit zwingend „lokalen“ Variablen zu Verfahren ist. Es gilt hier, drei wichtige Fälle zu unterscheiden. Einerseits gibt es die „globale“ Gruppe der im globalen Kontext deklarierten oder in der Parameterübergabe stehenden Variablen, welche eine lokale Verwendung finden sollen.

Andererseits gibt es die Gruppe aller lokal in der Subroutine und gleichzeitig außerhalb der parallelen Region deklarierten Variablen. Zuletzt gibt es noch eine Gruppe, zu denen die „echt“ lokal deklarierten Variablen gehören. Um die „echt“ lokal Deklarierten braucht man sich nicht zu kümmern, da sie für eine lokale Verwendung schon korrekt innerhalb der parallelen Region von jedem Unterprozess selbst angelegt werden. Die lokalen Variablen (in der Subroutine), die außerhalb der parallelen Region deklariert sind, werden dagegen nur dann automatisch korrekt von jedem Unterprozess lokal angelegt, wenn man die parallele Region (wie im letzten Unterkapitel schon erwähnt) nach außen und damit außerhalb der Subroutine verschiebt. Damit werden auch sie dann „echt“ lokal. Einige Vor- und Nachteile dieses Ansatzes werden im nachfolgenden Unterkapitel 4.3 erläutert.

Damit bleiben die lokalen Variablen der globalen Gruppe übrig, die ursprünglich global deklariert sind oder in der Parameterübergabe stecken. Generell muss man beachten, dass der Zugriff auf diese Variablen außerhalb der parallelen Region in einer Art *shared*-Kontext verwendet werden und nur innerhalb der Region die Arbeit auf lokale Kopien vorgesehen ist. Je nach OpenMP-Klausel kann dabei ein Kopieren der Werte hinein oder heraus beim Regionsübergang impliziert sein. Damit beschäftigt sich ausführlich das Kapitel 4.4.5, weshalb hier erst einmal die Wertekonsistenz bei einem Übergang vernachlässigt wird.

Für die Programmbereiche, in denen vorübergehend auf echten „lokalen“ Kopien (Variablen der globalen Gruppe) gearbeitet werden muss, fasst das nachfolgende Paket 4.2 drei mögliche Maßnahmen zusammen.

Maßnahmen 4.2 *Ersatz von* **private***-Spezifikationen*

1. *Spezifizierung durch* **threadprivate** *(siehe [13]) - Allerdings nur dann, wenn die Variable für alle auftretenden parallelen Regionen „lokal“ sein soll und darf.*
2. *Einführung einer neuen lokalen Zwischenvariable - Dies ist wenig aufwändig, kann aber zu kleinen Nachteilen im Speicherverbrauch oder der Rechenleistung führen.*
3. *Beibehaltung des globalen oder Parameterübergabe-Kontextes, allerdings mit einer Erhöhung der Dimensionierung der Variable → Jeder Unterprozess („OpenMP thread“) arbeitet dann auf einem eigenen Teilbereich.*

Der 3. Unterpunkt ist insbesondere dann wichtig, wenn es sich um mehrfach geschachtelte parallele Regionen handelt. Um die Problematik zu veranschaulichen, betrachtet man dazu einfach einen Fall, bei dem zwei geschachtelte parallele Regionen innerhalb einer Subroutine liegen und in beiden Regionen die Variable x in zusätzliche lokale Kopien aufgespalten werden soll. Durch eine neue lokale Zwischenvariable (2. Unterpunkt) in der Subroutine lässt sich hier nur die Erschaffung der lokalen Kopien für die erste parallele Region erfüllen. Die Erweiterung auf die zweite Region könnte man durch eine zusätzliche Funktionsmodularisierung des inneren Bereichs, im Sinne von Kapitel 4.3 erreichen. Wenn dies zu aufwändig wird, ist es oft einfacher, statt dessen eine mehrdimensionale Umstrukturierung entsprechend des 3. Unterpunktes vorzunehmen.

Im Detail muss man für zwei geschachtelte parallele Regionen entsprechend auch zwei zusätzliche Dimensionen für jede lokale Variable x einführen. Dabei ist zu beachten, dass man dann für die Variablen den Hauptspeicher über den ganzen Programmbereich reserviert halten muss, was ein verstärkter Nachteil gegenüber der 2. Lösung sein kann. Mehr Details zu der Verwendung zusätzlicher Dimensionen findet man im Abschnitt „Verwendung höher-dimensionaler Datenobjekte“ auf Seite 74.

In gleicher Art und Weise, wie man jetzt alle Datensichtbarkeitsbereichs-Attribute innerhalb der Unterroutine *foo* eliminiert, kann man auch in anderen Programmbereichen (z.B. im *main*-Teil) vorgehen. Als Resultat erhält man einen Programmcode, in dem stark reduziert und nur noch für bestimmte Ausnahmen ein Datensichtbarkeitsbereichs-Attribut spezifiziert werden muss. Das erleichtert eine automatische Weiterverarbeitung durch ein AD-Werkzeug, wie sie in Kapitel 4.4 benötigt wird.

Bezüglich des Aufwands für die Wartbarkeit ist anzumerken, dass das explizite Setzen der Datensichtbarkeitsbereichs-Attribute scheinbar nur durch eine nachträgliche Globalisierung der Variablen oder die anderen Maßnahmen eingetauscht wird (*Ersatzaufwand*).

Bei einer genaueren Betrachtung der geforderten Erweiterbarkeit (aus Kapitel 3) fällt auf, dass durch das Entfallen der expliziten Attributs-Spezifikationen (durch Maßnahmen 4.1 und 4.2) ein entscheidender Vorteil für die Wartbarkeit entsteht: Denn bei rein algorithmischen Erweiterungen ohne neue Variablen erweitert man einen wesentlich übersichtlicheren Programmcode ohne nennenswerten Zusatzaufwand für Attributs-Spezifikationen. Wären noch explizite Attributs-Spezifikationen vorhanden, müsste man diese wahrscheinlich zusätzlich nach pflegen.

In anderen Fällen handelt es sich um Erweiterungen mit zusätzlich einzuführenden Variablen. Für diese spielen insbesondere die zusammengefassten Datenstruktur-Richtlinien, wie sie in Kapitel 6 festgelegt und erläutert werden, eine wichtige Rolle. Diese Richtlinien sind natürlich auf die Parallelisierung gut abgestimmt. Deren Einhalten sorgt damit automatisch für die korrekten Datensichtbarkeitsbereich-Eigenschaften von neu eingeführten Variablen. So gesehen handelt es sich auch hier nicht um einen eingetauschten *Ersatzaufwand* anstelle der expliziten Datensichtbarkeitsbereich-Spezifizierung. Denn die (automatische) Voreinstellung der Datensichtbarkeitsbereichs-Attribute implizit durch den Deklarationspunkt der Variablen führt zu einer Bündelung (geringerer Aufwand), da

die Attribute nicht mehr für jede parallele Region oder Arbeitsverteilungs-Direktive einzeln spezifiziert werden müssen. Nur wenn man von einer nachträglichen Änderung des Datensichtbarkeitsbereichs-Attributs einer schon vorhandenen Variable ausgeht, handelt es sich noch um einen vergleichbar großen *Ersatzaufwand*.

4.3 Parallelisierungs-spezifische Modularisierung

Bisher wurden Strategien zur Entfernung der expliziten Datensichtbarkeitsbereichs-Attribute vorgestellt. Im Folgenden soll darüber hinaus der Umgang mit dem Eröffnen und Schließen einer parallelen Region allgemein erleichtert werden. Es empfiehlt sich, wie später noch erläutert wird, von *shared* als Voreinstellung auszugehen. Dafür muss keine Spezifikation von „*default(***shared***)*“ zu Beginn jeder parallelen Region einfügt werden (vergleiche dazu Abbildung 11(c)), denn der Compiler geht nach dem OpenMP-Standard 2.5 für Fortran bereits von *shared* als Standard-Voreinstellung aus.

Darüber hinaus empfiehlt es sich, die Beginn- und Ende-Direktiven der parallelen Region aus der jeweiligen Routine (z.B. *foo*) in eine eigene Zwischenroutine auszulagern. Näheres zu diesem Ansatz findet man auch in der Literatur [88, 89] unter dem englischen Begriff „outlining“. Dort geht es üblicherweise um eine hilfreiche Compiler-Technik für einen erleichterten Umgang mit den OpenMP-Unterprozessen. Im Gegensatz dazu geht es hier um Erleichterungen bei der Implementierung der Parallelisierung bzw. einer besseren Wartbarkeit. Das ist wichtig, denn man versucht dadurch eine Gleichbehandlung aller in einer Routine deklarierten Variablen zu erzwingen. Das hat zur Folge, dass man bei den Variablen nicht mehr zwischen außerhalb und innerhalb (der parallelen Region) unterscheiden muss. Denn ein Unterschied könnte sich sonst bei dem Übergang in die parallele Region ergeben, wenn einige Variablen entgegen der Voreinstellung mit *private* explizit deklariert sind (verwendungsabhängig). Diese erzwungene Konvention der Gleichbehandlung erhöht nicht nur die Übersichtlichkeit, sondern reduziert entsprechend auch die expliziten Datensichtbarkeitsbereichs-Attribute. Andererseits erhält man noch den positiven Nebeneffekt, dass man mit Hilfe der neuen Zwischenroutine unabhängig von der Voreinstellung wird. Es ist dann unerheblich, ob man abweichend von dem bisherigen *default*-Vorschlag auch mal *private* als Voreinstellung verwendet. Die Abbildung 17 zeigt eine solche komplette Umstellung.

In dem Beispiel der Abbildung 17 gleicht die Umstellung erst einmal einer Mischung aus den Varianten der Abbildungen 12 und 15 des vorherigen Unterkapitels 4.2. Nach der

```
      ! main routine
      ...
      double precision x(100)
      common /dummy/ x
      call omp_foo(100)
      ...

      ! OpenMP wrapper routine
      subroutine omp_foo(N)
      integer N
c$omp parallel
      call foo(N)
c$omp end parallel
      end subroutine
```

```
      subroutine foo(N)
      double precision x(100)
      double precision y(100)
      double precision tmp
      common /dummy/ x
      common /foo_dummy_shared/ y
c$omp do
      do i = 1, N
         tmp = x(i)
         y(i) = tmp * tmp
      end do
c$omp end do
      ... ! use vector [y]
      end subroutine
```

Abbildung 17: Eine zusätzliche Zwischenroutine *omp_foo* und exemplarischer „COMMON“-Block Lösung für die Variable *y* in *foo*.

Umstellung haben alle bisher lokal und direkt in *foo* deklarierten Variablen ein *private*-Attribut, da sie nun innerhalb der außerhalb von *foo* beginnenden parallelen Region deklariert und angelegt werden.

Darüber hinaus ist die ausgelagerte parallele Region nicht in die *foo* aufrufende Routine (z.B. *main*) hinein verlegt worden, sondern in eine neue Zwischenroutine (engl. „wrapper“). Dadurch entfällt bei der *main*-Routine die Notwendigkeit für zusätzliche Spezifikationen mittels Datensichtbarkeitsbereichs-Attributen an der Regionsgrenze. Diese Zwischenroutine (links-unten in der Abbildung) hier als *omp_foo* bezeichnet, soll weiter nichts beinhalten als alle bisherigen Funktionsparameter vom originalen *foo*, dem Eröffnen und Schließen der parallelen Region und dem dort enthaltenen eigentlichen Unterroutinen-Aufruf von *foo*. An dieser Stelle sei nochmal darauf hingewiesen, dass die Zwischenroutine nur dazu da ist, die parallele Region auf und zuzumachen und deshalb dort nur die Funktionsparameter deklariert und durchgereicht werden, die schon in der originalen Version von *foo* vorhanden waren. Außerdem ist es, anders als in Abbildung 17 gezeigt, auch möglich, die Zwischenroutine mit *foo* zu benennen und die ursprüngliche Routine mit *omp_foo*. Das hätte den Vorteil, dass der *main*-Programmteil gegenüber dem Urzustand immer noch unverändert einen „**call**“ auf *foo* enthalten würde und damit unverändert bleiben könnte. Nachfolgend wird die Zwischenroutine weiterhin mit *omp_foo* bezeichnet, um ihn speziell als eingefügte Modifikation zu kennzeichnen. Letztendlich kann man mit der Verwendung von Zwischenroutinen die eigentlichen Direktiven für das Eröffnen und Beenden der parallelen Region und gegebenenfalls explizitem Datensichtbarkeitsbereich zusätzlich gut verstecken.

An dieser Stelle wird ersichtlich, welchen Vorteil man in Bezug auf eine wechselnde Voreinstellung erhält. Für Zwischenroutinen ohne Funktionsparameter hat man eine komplette Entkoppelung von jeglicher Voreinstellung (mittels *default*), da es hier keine „lokal übergebenen“ Variablen gibt, auf die sich eine Voreinstellung auswirken könnte. Andererseits erhält man für die Weiterverarbeitung bei einer Programmcode-Transformation eine verbesserte Vereinbarkeit bei wechselnden Voreinstellungen (durch z.B. unterschiedliche *default*-Spezifikationen). Voraussetzung dafür sind lediglich zwei Punkte, die in dem nachfolgenden Grundannahmepaket 4.1 zusammengefasst sind.

Grundannahme 4.1 *Deklaration und Verschiebung nach Code-Transformation*

1. *Neue Variablen müssen durch das Transformations-Werkzeug im korrekten Deklarationskontext angelegt werden (unter anderem niemals nur in der Zwischenroutine „omp_foo“). Das ist normalerweise gegeben und kann beispielsweise zum Anlegen eines neuen Übergabeparameters (analog zu einem bereits bestehenden) führen. Dadurch hat der neue Übergabeparameter dann dasselbe Datensichtbarkeitsbereichs-Attribut, egal was für* **default** *spezifiziert ist.*

2. *Programmanweisungen und Deklarationen dürfen durch die Programmcode-Transformation nicht aus einer Unterroutine (z.B. „foo“) heraus oder hinein verschoben werden. Da die verwendeten Transformations-Werkzeuge dies gewährleisten konnten, wird es hiermit als Grundlage und als gegeben betrachtet.*

4.3.1 Eliminierung expliziter Datensichtbarkeitsbereichs-Attribute

In den vorherigen Kapiteln wurde separat erläutert, wie mit expliziten Datensichtbarkeitsbereich-Attributen einzelner Variablen umgegangen werden soll. An dieser Stelle soll zusammenfassend an einem allgemeineren Beispiel diese Anwendung, bis hin zur vollständigen Eliminierung der expliziten Attributsdeklarationen, demonstriert werden. Dieses Beispiel ist für ein besseres Verständnis sehr wichtig und ist deshalb sehr allgemein gehalten. Insbesondere geht es hier nicht auf komplizierte Datensichtbarkeitsbereich-Klauseln ein, die neben der Attributs-Festlegung auch Werte von Variablen kopieren (z.B. *firstprivate*). Das ist hier aber keine Einschränkung, denn für deren Handhabung sind bezüglich des Datensichtbarkeitsbereichs-Attributs die selben Maßnahmen erforderlich.

Das verallgemeinerte Richtlinienpaket 4.1 beschreibt die Strategie bei der Erstellung einer „neuen“ Software. Andererseits kann dies auch nachträglich als einmalige Vorberei-

tungsmaßnahme auf einem bestehenden Programmpaket erfolgen.

Richtlinien 4.1 *Ausschließliche Verwendung des Datensichtbarkeitsbereich-Automatismus statt expliziter Attributs-Spezifikation*

- *Die parallelen Regionen müssen entweder entsprechend Kapitel 4.3 in Zwischenroutinen ausgelagert oder aus der Subroutine heraus verschoben werden.*
- *Programmbereiche, die neu in eine parallele Region hinein verschoben werden, müssen dort seriell ausgeführt werden (beispielsweise durch eine* **master***-Direktive mit anschließender Synchronisation durch eine* **barrier***-Direktive).*
- *Als Voreinstellung (***default***) wird* **shared** *verwendet.*
- *Jedes Datenobjekt, welches mindestens einmal mit „geteilten“ Zugriff (***shared***) verwendet wird, muss durch die Maßnahmen 4.1 vorbereitet werden.*
- *Jedes Datenobjekt, welches zusätzlich an einigen Programmstellen „lokal“ (***private***) verwendet werden soll, benötigt zusätzlich Variablen entsprechend der zweiten oder dritten Variante des Maßnahmenpaketes 4.2.*
- *Jedes Datenobjekt, welches ausschließlich „lokal“ verwendet wird, muss durch die Maßnahmen 4.2 vorbereitet werden.*

Das resultierende mehrstufige Vorgehen bei einer Nachbearbeitung von einer bestehenden Software wird nun am folgenden komplexen Beispiel in Abbildung 18 veranschaulicht: Dieses Vorgehen stellt in analoger Weise den tatsächlichen Ablauf auch für das dieser Arbeit zugrunde liegende Software-Projekt nach.

Ausgangspunkt ist der linke Programmausschnitt für eine Subroutine *foo* mit einer einzelnen parallelen Region und innerer OpenMP-Arbeitsverteilung. In der Mitte von Abbildung 18 sieht man den erfolgreich transformierten Programmcode ohne explizite Datensichtbarkeitsbereichs-Attribute. Er steht damit beispielhaft für die Anwendung aller bis hierher vorgeschlagenen Maßnahmen. Ganz rechts sieht man eine Reihe von farbig markierten Hilfspfeilen, wobei ein grüner Pfeil eine gleichbleibende Nutzung der Variable bzw. ihres Datensichtbarkeitsbereichs markiert. Ein roter Pfeil steht dagegen für den Wechsel von einer Variable mit *shared*-Attribut zu einer mit *private*-Attribut (oder umgekehrt).

Im Detail ist der linke Ausgangscode (Abbildung 18) in insgesamt fünf Programmteile **A** bis **E** gegliedert, wobei **A** und **E** außerhalb der parallelen Region liegen und der Programmteil **C** zusätzlich über die `do`-Direktive ein OpenMP-Arbeitsverteilung vornimmt. Um das Beispiel möglichst allgemein zu halten, gibt es drei unterschiedliche Variablen deren Deklarationen erst einmal beliebig (<Deklaration>) vorgegeben sind.

Die Unterschiede drücken sich durch ein wechselnden bzw. gleichbleibenden Datensichtbarkeitsbereich aus. Auf die Variable *vss* soll durchgängig mit *shared*-Attribut zugegriffen werden. Bei der *vsp* soll innerhalb der parallelen Region auch *shared*, aber innerhalb der Schleifenparallelisierung (`do`-Direktive über Abschnitt **C**) das *private*-Attribut gelten. Als letztes steht die Variable *vpp* für einen durchgängigen *private*-Zugriff. Da in dem gesamten Beispiel *shared* als Voreinstellung gelten soll, muss man nur die expliziten *private*-Übergänge spezifizieren. Deshalb wird für *vpp* sowohl zu Beginn der parallelen Region, wie auch an der Schleifen-Direktive, eine explizite Spezifizierung vorgenommen. Für *vsp* muss es dagegen nur an der Schleifen-Direktive angegeben werden und für *vss* gar nicht.

Schrittweise Codetransformation

Entsprechend des Richtlinienpaketes 4.1 wird im Beispiel der Abbildung 18 die parallele Region als erstes nach außen über die Programmteile **A** bis **E** hinaus (in den *main*-Bereich) verschoben. Das ist meist zweckmäßiger, als eine minimale Zwischenroutine mit

Abbildung 18: Links: Programmcode im Ausgangszustand; Mitte: Mit den Richtlinien 4.1 konformer nachbearbeiteter Programmcode; Rechts: Variablenübergänge zwischen den einzelnen Programmteilen.
⬇ Gleichbleibendes Datensichtbarkeitsbereichs-Attribut; ⬇ wechselndes Datensichtbarkeitsbereich-Attribut (Umbenennung der Variablen mit und ohne „_p“)

vielen Übergabeparametern für die parallele Region und die Programmteile **B** bis **D** zu generieren. Dagegen kann man eine in den *main*-Bereich verschobene parallele Region von dort auch in eine Zwischenroutine kapseln und benötigt dann nur die selben Übergabeparameter, wie für das ursprüngliche *foo*.

Wenn man den weiteren Richtlinien 4.1 folgt, dann muss man als nächstes für alle Variablen einen *shared*-Zugriff sicherstellen, die einen solchen benötigen. In diesem Beispiel (Abbildung 18) geht man deshalb davon aus, dass in den Programmteilen **A** und **E** (also außerhalb der ursprünglichen parallelen Region) auf alle drei Variablen *shared* zugegriffen wird. Deswegen wurde jede ursprüngliche Variable dem Maßnahmenpaket 4.1 entsprechend im globalen Kontext angelegt (<**globale** Deklaration>). Die obere Hälfte in Abbildung 18 verdeutlicht die bisherigen Transformationen.

Der ganz rechte Teil der Abbildung zeigt zu den einzelnen Programmabschnitten die jeweilig verwendeten Variablen. Hier sieht man auch auf der Höhe der Programmabschnitte **A** und **E**, dass dort die global deklarierten Varianten der Variablen *vss*, *vsp* und *vpp* verwendet werden. Man sieht auch, wozu man die *master*-Direktive mit der *barrier*-Synchronisation benötigt. Da sowohl **A** und **E** jetzt innerhalb der parallelen Region liegen, muss ein Unterprozess stellvertretend für alle anderen die Berechnung vornehmen (oder **A** und **E** von allen redundant berechnet werden). Außerdem muss sicher gestellt werden, dass alle Unterprozesse zu Beginn des Programmteils **B** konsistent auf die Werte (von **A**) in den bearbeiteten Variablen zugreifen können. Analog dazu muss nach dem Ende von **D** auch die Datenkonsistenz sicher gestellt werden.

Betritt ein Unterprozess jetzt den Bereich **B**, muss nach Vorgabe auf einer „loka-

len“ Kopie von *vpp* gearbeitet werden. Das wird mit dem Maßnahmenpaket 4.2 sichergestellt, in dem bei diesem Beispiel für *vpp* eine neue lokale Variable *vpp_p* eingeführt wurde (<lokale Deklaration> *vpp_p*). Der Wechsel von *vpp* zu *vpp_p* wurde für den Übergang von **A** zu **B** mit einem roten Pfeil und die neu zu verwendende Variable *vpp_p* mit grauem Hintergrund markiert. Im Gegensatz dazu wird in **B** genauso wie in **A** noch auf *vss* und *vsp shared* zugegriffen und dies entsprechend mit einem grünen Pfeil verdeutlicht. Bei dem Übergang zum Programmteil **C** wird weiterhin auf dasselbe „lokale“ *vpp_p* und dasselbe *vss* (*shared*-Attribut) zugegriffen (grüne Übergangspfeile).

Neu ist hier allerdings der zusätzliche Wechsel von dem „geteilten“ *vsp* zu seiner „lokalen“ Variante *vsp_p*, was durch den entsprechenden roten Pfeil gekennzeichnet wird. Für die Übergänge von **C** zu **E** müssen die Variablen-Wechsel, wie in der Abbildung gezeigt, analog wieder zurück in umgekehrter Reihenfolge durchgeführt werden.

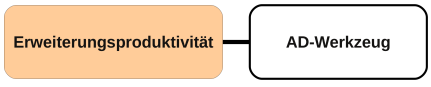
Das Beispiel zeigt anhand der wesentlichen Schritte des Richtlinienpaketes 4.1, dass auch vielfältige Mischungen von Übergangskonfigurationen verschiedener Variablen innerhalb eines Programmabschnittes handhabbar sind.

4.4 Erweiterte Automatisierung bei der Erzeugung von Ableitungscode

Dieses Kapitel widmet sich der *Vereinbarkeit* von parallelisierten Programmcode und AD-Werkzeug. Ein Ziel in diesem Kapitel ist es, die speziellen Anforderungen an das AD-Werkzeug zu erläutern, die für den Erhalt der Grundanforderung Parallelisierung notwendig sind. Ausgegangen wird dafür nicht mehr vom originalen Code der Modell-Simulation, sondern von dem erzeugten AD-Transformationscode. Im Detail sollte es sich dabei um einen von expliziten Datensichtbarkeitsbereichs-Attributen befreiten AD-Programmcode handeln, der entsteht, wenn man das AD-Werkzeug auf den entsprechend vorbereiteten Code der Modell-Simulation anwendet: (1) Die erste Teilanforderung an diese Transformation ist, dass die in dem Programmcode der originalen Modell-Simulation enthaltene Parallelisierung übernommen werden kann (Unterkapitel 4.4.1) und damit nutzbar bleibt; (2) Darüber hinaus soll in einem zweiten Schritt gewährleistet werden, dass diese Parallelisierung sich analog auf die Berechnung der Ableitungen erweitert lässt (Unterkapitel 4.5); (3) Unabhängig von der Machbarkeit wird im danach folgenden Unterkapitel 4.6 erläutert, wie man den Prozess der automatischen Erzeugung von Ableitungscode effizienter gestalten kann, indem man den Programmcode bezüglich der Notwendigkeit für die Ableitung restrukturiert.

Alle drei Aspekte zusammen ergeben eine Erweiterung der bisher beschriebenen Strategie für eine effiziente und nahezu automatisch ablaufende Erzeugung von Ableitungscode ohne negative Nebeneffekte auf andere Grundanforderungen.

Enthält das zu transformierende Programm eine Parallelisierung, erfordert dies eine spezielle Unterstützung vom AD-Werkzeug, um die ursprüngliche Parallelisierung auf die Ableitungsberechnung der Modell-Simulation zu reaktivieren und auszudehnen. Andererseits ist eine davon unabhängige Parallelisierung für die reine Ableitungsberechnung nicht eindeutig. Deswegen wird davon ausgegangen, dass die vom Anwender erwünschte Parallelisierungsstrategie nicht vom AD-Werkzeug unterstützt wird.



Normalerweise geht man bei der Erzeugung von Ableitungscode durch ein AD-Werkzeug weitestgehend von einem automatisierten Prozess aus. Aus der Annahme einer fehlenden Unterstützung der Parallelisierung durch das AD-Werkzeug, folgt die Notwendigkeit zu einer speziellen Vor- und Nachbearbeitung des erzeugten Programmcodes. Umso wichtiger ist es in dem hier vorgestellten Konzept, die sonst manuelle Nachbearbeitung voll automatisch zu ermöglichen. Insgesamt entspricht dies damit einer Verbesserung im Sinne der Erweiterungsproduktivität.

4.4.1 Eine die Parallelisierung erhaltene Erzeugung von Ableitungscode

An dieser Stelle wird der Einsatz der im letzten Kapitel vorgestellten Techniken vorausgesetzt, wodurch nur noch OpenMP-Direktiven erlaubt sind, die die parallelisierte Arbeitsverteilung regeln. Dabei handelt es sich z.B. um Schleifen-Direktiven wie „`c$omp do`“ in der Routine *foo* der Abbildung 17 (rechts). Weiter wird dabei vorausgesetzt, dass die Zwischenroutinen dem AD-Werkzeug entweder vorenthalten werden, oder vom AD-Werkzeug korrekt behandelt werden. Um etwaige Probleme mit dem AD-Werkzeug auszuschließen, wird deswegen vereinfacht angenommen, dass die Zwischenroutinen entfallen und vorübergehend alle Aufrufe von „**call** *omp_foo(100)*“ durch „**call** *foo(100)*“ ersetzt wurden (beispielhaft im *main*-Teil der Abbildung 17). Das kann vollautomatisch mit einfachen Ersetzungs-Werkzeugen (z.B. mit dem Text-Ersetzungs-Werkzeug „sed“) durchgeführt werden.

Eine wichtige Anforderung an das AD-Werkzeug ist zusätzlich das Beibehalten bestimmter OpenMP-Direktiven an festgelegten algorithmischen Programmstellen. Denn die Direktiven zur Arbeitsverteilung sind immer noch vorhanden und dürfen meist nicht aus ihrem algorithmischen Kontext entfernt werden. So gehört beispielsweise die OpenMP-Schleifendirektive „`c$omp do`“ unmittelbar vor die Fortran-Schleifendeklaration (siehe dazu die Routine *foo* in Abbildung 19`(1)`). Abhängig vom AD-Werkzeug genügt dafür die

```
(1)     subroutine foo(N)
        double precision x(100)
        double precision y(100)
        double precision tmp
        common /dummy/ x
        common /foo_dummy_shared/ y
c$omp   do
        do i = 1, N
          tmp = x(i)
          y(i) = tmp * tmp
        end do
c$omp   end do
        ... ! use vector [y]
        end subroutine
```

```
(2)     subroutine foo(N)
        double precision x(100)
        double precision y(100)
        double precision tmp
        common /dummy/ x
        common /foo_dummy_shared/ y
        call dummy_openmp('c$omp do')
        do i = 1, N
          tmp = x(i)
          y(i) = tmp * tmp
        end do
        call dummy_openmp( &
          'c$omp end do')
        ... ! use vector [y]
        end subroutine
```

```
(3)     subroutine g_foo(N)
        double precision x(100)
        double precision y(100)
        double precision g_x(100)
        double precision g_y(100)
        double precision tmp, g_tmp
        common /dummy/ x, g_x
        common /foo_dummy_shared/ y
        common /foo_dummy_shared/ g_y
        call dummy_openmp('c$omp do')
        do i = 1, N
          g_tmp = g_x(i)
          tmp = x(i)
          g_y(i) = 2 * tmp * g_tmp
          y(i) = tmp * tmp
        end do
        call dummy_openmp( &
          'c$omp end do')
        ... ! use vector [y]
        end subroutine
```

```
(4)     subroutine g_foo(N)
        double precision x(100)
        double precision y(100)
        double precision g_x(100)
        double precision g_y(100)
        double precision tmp, g_tmp
        common /dummy/ x, g_x
        common /foo_dummy_shared/ y
        common /foo_dummy_shared/ g_y
c$omp   do
        do i = 1, N
          g_tmp = g_x(i)
          tmp = x(i)
          g_y(i) = 2 * tmp * g_tmp
          y(i) = tmp * tmp
        end do
c$omp   end do
        ... ! use vector [y]
        end subroutine
```

Abbildung 19: `(1)` die originale Funktion *foo*; `(2)` mit automatisierter Ersetzung vorbereitet; `(3)` nach AD-Anwendung; `(4)` mit automatisierter Ersetzung nachbereitet. Die Routine *dummy_openmp* ist nicht weiter definiert, da sie keine Funktionalität enthält.

Angabe bestimmter Optionen. Andernfalls kann man versuchen, dies durch eine vorübergehende Verkapselung aller verbliebenen OpenMP-Direktiven und Klauseln in Dummy-Funktionen zu erreichen.

Eine einfache automatisierte Ersetzung könnte demnach jedes „`c$omp do`“ in „**call** *dummy_openmp*(`'c$omp do'`)“ automatisch umsetzen und alles nach der Transformation auf dem generierten AD-Code wieder rückgängig machen. Abbildung 19 zeigt einen solchen Werdegang (von links-oben) für eine originale Funktion *foo* (Abbildung 19`(1)`) und deren automatisch ersetzte Variante rechts daneben (Abbildung 19`(2)`). Danach geht es links-unten (Abbildung 19`(3)`) mit dem entsprechenden generierten Ableitungscode *g_foo* weiter und endet dann rechts-unten (Abbildung 19`(4)`) mit dem (wieder automatisch) nachbearbeiteten fertigen Ableitungscode. Falls das verwendete AD-Werkzeug OpenMP ausreichend unterstützt (teilweise in TAF [8]), sind dieser Übergänge automatisch vom AD-Werkzeug und damit ohne manuell erstellte Ersetzungsautomatik möglich.

Eine weitere Möglichkeit bietet den Vorteil, dass es bei einer einmaligen Umstellung des Programmcodes bleibt. Dabei wird vorausgesetzt, dass das Transformations-Werkzeug keine Anweisungsverschiebungen heraus oder hinein in eine Unterroutine erzeugt (siehe Grundannahmen 4.1 auf Seite 39). Dann kann man einen wichtigen Programmteil von innerhalb der Direktivengrenzen in eine eigene neue Unterroutine auslagern und damit sicherstellen, dass die Direktiven in unmittelbarer Nähe bleiben (vor und nach der neuen Unterroutine). Es muss insbesondere nicht mehr wegen ungünstiger Anweisungsverschiebungen vor- und nachbearbeitet werden. Natürlich kann dies im Allgemeinen die Übersicht verschlechtern. Deshalb gilt es abzuwägen, wann man diese Alternative einsetzt.

Für einen späteren Schnellzugriff und einer besseren Übersicht wird das noch einmal in Kurzform in dem Maßnahmenpaket 4.3 zusammengefasst.

Maßnahmen 4.3 *Erhaltung der OpenMP-Direktiven in unmittelbarer Nähe*

- *Vorübergehendes Verkapseln von OpenMP-Direktiven und Klauseln in einzelne Dummy-Funktionsaufrufe.*
- *Auslagern des Inhalts eines zusammenhängenden Direktiven-Bereichs in eine neue Unterroutine.*

Wichtig für alle weiteren Betrachtungen ist die Grundannahme, dass alle OpenMP-Direktiven nach der Codetransformation sich noch im korrekten Programmkontext befinden.

Auf der anderen Seite können viele Programmanweisungen auch so verschoben werden, dass der Programmkontext der OpenMP-Direktiven dabei nicht gestört wird. Damit sind dann ganz andere Probleme verbunden. Für eine Teilmenge dieser Anweisungsverschiebungen kann man Probleme ausschließen, wenn bestimmte Grundannahmen gelten. Durch zusätzliches Setzen bestimmter Einstellungen (Ausschalten von Optimierungen bei den AD-Werkzeugen) oder durch bestimmte Ersetzungsmaßnahmen (Kapitel 4.4.2) können auch die meisten anderen Probleme bewältigt werden. Für einen späteren Schnellzugriff und einer besseren Übersicht werden die Voraussetzungen für „unproblematische“ Anweisungsverschiebungen in dem folgenden Grundannahmenpaket 4.2 zusammengefasst.

Grundannahme 4.2 *Eigenschaften von unproblematischen Anweisungsverschiebungen*

- *Sie sind lokal und treten nur innerhalb der OpenMP-Direktivengrenzen auf.*
- *Durch Ersetzungen sind sie vermeidbar und werden damit unproblematisch.*
- *Es handelt sich um gutmütige Verschiebungen bei Schleifen-Direktiven.*

Für ein allgemeines Beispiel können die folgenden zwei Problemklassen dienen. Bei der ersten könnte es beispielsweise vorkommen, dass Berechnungen einer Variable von innerhalb einer parallelen Schleife nach außen verlegt wurden (Codeoptimierungen durch ein AD-

Werkzeug). Problematisch wird es dann, wenn für eine korrekte Berechnung der Variable vorher innerhalb der Schleife das *private*-Attribut benötigt wird und nun außerhalb *shared* erforderlich ist. Da diese Problematik durch Umstellungen in der ursprünglichen Anweisungsreihenfolge verursacht wird, wird sie im Weiteren als „Umstrukturierung entgegen der Reihenfolge“ (Kapitel 4.4.4) bezeichnet.

Die zweite Problematik kann unter anderem dann auftreten, wenn zu einer originalen Programmanweisung eine zusätzliche Ableitungsanweisung unmittelbar davor eingefügt worden ist. Das lässt sich nur schwer verhindern und wird im direkt nachfolgenden Kapitel „Verschiebung durch automatisch erzeugte Zusatzanweisungen“ gelöst.

Unabhängig von den Anweisungsverschiebungen gibt es eine bisher noch nicht erwähnte Fehlerquelle bei der AD-Programmcode-Transformation. Diese (eigentlich dritte) Klasse von Problemen ergibt sich, wenn neue Datenobjekte erstellt oder originale Variablen umgestellt werden und diese damit nicht mehr im korrekten Datensichtbarkeitsbereich-Kontext auftreten. Es wäre also denkbar, dass eine neue Variable global (d.h. für „geteilten“ Zugriff) angelegt wird, aber im Programmcode jeder Unterprozess die Arbeit mit einer „lokalen“ Kopie voraussetzt. Alle ursprünglichen Variablen, die beispielsweise aus einer lokalen Deklaration nach einer Programmcode-Transformation in den globalen Bereich verschoben wurden, könnten aus diesem Grund genauso zu einer fehlerhaften Parallelisierung führen.

Diese dritte Problemklasse trat bei den hier betrachteten AD-Werkzeugen nicht auf (siehe Grundannahme 4.1, Seite 39). Aufbauend auf der Grundannahme 4.1 wird im Weiteren vorausgesetzt, dass alle betrachteten Werkzeuge die nachfolgenden Konsistenzbedingungen (Grundannahme 4.3) einhalten.

Grundannahme 4.3 *Konsistenzbedingungen für die Programmcode-Transformation*

- *Keine Variable wird aus ihrem bisherigen Deklarationskontext heraus verschoben.*
- *Ableitungsobjekte, die für eine bestimmte in der Originalfunktion enthaltene Variable stehen, werden im selben oder einen analogen Deklarationskontext wie die originale Variable angelegt.*
- *Bei allen anderen neu angelegten Variablen handelt es sich um temporäre Zwischenvariablen, und sie werden lokal in der jeweiligen Routine angelegt.*

Hierbei bezieht sich der Deklarationskontext sowohl auf die Art und im weiteren Sinne auch auf den Ort. Im einzelnen heißt das, dass eine (originale) Variable deklariert in einem bestimmten globalen Bereich (z.B. benannter COMMON-Block) weder als lokale Variable umdeklariert, noch in einem anderen globalen Bereich verschoben werden darf.

4.4.2 Verschiebung durch automatisch erzeugte Zusatzanweisungen

Ab jetzt soll es um unterschiedliche Anweisungsverschiebungen entsprechend der ersten Problemklasse gehen. Ausgangspunkt ist eine kleine Rechen-Schleife mit einer einfachen explizit synchronisierten Berechnung im originalen Programmcode. Wie in der Einführung von Kapitel 4 erwähnt, kann es sich bei der synchronisierten Berechnung um eine (nicht parallele) nacheinander erfolgende Abarbeitung eines Programmabschnitts durch einzeln laufende Unterprozesse handeln. Für solche Synchronisationen kann es vorkommen, dass im erzeugten AD-Code (nach der Transformation) die Synchronisation ungünstig verschoben wurde. Links in der Abbildung 20 sieht man eine Beispiel-Schleife mit einer einfachen Summen-Berechnung auf u („geteilter“ Zugriff) und einer temporären Zwischenberechnung auf einer „lokalen“ Variable t (jeder Unterprozess benötigt hier seine eigene Kopie). Die Summen-Bildung selbst, ist durch eine *atomic*-Direktive (siehe Tabelle 2 im Anhang A) synchronisiert. Auf der rechten Seite sieht man den dazugehörigen korrekten Ableitungscode. Unter der Annahme, dass OpenMP vom AD-Werkzeug nicht unterstützt wird oder dem Werkzeug die Direktiven vorenthalten werden, ist von einem Fehlen der rot-invertierten

```
c$omp do
      do i = 1, N
         t = sin(y(i))
c$omp    atomic
         u = u + t
      end do
c$omp end do
```

```
c$omp do
      do i = 1, N
         g_t = cos(y(i)) * g_y(i)
         t = sin(y(i))
c$omp    atomic
         g_u = g_u + g_t
c$omp    atomic
         u = u + t
      end do
c$omp end do
```

Abbildung 20: Links: originale Schleife mit funktionstüchtiger Summenreduktion auf u; Rechts: dazugehöriger Ableitungscode mit zusätzlicher rot-invertierter *atomic*-Direktive. **Variablen**: t, g_t - Skalare mit *private*-Attribut; u, g_u - Skalare mit *shared*-Attribut; y, g_y - Vektoren mit *shared*-Attribut

atomic-Direktive auszugehen. Das führt dazu, dass anstatt von u nur sein Ableitungsobjekt g_u korrekt synchronisiert (durch *atomic*) und damit richtig aufsummiert wird.

Wie hier gezeigt wird, muss sichergestellt werden, dass das zweite rot-invertierte *atomic* ebenfalls an der richtigen Stelle vom AD-Werkzeug erzeugt wird, da sonst die Berechnung von u unsynchronisiert und damit fehlerhaft bleibt. Es gibt nun mehrere Möglichkeiten, diesen Problemfall zu verhindern, ohne auf zusätzliche Fähigkeiten des Transformations-Werkzeugs zuzugreifen oder manuell jedes Mal neu nachbessern zu müssen. Das Hauptproblem von *atomic* ist eigentlich, dass es sich bei seinem Direktiven-Bereich nur um eine einzelne nachfolgende Programmzeile handelt. Sie hat anders als z.B. das *critical*-Konstrukt keine Anfangs- und Ende-Direktive für einen größeren Bereich von Programmzeilen.

Ein Beispiel für die Verwendung einer *critical*-Direktive ist in Abbildung 21 gegeben. Auf der linken Seite sieht man (analog zum *atomic*-Beispiel) die originale Programm-

```
c$omp do
      do i = 1, N
         t = sin(y(i))
c$omp    critical
         u = u + t
c$omp    end critical
      end do
c$omp end do
```

```
c$omp do
      do i = 1, N
         g_t = cos(y(i)) * g_y(i)
         t = sin(y(i))
c$omp    critical
         g_u = g_u + g_t
         u = u + t
c$omp    end critical
      end do
c$omp end do
```

Abbildung 21: Links: originale Schleife mit funktionstüchtiger Summenreduktion auf u; Rechts: dazugehöriger Ableitungscode mit intakter Summenreduktion für u und g_u. **Variablen**: t, g_t - Skalare mit *private*-Attribut; u, g_u - Skalare mit *shared*-Attribut; y, g_y - Vektoren mit *shared*-Attribut

Schleife und rechts daneben den generierten Ableitungscode. Mit der Grundannahme 4.2 für Anweisungsverschiebungen hat sich hier für den *critical*-Bereich keine Programmanweisung aus dem korrekten Kontext heraus verschoben. Deshalb ist der transformierte AD-Code auf der rechten Seite korrekt, indem er sowohl u als auch g_u synchronisiert. Da eine *critical*-Region für einen ganzen Programmzeilen-Bereich genau dieselbe Synchronisation bedingt, wie sie die *atomic*-Direktive für nur genau eine Zeile hervorruft, ist eine Ersetzung die einfachste Lösung. Also ersetzt man auf dem originalen Programmcode alle *atomic*-Vorkommen durch *critical*-Regionen genau so, wie es der Übergang von Abbildung 20 zu 21 für die Beispielprogramm-Schleife zeigt (linke Seiten in beiden Abbildungen).

Gruppierung nach Problemgrad und Lösung

Im Nachfolgenden wird eine leichter zu verstehende Einteilung bezüglich des Grades des Problems und Art der Lösung erläutert. Für die folgenden Betrachtungen ist eine Trennung der Direktiven von den Klauseln sinnvoll, da sich die Problematik mit den Datensichtbarkeitsbereich-Klauseln hauptsächlich auf die Berechnung der eigentlichen Ableitungen beschränkt und weniger auf die Originalfunktion innerhalb des transformierten Codes. Deshalb werden zunächst die üblichen OpenMP-Direktiven hinsichtlich ihrer Problembehaftung gegenüber den Programmanweisungsverschiebungen gruppiert.

1. **gutmütig**: *parallel, do*
2. **nur lokal**: *sections, single, master, critical, ordered, barrier, task, taskwait*
3. **zu ersetzen**: *atomic, flush, workshare*

Innerhalb dieser drei Gruppen wurde die *threadprivate*-Direktive nicht eingeordnet, da sie in ihrer Wirkung eher den Datensichtbarkeitsbereich-Klauseln entspricht und deshalb in deren Kontext und Abschnitten in Kapitel 4.4.5 beschrieben wird. Die Direktiven *task* und *taskwait* gibt es erst ab OpenMP in Version 3.0 und definieren einzelne Teilaufgaben bzw. einen Warte-Bereich für die bisherige Abarbeitung.

Die erste Gruppe der „gutmütigen“ Direktiven hat den Vorteil, dass man etwaige Programmanweisungsverschiebungen über die Direktiven-Grenze hinaus und unter bestimmten Einschränkungen als unproblematisch ansehen kann. Dieser Vorteil wird anhand der OpenMP-Schleifen-Direktive *do* im Unterkapitel 4.4.4 (Problem bei „Umstrukturierungen entgegen der Reihenfolge“) genauer beleuchtet.

Im Gegensatz dazu bedingt die zweite Gruppe eine gewisse Einschränkung, denn es sind hier „nur lokale“ Verschiebungen innerhalb des Direktiven-Bereichs zulässig. Für die Synchronisationen mit eingrenzender Direktiven-Region (*sections, single, master, critical, ordered, task*) ist das leicht ersichtlich, da für sie analoge Aussagen wie für *critical* gelten (siehe Erläuterungen zu Abbildung 21). Wenn man für *barrier* und *taskwait* einfach den gesamten Programmbereich nach dieser Direktiven-Stelle (oder bis zu) als die dazu gehörende Direktiven-Region ansieht, führt das zu derselben Argumentation wie für *critical*. Bei den Arbeitsverteilungs-Direktiven in dieser zweiten Gruppe (z.B. *sections*) verhindert die Lokalität, dass damit keine Durchmischung der Teilaufgaben in den sonst unabhängigen Arbeitseinheiten entstehen kann (das wäre sonst problematisch).

Die letzte Gruppe (3.) von Direktiven wirft die stärksten Probleme auf, da sie oft auf unterschiedliche Art und Weise ersetzt werden müssen. Für die *atomic*-Direktive wurde bereits gezeigt, dass eine einfache Ersetzung durch ein *critical*-Konstrukt hilft. Dagegen kann man die *flush*-Direktive nicht einfach durch eine *barrier* ersetzen. Hier ist die Ersetzung mittels einer Dummy-Routine hilfreich. Diese Maßnahme wird zusammen mit der Handhabung der *workshare*-Direktive im nachfolgenden Unterkapitel 4.4.3 beschrieben.

Damit verbleiben noch die Klauseln, die nicht nur den Datensichtbarkeitsbereich beeinflussen. Dazu gehört beispielsweise die *reduction*-Klausel. Da sie indirekt bestimmte Rechenoperationen beinhaltet wird sie zusammen mit den „conditional compilation“-Programmanweisungen in Kapitel 4.5 ausgeführt. Andere Klauseln, die weder den Datensichtbarkeitsbereich noch Rechenoperationen beeinflussen (z.B. *schedule*), werden in dieser Arbeit nicht betrachtet, da sie die Berechnungen durch den transformierten Programmcode in ihrer Richtigkeit im Allgemeinen nicht beeinflussen.

4.4.3 Ersetzung weiterer OpenMP-Direktiven

Die Vorbereitungen für den vorübergehenden Einsatz einer Dummy-Routine veranschaulicht die Abbildung 22 in drei Schritten. Links befindet sich der originale Programmcode. Als erstes zerlegt man hier die *flush*-Direktive bezüglich ihrer einzelnen Variablen u und v (Mitte in Abbildung 22). Das ist ein einmaliger genereller Schritt der die vorüberge-

```
      u = 3.0 * u
      v = sin(v)
c$omp flush(u, v)
```

```
      u = 3.0 * u
      v = sin(v)
c$omp flush(u)
c$omp flush(v)
```

```
      u = 3.0 * u
      v = sin(v)
      call dummy_flush(u)
      call dummy_flush(v)
```

Abbildung 22: Links: originaler Programmcode mit Sammel-*flush*; Mitte: die Zerlegung in einzelne *flush*-Direktiven; Rechts: die leicht automatisierbare Dummy-Routinen-Ersetzung.
Variablen: u, v - Skalare mit *shared*-Attribut

hende automatische Dummy-Routinen-Ersetzung unterstützt, da dann nur noch Dummy-Varianten definiert werden müssen, die je einen einzelnen Übergabeparameter haben. Ganz rechts in dieser Abbildung sieht man dann den Einsatz der Dummy-Routine bevor der Code transformiert wird. Zu beachten ist hierbei, dass man solch eine Dummy-Routine für jeden Variablentyp anlegen sollte. Sicherheitshalber kann man dann diese Ersetzung für alle *flush*-Direktiven vornehmen und nicht nur für die ableitungs-aktiven Variablen. Dadurch ist es leichter, diesen Ersetzungsprozess zu automatisieren.

Angenommen, in diesem Beispiel ist nur v eine ableitungs-aktive Variable, dann wird entsprechend auch nur für v und g_v der Ableitungsaufruf „**call** *g_dummy_flush(v, g_v)*“ erzeugt. Abschließend kann man dann automatisch alle Aufrufe von „*dummy_flush*“ und „*g_dummy_flush*“ in Direktiven (`c$omp flush(u)` und `c$omp flush(v, g_v)`) zurück wandeln.

Ersatz für *workshare*

Zuletzt bleibt noch die *workshare*-Direktive übrig. Dabei handelt es sich um eine OpenMP-Arbeitsverteilung, die speziell auf die seit Fortran90 auftretenden Vektor-Operationen abgestimmt ist. Daraus ergibt sich zusätzlich zur generellen Anweisungsverschiebung noch ein weiteres Problem. Ein AD-Werkzeug könnte während der Codetransformation innerhalb des Direktiven-Bereiches nicht standardkonforme Zusatz-Programmanweisungen (z.B. skalare Zwischenberechnungen) einfügen. Da diese dann nicht mit der *workshare*-Direktive kompatibel sein müssen, sollte man die Direktive sicherheitshalber speziell ersetzen. Falls man eine Inkompatibilität ausschließen kann, ist *workshare* analog zu den Direktiven mit „nur lokalen“ Verschiebungen (z.B. *critical*) einzuordnen.

In der dieser Arbeit zugrunde liegenden Simulationssoftware wurde fast ausschließlich auf Fortran77-Programmcode [62, 90] gearbeitet, weshalb mit den *workshare*-Direktiven keine Erfahrungen gesammelt wurden. Trotzdem soll an dieser Stelle eine theoretische Betrachtung einen möglichen Lösungsweg aufzeigen. Dabei wird davon ausgegangen, dass man alle Fortran90/95-Vektoranweisungen [63, 91] in explizite Schleifen mit analoger OpenMP-Parallelisierung umformulieren kann. Das sieht dann beispielsweise wie in Abbildung 23 aus. Auf der linken Seite sieht man den ursprünglichen *workshare*-Code und

```
c$omp workshare
      u = u + v
      x = x * u
c$omp end workshare
```

```
c$omp do
      do i = 1, N
         u(i) = u(i) + v(i)
         x(i) = x(i) * u(i)
      end do
c$omp end do
```

Abbildung 23: Links: originale *workshare*-Direktive; Rechts: umgewandelte Ersatz-Schleife.
Variablen: u, v, x - Vektoren mit *shared*-Attribut

auf der rechten Seite den umgewandelten Ersatz-Code. Damit beruht der umgewandelte Programmcode auf besser handhabbaren OpenMP-Direktiven und passt sich damit in die anderen Lösungsstrategien ein.

Insgesamt lassen sich an dieser Stelle die Maßnahmen für die „zu ersetzenden“ Direktiven (Gruppe 3) in dem folgenden Maßnahmenpaket 4.4 zusammenfassen.

Maßnahmen 4.4 *Ersetzung von OpenMP-Direktiven*

- *Alle atomic-Direktiven jeweils durch eine critical-Region ersetzen.*
- *Alle flush-Direktiven mit vorübergehenden Dummy-Routinen (automatisiert) ersetzen.*
- *Alle workshare-Direktiven vorsichtshalber in explizite Ersatz-Schleifen mit entsprechender Parallelisierung umwandeln.*

4.4.4 Problem bei „Umstrukturierungen entgegen der Reihenfolge“

Diese Teilproblemklasse tritt bekanntermaßen nur bei so genannten „Quellcode zu Quellcode“-Transformations-Werkzeugen (engl. „source to source“) auf. Dagegen sind alle AD-Werkzeuge, die auf „Ersetzen der Rechenoperatoren“ (engl. „operator overloading“) beruhen, nicht von dieser Problematik betroffen, da sie in keiner Weise eine Quellcode-Optimierung oder sonstige größere Umstrukturierung vornehmen.

Die anderen AD-Werkzeuge (auch die in dieser Arbeit verwendeten) stellen durchaus die Anweisungsreihenfolge um. Allerdings sind viele der üblichen Umstrukturierungen lediglich nur geschickte Anweisungszerlegungen mit relativ lokalen Auswirkungen (Einhaltung von Grundannahme 4.2). Übrig bleiben diejenigen Umstrukturierungen, bei denen über OpenMP-Direktiven hinaus die Programmanweisungen verschoben werden.

Die nachfolgenden Abschnitte befassen sich mit dieser Verschiebungsproblematik am Beispiel einer OpenMP-Schleifendirektive. Dabei wird zuerst ein „gutmütiger“ Fall gezeigt, der resistent gegenüber möglichen Verschiebungsproblemen ist. Daran wird dann der Unterschied in einem nur leicht modifizierten Beispiel erarbeitet, welches zu Problemen führt. Hieraus ergeben sich dann zusätzliche Richtlinien, die es ermöglichen, für die gutmütigen Direktiven *do* und *parallel* auch übergreifende Umstrukturierungen seitens des Transformations-Werkzeugs hinzunehmen. Das wäre wünschenswert, wenn man von den Optimierungen durch die Codetransformation profitieren möchte.

Die nachfolgende Abbildung 24 veranschaulicht anhand einer Programm-Schleife diese gutmütigen Umstrukturierungen. Hierbei wird zuerst auf die linke Seite eingegangen (auch für Abbildung 25) und dann auf die Problemfälle der jeweiligen rechten Seite. Links oben in Abbildung 24 ist als erstes die originale Funktion dargestellt (eine einfache Schleife). Wichtige Grundannahme ist hier die *shared*-Voreinstellung für die Variablen y und v. Weiterhin wird die Variable *dlocal* als „lokal“ deklariert (mit *private*-Attribut). Unter diesem Codebeispiel (links-unten in der Abbildung 24) befindet sich der dazugehörige automatisch erzeugte Ableitungscode. Neu ist hier unter anderem die Variable *g_dlocal*, welche genau wie ihr originales Objekt *dlocal* auch „lokal“ deklariert ist. Analog dazu haben die Ableitungsobjekte *g_y* und *g_v* genau wie ihre Originale das *shared*-Attribut per Voreinstellung. Hierbei wurde noch keine Umstrukturierung entgegen der ursprünglichen Anweisungsreihenfolge vorgenommen.

Im Gegensatz dazu zeigt der linke Codeausschnitt in Abbildung 25 eine Möglichkeit für einen optimierteren Ableitungscode (durch Umstrukturierungen). Wie man hier sieht, wurde die Berechnung von *dlocal* und *g_dlocal* aus der Schleife heraus gezogen (Umstrukturierung entgegen der ursprünglichen Anweisungsreihenfolge), ohne die Richtigkeit der Parallelisierung zu beeinflussen. Hierbei könnten beide Variablen auch *shared* sein, ohne dass die Richtigkeit schaden nehmen würde. Enthält die Parallelisierung (wie hier gezeigt) keinerlei OpenMP implizierte Rechenoperationen, keine zusätzlichen Synchronisationen und keine expliziten Datensichtbarkeitsbereichs-Attribute, ist der generierte Ableitungscode trotz Optimierungen äußerst robust bezüglich der Parallelisierung. Der Grund dafür

```
c$omp do
      do i = 1, N
         dlocal = sin(v)
         y(i) = dlocal * dlocal
      end do
c$omp end do
```

```
c$omp do private(dglobal)
      do i = 1, N
         dglobal = sin(v)
         y(i) = dglobal * dglobal
      end do
c$omp end do
```

```
c$omp do
      do i = 1, N
         g_dlocal = cos(v) * g_v
         dlocal = sin(v)
         g_y(i) = 2 * dlocal * g_dlocal
         y(i) = dlocal * dlocal
      end do
c$omp end do
```

```
c$omp do private(dglobal)
      do i = 1, N                    Datenwettlauf
         g_dglobal = cos(v) * g_v
         dglobal = sin(v)
         g_y(i) = 2 * dglobal * g_dglobal
         y(i) = dglobal * dglobal
      end do
c$omp end do
```

Abbildung 24: Links-oben: Originalfunktion mit „lokaler“ Zwischenvariable; Links-unten: entsprechender Ableitungscode ohne Optimierungen; Rechts-oben: Originalfunktion mit global deklarierter Zwischenvariable („geteilter“ Zugriff) - nur vorübergehend innerhalb der Schleife „lokal“; Rechts-unten: entsprechender Ableitungscode mit gutmütigen „Datenwettlauf“ (engl. „data-race“) - siehe Abschnitt weiter unten.
Variablen: *dlocal*, *g_dlocal* - Skalare mit *private*-Attribut; *v*, *g_v*, *dglobal*, *g_dglobal* - Skalare mit *shared*-Attribut; *y*, *g_y* - Vektoren mit *shared*-Attribut

```
      g_dlocal = cos(v) * g_v
      dlocal = sin(v)
c$omp do
      do i = 1, N
         g_y(i) = 2 * dlocal * g_dlocal
         y(i) = dlocal * dlocal
      end do
c$omp end do
```

```
      g_dglobal = cos(v) * g_v
      dglobal = sin(v)    Anfangswert = 0
c$omp do private(dglobal)
      do i = 1, N
         g_y(i) = 2 * dglobal * g_dglobal
         y(i) = dglobal * dglobal
      end do
c$omp end do
```

Abbildung 25: Links: Ableitungscode mit Optimierungen; Rechts: optimierter Ableitungscode mit fehlerhafter Parallelisierung.
Variablen: *dlocal*, *g_dlocal* - Skalare mit *private*-Attribut; *v*, *g_v*, *dglobal*, *g_dglobal* - Skalare mit *shared*-Attribut; *y*, *g_y* - Vektoren mit *shared*-Attribut

ist, dass entsprechend den Vorzugsvarianten der „OpenMP-hiding“-Technik alle Variablen bei dem Bereichsübergang ihre Werte beibehalten. Damit bleibt eine Verschiebung über die Bereichsgrenze genau dann korrekt, wenn sie auch im seriellen Kontext richtig war.

Im Kontrast zu diesem gutmütigen Beispiel, soll jetzt anhand einer einfachen expliziten Datensichtbarkeitsbereich-Änderung die damit einhergehende Problematik veranschaulicht werden. Ein Beispiel dafür wird auf der jeweiligen rechten Seite der Abbildungen 24 und 25 dargestellt. Es ist im Wesentlichen analog zur linken Seite aufgebaut, allerdings mit einer wichtigen Änderung. Aus der Variable *dlocal* wurde nun *dglobal*, was sie damit als globale Variable und somit mit „geteilten“ Zugriff kennzeichnen soll. Aus Entwicklersicht hat sie nur außerhalb der Rechenschleife das *shared*-Attribut, denn für die Verwendung innerhalb der Schleife wurde *dglobal* vorübergehend und explizit mit dem *private*-Attribut deklariert.

Gutmütiger „Datenwettlauf“

Entsprechend sieht zum neuen Originalcode (rechts-oben in Abbildung 24) der generierte Ableitungscode wie darunter (rechts-unten) aus. Ein AD-Werkzeug, welches OpenMP nicht hinreichend unterstützt oder dem die Direktiven und Klauseln vorenthalten werden, belässt hier wahrscheinlich die OpenMP-Anweisung „`c$omp do` *private(dglobal)*“, wie sie

ist. Das führt dazu, dass für *dglobal* mit *private* gearbeitet wird, während für *g_dglobal* in der Schleife weiterhin das *shared*-Attribut gilt. Für *g_dglobal* würde es hier zu einem so genannten „kritischen Datenwettlauf“ (engl. „data-race“) kommen, da innerhalb der Schleife alle OpenMP-Unterprozesse gleichzeitig auf die selbe Variable *g_dglobal* schreiben würden. Da in diesem Beispiel alle Unterprozesse aber den selben Wert hineinschreiben, hätte das hier keine negativen Folgen („gutmütig“) für die Berechnung der Originalfunktion oder deren Ableitung.

Im Gegensatz dazu würde eine tiefer greifende Codeoptimierung (Umstrukturierung) zu einer inkorrekten Parallelisierung führen. Im Vergleich zu vorher (links in Abbildung 25) sorgt nun die explizite *private*-Deklaration in dem rechts gelegenen Programmbeispiel der Abbildung 25 für eine fehlerhafte Berechnung. Da *dglobal* noch genauso wie vorher als lokale Variable deklariert ist, führt die Verschiebung der Initialisierung von *dglobal* dazu, dass zu Beginn der Schleife ein undefinierter oder zu Null initialisierter Anfangswert verwendet wird. An dieser Stelle muss die Parallelisierung in jedem Fall manuell nach gebessert werden.

Deswegen gehört es zu dieser Strategie, solche expliziten Datensichtbarkeitsbereichs-Attribute in allen Programmteilen zu vermeiden, die zur Weiterverarbeitung an ein AD-Werkzeug weiter gereicht werden sollen.

Richtlinien 4.2 *Vermeidung expliziter Datensichtbarkeitsbereichs-Attribute*

- *Keine Verwendung von expliziten Datensichtbarkeitsbereichs-Attributen. Das ist erzielbar*
  - *durch die Ausnutzung der Vorzugsvarianten (OpenMP-Standard Voreinstellung),*
  - *oder durch die Anwendung des Maßnahmenpaketes 4.2.*

Durch Anwendung des Maßnahmenpaketes 4.2 kann man ein explizites (vorübergehendes) Umdeklarieren der Datensichtbarkeitsbereichs-Attribute vermeiden. Im letzten Beispiel (rechte Seite in Abbildung 25) könnte man dies sehr einfach nachbessern, indem man an der Stelle von *dglobal* eine temporäre neue lokale Variable verwendet. Das entspricht für dieses Beispiel genau der Lösung, wie sie die linke Seite in den Abbildungen 24 und 25 bereits zeigte.

4.4.5 Wertekonsistenz bei dem Bereichsübergang

Bis hierhin wurde an Beispielen gezeigt, welche Probleme explizite Datensichtbarkeitsbereich-Attribute nach sich ziehen können, und wie man einige Probleme der OpenMP-Direktiven umgehen kann. Geht man an dieser Stelle noch einmal zurück auf die in Kapitel 4 beschriebenen Probleme bei dem Übergang einer Variablen in eine neue parallele Region oder in eine OpenMP-Arbeitsverteilungs-Direktive, dann erkennt man, dass für die nicht-bevorzugten Varianten ($\ddot{U}_{pspP}$, $\ddot{U}_{pPpP}$, $\ddot{U}_{sPpP}$ und $\ddot{U}_{sPpA}$, siehe Tabelle 1 auf Seite 28) ein explizites Setzen von Datensichtbarkeitsbereichs-Attributen kennzeichnend ist.

In den nachfolgenden Abschnitten wird deshalb das explizite Setzen von Datensichtbarkeitsbereich-Attributen bzw. deren mögliche Vermeidung untersucht. Zur Vereinfachung geht man hier davon aus, dass dem Transformations-Werkzeug die Parallelisierung vorenthalten wird. Davon unabhängig bleibt die grundlegende Voraussetzung weiterhin notwendig, dass alle Anweisungsverschiebungen aus serieller Sicht korrekt bzw. konsistent sein müssen. Zusätzlich ist zu beachten, dass es nicht nur einfache Klauseln für das Setzen von Datensichtbarkeitsbereichs-Attributen gibt, sondern auch Kombinationen mit dem systematischen Hinein- oder Hinauskopieren der Anfangs- und Endwerte.

Darüber hinaus gibt es andere Klauseln (z.B. *reduction*), die nur bestimmte Kombinationen von Datensichtbarkeitsbereichs-Attributen zwischen außerhalb und innerhalb des neuen Direktiven-Bereiches zulassen. Diese fassen oft bestimmte Daten in Variablen zusammen (Aggregation) und geben die Ergebniswerte anschließend nach außen weiter. Die nachfolgende Aufzählung klassifiziert die einzelnen OpenMP-Klauseln und die *threadprivate*-Direktive in vier Gruppen.

K1: **nur Attribute**: *private, shared, threadprivate*

K2: **Attribute und Hineinkopieren**: *firstprivate, copyin*

K3: **Attribute und Herauskopieren**: *lastprivate, copyprivate*

K4: **Attribute und Daten-Aggregation**: *reduction*

Wie schon angedeutet, gliedert sich die *threadprivate*-Direktive in gewisser Weise in die erste Gruppe K1 ein. Sie wird, anders als bei den Klauseln, nicht beim Bereichsübergang spezifiziert, sondern bei der Variablen-Deklaration. Diese Direktive erzwingt eine *private*-Handhabung für alle einschließenden Variablen für jede neu auftretende parallele Region, insbesondere auch bei inneren (geschachtelten) parallelen Regionen. Bei diesem Übergang verhalten sich *threadprivate*-Variablen wie die $\ddot{U}_{pPpP}$-Variante aus Kapitel 4. Im Gegensatz dazu gilt für den Übergang zu einer Arbeitsverteilungs-Direktive innerhalb einer parallelen Region, dass der *private*-Status beibehalten wird, vergleichbar mit der $\ddot{U}_{pPpA}$-Variante.

Die letzte Gruppe K4 ist etwas komplizierter, weshalb ihr ein eigenes Kapitel 4.5 gewidmet wird. Bis dahin soll es nachfolgend um die Klauseln aus K1 bis K3 gehen. Sie unterscheiden sich nur durch das implizite Werte-Kopieren. Unabhängig davon gilt, dass bei einem Übergang in einen Bereich mit zusätzlichen neuen lokalen Kopien ein generelles anfängliches Hinein- und späteres Herauskopieren den Programmcode gegenüber Anweisungsverschiebungen stabilisieren kann. Das ist im Normalfall mit einem zusätzlichen Berechnungsaufwand verbunden und muss deshalb von Fall zu Fall abgewogen werden. Im Folgenden wird die Handhabung der verbliebenen Klauseln anhand ihres Einsatzzweckes und möglichen Vorkommens erläutert. Dazu werden weiter unten systematisch alle 8 gültigen Übergangsszenarien (vergleiche Tabelle 1, Seite 28), beginnend mit den Vorzugsvarianten, untersucht. Die nachfolgenden Definitionen sind die Grundlage für alle weiteren Betrachtungen:

- m - ist die Anzahl der Unterprozesse der äußeren parallelen Region.
- n - ist die Anzahl der Unterprozesse der inneren parallelen Region.
- $A_1, A_2 \cdots A_m$ - bezeichnet die m Unterprozesse der äußeren parallelen Region.
- $I_{11}, I_{21} \cdots I_{mn}$ - bezeichnet die Unterprozesse der inneren parallelen Regionen, wobei jede der m Gruppen (äußere parallele Region) aus genau n Unterprozessen besteht.
- $I_{k1}, I_{k1} \cdots I_{kn}$ - sind die n inneren Unterprozesse zu dem äußeren Unterprozess A_k.

Angenommen, man hat eine neue parallele Region (Unterprozesse $\in \{A_1, A_2 \cdots A_m\}$ und Prozessanzahl m) und die zu betrachtende Variable v ist „lokal“. Dann gibt es m entsprechende Kopien $\{v_1, v_2 \cdots v_m\}$ von v, siehe dazu die Abbildung 26. Eine runde Klammer bedeutet hier den Übergang in eine parallele Region und die Überschrift „*private*“ deklariert die Datensichtbarkeitsbereich-Eigenschaft der Variablen für diesen Bereich. Ein gestrichelter Pfeil von z.B. v nach v_1 kennzeichnet im Folgenden einen möglichen Werte-Übergang bzw. den Variablenfluss. Wenn dies nicht eindeutig ist, weil möglicherweise ein weiterer Pfeil auch zu v_m führt, wird der Variablen-Wert von v nicht unbedingt übernommen und die Anfangswerte von $v_1 \cdots v_m$ können damit undefiniert zu Beginn des neuen Bereiches sein.

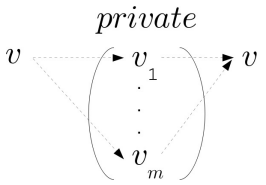


Abbildung 26: Ein *private*-Übergang der Variablen v in eine innere parallele Region.

Die *firstprivate*-Klausel aus Gruppe K2 kann deshalb verwendet werden, um zu Beginn einer parallelen Region (und einigen Arbeitsverteilungs-Direktiven) explizit dafür Sorge zu tragen, dass der Wert von v als erstes in die jeweilige lokale Kopie initialisiert wird. Eine Ausnahme bildet die *copyin*-Klausel, sie muss statt dessen verwendet werden, wenn es sich um eine *threadprivate*-Variable handelte. Analog markiert die Mündung mehrerer Pfeile auf ein Ziel auch die Uneindeutigkeit, welcher letzte Wert von $v_1 \cdots v_m$ in die Zielvariable v am Ende übernommen wird. Entsprechend kann dies auch zu einen undefinierten Anfangswert für die Zielvariable v am Ende bzw. außerhalb des Bereiches führen (nach der geschlossenen runden Klammer). Entsprechend hilft dann die *lastprivate*-Klausel der Gruppe K3, um sicher zu stellen, dass ein Endwert der lokalen Variablen $v_1 \cdots v_m$ als definierter Anfangszustand für außerhalb in v übernommen wird. Für den *single*-Direktivenbereich gilt der Sonderfall, dass man dazu die *copyprivate*-Klausel verwenden muss.

Dieses letzte Gesamtbeispiel lässt sich nun zu der Übergangsvariante $\ddot{U}_{pPsP}$ erweitern (links Abbildung 27). Für sie gilt, es gibt eine zusätzliche innere parallele Region

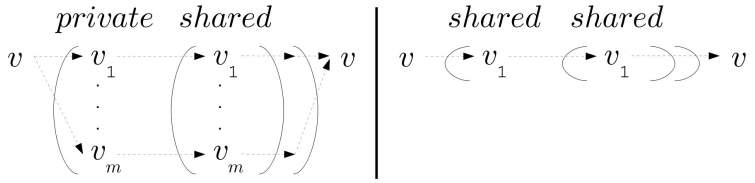


Abbildung 27: Übergang der Variablen v für eine innere parallele Region. Links: für Übergang $\ddot{U}_{pPsP}$; Rechts: für Übergang $\ddot{U}_{sPsP}$; Vergleiche Tabelle 1 auf Seite 28.

mit einer Gruppengröße n (Anzahl der Unterprozesse) und v soll in dieser das *shared* Datensichtbarkeitsbereichs-Attribut besitzen. Hierbei gehört zu einem äußeren Unterprozess A_k mit $(1 \leq k \leq m)$ entsprechend eine innere Gruppe $\{I_{k1}, I_{k2} \cdots I_{kn}\}$ von Unterprozessen. Für diesen Fall $\ddot{U}_{pPsP}$ (links Abbildung 27) greifen dann alle inneren Unterprozesse $I_{k1}, I_{k2} \cdots I_{kn}$ auf das selbe v_k des Unterprozesses A_k zu. Deswegen gib es auch auf der Ebene der inneren parallelen Region trotzdem insgesamt nur m Kopien von v. Oder anders ausgedrückt, es gibt für jede der m inneren Unterprozess-Gruppen je nur eine Variable v_k.

Wenn nun eine v_k-berechnende Programmanweisung aus dem inneren Bereich in den Äußeren verschoben wird (z.B. durch eine Optimierung), muss es sich um eine Berechnung handeln, die aus serieller Sicht verschiebbar ist. Das gilt eigentlich nur, wenn es sich dabei um eine Berechnung handelt, die für alle inneren Unterprozesse einer Gruppe gleich ist. Hierbei handelt es sich sozusagen um überflüssige Rechenarbeit, da ein Einzelner stellvertretend für alle anderen Unterprozesse $I_{k1}, I_{k2} \cdots I_{kn}$ die Berechnung übernehmen kann. In diesem Fall kann der Unterprozess A_k (außen) diese einmalige Berechnung auf v_k durchführen. Da in der inneren Region dieser von A_k berechnete Wert v_k für alle Unterprozesse dieser k-ten Gruppe gleich ist (innen *shared*), ist die Verschiebung unproblematisch. Natürlich funktioniert das analog auch in umgekehrter Reihenfolge bei einer Verschiebung von außen nach innen mit dem Unterschied, dass alle inneren Unterprozesse einer Gruppe k dann überflüssigerweise dieselbe Berechnung auf demselben v_k vornehmen würden. Dieser kritische „Datenwettlauf“ kann aber ignoriert werden, weil das v_k (wegen der angenommenen seriellen Verschiebbarkeit) innerhalb des inneren Bereiches nur gelesen oder nur geschrieben wird. Das führt dann höchstens zu Einbußen in der Rechenleistung. Für

die $\ddot{U}_{pPsP}$-Variante ist insgesamt noch anzumerken, dass man auf Grund des Werteerhaltes keine expliziten Direktiven mit Kopierfunktionalität benötigt. Das ist ein wichtiger Vorteil und gilt für alle vier Vorzugsvarianten ($\ddot{U}_{pPsP}$, $\ddot{U}_{sPsP}$, $\ddot{U}_{pPpA}$ und $\ddot{U}_{sPsA}$).

Die Variante $\ddot{U}_{sPsP}$ (rechts in Abbildung 27) unterscheidet sich nur darin, dass v auch in der äußeren Region *shared* ist und damit insgesamt nur ein einheitliches v (v_1) für alle äußeren und inneren Unterprozesse ($A_1 \cdots A_m, I_{11} \cdots I_{mn}$) existiert. Damit ist klar, wenn hier die Berechnung von v aus serieller Sicht verschiebbar ist, dann wird auch hier bei einer Verschiebung über die innere Bereichsgrenze (in beide Richtungen) dieselbe Rechenarbeit gegebenenfalls dupliziert oder verringert. Der Wert von v bleibt wie für die $\ddot{U}_{pPsP}$-Variante bei dem inneren Übergang erhalten. Damit ist die eigentliche Berechnung immer noch korrekt.

Der Unterschied zu den Varianten $\ddot{U}_{pPpA}$ und $\ddot{U}_{sPsA}$ besteht nun darin, dass hier keine neue innere Unterprozess-Gruppe erzeugt wird, sondern ein neuer Bereich mit einer OpenMP-Arbeitsverteilung. Die Abbildung 28 zeigt beide Varianten, wobei der Arbeitsverteilungsbereich durch eckige Klammern eingegrenzt wird. Um auch hier außer-

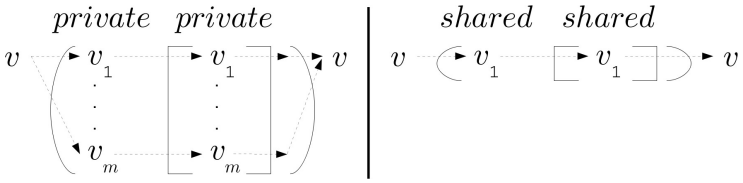


Abbildung 28: Übergang der Variablen v für eine OpenMP-Arbeitsverteilung. Links: für Übergang $\ddot{U}_{pPpA}$; Rechts: für Übergang $\ddot{U}_{sPsA}$; Vergleiche Tabelle 1 auf Seite 28.

halb und innerhalb des definierten Bereichs auf die selben Variablen(-Kopien) und damit auf die selben Werte zuzugreifen, müssen solche Variablen auf beiden Seiten das selbe Datensichtbarkeitsbereichs-Attribut aufweisen. Die $\ddot{U}_{sPsA}$ Variante entspricht deshalb genau der $\ddot{U}_{sPsP}$ mit außen und innen *shared*-Attribut für die Variable v. Während die $\ddot{U}_{pPsP}$-Variante innen ein *shared* benötigt, wenn außen *private* deklariert ist (Arbeit auf v_k), erreicht man dies bei einer OpenMP-Arbeitsverteilung genau dann, wenn auch innerhalb das *private*-Attribut beibehalten wird. Das entspricht genau der Kombination, wie sie in der $\ddot{U}_{pPpA}$-Variante gefordert wird. Damit kann man die Betrachtungen zur Richtigkeit von $\ddot{U}_{pPpA}$ und $\ddot{U}_{sPsA}$ ganz analog zu $\ddot{U}_{pPsP}$ und $\ddot{U}_{sPsP}$ abschließen. Ein direkter Vergleich zwischen den Abbildungen 27 und 28 in Hinblick des Zugriffs und Vorkommens aller Variablenkopien von v verdeutlicht das. Im Umkehrschluss kann man daraus folgern, dass es nur noch dann kritisch für eine korrekte Berechnung werden kann, wenn nach einer Verschiebung über die Grenze nicht mehr auf dieselben v_k oder v zugegriffen wird.

Dazu genügt es, den Wechsel zwischen verschiedenen parallelen Region auf einer Ebene zu betrachten ($\ddot{U}_{pspP}$-Variante aus Kapitel 4). In Abbildung 29 wird unter anderem ein v_1 (bzw. v) aus einer *shared*-Region zu den Kopien $v_1 \cdots v_m$ in einer *private*-Region. Allerdings enthält dieser Übergang von v zu v_k das Problem, dass der Wert von v irgendwie

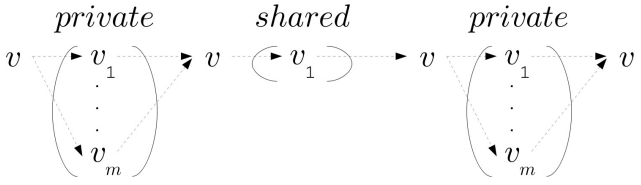


Abbildung 29: Übergang $\ddot{U}_{pspP}$ der Variablen v für verschiedene parallele Regionen auf gleicher Ebene. Vergleiche Tabelle 1 auf Seite 28.

in v_k übernommen werden können muss. Das gilt entsprechend auch umgekehrt bei einem Übergang von v_k zu v. Im Allgemeinen ist das nicht immer erforderlich, wie z.B. bei einer *private*-Region, in der v_k einfach überschrieben wird, ohne darin den vorherigen Wert von

v zu beachten. Wie schon erwähnt, kann man mit Hilfe der Klauseln aus K2 und K3 (Seite 52) explizit veranlassen, die Werte hinein bzw. heraus zu kopieren. Solche zusätzlichen OpenMP-Klauseln will man bei der „OpenMP-hiding"-Technik aber gerade vermeiden, da man sie sonst auch auf neue (automatisch erzeugte) Datenobjekte nachpflegen muss. Wie im Kapitel 4 bereits angedeutet wurde, soll die Problematik der $\ddot{U}_{pspP}$-Variante dadurch gelöst werden, indem man *shared* als Standard-Deklaration für alle von außen übergebenen Variablen v annimmt und die *private*-Regionen mit Hilfe von neuen lokalen Variablen v_lokal versorgt. Das entbindet den Programmcode bisher aber nur von der expliziten Deklaration von *private(v)*. Den Ersatz für das OpenMP-interne Hinein- und Herauskopieren kann man durch eine explizite Ausformulierung wie $v\_lokal = v$ und $v = v\_lokal$ erreichen.

Jetzt kann man die anderen nicht-bevorzugten Varianten $\ddot{U}_{pPpP}$ und $\ddot{U}_{sPpP}$ betrachten, siehe linke und mittlere Variante in Abbildung 30. Beiden ist gleich, dass es sich um einen

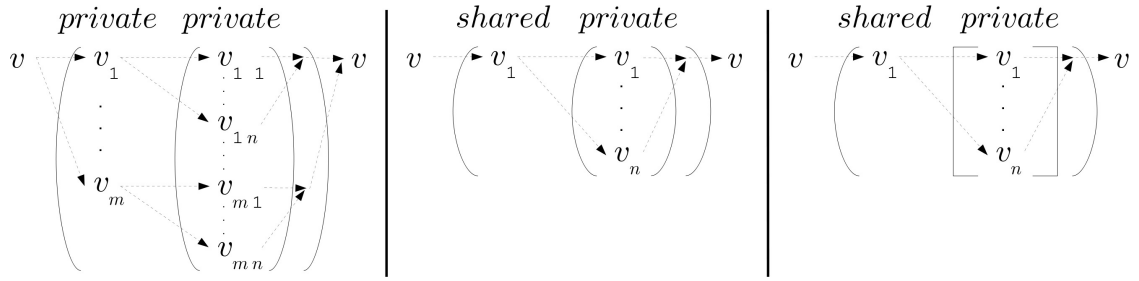


Abbildung 30: Verschiedene Übergangsvarianten der Variablen v. Links: für Übergang $\ddot{U}_{pPpP}$; Mitte: für Übergang $\ddot{U}_{sPpP}$; Rechts: für Übergang $\ddot{U}_{sPpA}$; Vergleiche Tabelle 1 auf Seite 28.

inneren Übergang in eine neue parallele Region handelt und die Variable $v_1 \cdots v_m$ (außen *private*) bzw. nur v_1 (außen *shared*) nun zusätzlich *private* in der inneren Region sein soll. Das ist wie bei der $\ddot{U}_{pspP}$-Variante auch nur durch eine explizite *private*-Deklaration (innen) zu erreichen. Für jedes v_k aus $v_1 \cdots v_m$ (bzw. das einzelne v_1) werden damit für alle Unterprozesse einer inneren Gruppe neue Kopien $v_{k1} \cdots v_{kn}$ bzw. $v_1 \cdots v_n$ angelegt. Sofern kein OpenMP impliziertes Werte-Kopieren angewendet wird, handelt es sich damit prinzipiell um die gleiche Problemstellung wie für die $\ddot{U}_{pspP}$-Variante. Deswegen kann man es auch genauso umgehen: neue lokale Variable anlegen und explizites Kopieren falls erforderlich.

Wie man leicht sieht, handelt es sich bei der noch nicht betrachteten $\ddot{U}_{sPpA}$-Variante (ganz rechts in Abbildung 30) um ein analoges Problem zu $\ddot{U}_{sPpP}$ (Mitte) für eine OpenMP-Arbeitsverteilung mit allen Konsequenzen und Lösungsmöglichkeiten.

Zusammenfassend kann man sagen, dass das implizite Verwenden der Datensichtbarkeitsbereich-Attribute entsprechend den Vorzugsvarianten unproblematisch ist. Alle nicht bevorzugten Varianten sind ebenfalls im Kontext einer Programmcode-Transformation korrekt anwendbar, wenn die explizite Datensichtbarkeitsbereich-Deklarationen und damit alle Klauseln und Direktiven aus K1 bis K3 (Seite 52) durch entsprechende Ersatz-Programmierung umgangen werden. Hierbei kann insbesondere auf alle *shared*-Klauseln verzichtet werden, da sie im Rahmen der realisierbaren Varianten überflüssig sind. Der generierte Transformationscode enthält damit eine funktionstüchtige Parallelisierung analog zu dem aus der originalen Modell-Simulation. Das gilt auch weiterhin nur für die Programmanweisungen der originalen Modell-Simulation und nicht für zusätzlich generierte Ableitungsanweisungen.

Betrachtet man alle bisher erläuterten OpenMP-Direktiven und Klauseln bzw. die dazugehörigen Strategien zum Umgehen etwaiger Probleme, dann wurde bis hierher ausschließlich dafür gesorgt, dass die Parallelisierung der originalen Funktion nach der Programmcode-Transformation erhalten bleibt. Das nächste Unterkapitel nimmt sich nun der Erweiterung der Parallelisierung auf die zusätzliche Ableitungsfunktionalität an.

4.5 Ableitungs-spezifische Reduzierung von OpenMP-Anweisungen

Eine einfache und doch effiziente Erweiterung der Parallelisierung auf den Code der Ableitungsberechnung erhält man, wenn die folgenden Grundannahmen 4.4 erfüllt werden können.

Grundannahme 4.4 *Erweiterung der Parallelisierung auf den Ableitungscode*

- *Die Datensichtbarkeitsbereichs-Attribute der Ableitungsobjekte gleichen denen der dazugehörigen originalen Objekte.*
- *Die Datensynchronisation (mit impliziten Rechenoperationen) auf den Ableitungsobjekten wird im selben Kontext und analog zu den dazugehörigen originalen Objekten vorgenommen.*
- *Die Transformation der expliziten Rechenoperationen auf den Ableitungsobjekten ist korrekt (entspricht der Ableitung der originalen Rechenoperationen).*

Mit diesen Grundannahmen unterscheidet sich das Ableitungsobjekt nur in dem eigentlich berechneten Wert, nicht aber in der Datenkonsistenz, von dem der originalen Variable. Daraus kann man folgern, dass innerhalb einer Parallelisierung jeder Unterprozess genau dann auf seine Ableitungswerte korrekt zugreifen kann, wenn dies gerade auch für seine originalen Funktionswerte gilt. Damit ergibt sich insgesamt eine fehlerfreie Parallelisierung auf den Ableitungsobjekten, wenn die Parallelisierung auch für die originale Funktion korrekt ist. Letzteres war aber Voraussetzung, womit sich der Kurzbeweis schließt.

Angenommen, sowohl die Datensichtbarkeitsbereichs-Attribute, wie auch eine korrekte Transformation der Rechenoperation können vorausgesetzt werden, dann bleibt noch die Forderung nach der Datensynchronisation übrig. Für einige OpenMP-Direktiven (z.B. *flush*) wurde bereits erläutert, wie die Werte-Synchronisation auf die Ableitungsobjekte ausgedehnt werden kann. Offen sind aber noch OpenMP-Klauseln und Konstrukte, in denen (implizite) Rechenoperationen mit einer Synchronisation kombiniert werden. Hier muss man eine explizite Trennung der Rechenoperation von der Synchronisation durchführen und für beide sicherstellen, dass bei der Programmcode-Transformation auch die Rechenoperation berücksichtigt wird. So etwas wird im nachfolgenden Unterkapitel für die *reduction*-Klausel gezeigt. Andererseits gibt es Konstrukte, die eine Art versteckte Rechenoperation enthalten. Damit befasst sich dann das darauf folgende Unterkapitel 4.5.2.

Um beide Aspekte einfacher handhaben zu können, wird im Folgenden angenommen, dass keine expliziten Datensichtbarkeitsbereichs-Attribute mehr verwendet werden und der generierte Transformationscode eine korrekte Parallelisierung der originalen Modell-Simulation durch die in den letzten Kapiteln beschriebenen Maßnahmen enthält. Außerdem wird vorausgesetzt, dass neu eingefügte Datenobjekte (z.B. Ableitungsvariablen) analog zu ihren originalen Objekten die gleichen Datensichtbarkeitsbereichs-Attribute aufweisen. Es lassen sich damit die wichtigsten verbliebenen Probleme auf eine grundlegende Ursache zurückführen.

In diesen Fällen werden einzelne Variablen spezifiziert und nur auf diesen werden bestimmte Rechenoperationen bzw. Synchronisationen ausgeführt. Das trifft nicht nur auf die in der Gruppe K4 (Seite52) enthaltene *reduction*-Klausel zu, sondern auch auf die so genannten „conditional compilation“-Programmanweisungen. Gemeinsam ist beiden, dass sie genauso wie die *flush*-Direktive sich auf einzelne Variablen beziehen können und nur deren Werte-Konsistenz beeinflussen. Im Gegensatz zu *flush* reicht es aber nicht aus, eine Dummy-Routine vorübergehend einzusetzen. Diese Möglichkeit kann man verwenden, wenn man sich bei den allgemeinen Variationsmöglichkeiten einschränken würde. Das wäre aber kein genereller Ansatz mehr, weswegen sich die nachfolgenden beiden Unterkapitel mit der Umgehung dieser Beschränkung beschäftigen. Im Wesentlichen verfolgt man dabei das Ziel, die enthaltenen Rechenoperationen explizit und damit sichtbar für

das Transformations-Werkzeug umzuformen. Hierfür müssen auch enthaltene Synchronisationen so konvertiert werden, dass sie wie im bisherigen handhabbaren Kontext sowohl für die originalen Variablen als auch für neue Ableitungsobjekte funktionieren.

4.5.1 Behandlung der OpenMP-*reduction*-Klausel

Als Anschauungsbeispiel dient das nachfolgende Programmbeispiel in Abbildung 31 mit der *reduction*-Klausel. Hier ist auf der linken Seite die originale Funktion gezeigt, in der auf

```
c$omp do reduction(+: u)
      do i = 1, N
         u = u + sin(y(i))
      end do
c$omp end do
```

```
c$omp do reduction(+: u, g_u)
      do i = 1, N
         g_u = g_u + cos(y(i)) * g_y(i)
         u = u + sin(y(i))
      end do
c$omp end do
```

Abbildung 31: Anschauungsbeispiel mit *reduction*-Klausel. Links: die originale Funktion; Rechts: der korrekte Ableitungscode.
Variablen: u, g_u - Skalare mit *shared*-Attribut; y, g_y - Vektoren mit *shared*-Attribut

der Variablen u eine Reduktion durchgeführt werden soll (analog zu dem *atomic*-Beispiel aus Abbildung 20). Allerdings sorgt jetzt die *reduction*-Klausel dafür, dass die Summe über alle lokalen Teilergebnisse von u am Ende der Schleife gebildet wird.

Auf der rechten Seite der Abbildung 31 sieht man das Ergebnis nach einer AD-Transformation, wenn OpenMP korrekt erweitert werden würde. Im Detail muss dazu die hier rot-invertierte Variable g_u mit in die Klausel aufgenommen werden. Man sieht hier sehr einfach, dass die Berechnung der Ableitungsfunktion nicht korrekt wäre, wenn diese Aufnahme fehlt. Bevor ein Lösungsweg erläutert wird, gibt es noch eine Besonderheit zu beachten. Die in diesem Beispiel enthaltene *reduction*-Klausel impliziert in der Regel einen Datensichtbarkeitsbereich-Wechsel für u bzw. g_u von außen „geteilt“ zu innen „lokal“. Das bedeutet, dass hier zusätzlich zu der Summenbildung und Synchronisation die *private*-Charakteristik innerhalb des Arbeitsverteilungs-Direktivenbereiches umgesetzt werden muss.

Für einen allgemeinen Lösungsansatz werden ausgehend vom *reduction*-Beispiel nacheinander die drei Facetten, beginnend mit dem Datensichtbarkeitsbereichs-Attributswechsel, der Rechenoperation und der Synchronisation betrachtet. Als erstes wird links in der Abbildung 32 eine neue „lokal“-angelegte Variable *u_local* eingeführt (grauer Hintergrund). Diese realisiert den schon angesprochenen vorübergehenden *private*-Status von u und ist

```
c$omp do reduction(+: u)
      do i = 1, N
         u_local = u_local &
                 + sin(y(i))
      end do
c$omp end do
```

```
c$omp do
      do i = 1, N
         u_local = u_local &
                 + sin(y(i))
      end do
c$omp end do
      u = u + u_local
```

```
c$omp do
      do i = 1, N
         u_local = u_local &
                 + sin(y(i))
      end do
c$omp end do
c$omp critical
      u = u + u_local
c$omp end critical
```

Abbildung 32: Allgemeine Lösung für *reduction*-Klausel. Links: *private*-Attributs-Umsetzung; Mitte: explizite Rechenoperation; Rechts: Synchronisation auf Rechenoperation.
Variablen: *u_local* - Skalar mit *private*-Attribut; u - Skalar mit *shared*-Attribut; y - Vektor mit *shared*-Attribut

insbesondere auch dann hilfreich, wenn es mehr als eine Modifikation von u (oder jetzt

auf *u_local*) innerhalb des Schleifenbereiches gibt (Berechnungen auf lokaler Kopie von u). Während für *u_local* die Lokalität mit *private*-Attribut sichergestellt werden muss, bleibt u eine Variable mit „geteilten“ Zugriff. Danach muss die eigentliche Rechenoperation explizit ausformuliert werden. Das sieht dann beispielsweise, wie der mittlere Codeausschnitt in Abbildung 32 aus. Hier wurde die in der Klausel definierte Rechenoperation „(+: u)“ explizit in die Anweisung „$u = u + u\_local$“ hineingezogen (grau markiert außerhalb der Schleife). Letztendlich fehlt noch die Synchronisation, die man mit Hilfe einer *critical*-Direktive etablieren kann. Wie ganz rechts in Abbildung 32 zu sehen ist, reicht es dazu, den Synchronisationsbereich ausschließlich über der expliziten Rechenoperation „$u = u + u\_local$“ aufzubauen.
Anmerkung: Oft ist für die Unterprozesse eine Synchronisation von u notwendig, was mit einer zusätzlichen *barrier*-Direktive nach der *critical*-Region und einer *nowait*-Spezifikation auf der „*end do*“-Direktive gewährleistet werden kann.

Das nachfolgende Maßnahmenpaket 4.5 fasst für eine bessere Übersicht diesen generellen Ansatz in drei Schritten zusammen.

Maßnahmen 4.5 *Ersetzung „reduction“-Klausel*

- *Einführung neuer lokaler Datenobjekte und vollständige Ersetzung innerhalb des Schleifenbereiches für alle spezifizierten* **reduction***-Variablen.*
- *Explizite Ausformulierung der eigentlichen* **reduction***-Rechenoperationen auf den dazugehörigen* **shared***-Variablen.*
- *Etablierung einer neuen* **critical***-Region über den expliziten Rechenoperationen.*

In Abbildung 33 wird der dazugehörende generierte Ableitungscode aufgezeigt. Dieser

```
c$omp do
      do i = 1, N
         g_u_local = g_u_local + cos(y(i)) * g_y(i)
         u_local = u_local + sin(y(i))
      end do
c$omp end do
c$omp critical
      g_u = g_u + g_u_local
      u = u + u_local
c$omp end critical
```

Abbildung 33: Erfolgreich transformierter Ableitungscode basierend auf dem rechten Programmcode der Abbildung 32 als originale Funktion.
Variablen: *u_local*, *g_u_local* - Skalare mit *private*-Attribut; u, *g_u* - Skalare mit *shared*-Attribut; y, *g_y* - Vektoren mit *shared*-Attribut

enthält nicht nur eine korrekte Parallelisierung für die Berechnung der originalen Funktion mit dem Endergebnis in u, sondern auch eine korrekte parallelisierte Ableitungsberechnung in *g_u*. In Hinsicht auf die Effizienz der Berechnungen ist anzumerken, dass die *critical*-Region gegenüber der *reduction*-Klausel durchaus Nachteile haben kann. Sie sind aber oft vernachlässigbar, solange es im Vergleich zum Rechenaufwand für die *critical*-Region genug Rechenarbeit auf den lokalen Datenobjekten innerhalb der Schleife gibt.

An diesem letzten Beispiel wird auch deutlich, dass ohne diese einmaligen Vorverarbeitungsschritte ein AD-Werkzeug eine entsprechend aufwändige Unterstützung der Parallelisierung bereitstellen muss, egal ob es sich dabei um Techniken mit „Quellcode zu Quellcode“-Transformation oder „Ersetzen der Rechenoperatoren“ handelt. Insbesondere letzteres wurde bisher als unproblematisch eingestuft. Vereinzelt gibt es auch Simulationscodes, die trotz ihrer Komplexität mit einer verhältnismäßig einfachen Parallelisierung

auskommen. Ein Beispiel dafür wird in [92] vorgestellt. Hier ist keine aufwändige Unterstützung durch das AD-Werkzeug erforderlich, dafür ist dies aber auch nicht so ohne weiteres auf andere Software-Projekte übertragbar.

Betrachtet man aber die Details aus dem *reduction*-Beispiel genauer, muss auch eine mit „Ersetzen der Rechenoperatoren“ arbeitende Werkzeugumgebung die OpenMP-Direktiven und Klauseln explizit unterstützen. Wenn das nicht der Fall ist, kann allein schon bei der *reduction*-Klausel mit Multiplikationsoperator „$*$“ inkorrekter Ableitungscode entstehen. Es müssen also nahezu alle OpenMP-Klauseln (Datensichtbarkeitsbereichs-Attribute eingeschlossen) und einige Direktiven (z.B. *flush* etc.) korrekt ersetzt (in der Fachsprache: „überladen“) werden. Fehlt diese spezielle OpenMP-Unterstützung kann man alle Vorbereitungsmaßnahmen analog wie für die „Quellcode zu Quellcode“-Werkzeuge verwenden, um korrekten Ableitungscode und Parallelisierung zu erzeugen.

4.5.2 „conditional compilation“-Programmanweisungen

Eine abschließende, aber dafür allgemeinere Problematik ergibt sich für die sogenannten „conditional compilation“-Programmanweisungen, für die man links in der Abbildung 34 ein Beispiel findet. Kennzeichnend sind hier die rot eingefärbten OpenMP-Pragmas `c$`,

```
c$    u = u + v                 openmp_compiled = .false.
c$    call foo(u)           c$  openmp_compiled = .true.
                                if (openmp_compiled) then
                                   u = u + v
                                   call foo(u)
                                end if
```

Abbildung 34: Allgemeine Lösung für „conditional compilation“-Programmanweisungen. Links: originaler Beispielcode; Rechts: mit expliziter **if**-Abfrage.
Variablen: u, v - Skalare mit *private*-Attribut

die dafür sorgen, dass der danach grün-markierte Programmcode nur dann vom Compiler kompiliert und beachtet wird, wenn die OpenMP-Parallelisierung eingeschaltet wurde. Der darin enthaltene Programmcode ist unter AD-Aspekten relevant, wenn er aktive Variablen enthält. Geht man auch hier wieder von einer Nicht-Beachtung seitens des AD-Werkzeugs aus, müssen auch solche Zeilen explizit umgeschrieben werden, damit sie in korrekten Ableitungscode transformiert werden können.

Am einfachsten geht das, indem man eine explizite **if**-Abfrage einbaut. Veranschaulicht wird das in dem rechten Teil der Abbildung 34. Hier wurde in 5 Schritten (Maßnahmenpaket 4.6) der Zweizeiler von links in einen 6 Zeilen langen Block umgewandelt.

Maßnahmen 4.6 *Ersetzung „conditional compilation“-Programmanweisung*

- *Eine neue lokale* **logical***-Variable einfügen.*
- *Diese mit dem „Falsch“-Wert vorinitialisieren.*
- *Eine neue „conditional compilation“-Programmanweisung einbauen, um die neue Variable gegebenenfalls mit dem „Wahr“-Wert zu überschreiben.*
- *Eine* **if***-Abfrage über die ursprünglichen Programmanweisungen spannen, die vom Wahrheitsgehalt der* **logical***-Variable abhängig ist.*
- *Das „conditional compilation“-Pragma der ursprünglichen Programmanweisungen entfernen.*

Das Beispiel in Abbildung 34 funktioniert unter der Annahme, dass das Transformations-Werkzeug die **if**-Abfrage nicht weg optimiert hat. Wenn das nicht unterbunden werden

kann, lässt sich dies oft durch geeignete Werkzeug-Parameter oder ein verstecktes Vorinitialisieren (des „Falsch“-Wertes) in einer Dummy-Routine erreichen.

4.6 Ableitungs-spezifische Modularisierung

Ein gänzlich anderer Aspekt, der ausschließlich die Verträglichkeit mit dem AD-Werkzeug und die Erweiterungsproduktivität positiv beeinflusst, ist eine tiefer greifende Funktionsmodularisierung über eine Auswahl von abzuleitenden Routinen hinaus. Dies betrifft die Extraktion von nicht abzuleitenden Programmcode aus größeren Routinen, die Behandlung von nicht-standard Spracherweiterungen, und manuelle Anpassungen um die Konvergenz oder die Effizienz der Ableitungsberechnungen durch Ausnutzung numerischen Wissens zu optimieren.

4.6.1 Minimierung des Aufwands für die Programmtransformation

Der zeitliche Aufwand für die Durchführung einer AD-Programmcode-Transformation wird minimiert, wenn man möglichst viele (zusammenhängende) Codeteile neu in eigene Unterroutinen auslagert für die mindestens eine der folgenden zwei Eigenschaften zutrifft.

E1: Es ist keine Berechnung der abzuleitenden Zustandsgrößen der Modell-Simulation enthalten, noch wird sie hierdurch beeinflusst oder gesteuert.

E2: Es handelt sich um so genannten Einlese-Code, der aus externen Quellen die Werte einliest bzw. initialisiert.

Im Allgemeinen gehören dazu z.B. alle Ausgaberoutinen, die die berechneten Daten nur konvertieren oder in Dateien oder auf den Bildschirm schreiben und die meist problemlos in separate Routinen ausgelagert werden können.

Eine Besonderheit stellt der Punkt E2 dar, denn eingelesene Werte können die Modell-Simulation und damit potentiell auch die Ableitungsberechnung beeinflussen. Aus AD-Sicht werden eingelesene Werte aus externen Quellen aber als Konstanten angesehen, weswegen die Ableitungsobjekte an diesen Stellen zumeist zu Null initialisiert werden müssen.

Tritt das zu Null-Setzen nur an definierten Stellen ganz zu Anfang und damit vor Beginn der eigentlichen Modell-Simulation auf, fällt die zu Null-Setzung meist mit der Erstinitialisierung der Ableitungsobjekte zusammen. In solchen Fällen wird es dann einfacher und übersichtlicher, wenn man die Erstinitialisierung gesondert in einer eigenen Routine für alle Ableitungsobjekte abarbeitet. Dann kann dort erst der Speicher reserviert und dann zu Null gesetzt werden (Initialisierungsphase der Ableitungsobjekte). Wenn das generell so gehandhabt wird, ist es zusätzlich sinnvoll, solche Einlese-Abschnitte zusammenzufassen und dem AD-Werkzeug entsprechend vorzuenthalten. Dies ist oft auch in Hinblick auf die Nutzung von ccNUMA-Fähigkeiten (Kapitel 2.5.1) wichtig, wenn man dafür die Rechenleistung optimieren möchte (siehe Kapitel 6).

Abschließend sei noch darauf hingewiesen, dass viele „Quellcode zu Quellcode“-Transformations-Werkzeuge (z.B. in TAPENADE [9]) für eine statische Datenfluss-Analyse verschiedene Analysetechniken (siehe dazu unter TBR [93, 94, 95]) einsetzten. Dabei kann es vorkommen, dass bei einer großen Menge von Codezeilen potentiell auch Variablen bzw. Felder als *aktiv* markiert werden können, die nicht wirklich die Ableitungsberechnung beeinflussen.

Angenommen, eine *aktive* Variable a beeinflusst das erste Element eines Vektorfeldes v, wobei v selbst über sein erstes Element $v(1)$ auf die Variable e und über sein letztes $v(L)$ auf die Variable l einwirkt. Zusätzlich soll hier gelten, das l eine weitere Variable w beeinflusst. Dann gib es eigentlich nur einen „ableitungs-aktiven“ Datenpfad $a \rightarrow v(1) \rightarrow e$. Wegen üblicher „konservativer“ Annahmen, wird hierbei aber oft das Feld $v(1 \cdots L)$ komplett als *aktiv* markiert. Das kann damit auch zu der überflüssigen Aktivierung von l und w ($v(L) \rightarrow l \rightarrow w$) führen.

Daraus resultiert oft unnötig viel Arbeitsspeicher und Rechenaufwand (wg. Ableitungsobjekte für l und w). Hat man dagegen den Programmcode stark auf die wesentlichen Teile ausgedünnt (ohne den Code für $l \to w$), haben die Analysetechniken es einfacher, wodurch oft weniger Ableitungsobjekte (nicht mehr für w) erzeugt werden, was insgesamt den Speicherverbrauch und die Rechenarbeit senkt.

4.6.2 Umgehung unnötiger Probleme von Programmteilen

Neben dem auf Masse ausgerichteten Aussparen von möglichst viel komplizierten Programmcode gibt es auch konkrete Probleme mit AD-Werkzeugen, die sich nur durch Extrahieren (Verkapseln von Programmteilen) und durch Vermeidung von AD auf diesem Code umgehen lassen. Probleme treten oft dann auf, wenn Spezial-Erweiterungen der Programmiersprache verwendet wurden (z.B Cray-Pointer, Präprozessor-Direktiven, Parallelisierung etc.), die vom AD-Werkzeug nicht unterstützt werden. Zusammenfassen lässt sich das in einem kleinen Richtlinienpaket 4.3 für das Verkapseln von Programmcode. Die Zielausrichtung liegt dabei auch bei einer Verbesserung im Sinne der Erweiterungsproduktivität (wg. Nachbearbeitung).

Richtlinien 4.3 *Bedingungen für eine allgemeine Verkapselung von Programmcode*

- *Der Code ist für das Werkzeug nicht korrekt bearbeitbar und enthält auch keine ableitungs-relevanten Berechnungen (z.B. für einige OpenMP-Zwischenroutinen Kapitel 4.3).*
- *Immer dann, wenn der Code nicht zur Ableitungsberechnung benötigt wird und durch die Verkapselung keine nennenswerten Nachteile für die Rechenleistung oder die Wartbarkeit entsteht (z.B. komplizierte ccNUMA-Speicherreservierung).*

Die Erfahrungen in dem zugrunde liegenden Software-Projekt zeigten, dass sich der überwiegende Teil der kritischen Fälle durch Verkapselung vermeiden lässt (z.B. Daten-Konvertierung oder komplizierte Ein- und Ausgabe-Konstrukte für den Dateizugriff). Die verbliebenen kritischen Fälle konnten durch ein Umschreiben des Programmabschnitts gelöst werden (beispielsweise wie im nachfolgenden Unterkapitel 4.6.3).

4.6.3 Umgang mit Spezialcode - manuelle Ableitungen

Es gibt Spezialfälle, welche immer einer manuellen Nachbearbeitung bedürfen oder die generell anders gehandhabt werden sollen als einfach durch ein AD-Werkzeug. Dabei geht es um Verbesserungen in der Ableitungsberechnung allgemein und speziell in deren Steuerbarkeit bezüglich der Konvergenz und der Genauigkeit. In Einzelfällen kann es sich auch um manuell erstellten Code zur Ableitungsberechnung (z.B. analytische Teillösungen) handeln, der aus Gründen der Effizienz dem automatisch erzeugten Ableitungscode vorzuziehen ist.

Beispielsweise wird angenommen, es gibt eine einfache „**do while**“-Schleife wie links-oben in Abbildung 35. Es soll sich dabei um eine Konvergenzschleife über einen linearen oder nicht-linearen Prozess (*update a*) handeln. Weiterhin wird dieser iterative Prozess von der Konvergenzeigenschaft ($a - a\_old$ soll kleiner als 10^{-8} sein) über der Zustandsgröße a gesteuert. Vom AD-Werkzeug wird daraus dann ein Ableitungscode erzeugt, bei dem die Konvergenzsteuerung immer noch durch die Abfrage auf der originalen Zustandsgröße a beruht (siehe rechts-oben in Abbildung 35). Die Ableitungen konvergieren im Allgemeinen aber langsamer (siehe z.B. [96]) und deshalb empfiehlt es sich, im Ableitungscode die Konvergenzkriterien auch auf die Ableitungsobjekte zu erweitern. Wenn dies nicht innerhalb des AD-Werkzeugs durch entsprechende Zusatzinformationen unterstützt wird, kann man es auch mit Hilfe einer weiteren Funktionskapselung erreichen.

```
do while(a - a_old > 1.0d-8)
   a_old = a

   ... ! update a
end do
```

```
do while(a - a_old > 1.0d-8)
   a_old = a
   g_a_old = g_a
   ... ! update a, g_a
end do
```

```
conv = conv_fkt(a, a_old)
do while(conv)
   a_old = a

   ... ! update a
   conv = conv_fkt(a, a_old)
end do
```

```
conv = g_conv_fkt(a, g_a, a_old, g_a_old)
do while(conv)
   a_old = a
   g_a_old = g_a
   ... ! update a, g_a
   conv = g_conv_fkt(a, g_a, a_old, g_a_old)
end do
```

Abbildung 35: Beispiel Konvergenzschleife. Links-oben: originale Schleife; Rechts-oben: Standard-AD Ableitung der originalen Schleife; Links-unten: verbesserte Schleife; Rechts-unten: Ableitung der verbesserten Schleife.
Variablen: *a*, *a_old*, *g_a*, *g_a_old* - Skalare (mit *private*-Attribut)

Im Beispiel links-unten (Abbildung 35) lagert man die Konvergenzabfrage ($a - a\_old > 1.0d\text{-}8$) des originalen Codes deshalb in eine eigene Funktion $conv\_fkt$ aus. Der Wahrheitsgehalt wird hier über die logische Variable *conv* in der „**do while**“-Schleife weiterverarbeitet. Angenommen, die Funktion mit der Konvergenzabfrage $conv\_fkt$ wird in dem AD-Werkzeug als *aktiv* behandelt, dann erhält man den dazu gehörenden Ableitungscode, wie rechts-unten in Abbildung 35. An dieser Stelle ist es nun möglich, die automatisch erzeugte Konvergenzfunktion g_conv_fkt durch eine manuell erstellte zu ersetzen, welche sowohl die Konvergenz der originalen Zustandsgröße a als auch die des Ableitungsobjektes g_a berücksichtigt.

Alternativ und ergänzend kann man diesen ganzen „**do while**“-Iterationsprozess durch eine Verkapselung noch mehr verbessern. Hierunter kann man auch eine Anwendung der Technik des „verspäteten Einsatzes“ zählen, siehe dazu Kapitel 2.2 und in der Literatur [47, 48, 49]. Hier beginnt die Ableitungsberechnung erst, wenn die originalen Zustandsgrößen bereits konvergiert sind.

Im Detail geht man am Beispiel einer Fixpunktiteration dazu in fünf Schritten vor.

- Als erstes kapselt man die Fixpunktiteration, falls nicht schon so vorgegeben, in eine extra Routine $FIter$.
- Danach fügt man eine zusätzliche Zwischenroutine $Wrapper\_FIter$ ein, die nichts weiter aufruft als das ursprüngliche $FIter$.
- Anschließend stellt man den sonstigen originalen Programmcode auf den entsprechenden längeren Funktionspfad (über $Wrapper\_FIter$ statt direkt $FIter$) um.
- Man generiert mit Hilfe des AD-Werkzeugs die neuen Routinen $g\_Wrapper\_FIter$ und g_FIter.
- Zuletzt erstellt man manuell eine Version für $g\_Wrapper\_FIter$ die im Gegensatz zur AD-generierten, zuerst das originale $FIter$ aufruft (auskonvergieren lässt) und danach die Ableitungsberechnung mit g_FIter anstößt.

Sofern die Schnittstellen für $Wrapper\_FIter$ bzw. $FIter$ genug verallgemeinert sind, sollte sie im späteren Verlauf kaum geändert werden müssen. Damit muss auch die einmal manuell erstellte $g\_Wrapper\_FIter$-Routine später kaum noch angepasst werden. Die doppelte Funktionskapselung erhöht so die robusten Eigenschaften gegenüber den meisten Änderungen im sonstigen Programmcode (speziell bei Verschiebungen des Aufrufs von $FIter$ bzw. $Wrapper\_FIter$).

Einen etwas anderen Fall stellt der Lösungsalgorithmus eines linearen Gleichungssystems dar. Angenommen, das lineare Gleichungssystem (8) besteht aus einer Matrix A und einer rechte Seite b.

$$A \cdot x = b \tag{8}$$

Die gesuchte Lösung von (8) soll x sein und alles ist zusammen in der Routine *solver* verkapselt. Weiterhin soll es sich hier um einen iterativen Gleichungslöser [10, 11, 97] mit Konvergenzsteuerung handeln. Dann empfiehlt die Literatur [98, 99] (allgemeiner im anderem Zusammenhang auch in [96, 48]), nicht den gesamten Löseralgorithmus durch das AD-Werkzeug weiter zu verarbeiten. Stattdessen kann man unter der Annahme, dass sowohl A, b als auch x aktive Variablen sind, die Ableitung auch analytisch und ausgehend von (8) als

$$A' \cdot x + A \cdot x' = b' \tag{9}$$

schreiben, was sich direkt in

$$A \cdot x' = c \quad \text{mit} \quad c = b' - A' \cdot x \tag{10}$$

umformen lässt. Daraus folgt, dass das Gleichungssystem mit veränderter rechten Seite c (eine pro Ableitung) gelöst wird, wobei der Programmcode des Gleichungssystemlösers nicht abgeleitet werden muss. Die in (10) beschriebene Variante ist oft auch effizienter als ein automatisch erzeugter Ableitungscode.

Dies ist ein Beispiel, bei dem wegen einer Optimierung für bessere Rechenleistung eine Modularisierung stattfindet, die dem AD-Werkzeug vorenthalten werden soll. Mit wenigen Schritten kann man dann einen Programmcode für die analytische Ableitung von *solver* erstellen, der zweimal die ursprüngliche Version von *solver* aufruft (einmal mit rechter Seite b und einmal mit c) und dazwischen c berechnet. Auch hierfür gilt, dass dies nur dann mit wenig Aufwand gelingt und nur selten erneuert werden muss, wenn *solver* ausreichend verallgemeinert und verkapselt ist.

Abschließend zeigt das Richtlinienpaket 4.4 als Ergänzung zu 4.3 eine Zusammenfassung der weiteren Gründe zur Modularisierung bzw. Verkapselung von Programmteilen in extra Funktionen oder Routinen.

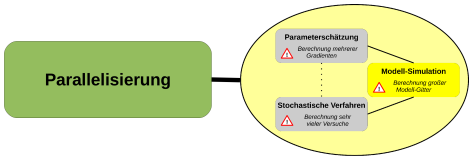
Richtlinien 4.4 *Gründe für zusätzliche Verkapselung von Programmcode*

1. *Wenn man die Berechnungszeit der Ableitungen oder den resultierenden Speicherverbrauch verkürzen kann, z.B. durch Einsatz analytischer Ableitungen oder Code mit besonderen Optimierungen.*

2. *Wenn sich damit manuelle Nachbearbeitungen allgemein umgehen lassen. Z.B. für eine bessere Genauigkeit mittels einer Anpassung der Konvergenzabfrage oder Anwendung von dem „verspäteten Einsatz“.*

3. *Wenn eine manuelle Umstrukturierung im Ableitungscode (im Sinne von 2.) weniger oft angepasst werden muss. Z.B. die doppelte Verkapselung für den „verspäteten Einsatz“ (siehe Kapitel 2.2).*

5 Teilkonzept - Parallelisierungsstrategien



Für eine der wichtigen Grundanforderungen, die Parallelisierung, gibt es viele mögliche Ansätze. Deshalb werden in den nachfolgenden Abschnitten dieses Kapitels alle „relevanten“ Parallelisierungsstrategien anhand eines Beispielprogramms und hinsichtlich der Unterschiede in ihrem (1) Berechnungsaufwand, (2) dem Speicherverbrauch und (3) dem Synchronisationsanteil bewertet und verglichen. Zu allen drei Merkmalen wird eine gewichtete und begründete Bewertung mittels Performance-Modellen durchgeführt. Allerdings geht es nicht nur ausschließlich um eine effiziente Verteilung der Rechenarbeit, sondern auch um eine Parallelisierung, die übersichtlich, leicht zu pflegen und möglichst unabhängig von den meisten zu erwartenden Erweiterungen und Änderungen am Programmcode ist.

In Abbildung 36 wird ein Überblick der grundsätzlich zu unterscheidenden Berechnungsaufgaben gezeigt, in dem zwischen einer inneren und einer äußeren Parallelisierung unterschieden wird. Die (rot markierte) rechte Parallelisierung steht in dieser Abbil-

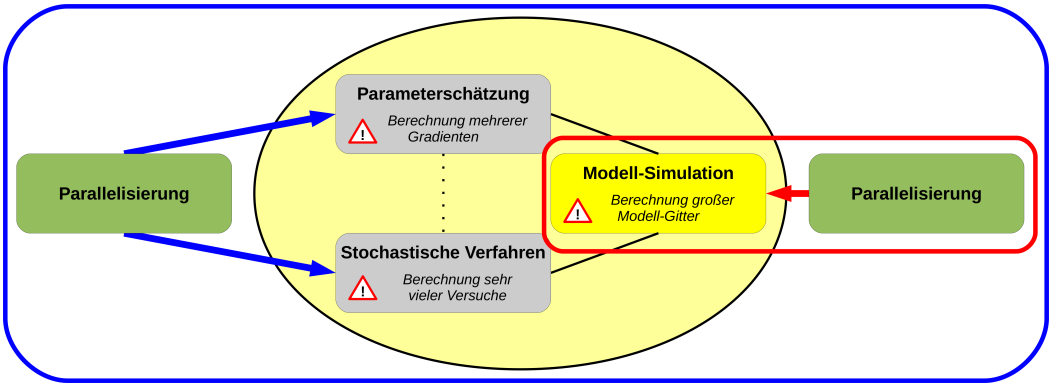


Abbildung 36: Zwei von einander unabhängige Parallelisierungsstufen; Links die äußere und rechts die innere Parallelisierungsschicht.

dung für die innere Parallelisierungsschicht. Sie fasst alle Teilaufgaben zusammen, die bei der originalen Modell-Simulation (Vorwärtsberechnungen) parallelisiert berechnet werden können und wird deswegen auch als „originale Parallelisierung“ bezeichnet. Sie steht aber auch für die parallelisierte Berechnung einer einzelnen Richtungsableitung, da die originale Parallelisierung, durch die im letzten Kapitel 4 beschriebenen Richtlinien und Maßnahmen auch auf die Berechnung der Ableitungsanweisungen ausdehnt wird. Der Vollständigkeit halber werden in dem nachfolgenden Unterkapitel 5.1 einige grundlegende Details zu der hier verwendeten originalen Parallelisierung ausgeführt.

Auf der anderen Seite liegt die blau-markierte äußere Parallelisierung (links in Abbildung 36) die für die Mehrfachberechnung von Modell-Simulationen (stochastische Versuche) oder Richtungsableitungen steht. Es wird hier also immer vorausgesetzt, dass man eine bestimmte Anzahl von verschiedenen und unabhängig berechenbaren Modell-Simulationen oder Gradienten benötigt. Letztere werden in dieser Arbeit immer durch den automatisch erzeugten Programmcode eines AD-Werkzeugs berechnet. Ein systematischer Vergleich findet in Kapitel 5.2 statt. Die Parallelisierung für die Modell-Simulationen wird in Kapitel 5.3 erläutert.

Diese zahlreichen Möglichkeiten zur parallelen Berechnung mehrerer Ableitungen oder Modell-Simulationen sind alle für die zugrunde liegende Anwendungsfragestellung relevant und werden umfassend verglichen. Diese Gegenüberstellung ist in der Vollständigkeit und in Bezug auf die Vergleichskriterien neu. Neu ist damit auch die letztendliche Auswahl in

Hinblick auf die in dieser Arbeit vorgestellten Softwaretechniken, um die hier definierten Grundanforderungen besonders gut erfüllen zu können.

Um eine größere Verallgemeinerbarkeit zu erreichen, werden alle Strategiebetrachtungen losgelöst von den vielfältigen Möglichkeiten einer konkreten originalen Parallelisierung durchgeführt. Deren Speicherverbrauch, Rechenaufwand und Synchronisation sind also nicht Gegenstand des Vergleiches. Aus diesem Grund nehmen in dem Vergleich die Performance-Modelle immer einen allgemeinen Bezug auf die Eigenschaften der Vorwärtssimulation in Form der folgenden Kenngrößen:

M_o(N): Allgemeiner Speicherverbrauch

T_o(N,1): Serieller Berechnungsaufwand bzw. Zeitverbrauch

T_o(N,$\mathbf{t}$): Paralleler Berechnungsaufwand für $\mathbf{t}$ Unterprozesse, d.h. maximaler Berechnungsaufwand für einen einzelnen der $\mathbf{t}$ Unterprozesse

S_o(N): Anzahl an Programmanweisungen, zu denen durch die AD-Transformation Ableitungsanweisungen erzeugt werden (Optimierungen werden hier vernachlässigt)

A_o(N): Anzahl der vom Programmcode „explizit“ zu reservierenden Datenelemente (implizite Speicherreservierungen oder explizite von außerhalb des Betrachtungsbereichs werden hier nicht mitgezählt)

Hierbei steht der Index o für den „originalen“ Programmcode (einfache Vorwärtssimulation) und N steht für eine allgemeine Modellgröße bzw. für die zu betrachtende Datenmenge. Wichtig ist außerdem, dass es sich in der vorliegenden Arbeit bei dem Berechnungsaufwand (wie bei $T_o(\mathrm{N}, \mathbf{t})$ angedeutet) um eine grobe Schätzung für die maximale Laufzeit eines Unterprozesses und damit um eine Schätzung der Gesamtlaufzeit handelt. Damit sind nicht die tatsächlichen Rechenschritte oder Operationen gemeint, denn Wartezeiten werden mit einbezogen.

Bei der gleichzeitigen Kombination der inneren mit der äußeren Parallelisierung erhält man eine Hierarchie von Unterprozessen. Ausgegangen wird davon, dass in der äußeren Parallelisierung eine Gruppe von Unterprozessen eröffnet wird und anschließend alle die innere parallele Region erreichen. Dann eröffnet an dieser Stelle jeder dieser äußeren Unterprozesse eine eigene innere Gruppe von neuen Unterprozessen, womit dann mehrere innere Gruppen von Unterprozessen in der Ausführung sind. Hierbei kommt es unter anderem auch darauf an, welche Parallelisierungsalgorithmen sich für die innere Parallelisierung eignen bzw. was bei der Auswahl und deren Implementation beachten werden muss, damit sie in Kombination mit der äußeren einsetzbar ist.

5.1 Parallelisierung der Vorwärtssimulation

Die innere Parallelisierung [100] gliedert sich grob in zwei Regionen. Die erste bildet einen parallelisierten Bereich rund um die Aufstellung des linearen Ersatzgleichungssystems (siehe Kapitel 2.1) für jeden physikalischen Prozess. Die Zweite ist eine unabhängige Region über die parallelisierten linearen Gleichungssystemlöser (in dieser Arbeit werden CG und BiCGStab betrachtet). In Abbildung 37 kann man die Aufteilung anhand der zwei rot gestrichelten Bereiche erkennen. Die Gleichungssystem-Aufstellung wird innerhalb der braun gefärbten Physik-Module vorgenommen, welche für die Berechnung der einzelnen physikalischen Zustandsgrößen zuständig sind. Hierbei wird nacheinander für die jeweils aktiven physikalischen Zustandsgrößen das Ersatzsystem aufgestellt und anschließend mit dem linearen Gleichungssystemlöser gelöst, bevor die nächste Zustandsgröße berechnet wird. Die nacheinander ausgeführten Berechnungen der einzelnen Zustandsgrößen werden durch die nicht-lineare Iterationsschleife gekoppelt.

Betrachtet man das Aufstellen des Ersatzsystems etwas genauer, findet man eine ausschließliche Abhängigkeit der Berechnungen vom letzten Zustand vor dem Betreten des

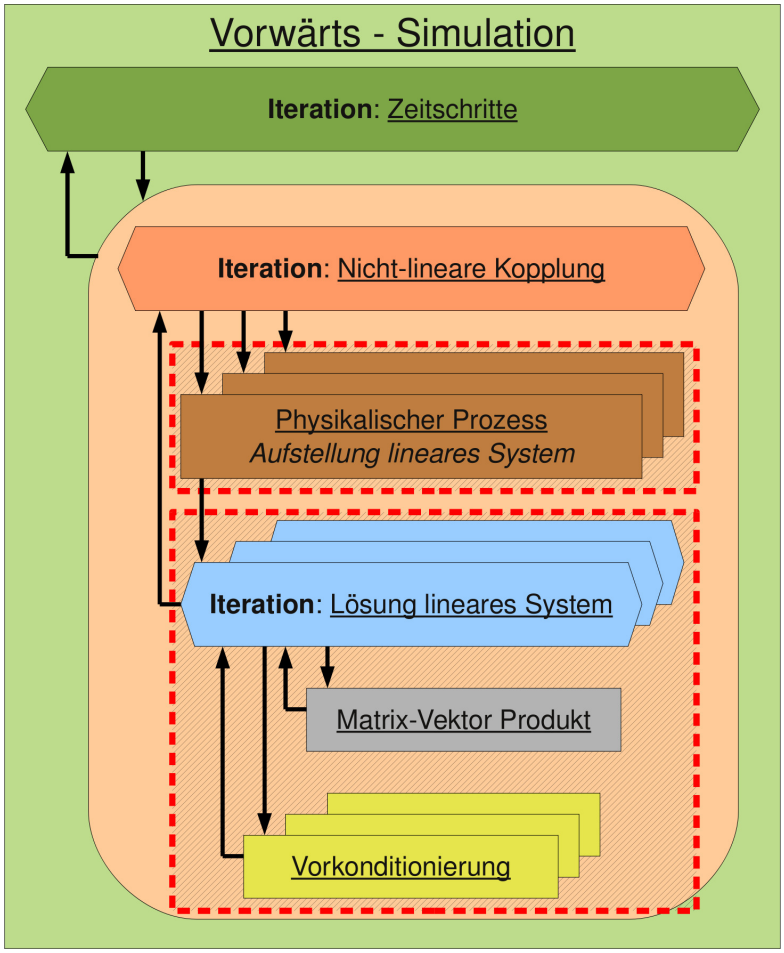


Abbildung 37: Schematischer Ablauf der Vorwärtssimulation. Parallele Regionen sind durch die rot gestrichelten Bereiche gekennzeichnet.

jeweiligen Physik-Moduls. Mit dieser Voraussetzung ergibt sich eine weitestgehende Unabhängigkeit aller innerhalb eines Moduls stattfindenden Berechnungen und damit eine parallelisierbare Aufstellung des Ersatzgleichungssystems. Im Detail kann nahezu jede einzelne Zeile der Systemmatrix und der dazugehörigen rechten Seite weitestgehend unabhängig von den anderen berechnet werden. Einzige Ausnahme bilden hier die nachträglichen Modifikationen bezüglich der Dirichlet- und Neumann-Randbedingungen. Diese teilweise seriell ausgeführten Randwerte-Modifikationen sind nur ein Grund für die zweiteilige Parallelisierung.

Ein anderer Grund für diese Trennung liegt in der schon erwähnten Möglichkeit, dass der Gleichungslöser später bei der Erzeugung von Ableitungscode nicht als „Blackbox“ (Fachbegriff für die Verwendung von automatisch erzeugten AD-Code ohne diesen zu untersuchen oder zu ändern) beibehalten werden soll. Hier kann stattdessen ein manuell erstellter und optimierter Ableitungscode oder eine externe Lösung in Form eines Bibliotheksaufrufs eingesetzt werden. Neben einer Vielzahl von manuell optimierten bzw. parallelisierten Gleichungslösern findet sich hier auch die Aufrufmöglichkeit von externen, nicht-parallelisierten Gleichungslösern (z.B. aus LAPACK [27]). Deswegen wird, falls vorhanden, die zweite parallele Region explizit neu für das Lösen des eigentlichen linearen Gleichungssystems eröffnet (unterer rot-gestrichelter Bereich in Abbildung 37).

An dieser Stelle kann man in Hinblick auf die spätere, parallelisierte Mehrfachausführung (äußere Parallelisierung) eine wichtige Forderung (Grundannahme 5.1) formulieren.

Grundannahme 5.1 *Bedingung für eine mehrfache Ausführbarkeit*

Der Programmcode bzw. die Algorithmen innerhalb der parallelen Regionen (der Vorwärtssimulation) müssen für die gleichzeitige Nutzung durch verschiedene Unterprozesse ausgelegt sein (engl. „thread-safe“).

Diese Annahme 5.1 bedingt zweierlei Unabhängigkeiten. Einerseits müssen die verwendeten Implementierung innerhalb einer solchen parallelen Region unabhängig von einander ausführbar sein. Andererseits muss es möglich sein, dass gleichzeitig mehrere Gruppen von Unterprozessen ihre parallelen Regionen unabhängig auf verschiedenen Datenbereichen bearbeiten können.

Im ersten Fall kann es sich beispielhaft um verschiedene logisch unabhängige Berechnungen auf unterschiedlichen Zeilen innerhalb eines Datenvektors handeln. Die Berechnung soll dabei für die eine Zeile nicht von den Ergebnissen in einer anderen abhängen. Für den zweiten Fall muss man meistens sicherstellen, dass die unterschiedlichen Gruppen von Unterprozessen verschiedene Datenvektoren bearbeiten können. Dies scheint auf den ersten Blick einfacher erfüllbar, führt aber oft bei der Verwendung externer Bibliotheken zu Problemen. Diese treten oft bei einer integrierten Parallelisierung für die Verteilung der Rechenarbeit auf mehrere Rechenkerne auf. Genauso oft kommen dabei aber globale Schnittstellen zum Einsatz, die im Nachhinein nicht ohne Weiteres die gleichzeitige Verwendung in mehreren Unterprozess-Gruppen unterstützen. Näheres zu einer Lösungsstrategie wird in Kapitel 6 beschrieben.

Unabhängig von den manuell implementierten Gleichungslösern wird auf den externen LAPACK-Löser gleichzeitig mit verschiedenen Unterprozess-Gruppen zugegriffen, obwohl der Löser selbst nicht parallelisiert auf einer inneren Gruppe ausgeführt wird. Bei den manuell implementierten Lösern handelt es sich ausschließlich um „konjugierte Gradienten“-Verfahren [10, 11, 101, 102] mit wahlweiser Vorkonditionierung [26, 103, 104, 105]. Hier wird unter anderem massiver Gebrauch von BLAS1-Routinen der BLAS-Bibliothek [106, 107] gemacht.

Der dieser Arbeit zugrunde liegende Simulationscode hat eine weitere besondere Eigenschaft. Er enthält Algorithmen, die für stark bzw. fein diskretisierte Modellgitter geeignet sind. Das bedeutet einerseits eine hohe Rechenlast die mittels Parallelisierung bewältigt wird. Andererseits belasten die verwendeten Algorithmen sehr intensiv den Hauptspeicher, was sich nicht nur im Gesamtverbrauch des zur Berechnung nötigen Hauptspeichers, sondern auch in einem hohen Datenverkehr zwischen Speicher und Prozessorkern äußert. Deshalb ist maßgeblich eine hohe Hauptspeicherbandbreite für eine gute Skalierbarkeit (im Sinne der Parallelisierung) und damit für eine kurze Berechnungszeit nötig.

Übrig bleibt für die späteren Vergleiche einmal der maximale Speicherverbrauch, der in den hell-gelben Übersichten mit **mem** abgekürzt wird und zum anderen der maximale Berechnungsaufwand. Letzterer wird mit **cpu** abgekürzt und wird in Abhängigkeit von der jeweiligen Anzahl der Unterprozesse angegeben. Beide Kriterien erhöhen sich mit zunehmend verfeinerter Modell-Diskretisierung, weswegen die Vergleiche bezüglich einer fiktiven Größe N und damit unabhängig von der tatsächlichen Modellgröße betrachtet werden. Das selbe gilt auch für die Anzahl zu berechnender Zeitschritte oder sonstige modellspezifische Konfigurationen, für die insgesamt das N als Näherung steht. Die einzigen Ausnahmen bilden hier die Anzahl der für die Parameterschätzung benötigten Gradienten P oder die Anzahl der zu berechnenden stochastischen Versuche V. Sie werden zur Skalierung der Vergleichskriterien eingesetzt.

Beginnend mit dem Berechnungsaufwand soll die in Abbildung 38 aufgeführte Beispielroutine einen typischen Simulationsverlauf der originalen Modell-Simulation charakterisieren. In der abgebildeten Routine *vdiv* wird im Wesentlichen ein Skalar α elementweise durch einen Vektor x dividiert und das Resultat elementweise in dem Vektor y gespeichert. Entsprechend gibt es hier N Divisions-Operationen. Ohne Parallelisierung wird hierfür der

```
subroutine vdiv(N,α,x,y)
  integer N, i
  real x(N), y(N), α
  do i = 1, N
    y(i) = α/x(i)
  end do
end subroutine vdiv
```

Abbildung 38: Beispielfunktion *vdiv* stellvertretend für den Speicherverbrauch und Berechnungsaufwand einer Modell-Simulation.
□ (dicke Umrandung) Markierung für Berechnungsroutine

serielle Berechnungsaufwand (N Rechenoperationen) mit **cpu** $= T_o(\mathrm{N}, 1)$ definiert. Analog dazu wird der Speicherverbrauch einfach mit **mem** $= M_o(\mathrm{N})$ spezifiziert, wobei es sich im Detail um $2 \cdot \mathrm{N} + 1$ Datenelemente für die beiden Vektoren x, y und dem Skalar α handelt (die Variablen N und i werden vernachlässigt). Das ist allerdings noch nicht ausreichend, um auch bei einer Parallelisierung die dahinter liegenden Datenstrukturen bzw. deren Speicherverbrauch überschaubar zu machen.

Für ein besseres Verständnis ist es deshalb wichtig, die möglichen Datenstrukturen schematisch und gesondert zu veranschaulichen. Im Folgenden wird deshalb oft zu den Parallelisierungsstrategien ein Schema mit blauen Datenblöcken gezeigt. Zusätzlich werden Ableitungsobjekte grau statt blau eingefärbt. In der Abbildung 39 sind die drei häufigsten Varianten dargestellt. Sie zeigen von links nach rechts jeweils drei Datenobjekte (Blöcke),

Abbildung 39: Datenstruktur und Speicherverbrauch. Links: Mehrfachnutzung zu unterschiedlichen Zeitpunkten; Mitte: jeweils einzelne Datenobjekte; Rechts: höherdimensionales Datenobjekt.

wobei die Anordnung der drei Datenblöcke die unterschiedlichen Strukturen kennzeichnet. Die erste (linke) Variante steht für eine zeitliche Mehrfachnutzung des selben Datenobjektes, d.h. dieselbe Vektorvariable wird in diesem Beispiel für insgesamt drei verschiedene unabhängige Berechnungen eingesetzt. Der entsprechende Hauptspeicher kann für die jeweilige Berechnung wieder verwendet werden, weshalb man nur ein Datenobjekt benötigt und damit in der Darstellung nur einen aktiven Block vollständig sieht.

Bei der mittleren Variante verwendet dagegen jeder Unterprozess für seine Berechnung eine eigene Vektorvariable (bzw. Kopie des Datenobjektes) und muss somit das Datenobjekt insgesamt mehrmals im Hauptspeicher anlegen. Deswegen sind hier die drei Datenblöcke deutlich getrennt und sichtbar vereinzelt dargestellt. Dies ist unabhängig davon, ob zu einem Zeitpunkt tatsächlich vielleicht nur ein Datenobjekt vorhanden ist; bei einer Parallelisierung könnten es auch mehrere sein.

Der Unterschied zur letzten Variante (ganz rechts in Abbildung 39) besteht hier deshalb hauptsächlich in der Anzahl der einzeln erzeugten Datenobjekte. Hier handelt es sich nur um ein einzelnes Vektorobjekt mit einer erhöhten Dimensionsanzahl. Die Größe der zusätzlichen Dimension entspricht dafür aber der Anzahl der benötigten Kopien bzw. einzelnen Datenbereiche.

5.1.1 Vorwärtssimulation - Konzeptuelle Richtlinien

Unabhängig von den bisher schon angesprochenen funktionellen Modularisierungen oder technischen Maßnahmen gibt es eine Reihe von wichtigen Richtlinien für das Hochleistungsrechnen (engl. „high performance computing“) und einen dafür tauglichen Simulationscode. Die allgemeinen Anforderungen und eigenen Erfahrungen im Softwareprojekt lassen sich in Hinsicht auf (1) schonende Ressourcennutzung, (2) leistungsfähige Datenplatzierung, (3) Datenstrukturen für effizienten Zugriff und (4) hoch skalierender Parallelisierung zu einer Reihe von Richtlinien zusammenführen. Hierbei profitierte das im Rahmen dieser Forschungsarbeit erstellte Softwareprojekt maßgeblich von einer Ausrichtung auf diese Pakete. Das erste Richtlinienpaket 5.1 beschäftigt sich mit der allgemeinen Datenverwaltung und besagt außerdem, wann Ersatzfunktionen statt Datenfelder verwendet werden sollten.

Richtlinien 5.1 *Allgemeiner Umgang mit den Datenobjekten und Ersatzfunktionen*

- *Große Datenfelder auflösen und durch Funktionsaufrufe ersetzen (Ersatzfunktionen), um die Speicheranforderungen gering zu halten. Das ist günstig bei selten verwendeten Werten, die leicht zu berechnen sind.*
- *Ein Datenfeld für Zwischenwerte immer dann anlegen, wenn die Werte sehr oft benötigt werden und eine sonst notwendige Neuberechnung (durch Ersatzfunktionen) die Rechenleistung zu sehr negativ beeinflusst. Insbesondere gilt das dann, wenn die Neuberechnung eine Synchronisation oder unregelmäßige Speicherzugriffe erfordert. Vorsicht erfordern Datenfelder, die selbst auch sehr unregelmäßige Zugriffe auslösen.*
- *Zweckmäßige Parameter-Listen bei der Funktionsdeklaration einsetzen. Bei Fortran handelt es sich oft nur um Referenzen, aber eine lange Liste kann sowohl die Rechenleistung wie auch die Übersicht stark mindern. Einsparungen erreicht man durch globale Übergaben (z.B. innerhalb von Fortran-Modulen).*
- *Speicherverwaltung möglichst selten aufrufen. Wiederverwendbarkeit von Datenobjekten reduziert den Gesamtverbrauch und Kosten durch die Speicherverwaltung; kann z.B. bei temporären Feldern häufig verwendeter Routinen vorkommen. Statt die Routine die temporären Felder ständig neu anlegen zu lassen, ist es oft besser, sie permanent aktiv zu halten und gegebenenfalls anderen Routinen zur Verfügung zu stellen.*
- *Arbeit auf lokalen Datenbereichen prozessornah bevorzugen. Unabhängig von Cache-Optimierungen ist es bei ccNUMA-Architekturen immer dann vorteilhaft, wenn man besonders häufig Daten mit dem Hauptspeicher austauschen muss. Eine Lösung besteht in dem Anlegen von lokalen Datenkopien und muss deshalb gegenüber der Erhöhung des Speicherverbrauches abgewogen werden.*

Das zweite Richtlinienpaket 5.2 beschäftigt sich näher mit der Strategie für die Datensichtbarkeitsbereichs-Attribute (Ausnahme Unterpunkt D4). Das kann man zunächst unabhängig von einem konkreten Algorithmus und dessen Parallelisierung aber abhängig von dem verwendeten Rechnersystem betrachten.

Denn die im Kapitel 4.1 beschriebene uneingeschränkte Flexibilität für die Datensichtbarkeitsbereich-Attribute ermöglicht eine von der Parallelisierung unabhängige Betrachtung, solange die Datenobjekte die dort geforderten Attribute aufweisen und die Parallelisierung oder der Algorithmus nicht einschränkt, durch welche Maßnahme (z.B. lokal oder global deklariert) das bewerkstelligt werden muss. Das bedeutet, dass die Parallelisierung grundsätzlich frei in der Wahl bleibt, welcher Unterprozess, was zu berechnen hat und was für ein Zugriff auf die Daten ihm erlaubt ist. Das ist wichtig in Bezug auf

die Verallgemeinerbarkeit bzw. Übertragbarkeit der im Kapitel 4 vorgestellten „OpenMP-hiding“ Softwaretechnik.

Bezüglich des Rechnersystems macht es dagegen schon einen Unterschied, durch welche technische Maßnahme die Datenobjekte eher prozessornah platziert sind und wann nicht. Deswegen fasst das Richtlinienpaket 5.2 die gemachten Erfahrungen aus der Sicht des Rechnersystems in einer bewährten Strategieübersicht zusammen.

Richtlinien 5.2 *Allgemeine Strategie für Datensichtbarkeitsbereichs-Attribute und Synchronisation*

*D1: Grundsätzlich sollen alle Datenvariablen innerhalb einer parallelen Region als lokale Kopie (***private***-Attribut) vorliegen.*

D2: Als nahezu gleichwertig gelten auch global und damit als **shared** *angelegte Datenobjekte, wenn jeder Unterprozess nur auf einem eigenen (großen) Teilbereich dieser Variable zugreift.*

D3: Davon abweichend können große Datenobjekte, die nach einer Erstinitialisierung nie modifiziert, sondern nur gelesen werden, als **shared** *zur Verfügung gestellt werden. Das ist eine wesentliche Maßnahme, um den Speicherverbrauch zu reduzieren.*

D4: Redundante Berechnungen für mehrere Unterprozesse sind Synchronisationen vorzuziehen, es sei denn, daraus ergeben sich zu schwerwiegende Nachteile in der Rechenleistung.

Der erste Unterpunkt D1 steigert die Leistung, da die „lokalen“ Variablen hauptsächlich im lokalen (schnellen) Speicherbereich angelegt werden (wichtig für ccNUMA). Außerdem darf hier nur jeweils ein Unterprozess darauf zugreifen, wodurch die Bedingung für eine mehrfache Ausführbarkeit (Grundannahme 5.1) gewährleistet wird.

Sowohl für D2 als auch D3 ist es auf ccNUMA-Systemen entscheidend, die Datenobjekte mit Hilfe einer verteilten bzw. parallelisierten Speicherreservierung durchzuführen, um die ansonsten auftretenden Nachteile für die Recheneffizienz abzufangen.

5.2 Parallelisierung des automatisch erzeugten Transformationscodes

Aufgrund des erhöhten Berechnungsaufwands für Gradienten ist eine Parallelisierung von automatisch erzeugten Ableitungscode notwendig. In der Literatur finden sich zahlreiche Beschreibungen für nachträgliche Parallelisierungen (z.B. in [108, 109, 110]).

Die vorliegende Arbeit betrachtet dagegen eine automatisierte Weiterverwendung einer bereits bestehenden (originalen) Parallelisierung, z.B. durch die in Kapitel 4 beschriebene „OpenMP-hiding“-Technik. Damit kann man sowohl die originale Berechnungsfunktion als auch die eigentliche Gradientenberechnung beschleunigen. In den nachfolgenden Unterkapiteln 5.2.1 und 5.2.2 werden deshalb alle Beispiele mit der originalen Parallelisierung ausgeführt, wobei $\mathbf{t}_\mathrm{i}$ für die Anzahl der an der parallelen Berechnung beteiligten Unterprozesse steht.

5.2.1 Die originale Parallelisierung mit *Skalar-Gradienten*-Modus

Man unterscheidet bei der Ableitungsberechnung üblicherweise zwei gängige Betriebsmodi, die in der Anzahl der berechenbaren Gradienten von einander abweichen. Die erste Betriebsart kann als *Skalar-Gradienten*-Modus bezeichnet werden. Dabei wird bei der Verwendung einer „Quellcode zu Quellcode“-Transformation ein speziell für diesen Fall eingeschränkter Programmcode erzeugt. Dieser kann nur solche Ableitungsobjekte anlegen und handhaben, die für die Berechnung einer einzelnen Richtungsableitung notwendig

sind. Mit einer zusätzlichen äußeren Schleife und unterschiedlicher Initialisierung lassen sich dann aber durchaus mehrere Ableitungen berechnen.

In Abbildung 40 wird dies anhand eines einfachen Beispiels und ausgehend von dem originalen Programmcode verdeutlicht. Auf der linken Seite dieser Abbildung, wird die

```
module caller_shared
  ! N : vector size
  integer, parameter :: N=100


  ! x,y : global data vectors
  real x(N)
  real y(N)
end module caller_shared
```

```
! original caller routine
use caller_shared
real α


...
call omp_vdiv(N,α,x,y)
...
```

```
module g_caller_shared
  ! N : vector size
  integer, parameter :: N=100
  ! P : number of gradients
  integer, parameter :: P=10
  ! x,y : global data vectors
  real x(N), g_x(N)
  real y(N), g_y(N)
end module g_caller_shared
```

```
! AD caller routine,SG-Mode(einfach außen)
use g_caller_shared
real α, g_α
do j = 1, P
  ... ! seeding [j] in g_x(1:N),g_α
  ...
  call g_omp_vdiv(N,α,g_α,x,g_x,y,g_y)
  ...
  ... ! evaluate [j] from g_y(1:N)
end do
```

Abbildung 40: Links: Fortran-Modul (grauer Hintergrund) und Funktionsaufruf der originalen Routine (grau-schraffierter Hintergrund); Rechts: Fortran-Modul und mehrmaliger Funktionsaufruf der Ableitungsroutine in einer Schleife über P.
■ Wichtige Kennzeichnung für die Berechnung der Ableitungen; ■ Fortran-Modul Definition; ■ Hauptabschnitt Funktionsaufruf;

originale Variablen-Deklaration im Modul „caller_shared“ und darunter der ursprüngliche Aufruf der aus Abbildung 38 bekannten Routine *vdiv* gezeigt. Hierbei wird allerdings *omp_vdiv* statt *vdiv* aufgerufen, um anzudeuten, dass es sich um eine parallelisierte Version von *vdiv* handelt. Für den Ableitungscode auf der rechten Seite sind die wichtigsten Unterschiede gegenüber der originalen Version mit grünem Hintergrund hervorgehoben. Dies sind einerseits die Konstante P und die Ableitungsobjekte der Vektoren x, y (gekennzeichnet durch g_x, g_y) in dem neuen Fortran-Modul „g_caller_shared“. Zum Anderen findet sich rechts-unten in der Ableitungsversion des Funktionsaufrufs die lokale Deklaration des Ableitungsobjektes g_α und eine Schleife mit j von 1 bis P für die Nacheinanderberechnung aller P Richtungsableitungen.

In dieser einfachen Schleife befinden sich zwei zusätzliche grün markierte Kommentare. Dabei steht „seeding“ (engl. Fachbegriff) für eine spezielle Initialisierung zur Berechnung der j-ten Ableitung. Entsprechend gibt es nach der j-ten Gradientenberechnung (in *g_omp_vdiv*) einen (mit „evaluate“ gekennzeichneten) Vermerk, dass anschließend das j-te Ergebnis (hier in g_y gespeichert) weiterverarbeitet oder kopiert wird. Im Folgenden wird nicht weiter auf die jeweiligen Details für die Initialisierung und die Weiterverarbeitung näher eingegangen, und für alle späteren Vergleiche wird die darin enthaltene Rechenarbeit oder der Speicherverbrauch vernachlässigt. Wegen kleinen aber wichtigen Unterschieden werden sie dennoch immer mit aufgeführt. Abschließend sei noch angemerkt, dass diese Variante als *Skalar-Gradienten*-Modus mit *einfachen* Datenobjekten und einer Speicherreservierung *außerhalb* des hier gezeigten Funktionsaufrufs bezeichnet wird (zusammen kurz `SG-Mode (einfach außen)`).

Der Vollständigkeit halber zeigt die Abbildung 41 die Parallelisierung von *vdiv* bzw.

g_vdiv. Entsprechend der in Kapitel 4.3 vorgeschlagenen Modularisierung findet eine Auf-

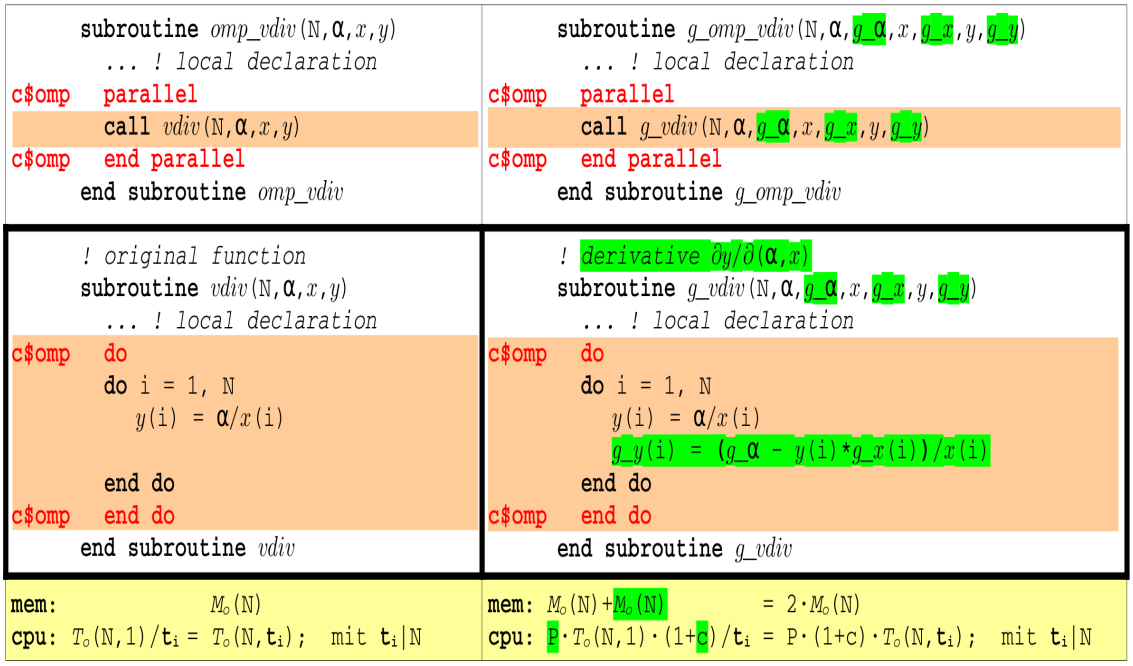


Abbildung 41: Links: Zwischenroutine *omp_vdiv* und parallelisierte Original-Routine *vdiv* (dicke Umrandung); Rechts: analog dazu erzeugter Ableitungscode mit *originaler* Parallelisierung. Das Innere der rot markierten parallelen Region wird mit einem Hintergrund in orange hervorgehoben.
Innere parallele Region; Wichtige Kennzeichnung für die Berechnung der Ableitungen; Zwischenroutine; Berechnungsroutine; Performance-Modelle

teilung der in Abbildung 38 eingeführten Beispielfunktion in eine Zwischenroutine (hier *omp_vdiv*) und die eigentliche Funktion *vdiv* (schwarze Umrandung) statt. Die Schlüsselwörter für den Anfang (**c$omp parallel**) und das Ende (**c$omp end parallel**) einer parallelen Region werden hier in *omp_vdiv* mit einem kräftigen Rot hervorgehoben. Der dazugehörige innere Bereich wird deshalb mit einer schwächeren Einfärbung (hier rosa Hintergrundfarbe) gekennzeichnet. Das ist nötig, um sie später von einer zusätzlichen parallelen Region (mit blauer Einfärbung) unterscheiden zu können. Auf der rechten Seite (rechts in Abbildung 41) befindet sich, wie schon zuvor, der erzeugte Programmcode für die Ableitungsberechnung von $\partial y/\partial(\alpha, x)$. Die wesentlichen Unterschiede zur originalen Version sind auch hier wieder mit grünem Hintergrund hervorgehoben.

Bei der Betrachtung des eigentlichen Berechnungsaufwands (rechts-unten in Abbildung 41) fällt auf, dass für die Ableitungsberechnung eine einzelne neue Programmzeile in *g_vdiv* hinzugekommen ist, die ihre Resultate in $g\_y(\mathrm{i})$ speichert. Damit erhöht sich der Berechnungsaufwand **cpu** (seriell betrachtet) erst einmal von $T_o(\mathrm{N}, 1)$ auf

$$\mathrm{P} \cdot T_o(\mathrm{N}, 1) + 3 \cdot \mathrm{P} \cdot T_o(\mathrm{N}, 1) = \mathrm{P} \cdot (1 + 3) \cdot T_o(\mathrm{N}, 1),$$

denn die Berechnung von $y(\mathrm{i})$ enthält eine und $g\_y(\mathrm{i})$ drei weitere Rechenoperationen für jedes der N Datenelemente pro Vektor und für P Richtungsableitungen. Da das Verhältnis von neuen Rechenoperationen (für die Ableitungsberechnung) gegenüber den originalen Operationen schwanken kann, werden im Folgenden die zusätzlichen Ableitungsoperationen allgemein auf eine Konstante c geschätzt. Damit ergibt sich $\mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, 1)$.

Hier kommt jetzt noch die Anzahl der verwendeten Unterprozesse $\mathbf{t}_\mathrm{i}$ hinzu, wobei der Index „i“ für die *innere* Parallelisierung (orange-markiert in Abbildung 41) steht. Idealerweise soll für den originalen Programmcode (allgemein) eine perfekte originale Parallelisierung mit $T_o(\mathrm{N}, 1)/\mathbf{t}_\mathrm{i} = T_o(\mathrm{N}, \mathbf{t}_\mathrm{i})$ und der Nebenbedingung $\mathbf{t}_\mathrm{i}|\mathrm{N}$ (mathematische Abkürzung für $\mathbf{t}_\mathrm{i}$ teilt N) gelten, genau wie in dem Beispielcode. Dann ergibt sich für die

Berechnung aller Ableitungen ein Berechnungsaufwand von $\mathbf{cpu} = \mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, 1)/\mathbf{t_i}$ pro Unterprozess und damit insgesamt $\mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, \mathbf{t_i})$ gegenüber von nur $T_o(\mathrm{N}, \mathbf{t_i})$ der originalen Funktion. Siehe dazu auch den hell-gelben Bereich mit den Performance-Modellen ganz unten in Abbildung 41.

Ausgehend von $\mathbf{mem} = M_o(\mathrm{N})$ für die originalen Funktion verdoppelt sich der Speicherverbrauch für die nacheinander stattfindende Ableitungsberechnung auf insgesamt $2 \cdot M_o(\mathrm{N})$. Denn zu jedem originalen Datenobjekt kommt hier nur ein Ableitungsobjekt, wie in Abbildung 42 angedeutet, mit gleicher Größe hinzu. Andererseits kann man auch

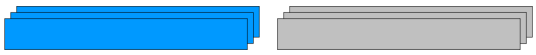


Abbildung 42: Datenstruktur-Speicherschema mit äußerer Speicherreservierung und einfachen Datenobjekten, für die Mehrfachnutzung zu unterschiedlichen Zeitpunkten.

die anderen beiden Datenstrukturvarianten links und rechts in Abbildung 43 (analog zu Abbildung 39) nachbilden. Dazu muss lediglich das Fortran-Modul und der Teil mit dem Funktionsaufruf umgeschrieben werden. Im Anhang B.1 finden sich deshalb die dazu-

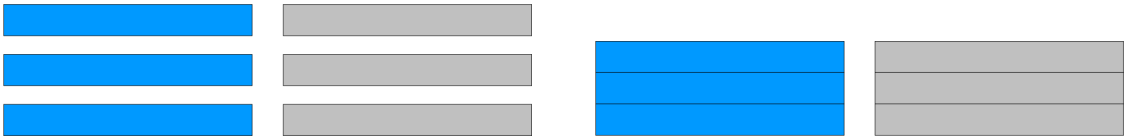


Abbildung 43: Datenstruktur-Speicherschema. Links: mit innerer *private*-Speicherreservierung; Rechts: mit äußerer Speicherreservierung und höher-dimensionalen Datenobjekten.

gehörigen Programmabschnitte in den Abbildungen 64 und 65. Für beide gelten die selben Schätzungen bezüglich des Berechnungsaufwands von $\mathbf{cpu} = \mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, \mathbf{t_i})$. Bei dem Speicherverbrauch folgt die „innen *private*“-Variante (Abbildung 64; Anhang B.1) dem bisherigen mit $\mathbf{mem} = 2 \cdot M_o(\mathrm{N})$, denn sie legt zwar innerhalb der Gradientenschleife für jede Ableitungsberechnung neue Datenvektoren im Speicher mit **allocate(...)** an, gibt aber vorher den Speicher der letzten Berechnung mit **deallocate(...)** wieder frei. Allerdings kann das ständig neue Reservieren und Freigeben von vielen Variablenobjekten die Effizienz stark negativ beeinflussen. Dabei kann es sich um Nachteile in dem Berechnungsaufwand und bei der Synchronisation handeln. Dieser Effekt kann sich noch dadurch verstärken, wenn die expliziten Speicherreservierungen innerhalb der Ableitungsfunktion (bzw. originalen Funktion) noch öfters (z.B. in einer Schleife) auftreten. Diese Variante steht deshalb stellvertretend für eine ganze Klasse mit innerer „lokaler“ Speicherreservierung.

Mit der Verwendung von höher-dimensionalen Datenobjekten (rechts in Abbildung 43 und Programmbeispiel in Abbildung 65; Anhang B.1) ergibt sich dann ein anderer Nachteil. Die Datenobjekte werden zwar wie bei der ersten Variante (Abbildung 42) außen einmalig angelegt, aber man muss P-mal soviel Speicher gleich für alle Datenobjekte anlegen ($\mathbf{mem} = 2 \cdot \mathrm{P} \cdot M_o(\mathrm{N})$). Das ist eigentlich für die originalen Variablenobjekte nicht nötig, wir nehmen dies hier aber aus praktischen Gründen an (Erklärung im nachfolgenden Abschnitt) und folgenden damit dem zuvor abgebildeten Speicherschema (rechts in Abbildung 43). So lange man nur die originale Parallelisierung verwenden will, ist dieser große Nachteil zu beachten. Er fällt, wie später noch gezeigt wird, aber nicht mehr so stark ins Gewicht, sobald man andere Arten der Parallelisierung betrachtet.

Verwendung höher-dimensionaler Datenobjekte

Ein Vorteil bei der Verwendung von höher-dimensionalen Datenobjekten ist es, dass jede Richtungsableitung direkt auf ihrem eigenem Teilbereich von g_x, g_y, g_a ungestört und

frei von Nebeneffekten berechnet werden kann. Angenommen die Vektorvariable y wurde mit $y(\mathrm{N})$ angelegt (N = Vektorlänge) und man benötigt drei Richtungsableitungen (P = 3), dann kann das dazugehörige Ableitungsobjekt mit $g\_y(\mathrm{N}, 3)$ statt mit $g\_y(\mathrm{N})$ dimensioniert werden.

Dies erhält man automatisch, wenn die originale Variable schon mit $y(\mathrm{N}, \mathrm{P})$ neu dimensioniert wurde, wie zuletzt vorgeschlagen. Das wird später noch einmal wichtig, sobald man wegen einer anderen Parallelisierung diese zusätzliche Dimension auch auf den originalen Variablenobjekten benötigt. Mit der zusätzlichen Dimension ist die erste Richtungsableitung jetzt auf dem Teilbereich $g\_y(1 : \mathrm{N}, \mathbf{1})$ und die Zweite und Dritte in $g\_y(1 : \mathrm{N}, \mathbf{2})$ und $g\_y(1 : \mathrm{N}, \mathbf{3})$ direkt berechenbar.

Die zusätzliche Dimension in der Größenordnung der Anzahl der Richtungsableitungen P wird im Folgenden auch als „Ableitungsdimension“ und bei den stochastischen Versuchen als „Versuchedimension“ (mit der Anzahl V) bezeichnet.

Um den Vorteil der höher-dimensionalen Datenobjekte zu nutzen, muss man die Ableitungsdimension auf allen notwendigen Datenvariablen einführen. Außerdem muss man dabei sicher stellen, dass eine einzelne Ableitungsberechnung auch nur auf dem jeweiligen Teilbereich arbeitet. Dazu kann man entweder, wie in den Beispielen gezeigt, nur den jeweiligen Teilbereich weiter geben oder eine lokale Variable mit der jeweiligen Indexnummer (hier 1 bis 3) übergeben. Im Hinblick auf eine spätere Parallelisierung mit vielen globalen Datenobjekten ist eine lokale Indexvariable sinnvoller.

Die spezielle Initialisierung (für die Gradientenberechnung) lässt sich direkt schon von Anfang an auf den höher-dimensionalen Datenobjekten vornehmen und ein Überführen der Ableitungsergebnisse (bei der Weiterverarbeitung) in zusätzliche Datenobjekte würde wegfallen.

5.2.2 Die originale Parallelisierung mit *Vektor-Gradienten*-Modus

Im Gegensatz zu dem *Skalar-Gradienten*-Modus bietet der im Folgenden als *Vektor-Gradienten*-Modus bezeichnete die Möglichkeit, direkt mit einem Lauf mehrere Richtungsableitungen berechnen zu lassen. Den Unterschied zeigt das in der Abbildung 44 aufgeführte Programmbeispiel. Grundsätzlich wird dabei so vorgegangen, dass zu fast jeder originalen Berechnungsanweisung auch direkt eine Gradientenschleife im Ableitungscode eingefügt wird. Eine zu diesem Zweck eingefügte Gradientenschleife ist mit grünem Hintergrund in der dick-umrandeten Routine *g_vdiv* markiert. Da diese Gradientenschleifen (im Allgemeinen mehrere) nun direkt im Ableitungscode vorkommen, gestaltet sich der Funktionsaufruf (links daneben in Abbildung 44) entsprechend einfach. Auch hier arbeitet man üblicherweise mit höher-dimensionalen Datenobjekten ähnlich dem zuletzt vorgestellten im *Skalar-Gradienten*-Modus.

Es gibt aber zwei Unterschiede. Zum einen ist die P-Dimension als erste für die Ableitungsobjekte definiert worden (siehe grün-markierte Ableitungsobjekte im Modul „g_caller_shared“), um in Fortran die Berechnung der Gradientenschleife mit einem sehr effizienten Datenzugriff zu ermöglichen. Der zweite Unterschied betrifft die spezielle Initialisierung. Bei dem *Skalar-Gradienten*-Modus hatte man noch die Möglichkeit, sie für jede Gradientenberechnung gesondert durchzuführen; jetzt muss man vor dem Ableitungsaufruf alle Gradienten vollständig initialisieren (im Programmcode der Abbildung 44 mit „**full** seeding“ gekennzeichnet).

Zur Schätzung des Berechnungsaufwands kann man verallgemeinert von $T_o(\mathrm{N}, 1)$ für eine originale Funktion mit von N abhängigen Berechnungsanweisungen ausgehen. Unter der bisherigen Annahme, dass zu jeder originalen Programmanweisung genau c-viele Ableitungsanweisungen (im *g_vdiv* Beispiel ist $c = 3$) hinzukommen, kann man auch hier für eine einzelne Ableitungsberechnung von einem seriellen Gesamtaufwand $\mathbf{cpu}_1 = T_o(\mathrm{N}, 1) + c \cdot T_o(\mathrm{N}, 1) = (1 + c) \cdot T_o(\mathrm{N}, 1)$ ausgehen.

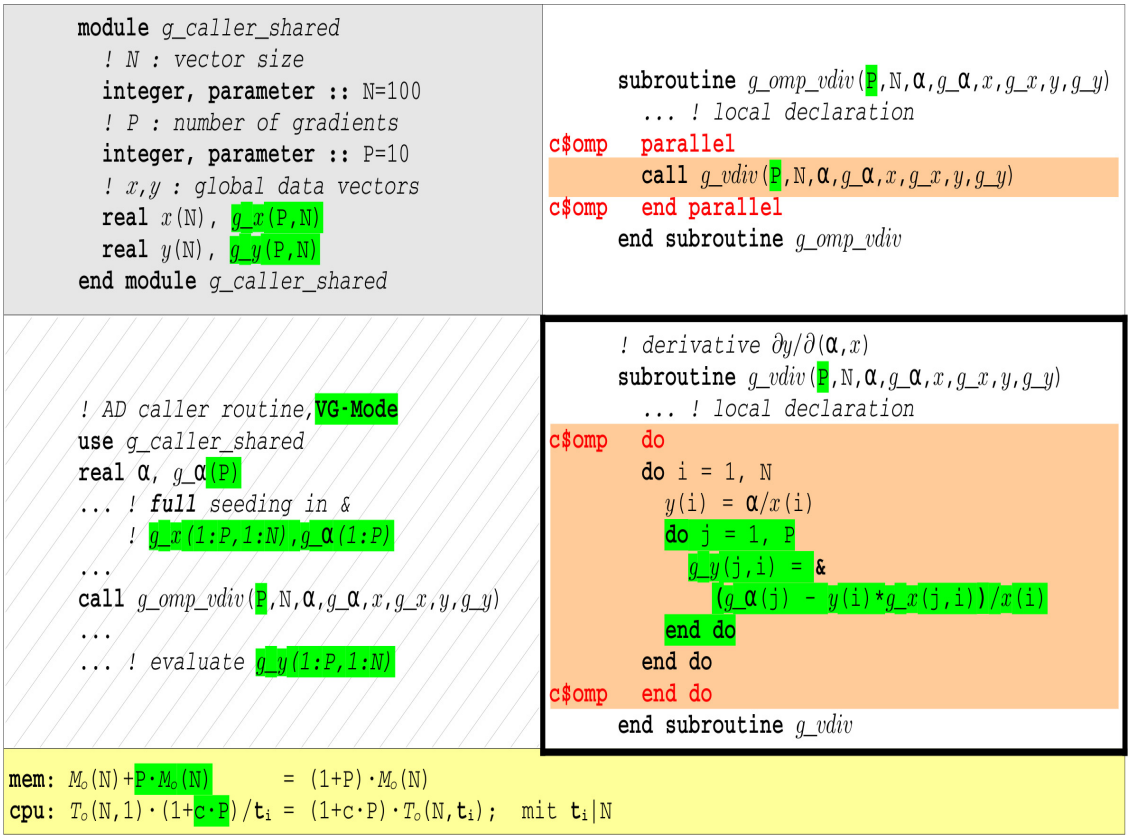


Abbildung 44: Ableitungsberechnung im *Vektor-Gradienten*-Modus mit *originaler* Parallelisierung.
Innere parallele Region; Wichtige Kennzeichnung für die Berechnung der Ableitungen; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

Für alle P Richtungsableitungen zusammen (im *Vektor-Gradienten*-Modus) erhält man entsprechend

$$\mathbf{cpu}_{\mathrm{VG}} = (1 + c \cdot \mathrm{P}) \cdot T_o(\mathrm{N}, 1),$$

wobei hier wichtig ist, dass P nicht die originalen Programmanweisungen mit skaliert. Dagegen gilt für den *Skalar-Gradienten*-Modus ein Aufwand von

$$\mathbf{cpu}_{\mathrm{SG}} = \mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, 1),$$

da die originalen Programmanweisungen genauso oft wiederholt werden müssen, wie die Anzahl der Richtungsableitungen P. Daraus ergibt sich dann ein Verhältnis von

$$\frac{\mathbf{cpu}_{\mathrm{SG}}}{\mathbf{cpu}_{\mathrm{VG}}} = \frac{\mathrm{P} \cdot (1 + c) \cdot T_o(\mathrm{N}, 1)}{(1 + c \cdot \mathrm{P}) \cdot T_o(\mathrm{N}, 1)} = \frac{(1 + c)}{(1/\mathrm{P} + c)} \tag{11}$$

und für ein hinreichend großes P erhält man:

$$\lim_{\mathrm{P} \to \infty} \frac{(1 + c)}{(1/\mathrm{P} + c)} = 1/c + 1. \tag{12}$$

Normalerweise gilt $c \geq 1$, wobei hier aber keine Aktivitätsanalyse vom AD-Werkzeug berücksichtigt wird. Für diese Analyse geht man davon aus, dass nur für die zur Ableitungsberechnung nötigen Rechenoperationen (der originalen Funktion) auch Ableitungsoperationen generiert werden. Da es deswegen für einen allgemeinen Fall mehr originale Rechenoperationen als Ableitungsoperationen geben kann, soll hier c als durchschnittliches

Verhältnis (AD-Operationen zu originalen Operationen) angenommen werden. Für einen typischen Durchschnittswert (auch nach eigenen Erfahrungen) von $c \approx 1$, ergibt sich aus $1/c + 1$ ein Verhältnis von ungefähr 2. Hieraus folgt, dass sich typischerweise durch den *Vektor-Gradienten*-Modus in etwa die Hälfte des Berechnungsaufwands gegenüber dem *Skalar-Gradienten*-Modus einsparen lässt. Das gilt dann auch unter der Einbeziehung der Parallelisierung mit $\mathbf{t}_\mathrm{i}$ Unterprozessen.

Üblicherweise werden bei dem *Vektor-Gradienten*-Modus nur die Ableitungsobjekte automatisch mit einer zusätzlichen Ableitungsdimension erzeugt. Die Abbildung 45 verdeutlicht das in Form eines Schemas zum Speicherverbrauch.

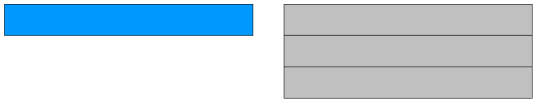


Abbildung 45: Datenstruktur-Speicherschema typisch für *Vektor-Gradienten*-Modus.

Die Schätzung des Speicherverbrauchs ergibt $\mathbf{mem}_{\mathrm{VG}} = (1 + \mathrm{P}) \cdot M_o(\mathrm{N})$. Vergleicht man das mit dem *Skalar-Gradienten*-Modus mit höher-dimensionalen Datenobjekten (rechts in Abbildung 43, $\mathbf{mem}_{\text{SG-hd}} = 2 \cdot \mathrm{P} \cdot M_o(\mathrm{N})$), erkennt man leicht, dass sich ein ähnlicher Vorteil ergibt. Im Detail berechnet sich das Verhältnis für den Speicherverbrauch dann wie

$$\frac{\mathbf{mem}_{\text{SG-hd}}}{\mathbf{mem}_{\mathrm{VG}}} = \frac{2 \cdot \mathrm{P} \cdot M_o(\mathrm{N})}{(1 + \mathrm{P}) \cdot M_o(\mathrm{N})} = \frac{2}{(1/\mathrm{P} + 1)}. \tag{13}$$

Der Speicherverbrauch vom *Vektor-Gradienten*-Modus ist damit in etwa auch nur halb so groß wie bei dem *Skalar-Gradienten*-Modus mit vollständig höher-dimensionaler Speicherreservierung (für hinreichend großes P). Zu beachten ist hierbei, dass beide parallelisierten Varianten für $\mathrm{P} \gg 1$ immer noch deutlich mehr benötigen, als die am Anfang eingeführten seriellen Versionen im *Skalar-Gradienten*-Modus ($\mathbf{mem} = 2 \cdot M_o(\mathrm{N})$).

5.2.3 Die Varianten der Gradientenparallelisierung

Dieses Unterkapitel handelt ausschließlich von der äußeren Parallelisierung direkt über der Menge der zu berechnenden Richtungsableitungen. Dazu muss man einen Blick auf den schematischen Ablaufplan der Parameterschätzung werfen. Er wird zusammen mit einer blau-gestrichelten Markierung in Abbildung 46 dargestellt. Dieser blau-markierte Bereich kennzeichnet den durch die äußere Parallelisierung erfasste Programmregion. Hier sind weder die Konvergenzsteuerung der Parameterschätzung noch die Optimierer von der Parallelisierung betroffen, denn diese können im Allgemeinen von ihrem Berechnungsaufwand her gegenüber der vollständigen Berechnung der Jacobi-Matrix vernachlässigt werden.

Zusätzlich zu dem schon aus dem Kapitel 2.2 bekannten, kann man für die hell-gelbe Iterationsschleife „**Iteration**: Jacobi-Matrix“ sagen, dass es sich nicht wie hier angedeutet, um die explizite äußere Gradientenschleife für den *Skalar-Gradienten*-Modus handeln muss. Es könnte auch eine interne Schleife sein, wie für den *Vektor-Gradienten*-Modus.

Für beide gilt, dass die im nachfolgenden Kapitel ausgeführten Gradientenparallelisierungen auf der sogenannten äußeren Parallelisierung die jeweilige Rechenarbeit auf $\mathbf{t}_\mathrm{a}$ Unterprozesse aufteilt. Die vorherige innere Parallelisierung wird hier erst einmal als entfernt betrachtet, weshalb die Performance-Modelle hier nur Bezug auf den seriellen Berechnungsaufwand $T_o(\mathrm{N}, 1)$ nehmen. Für die nachfolgenden **cpu**-Schätzungen der maximalen Rechenarbeit pro Unterprozess soll deshalb immer $\mathbf{t}_\mathrm{a}|\mathrm{P}$ gelten (P ist die Anzahl der Richtungsableitungen).

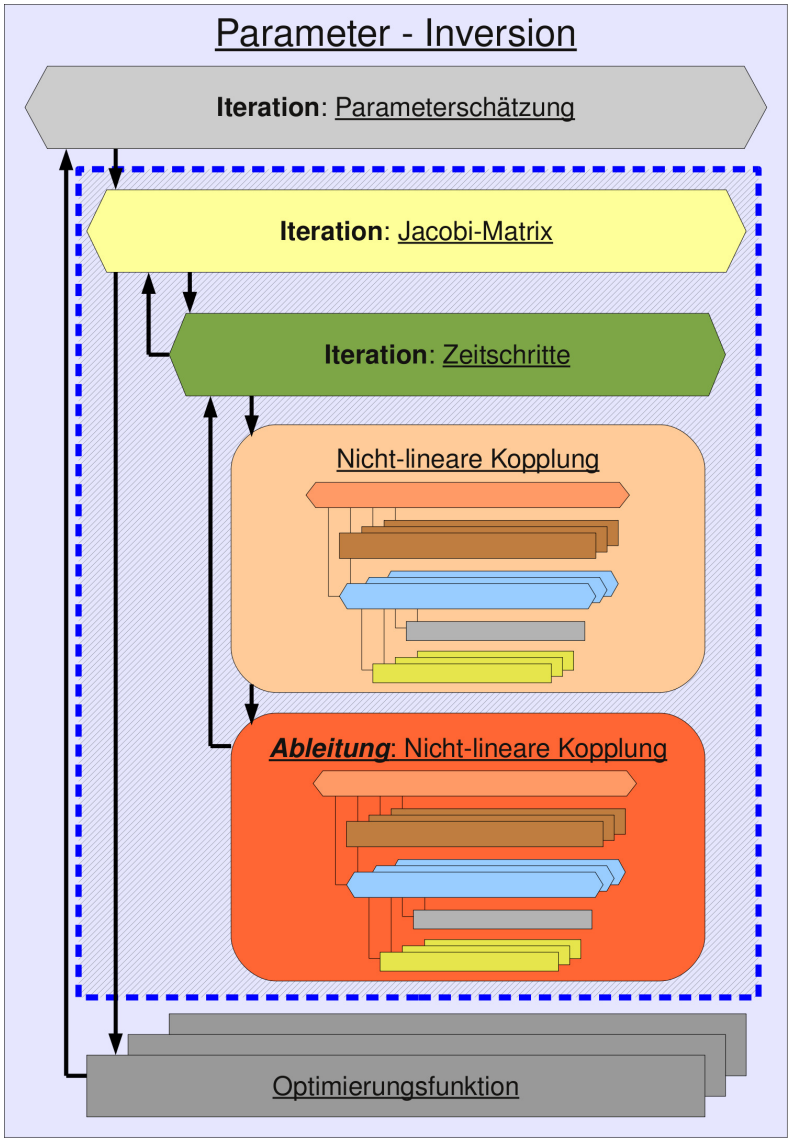


Abbildung 46: Schematischer Ablauf der Parameterschätzung (Inversion). Die parallele Region ist durch den blau-gestrichelten Bereich gekennzeichnet.

Parallelisierung mit der *Einzel*-Variante

Gerade für den rechnerisch effizienten Ableitungscode im *Vektor-Gradienten*-Modus findet man in der Literatur (allgemein: [58, 47], OpenMP: [108] oder Werkzeuge: [7]) ausreichend viele Informationen. Der einfachste Ansatz wäre, wie in Abbildung 47 gezeigt, wenn man einfach auf jeder Gradientenschleife eine eigene parallele Region (dunkel-blaue Direktiven) eröffnet, um die Berechnung (hell-blauer Hintergrund) der Richtungsableitungen auf einzelne Unterprozesse zu verteilen. Der Vorteil ist hier, dass sich die Datenstrukturen und der Funktionsaufruf von der zuletzt vorgestellten *Vektor-Gradienten*-Modus Variante mit originaler Parallelisierung (Abbildung 44) nicht unterscheidet. Dementsprechend bleibt der Speicherverbrauch äquivalent bei $\mathbf{mem} = (1 + \mathrm{P}) \cdot M_o(\mathrm{N})$.

Da man sofort nach jeder Gradientenschleife die parallele Region wieder schließt, muss man für jede weitere Gradientenschleife jeweils eine eigene neue parallele Region verwenden. Daraus resultiert ein sehr hoher Synchronisationsanteil. Bei einer Parallelisierung mit $\mathbf{t}_\mathrm{a}$ Unterprozessen führt dieser dazu, dass der theoretische Berechnungsaufwand von nur $\mathbf{cpu} = (1 + c \cdot \mathrm{P}/\mathbf{t}_\mathrm{a}) \cdot T_o(\mathrm{N}, 1)$ durch die zeitraubende Synchronisation dominiert wird.

```
! VG-Mode(Einzel-Var.)
subroutine g_omp_vdiv(P,N,α,g_α,x,g_x,y,g_y)
  ... ! local declaration
  call g_vdiv(P,N,α,g_α,x,g_x,y,g_y)
end subroutine g_omp_vdiv
```

```
      ! derivative ∂y/∂(α,x)
      subroutine g_vdiv(P,N,α,g_α,x,g_x,y,g_y)
        ... ! local declaration
        do i = 1, N
          y(i) = α/x(i)
c$omp     parallel do
          do j = 1, P
            g_y(j,i) = &
              (g_α(j) - y(i)*g_x(j,i))/x(i)
          end do
c$omp     end parallel do
        end do
      end subroutine g_vdiv
```

```
mem:   (1+P)·M_o(N)
cpu:   (1+c·P/t_a)·T_o(N,1);  mit t_a|P
sync:  S_o(N)·PR;  mit PR » fs
```

Abbildung 47: *Einzel*-Variante im *Vektor-Gradienten*-Modus und Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Zwischenroutine; Berechnungsroutine; Performance-Modelle

Verschiedene Formen der Synchronisation

Aus diesem Grund wird ab jetzt zusätzlich auch der Synchronisationsanteil (Schlüsselwort **sync**) mit geschätzt und zu einem besseren Vergleich mit herangezogen. Für die originale Parallelisierung wurde dies bisher nicht berücksichtigt, denn von besonderem Interesse sind hier nur die Gradientenparallelisierungen. Wegen der Verallgemeinerbarkeit geht man deshalb für die originale Parallelisierung einfach von einer effizienten und vor allem nicht verbesserbaren Parallelisierung aus.

Es gibt verschiedene Arten von Synchronisationen mit unterschiedlichen Ursachen, die in den kommenden Beispielen eine Rolle spielen. Die nachfolgende Übersicht zeigt die wichtigsten Synchronisationen, beginnend mit dem höchsten und damit ungünstigsten Einfluss auf die Rechenleistung bzw. deren Skalierbarkeit.

PR: Der Beginn und das Ende einer **p**arallelen **R**egion enthält neben einer allgemeinen Unterprozess-Barriere („thread barrier“) auch das Eröffnen und Schließen einer neuen Gruppe von Unterprozessen. Das ist hier die *stärkste* Synchronisation mit dem höchsten negativen Einfluss auf die Skalierbarkeit einer Parallelisierung.

B: Eine allgemeine Unterprozess-**B**arriere (z.B. explizit durch eine *barrier*-Direktive) hat oft einen um so höheren Einfluss, je mehr Unterprozesse beteiligt sind. Sie kann zu sehr vielen Datensynchronisationen mit starken Einbußen in der Skalierbarkeit führen.

Bdo: Die implizite Unterprozess-**B**arriere kann nach jedem Ende eines **do**-Arbeitsverteilungsbereiches vorkommen, sofern die *nowait*-Klausel nicht spezifiziert wurde. Zusätzlich beinhaltet sie auch jede sonstige Synchronisation zum Aushandeln des nächsten Arbeitspaketes für die beteiligten Unterprozesse.

A(**t**): Die Speicherreservierung ist auf einigen Systemen (z.B. ScaleMP) ein kritischer Engpass. Besonders nachteilig wird es genau dann, wenn sehr viele Unterprozesse gleichzeitig versuchen eine Speicherreservierung durchzuführen. Der Gesamteinfluss ist von der Anzahl **t** der beteiligten Unterprozesse und von der Menge der zu reservierenden Daten $A_o(\mathrm{N})$ abhängig (siehe dazu S.66). Deswegen wird diese Synchronisation vollständig immer mit $\mathbf{A}(\mathbf{t}) \cdot A_o(\mathrm{N})$ abgeschätzt.

fs: Eine Fehlaufteilung beim Datenzugriff (engl. „**f**alse-**s**haring“) kann auftreten, wenn zwei Unterprozesse auf zwei sehr nahe bei einander liegende Datenelemente zugreifen. Das kann dann zu zusätzlicher Synchronisation zwischen den Unterprozessen führen. Teilt man die Berechnung eines sehr keinen Vektors auf sehr viele Unterprozesse auf, können die dadurch verursachten Synchronisationen die Skalierbarkeit entscheidend beeinflussen.

[ ]: Das Auftreten einiger Synchronisationen kann durch besondere Umstrukturierungen so minimiert werden, dass sie damit teilweise vernachlässigt werden können. Gekennzeichnet werden vernachlässigbare oder stark abgeschwächte Synchronisationen durch zusätzliche eckige Klammern (z.B. **[fs]** statt **fs**).

Die letzte Gradientenparallelisierung im *Vektor-Gradienten*-Modus (*Einzel*-Variante; Abbildung 47) enthält schon in diesem kleinen Beispiel die Erzeugung einer parallelen Region für jeden der N Durchläufe (i-Schleife von 1 bis N). Falls es in der Schleife jedoch mehr als eine Ableitungsanweisung geben würde, könnten so auch mehrere Gradientenschleifen und damit mehrere parallele Regionen pro Iteration erzeugt werden. Die entsprechende Anzahl wird mit $S_o(\mathrm{N})$ (siehe S.66) bezeichnet und führt zu einer Schätzung für die Synchronisation mit $\mathbf{sync} = S_o(\mathrm{N}) \cdot \mathbf{PR}$. Damit ist ein akzeptables Skalierungsverhalten praktisch ausgeschlossen.

Erfahrungen (aufbauend auf den Arbeiten in [111]) zeigten, dass alle Parallelisierungen, bei denen für ein so einfaches Beispiel Synchronisationen in Form von parallelen Regionen oder Unterprozess-Barrieren potentiell in der Größenordnung von $S_o(\mathrm{N})$ auftreten, zu viel in der Skalierbarkeit bzw. der Effizienz einbüßen. Darüber hinaus wird im Folgenden angenommen, dass man Probleme mit der Fehlaufteilung beim Datenzugriff (siehe **fs**) vernachlässigen kann, wenn gleichzeitig Synchronisationen durch parallele Regionen oder Unterprozess-Barrieren in diesem hohen Maße (Größenordnung $S_o(\mathrm{N})$) vorkommen. Mathematisch lässt sich das mit: $\mathbf{PR}, \mathbf{B}, \mathbf{Bdo} \gg \mathbf{fs}$ ausdrücken.

Parallelisierung mit der *Master*-Variante

Eine erste Verbesserung gegenüber der *Einzel*-Variante ist in [108] beschrieben. Bei dieser wird nicht jedes Mal über den einzelnen Gradientenschleifen eine neue parallele Region eröffnet. Wie in Abbildung 66 (Anhang B.2) aufgezeigt wird, wird stattdessen die notwendige parallele Region ganz nach außen über den Funktionsaufruf von *g_omp_vdiv* gezogen. Dafür müssen aber alle originalen Programmanweisungen (Zeile „$y(\mathrm{i}) = \cdots$“ in *g_vdiv*) in einem OpenMP-*master*-Konstrukt seriell und teilweise auch synchronisiert (OpenMP-*barrier*) ausgeführt werden. Hier gibt es Ausnahmen, wie z.B. originale Schleifenkonstrukte (i=1, N in Abbildung 66), sie müssen entsprechend der beschriebenen Strategie von jedem Unterprozess vollständig („lokale“ Kopie) und damit redundant ausgeführt werden.

Sowohl für den Speicherverbrauch **mem** (Schema in Abbildung 48) als auch für den Berechnungsaufwand **cpu** gibt es keine wesentlichen Unterschiede gegenüber der *Einzel*-Variante, wenn man die redundanten Ausführungen der Schleifenkonstrukte vernachlässigt. Allerdings belegen Messungen, dass die vielen erforderlichen *barrier*-Synchronisationen

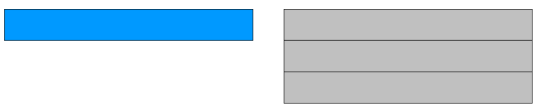


Abbildung 48: Datenstruktur-Speicherschema für *Master*-Variante.

nach den *master*-Konstrukten zwar deutlich effizienter sind als das ständige Eröffnen einer ganzen parallelen Region, aber andererseits immer noch die Skalierbarkeit stark beeinträchtigen. Hinzu kommt die Schleifenparallelisierung an sich, die je nachdem, wie sie konfiguriert ist (mit oder ohne *nowait*), zusätzliche Synchronisation (implizit man Ende

der Schleife) verursachen kann. Trotz Verbesserung muss insgesamt der verbliebene Synchronisationsanteil auf $\mathbf{sync} = S_o(\mathrm{N}) \cdot \mathbf{B} + S_o(\mathrm{N}) \cdot \mathbf{Bdo}$ geschätzt werden, was auch diesen Ansatz nur wenig geeignet für das Hochleistungsrechnen macht.

Parallelisierung mit der *Private*-Variante

Weitere Verbesserungen kann man erzielen, wenn man statt der Verwendung der *master*-Konstrukte (und ihrer anschließenden *barrier*-Synchronisation) einfach redundante Berechnungen der originalen Programmanweisungen auf allen Unterprozessen ausführen lässt.

Dann bleiben, wie in Abbildung 67 (Anhang B.2) ausgeführt, nur noch Synchronisationen auf den Gradientenschleifen selbst übrig. Diese können erforderlich sein, um einen „kritischen Datenwettlauf“ zu vermeiden. Bei statischer Aufteilung (der Schleifen-Indizes) auf die Unterprozesse oder wenn ein „kritischer Datenwettlauf“ ausgeschlossen werden kann, dürfen einige Synchronisationen der Gradientenschleifen entfallen (z.B. durch Verwendung der *nowait*-Klausel). Damit wird potentiell noch weniger synchronisiert. Deshalb wird der Synchronisationsanteil der impliziten Barrieren am Schleifenende abgewertet und für die Modellberechnung (Schätzung) mit eckigen Klammern $[\cdots]$ dargestellt.

Das ist auch der Grund, weshalb ein anderer Effekt hier in den Vordergrund treten kann. Eigene Messungen ergaben, dass besonders auf einem ScaleMP-System ständiges Reservieren und Freigeben von Speicher schnell zum Engpass $\mathbf{A}(\mathbf{t})$ für die Effizienz und Skalierbarkeit werden kann. Da man in dieser Beispielvariante nur die originalen Datenobjekte nachträglich „lokal“ anlegt, müssen auch nur diese $A_o(\mathrm{N})$-Objekte (in Abbildung 67 nur die Vektoren x, y) beim Eintritt in die parallele Region aufwändig von jedem der $\mathbf{t}_\mathrm{a}$ Unterprozesse reserviert werden. Daraus ergibt sich insgesamt eine Schätzung mit $\mathbf{sync} = [S_o(\mathrm{N}) \cdot \mathbf{Bdo}] + \mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$ für die *Private*-Variante.

Da jeder Unterprozess redundante Berechnungen auf den originalen Datenobjekten ausführen muss, benötigt jeder eine eigene „lokale“ Kopie zum Arbeiten. Hieraus resultiert ein zusätzlicher Speicherverbrauch von $\mathbf{t}_\mathrm{a} \cdot M_o(\mathrm{N})$. Das dieser Variante entsprechende Speicherschema wird in der Abbildung 49 veranschaulicht. Man erkennt hier leicht, dass

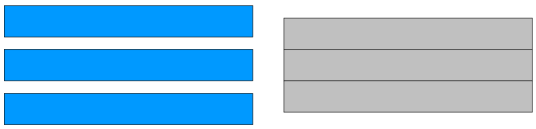


Abbildung 49: Datenstruktur-Speicherschema mit „lokalem“ Zugriff auf originale Datenobjekte und Gradientenobjekte mit höherer Dimensionierung („geteilter“-Zugriff).

der Speicherverbrauch mit $\mathbf{mem} = (\mathbf{t}_\mathrm{a} + \mathrm{P}) \cdot M_o(\mathrm{N})$ gegenüber den letzten Varianten ($\mathbf{mem} = (1 + \mathrm{P}) \cdot M_o(\mathrm{N})$ und Schema in Abbildung 48) etwas größer ausfallen kann.

Anders verhält es sich mit dem Berechnungsaufwand, denn seriell gesehen hat ein einzelner Unterprozess einen Aufwand von $\mathbf{cpu} = (1 + c \cdot \mathrm{P}) \cdot T_o(\mathrm{N}, 1)$. Aber für $\mathbf{t}_\mathrm{a}$-Unterprozesse müssen $\mathbf{t}_\mathrm{a}$-viele redundante Berechnungen ausgeführt werden. Das ergäbe eigentlich $\mathbf{cpu} = (\mathbf{t}_\mathrm{a} + c \cdot \mathrm{P}) \cdot T_o(\mathrm{N}, 1)$, wobei aber die P Richtungsableitungen und auch die redundanten Berechnungen auf $\mathbf{t}_\mathrm{a}$-Unterprozesse aufgeteilt werden. Letztendlich führt das mit $(\mathbf{t}_\mathrm{a} + c \cdot \mathrm{P})/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N}, 1)$ wieder zu der bisherigen Schätzung $(1 + c \cdot \mathrm{P}/\mathbf{t}_\mathrm{a}) \cdot T_o(\mathrm{N}, 1)$ für den maximalen Berechnungsaufwand eines Unterprozesses.

Parallelisierung mit der *Intervall*-Variante

Ein anderer Ansatz (vorgestellt in [109]) versucht nun, die verbliebenen Synchronisationen auf den Gradientenschleifen zu eliminieren. Dazu wird statisch die Zuordnung aufgeteilt, welcher Unterprozess welche Richtungsableitung zu berechnen hat. Im Detail teilt man die Gesamtzahl P der zu berechnenden Richtungsableitungen in feste Intervalle auf. Für den j-ten Unterprozess könnte das z.B. [j_min, j_max] sein. Das kann man zu Beginn vornehmen,

gleich nach dem Eröffnen der parallelen Region. Links-unten in der Abbildung 50 wird das im Funktionsaufruf-Teil durch den dunkel-blau markierten Kommentar „*compute interval* [j_min, j_max]“ angedeutet.

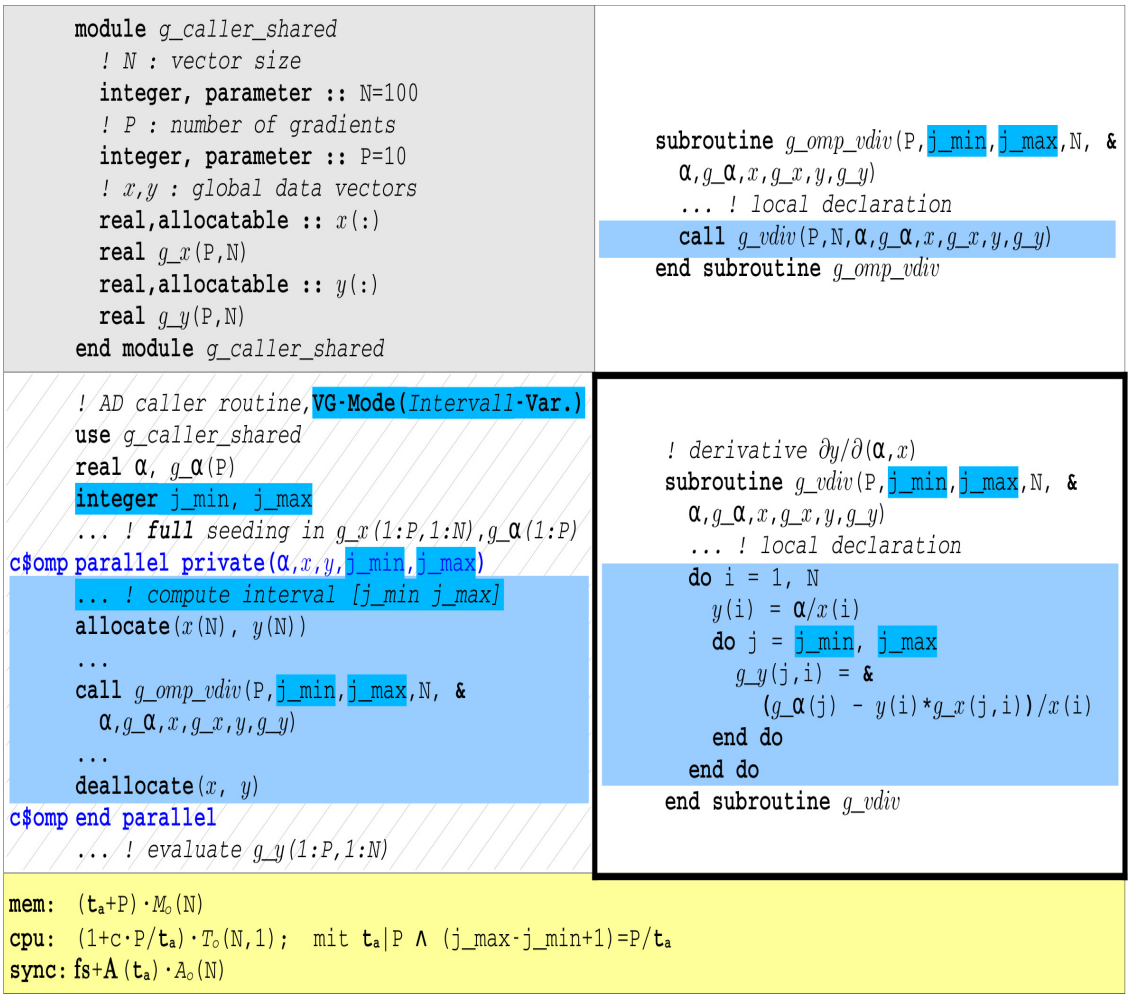


Abbildung 50: *Intervall*-Variante im *Vektor-Gradienten*-Modus und Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung für die Berechnung der Ableitungen; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

Danach entspricht die generelle Vorgehensweise zunächst der *Private*-Variante, mit Ausnahme des Beginns der ursprünglichen Gradientenschleife (siehe *g_vdiv* rechts-unten). Hier ist der Code so verändert, dass für jeden Unterprozess eine einfache Schleife ohne OpenMP-Direktiven steht. Sie zählt allerdings nicht mehr für j von 1 bis P, sondern nur noch dem eigenen Intervall entsprechend von j_min zu j_max (dunkel-blauer Hintergrund).

Der Speicherverbrauch und der Berechnungsaufwand haben sich gegenüber der *Private*-Variante nur unerheblich verändert und werden deshalb wieder analog bewertet. Dagegen fallen erstmals alle expliziten OpenMP-Synchronisationen ganz heraus. Übrig bleibt die Speicherreservierung $\mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$ im Sinne der Synchronisation, welche auch im Programmcode sichtbar ist. Nicht sichtbar dagegen ist der bisher vernachlässigte Effekt der Fehlaufteilung beim Datenzugriff.

Problem der Fehlaufteilung beim Datenzugriff

Dies ist eine für die Skalierbarkeit wichtige Besonderheit und wird hier deshalb generell für die Parallelisierungen im *Vektor-Gradienten*-Modus hervorgehoben. Alle bisher aufgeführ-

ten *Vektor-Gradienten*-Varianten setzten bei den Ableitungsobjekten auf eine zusätzliche Dimension P entsprechend der Anzahl der Richtungsableitungen (z.B. $g\_x(\mathrm{P},\mathrm{N})$ statt $g\_x(\mathrm{N})$). Diese wird, abhängig vom AD-Werkzeug, sehr oft für einen effizienten Speicherzugriff ausgelegt.

Im Detail wird dabei dafür gesorgt, dass die Berechnungen in der Gradientenschleife effizient und systematisch auf die einzelnen Daten zugreifen können. Das ist in der Regel dann der Fall, wenn die Daten der P einzelnen Richtungsableitungen direkt hintereinander im Speicher stehen. Angenommen, man hat zu einem Vektor x der Länge N die Ableitungsvariable g_x mit $g\_x(\mathrm{P},\mathrm{N})$ dimensioniert, dann können in der Gradientenschleife sehr effizient mittels einer Datenvorladung (Fachbegriff engl. „data prefetching“) über der ersten Dimension von g_x die Daten gelesen bzw. geschrieben werden. Denn in Fortran stehen die Daten der ersten Dimension direkt hintereinander im Speicher. Hat man aber gerade diesen Zugriff auf mehrere Unterprozesse verteilt, kann es vorkommen, dass zwei unterschiedliche Unterprozesse auf direkt benachbarte Daten zugreifen müssen. Liegen mehrere Datenelemente unterschiedlicher Unterprozesse in der selben „Cache-Zeile“, dann kann es zu einem Effekt der Fehlaufteilung (engl. „false-sharing“) kommen.

Bei einer Fehlaufteilung müssen die zu den Unterprozessen gehörenden Prozessorkerne mitunter direkt kommunizieren. Dies kann besonders viel Synchronisationszeit erfordern, wenn die Prozessorkerne sich keinen gemeinsamen Prozessorcache teilen (unabhängige Prozessoren oder Systeme).

Diesen Nachteil und damit die Häufigkeit von Fehlaufteilungen kann im Verhältnis zur sonstigen Rechenarbeit verringert werden, wenn es sehr viele Richtungsableitungen pro Unterprozess zu berechnen gibt, oder man die Dimension über die Anzahl der Richtungsableitungen von der ersten (inneren) auf die letzte (äußere) verschiebt. Im letzteren Fall müsste man also g_x mit $g\_x(\mathrm{N},\mathrm{P})$ dimensionieren. Dann greifen unterschiedliche Unterprozesse auf weit entfernte Datenelemente zu, die nur selten (an den Grenzen) in einer gemeinsamen „Cache-Zeile“ mit Anderen liegen.

Dieser Vorteil wird dafür aber mit dem neuen Nachteil erkauft, dass kaum noch ein effizientes Datenvorladen eingesetzt werden kann. Der Hauptgrund liegt eigentlich darin, dass außen die originale Schleife sich über die N Datenelemente erstreckt, während innen jeweils Schleifen über die P Gradienten stehen. Hier wird dann sehr oft (weil innere Schleife) auf Daten mit dem großen Abstand N zugegriffen.

Parallelisierung mit der *Spaltungs*-Variante

Im Folgenden werden zwei Lösungsmöglichkeiten für die Umgehung der Fehlaufteilung beim Datenzugriff vorgestellt. Bei der ersten handelt es sich um eine Mischung aus innerer und äußerer Dimensionierung. Hierfür spaltet man die P zu berechnenden Richtungsableitungen in beispielsweise K annähernd gleichgroße Gruppen von je ca. G Richtungsableitungen auf, vergleichbar dem „strip-mining“-Ansatz [112, 113]. Während es in diesen Veröffentlichungen eher um die Parallelisierungstechnik an sich geht, ist die Motivation hier das Vermeiden der Fehlaufteilung beim Datenzugriff. Das ist auch der Grund, warum es hier im Detail so ausführlich noch einmal aufgegriffen wird, denn das besondere Interesse gilt hier den Datenstrukturen bei einer reinen Parallelisierung mit OpenMP.

Für ein Beispiel soll idealerweise K als echter Teiler von P angenommen werden, also $\mathrm{P} = \mathrm{K} \cdot \mathrm{G}$. Ebenfalls empfiehlt es sich, dass K gleichzeitig auch der Anzahl der maximal zu Verfügung stehenden Prozessorkerne (und Unterprozesse) entspricht. Dann kann man eine Schleife über diese K Gruppen von Ableitungen nach außen ziehen, während ganz innen eine einfache lokale Schleife bis G läuft. Daher berechnet jeder der K Unterprozesse seine eigene Gruppe von G Richtungsableitungen. Hierfür bietet sich eine $(\mathrm{G},\mathrm{N},\mathrm{K})$-Dimensionierung von g_x an, gezeigt im *g_caller_shared*-Modul der Abbildung 68 des Anhangs B.2.

Die aus der *Intervall*-Variante bekannten Teilintervalle [j_min, j_max] sind hier auf die lokal zu berechnenden G Richtungsableitungen abgebildet (in *g_vdiv* rechts-unten in Ab-

bildung 68). Die Parallelisierung über der Anzahl der Gradienten wird hier nur auf der m-Schleife von 1 bis K durchgeführt (links-unten Abbildung 68). Dabei muss man sicherstellen, dass jeder Unterprozess genau einen Durchlauf ausführt und den jeweiligen m-Index nach unten bis *g_vdiv* durchreicht. Dadurch erhält man die Vorteile, dass die Fehlaufteilung beim Datenzugriff nur noch selten auftritt (nur an den Grenzen mit Abstand $\mathrm{G} \cdot \mathrm{N}$) und gleichzeitig ein effizientes Datenvorladen innerhalb eines Unterprozesses bezüglich G möglich ist. Im Nachfolgenden wird deshalb von der Vernachlässigbarkeit der Fehlaufteilungen beim Datenzugriff ausgegangen.

Für den Berechnungsaufwand ergibt sich hier ein kleiner Unterschied gegenüber der *Intervall*-Variante, da durch die äußere m-Schleife (von 1 bis K) jetzt K statt $\mathbf{t}_\mathrm{a}$ redundante Berechnungen der originalen Programmanweisungen durchgeführt werden. Das ergibt einerseits einen Faktor von $\mathrm{K}/\mathbf{t}_\mathrm{a}$ und andererseits den für jeden Unterprozess gleichen Aufwand $T_o(\mathrm{N}, 1) \cdot (1 + c \cdot \mathrm{G})$. Zusammenfassend erhält man damit die Schätzung $\mathbf{cpu} = (\mathrm{K} + c \cdot \mathrm{P})/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N}, 1)$.

Bezüglich des Speicherverbrauches sind beide Varianten vergleichbar, weshalb **mem** analog geschätzt werden kann. Der einzige wesentliche Unterschied findet sich bei dem Synchronisationsaufwand, der auf nur noch $\mathbf{sync} = [\mathbf{fs}] + \mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$ geschätzt wird.

Parallelisierung durch *Skalare*-Varianten

Eine zweite Möglichkeit zur Vermeidung von Fehlaufteilungen beim Datenzugriff ergibt sich, wenn man die Gradientenschleife über P von innen nach außen zieht. Das ist einer der zu erhoffenden Vorteile, wenn man den *Skalar-Gradienten*-Modus einsetzt, siehe Kapitel 5.2.1. Im Gegensatz zu dem *Vektor-Gradienten*-Modus befindet sich bei der Berechnung mehrerer *skalarer Gradienten* die Schleife über die P Richtungsableitungen von vornherein ganz außen. Dem zufolge greift ein Unterprozess bei der Berechnung einer einzelnen Richtungsableitung fast immer folgerichtig auf die N Datenelemente eines Vektors x bzw. g_x zu. Das soll an dieser Stelle als gegeben angenommen werden, weil von einem effizienten und optimierten Programmcode der originalen Modell-Simulation ausgegangen wird.

Möchte man nun die Berechnung der einzelnen Gradienten auf verschiedene Unterprozesse aufteilen, hat man laut Kapitel 5.2.1 bezüglich der verwendbaren Datenstrukturen mehrere Möglichkeiten. Dabei fällt die in Abbildung 42 vorgestellte Mehrfachnutzung mehr oder weniger aus, da verschiedene Unterprozesse zum gleichen Zeitpunkt jeweils ihre eigenen Arbeitskopien der Datenobjekte benötigen. Die Mehrfachnutzung war mit $\mathbf{mem} = 2 \cdot M_o(\mathrm{N})$ sehr speichereffizient, aber wie dort schon angedeutet wurde, erfordert dies einen höheren Speicherverbrauch in der Größenordnung von $2 \cdot \mathrm{P} \cdot M_o(\mathrm{N})$, da man bis zu P Kopien benötigt. Die in Abbildung 43 beschriebenen zwei Möglichkeiten („lokale“ Speicherreservierung oder höher-dimensionale Datenobjekte mit „geteilten“ Zugriff) stellen dies sicher und eignen sich damit hervorragend für eine Gradientenparallelisierung.

In der nachfolgenden Abbildung 51 sieht man den Beispielcode in einer Version mit den höher-dimensionalen Datenobjekten aufbauend auf der ursprünglichen Variante mit originaler Parallelisierung (siehe Abbildung 65 im Anhang B.2).

Zum Vergleich findet man im Anhang B.2 in Abbildung 69 auch die zweite Alternative (*Skalare*-Variante 2) analog mit „lokaler“ Speicherreservierung. Dort sieht man, dass es möglich ist, den Speicherverbrauch auf nur $\mathbf{mem} = 2 \cdot \mathbf{t}_\mathrm{a} \cdot M_o(\mathrm{N})$ zu beschränken, also abhängig von $\mathbf{t}_\mathrm{a}$ anstatt von P. Angenommen, man hat nur $\mathbf{t}_\mathrm{a}$ Unterprozesse (mit $\mathbf{t}_\mathrm{a} < \mathrm{P}$ und $\mathbf{t}_\mathrm{a} | \mathrm{P}$) zur Verfügung, dann benötigt man eigentlich auch nur $\mathbf{t}_\mathrm{a}$ Kopien gleichzeitig während der parallelen Berechnung. (Speicher für die spezielle Initialisierung und Weiterverarbeitung für alle P Gradienten wird weiterhin vernachlässigt.) Diese Verbesserung lässt sich für die Version mit höher-dimensionalen Datenobjekten (*Skalare*-Variante 3) nachholen und wird im Anhang B.2 in Abbildung 70 gezeigt.

Der Berechnungsaufwand ist für alle *Skalar-Gradienten*-Modus Varianten mit $\mathbf{cpu} = (1 + c) \cdot \mathrm{P}/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N}, 1)$ gleich, da bei allen die Routine *g_vdiv* genau P-mal aufgerufen wird und dann jeweils Berechnungen in der Größe von $(1 + c) \cdot T_o(\mathrm{N}, 1)$ statt finden.

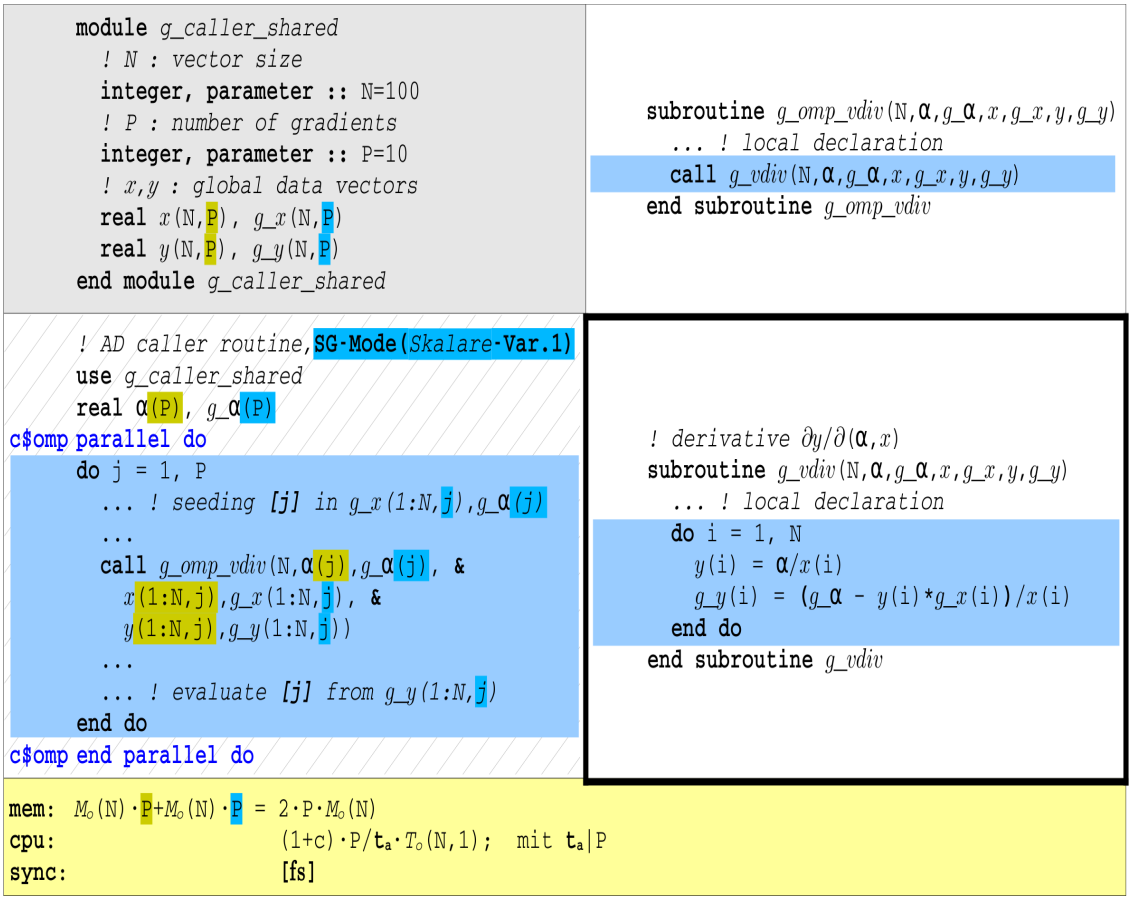


Abbildung 51: *Skalare*-Variante 1 mit höher-dimensionalen Datenobjekten, ein um Faktor P höheren Speicherverbrauch für jedes Datenobjekt und Gradientenparallelisierung. Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Wichtige Kennzeichnung für neue höher-dimensionale Datenobjekte; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

Insgesamt spielen noch zwei wesentliche Effekte bei der Synchronisation eine Rolle. Für die *Skalare*-Variante 2 mit lokaler Speichernutzung kann deren Reservierung (mit $\mathbf{sync} = 2 \cdot \mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$) noch immer zu einem Problem werden. Ein Grund ist die jetzt doppelte Anzahl von Speicherreservierungen (zusätzlich für die Ableitungsobjekte). Auf der anderen Seite muss an dieser Stelle hervorgehoben werden, dass generell die lokalen Speicherreservierungen auch innerhalb der Schleife j=1, P ($\mathbf{sync} = 2 \cdot \mathrm{P}/\mathbf{t}_\mathrm{a} \cdot \mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$) oder noch tiefer innerhalb der i-Schleife der *g_vdiv*-Routine ($\mathbf{sync} = 2 \cdot \mathrm{P}/\mathbf{t}_\mathrm{a} \cdot \mathrm{N} \cdot \mathbf{A}(\mathbf{t}_\mathrm{a}) \cdot A_o(\mathrm{N})$) stattfinden könnte. Solch ein drastisches Anwachsen (für $\mathbf{t}_\mathrm{a} \ll \mathrm{P}$) muss also umgangen werden.

Dagegen ist der zweite Effekt mit der Fehlaufteilung beim Datenzugriff für die Varianten mit höher-dimensionalen Datenobjekten eher selten, solange man die P-Dimension als letzte äußere Dimension bei den Datenobjekten definiert. Durch diese starke Verminderung der Fehlaufteilungen beim Datenzugriff (in den Performance-Modellen durch **[fs]** verdeutlicht) erhält man sehr gute Voraussetzungen für eine hohe Skalierbarkeit.

Damit liegen alle zuletzt hier vorgestellten *Skalare*-Varianten vom Synchronisationsanteil her in etwa auf dem Niveau der *Spaltungs*-Variante. Abschließend können die *Skalare*-Varianten und die *Spaltungs*-Variante als gleich gute Lösungen angesehen werden.

Vorteil durch Lastausgleich

Ein anderer wichtiger Vorteil bei dem *Skalar-Gradienten*-Modus ist ein automatischer Lastausgleich der Rechenarbeit, wenn viele Richtungsableitungen mit stark unterschiedlichem Rechenaufwand auftreten. Das kann zum Beispiel dann auftreten, wenn der Ableitungscode einen numerisch iterativen Prozess beinhaltet, der schon in dem originalen Programmcode der Modell-Simulation enthalten war.

Angenommen, ein iterativer Prozess steuert in einer Programmfunktion `f` die Genauigkeit der im Sinne der Ableitungsberechnung aktiven Zustandsgrößen $\mathbf{H}$ und $\mathbf{T}$. Dann kann sich der Berechnungsaufwand der einzelnen Richtungsableitungen bezüglich verschiedener Parameter $\mathbf{k}$ und λ erheblich von einander unterscheiden. Für ein Beispiel sollen in `f` sowohl $\mathbf{H}$ als auch $\mathbf{T}$ iterativ berechnet werden, aber nur $\mathbf{H}$ von $\mathbf{k}$ abhängen und $\mathbf{T}$ nur von λ. Dann kann es vorkommen, dass man für die Ableitungsberechnung von `f` nach $\mathbf{k}$ mehrere Iterationen zur Berechnung von $\partial\mathbf{H}/\partial\mathbf{k}$ benötigt, aber nur eine einzige oder keine für $\partial\mathbf{T}/\partial\mathbf{k}$, da sie wegen fehlender Abhängigkeit gleich Null ist. Analog soll gelten, dass für die Berechnung von $\partial\mathtt{f}/\partial\lambda$ nur $\partial\mathbf{T}/\partial\lambda$ viele Iterationen (und somit Rechenzeit) benötigt, $\partial\mathbf{H}/\partial\lambda$ dagegen sehr wenig.

Die Abbildung 52 illustriert dies an einem kleinen Laufzeit-Diagramm. Hier ist als ers-

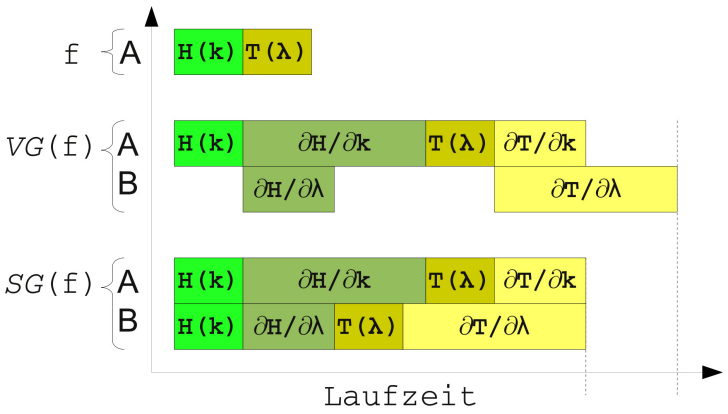


Abbildung 52: Laufzeit-Beispiel für die originale Funktion `f` (Berechnung von $\mathbf{H(k)}$ und $\mathbf{T}(\lambda)$) auf einem Unterprozess A (oben); Ableitungen im *Vektor-Gradienten*-Modus (*VG*(`f`), in der Mitte) und im *Skalar-Gradienten*-Modus (*SG*(`f`), ganz unten), wobei $\partial\mathtt{f}/\partial\mathbf{k}$ auf Unterprozess A und $\partial\mathtt{f}/\partial\lambda$ auf Unterprozess B berechnet werden.

tes (oben), zum Vergleich, die ursprüngliche Laufzeit der originalen Funktion `f` auf einem einzelnen Unterprozess A aufgeführt. Ausgehend davon folgt für die Ableitungsberechnung im *Vektor-Gradienten*-Modus zuerst die serielle Berechnung von $\mathbf{H(k)}$ auf A und dann erst die parallele Berechnung von $\partial\mathbf{H}/\partial\mathbf{k}$ und $\partial\mathbf{H}/\partial\lambda$ auf A und B (untereinander dargestellt). Erst wenn die längste Berechnung der beiden (in dem Fall $\partial\mathbf{H}/\partial\mathbf{k}$) beendet ist, wird seriell auf dem Unterprozess A das $\mathbf{T}(\lambda)$ berechnet. Nachdem die danach folgende parallele Berechnung von $\partial\mathbf{T}/\partial\mathbf{k}$ und $\partial\mathbf{T}/\partial\lambda$ fertig ist, endet die gesamte Ableitungsberechnung.

Für die Ableitungsberechnung im *Skalar-Gradienten*-Modus rechnen dagegen beide Unterprozesse A und B unabhängig von einander die Teile der originalen Funktion $\mathbf{H(k)}$, $\mathbf{T}(\lambda)$ aus und jeweils nur ihre eigenen Ableitungen nach entweder $\mathbf{k}$ oder λ. Wie man leicht sieht, ist in diesem Beispiel der *Skalar-Gradienten*-Modus kompakter und schneller, da er keinerlei Wartezeiten oder sonstige Synchronisation zwischen den Unterprozessen enthält.

Verstärkend kann hier noch hinzu kommen, dass bei einem iterativen Prozess durch unterschiedliche Initialisierungen auch deren Konvergenz (z.B. ausgedrückt durch die Anzahl der Berechnungsschritte) stark von einander abweichen kann. Das kann, wie später noch gezeigt wird, sowohl bei der Gradientenberechnung als auch bei den stochastischen Versuchen für reale Anwendungsfälle großen Einfluss auf die (parallele) Berechnungszeit haben.

Bei der Berechnung von Ableitungen im *Vektor-Gradienten*-Modus ist eine automatische Gleichverteilung der Rechenarbeit nur schwer zu erreichen und meist mit erheblichen manuellen Programmieraufwand verbunden. Für den *Skalar-Gradienten*-Modus gibt es dagegen den Vorteil, dass bei einer genügend großen Anzahl P von Richtungsableitungen, sich jeder Unterprozess dynamisch immer einen neuen Auftrag für eine neue Richtungsableitung holen kann, sobald er mit der letzten fertig ist. Dadurch können manche Unterprozesse entsprechend ihrer Auslastung mehrere Richtungsableitungen (aber dafür kürzere) berechnen als andere und es entsteht ein hilfreicher Lastausgleich. Dieser ist im Allgemeinen nicht ideal, kann aber zu einem großen Vorteil in Bezug auf die Gesamtberechnungszeit gegenüber der *Vektor-Gradienten*-Berechnung führen.

5.2.4 Kombinierte geschachtelte Parallelisierung

Ausgehend von den unterschiedlichen Möglichkeiten der Parallelisierung auf einer Ebene (nur originale oder nur Gradientenparallelisierung) liegt es nahe, diese verschiedenen Ebenen mit einander zu kombinieren, siehe dazu auch [111]. Erfahrungen mit realitätsnahen Fragestellungen (siehe Inversions-Beispiele in Kapitel 6.2.2 und synthetische Grenzen von dem Rechnersystem Nehalem-EX in Kapitel 6.2.1) zeigten, dass zahlreiche Beschränkungen in jeder einzelnen Ebene eine Kombination geradezu zwingend notwendig machen kann.

Angenommen es kann für das der Simulation zugrunde liegende Modell nur eine beschränkte Menge von unabhängig berechenbaren Arbeitseinheiten definiert werden (z.B. durch die vorgegebene Diskretisierung). Dann begrenzt diese Menge parallel berechenbarer Einheiten damit die maximale Anzahl der nutzbaren Unterprozesse (Prozessorkerne). Diese maximale Anzahl soll im Weiteren mit I bezeichnet werden. Hiermit ist für ein gegebenes Modell aber auch die stärkste Verkürzung der Gesamtberechnungszeit beschränkt. Dies ist unabhängig davon, wie viele Prozessorkerne zur Verfügung stehen. Andererseits könnte die innere Parallelisierung (wegen ihres Synchronisationsaufwands) generell nur innerhalb eines Rechenknotens (eines ScaleMP-Systems, Kapitel 2.5.3) ausreichende Skalierungseigenschaften aufweisen. Hieraus würde sich auch eine Beschränkung von I auf die Anzahl der Rechenkerne pro Rechenknoten ergeben. Damit folgt insgesamt eine Begrenzung für die sogenannte innere Parallelisierungsebene.

Analog beschränkt aber auch die Anzahl P der zu berechnenden Richtungsableitungen die Parallelisierung in der zweiten, der äußeren Ebene. Funktioniert die innere Parallelisierung auch für die Berechnung einer einzelnen Richtungsableitung (Vorarbeiten siehe Kapitel 4), kann man beide Ebenen kombinieren. Bei einer erfolgreichen gleichzeitigen Kombination beider Parallelisierungsebenen kann man daher bis zu $I \cdot P$ unabhängig berechenbare Arbeitseinheiten erwarten. Theoretisch kann man damit die Berechnungszeit auf das $1/(I \cdot P)$-Fache gegenüber einer seriellen Berechnung verkürzen, was deutlich schneller ist als nur das $1/I$- bzw. $1/P$-Fache.

Wie im letzten Unterkapitel bereits erläutert wurde, gibt es bei der Gradientenparallelisierung hinsichtlich ihrer Rechenleistung in Bezug auf den Berechnungsaufwand und den Synchronisationsanteil drei als gleichwertig erscheinende Strategien. Stellvertretend soll hier im Detail die auf der *Spaltungs*-Variante beruhende Kombination untersucht werden. Der für die geschachtelte Parallelisierung nötige Beispielcode ist in der nachfolgenden Abbildung 53 aufgeführt.

Deutlich sieht man hier die mit ocker-farbigen Hintergrund markierten OpenMP-Direktiven der geschachtelten parallelen Regionen. Die erste und äußere Region wird wie immer in dem Funktionsaufruf-Teil eröffnet. Der mit blauen Hintergrund gekennzeichnete innere Bereich (links-unten) erstreckt sich hier auch über die Eröffnung der zweiten inneren parallelen Region in der Zwischenroutine *g_omp_vdiv* (rechts-oben). Bis hierhin existieren nur $\mathbf{t}_\mathrm{a}$ Unterprozesse, aber mit Beginn der inneren Region eröffnet sich jeder der $\mathbf{t}_\mathrm{a}$ Unterprozesse eine eigene „innere“ Gruppe mit $\mathbf{t}_\mathrm{i}$ Unterprozessen.

Der nun folgende innerste Bereich (orange-blau-gestreifter Hintergrund, rechts-unten

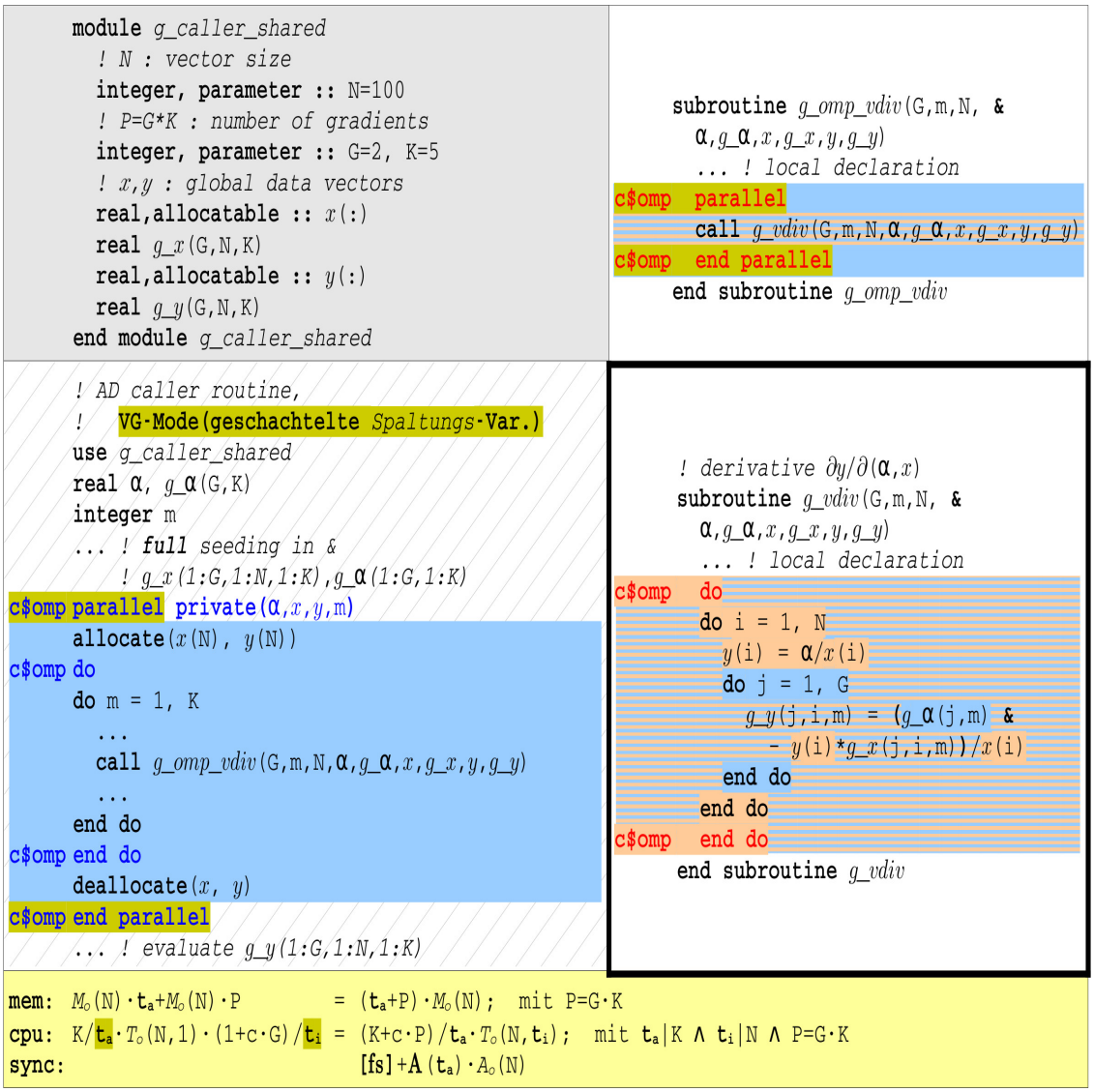


Abbildung 53: Geschachtelte Parallelisierung, aufbauend auf *Spaltungs*-Variante. Äußere parallele Region; Wichtige Kennzeichnung für innere parallele Region; Geschachtelte parallele Region; Wichtige Kennzeichnung für geschachtelte Region; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

in Abbildung 53) wird also auf einer Gesamtmenge von $\mathbf{t}_\mathrm{a} \cdot \mathbf{t}_\mathrm{i}$ Unterprozessen ausgeführt. Jede der „äußeren“ Gruppen berechnet in *g_vdiv* ausschließlich die für ihren äußeren Unterprozess festgelegte Menge G von Richtungsableitungen, mit Hilfe der originalen Parallelisierung (auf $\mathbf{t}_\mathrm{i}$ Unterprozessen).

In der Routine *g_vdiv* sind für eine bessere Übersicht die Teile des Programmcodes mit entweder orange oder hell-blauen Hintergrund eingefärbt, wenn es Unterschiede nur bezüglich einer der beiden parallelen Regionen gibt. So ist die i-Schleife von 1 bis N für alle Gruppen gleich und die Abarbeitung ist nur innerhalb einer $\mathbf{t}_\mathrm{i}$-Gruppe für die Unterprozesse unterschiedlich (Hintergrund in orange). In analoger Weise kann man die Berechnungszeile für *g_y*(j,i,m) = ··· in verschiedene Zugehörigkeiten zerlegen. Hier greifen beispielsweise alle $\mathbf{t}_\mathrm{i}$ Unterprozesse einer Gruppe auf dieselben *g_α*-Werte zu (hell-blauer Hintergrund), sind gleichzeitig aber unterschiedlich für jede Gruppe. Der Zugriff auf *y*(i) ist für jeden Unterprozess einer $\mathbf{t}_\mathrm{i}$-Gruppe ein anderer (markiert durch Hintergrund in orange), der Vektor selbst ist aber für alle Gruppen gleich. Damit zeigt der gemischte Hintergrund für *g_y* an, dass er eindeutig nur für seine bestimmte Kombination aus $(\mathbf{t}_\mathrm{a}, \mathbf{t}_\mathrm{i})$

Unterprozessen erfolgt und damit unterschiedlich bezüglich aller Gruppen-Zugehörigkeiten ist.

In ähnlicher Weise können auch die auf den *Skalar-Gradienten*-Modus beruhenden Varianten mit der originalen Parallelisierung kombiniert werden. Für die Version mit höherdimensionalen Datenobjekten (*SG*-Variante 3) findet man im Anhang B.3 (Abbildung 71) den entsprechenden Beispielcode mit geschachtelter Parallelisierung.

Ist die innere Parallelisierungsebene wirklich unabhängig von der äußeren, also vom Programmcode her ohne technische Nebeneffekte, kann man die Bewertung (in Bezug auf Speicherverbrauch und Synchronisationsaufwand) von der zugrunde liegenden Gradientenparallelisierung ableiten. Das ist an dieser Stelle aus zweierlei Gründen sinnvoll. Einmal enthalten die Betrachtungen zur Gradientenparallelisierung schon eine Bewertung zu der Speicheranforderung, die sich durch die zusätzliche innere Parallelisierung kaum ändert. Der zweite Grund liegt darin, dass der Synchronisationsanteil nicht im Detail für die innere Parallelisierung betrachtet wurde, und somit unabhängig davon ein Rückschluss nur aus der Gradientenparallelisierung gezogen werden kann. Diese Unabhängigkeit ist unter anderem in Hinblick auf eine Verallgemeinerbarkeit wichtig.

Übrig bleibt der Gesamtberechnungsaufwand, der sich für die geschachtelte *Spaltungs*-Variante mit $\mathbf{cpu} = (\mathrm{K} + c \cdot \mathrm{P})/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N}, \mathbf{t}_\mathrm{i})$ und für die geschachtelten *Skalare*-Varianten erwartungsgemäß mit $\mathbf{cpu} = (1 + c) \cdot \mathrm{P}/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N}, \mathbf{t}_\mathrm{i})$ angeben lässt. Alle liegen damit in einer Größenordnung, die bis auf eine kleine Konstante von P und T_o abhängt und im Wesentlichen die Berechnungen um einen Faktor $1/(\mathbf{t}_\mathrm{a} \cdot \mathbf{t}_\mathrm{i})$ verkürzen kann.

5.2.5 Bewertung und Vergleich

Für einen vollständigen Vergleich muss an dieser Stelle ein noch nicht angesprochener aber sehr wichtiger Unterschied hervorgehoben werden. Dieser ergibt sich aus den Anforderungen an das AD-Transformations-Werkzeug. Für die Umsetzung der *Intervall-* oder *Spaltungs*-Variante benötigt man eine Programmcode-Transformation auf dem generierten Ableitungscode, der die Berechnung in den Gradientenschleifen auf die Intervallgrenzen [j_min, j_max] oder auf die doppelte Dimensionierung mit G und K anstatt nur mit P umstellt. Im Detail muss dazu für eine zusätzliche Parameterübergabe dieser Grenzen bzw Indizes von ganz außen in die inneren Routinen gesorgt werden. Hierfür benötigt man entweder eine Erweiterung oder Unterstützung des AD-Werkzeugs selbst oder ein zusätzliches Transformations-Werkzeug. Das stellt ein nicht unerhebliches Problem dar, weshalb eine solche Anforderung als zusätzliches Bewertungskriterium für die abschließende Entscheidungsfindung mit eingeht.

Im Weiteren konnte die Unterstützung der Intervallgrenzen bzw. doppelten Dimensionierung seitens des AD-Werkzeugs nicht vorausgesetzt werden. Darüber hinaus war die Konfiguration oder Erstellung eines externen Werkzeugs für die nachträgliche Transformation nicht gegeben bzw. im Rahmen der Entwicklungsarbeit nicht gewünscht. Deshalb mussten die auf der *Intervall-* bzw. *Spaltungs*-Variante beruhenden Strategien für das dieser Arbeit zugrunde liegende Softwareprojekt als unerfüllbar angesehen werden. Eine für diesen Zweck nachträgliche manuelle Nachbearbeitung nach jeder AD-Neugenerierung ist auf Grund der im Rahmen dieser Arbeit definierten Grundanforderungen nach einer hohen Erweiterungsproduktivität ein klarer Nachteil. Damit ergibt sich ein zusätzliches Manko neben dem schon ohnehin vorhandenen Nachteil eines nur schwer implementierbaren Lastausgleichs.

Ausgehend von der Tatsache, dass für die heutigen Hochleistungsrechner der Hauptspeicher immer noch ein limitierender Faktor für die Berechnung großer Simulationen sein kann, darf man die Frage nach speichereffizienten Algorithmen nicht vernachlässigen. Auf der anderen Seite nützt es nichts, wenn man Dank einer speichereffizienten Implementation das Simulationsmodell zwar laden, aber aufgrund einer unverhältnismäßig hohen Berechnungszeit nicht schnell genug berechnen kann. Deswegen sollen der Speicherverbrauch und der Berechnungsaufwand kurz gegeneinander abgewogen werden.

Ein vollständiger Vergleich aller zuvor aufgeführten reinen Gradientenparallelisierungen untereinander ergibt, dass ein deutlicher Vorteil bei dem Speicherverbrauch nicht die damit einhergehende Beeinträchtigung der Rechenleistung aufwiegen kann. Die Betrachtung der reinen Gradientenparallelisierungen soll hier stellvertretend für die geschachtelten Parallelisierungen ausreichend sein, da sich deren Charakteristiken übertragen lassen.

Stellvertretend für die sparsamen Parallelisierungen (bezüglich Speicherverbrauch und Berechnungsaufwand) sollen die *Einzel*-Variante („EV“, auf S. 79) und als extremes Gegenbeispiel die *Skalare*-Variante 1 („SV1“, auf S. 85) betrachtet werden. Die resultierenden Verhältnisse für den Speicherverbrauch kann man in Gleichung (14) und für den Berechnungsaufwand in Gleichung (15) sehen.

$$\frac{\mathbf{mem}_{\,\mathrm{SV1}}}{\mathbf{mem}_{\,\mathrm{EV}}} = \frac{2 \cdot \mathrm{P} \cdot M_o(\mathrm{N})}{(1+\mathrm{P}) \cdot M_o(\mathrm{N})} = \frac{2}{(1/\mathrm{P}+1)} \tag{14}$$

$$\frac{\mathbf{cpu}_{\,\mathrm{SV1}}}{\mathbf{cpu}_{\,\mathrm{EV}}} = \frac{(1+c) \cdot \mathrm{P}/\mathbf{t}_\mathrm{a} \cdot T_o(\mathrm{N},1)}{(1+c \cdot \mathrm{P}/\mathbf{t}_\mathrm{a}) \cdot T_o(\mathrm{N},1)} = \frac{(\mathrm{P}/\mathbf{t}_\mathrm{a} + c \cdot \mathrm{P}/\mathbf{t}_\mathrm{a})}{(1+c \cdot \mathrm{P}/\mathbf{t}_\mathrm{a})} = \frac{(1+c)}{(\mathbf{t}_\mathrm{a}/\mathrm{P}+c)} \tag{15}$$

Für den Speicherverbrauch kann man wegen des gleichen Ergebnisses auch zu der selben Schlussfolgerung gelangen, wie sie schon für Gleichung (13) auf Seite 77 ausgeführt wurde. Damit würde der maximal erzielbare Speichervorteil ungefähr bei 50% liegen.

Dagegen muss man hinsichtlich des Berechnungsaufwands zwei Fälle unterscheiden. Einmal erhält man für $\mathbf{t}_\mathrm{a} = \mathrm{P}$ Unterprozesse nach Gleichung (15) einen Faktor von 1, also keine Vorteil. Zum Anderen erhält man mit $\mathbf{t}_\mathrm{a} = 1$ wieder ein analoges Ergebnis (Faktor $2,0$) zu dem von Gleichung (11) (mit $c \approx 1$ auf Seite 76) und kann damit den maximalen Vorteil wiederum auf Einsparungen von 50% schätzen. Allerdings lässt sich diese Einsparung nur bei serieller Berechnung ($\mathbf{t}_\mathrm{a} = 1$) erzielen, was dem eigentlichen Sinn der Parallelisierung entgegen steht.

Zudem sind die in den Synchronisationsanteilen steckenden Nachteile bei den sparsameren Varianten so gravierend, dass sie für ein leistungsfähiges bzw. effizientes Parallelisierungskonzept nicht mehr in Frage kommen. Bei den unterschiedlichen Parallelisierungsansätzen sind letztendlich die verwendeten Datenstrukturen und der Synchronisationsaufwand wichtiger als der Speicherverbrauch oder der Berechnungsaufwand.

Es verbleiben hiermit einzig die auf dem *Skalar-Gradienten*-Modus (*Skalare*-Variante 2 und 3) beruhenden kombinierten bzw. geschachtelten Parallelisierungen als finale Konzeptstrategie(n) übrig.

Ausgehend von diesem Ergebnis wurden die vorgestellten Techniken eingesetzt und auf das geophysikalische Softwareprojekt angewendet. In einer ersten Version wurde die Parallelisierung auf einer modernen Multiprozessor-Plattform getestet. Die nachfolgende Abbildung 54 stammt aus der Veröffentlichung [114] und zeigt eindrucksvoll, wie die Kombination aus originaler und zusätzlicher Gradientenparallelisierung (grüne Markierungen: comb.par.) schneller die Ergebnisse berechnen kann, als die jeweilige Parallelisierung alleine.

Für den Vergleich zeigen die blauen Werte den Beschleunigungsfaktor (engl. „speedup“) für die reine originale Parallelisierung (orig.par.) an. Der rote Graph steht dagegen für die reine Gradientenparallelisierung (deriv.par.). Zudem ist für die geschachtelte Parallelisierung (comb.par. als grüner Graph) die verwendete Unterprozess-Konfiguration angegeben. Über den grünen Dreiecken zeigt die jeweilige Konfiguration ($\mathbf{t}_\mathrm{a} \times \mathbf{t}_\mathrm{i}$) an, dass $\mathbf{t}_\mathrm{a}$ Unterprozesse für die „äußere“ Gradientenparallelisierung und gleichzeitig jede „innere“ Gruppe aus $\mathbf{t}_\mathrm{i}$ Unterprozessen besteht.

Im Detail handelt es sich bei dem Datenstrukturschema um eine gute Mischung aus den Strategien mit der lokalen Speicherreservierung und der mit höher-dimensionalen Datenobjekten („geteilter“ Zugriff). Zugrunde liegt eine frühe Optimierung für Programmteile, die besonders stark von der Speicherbandbreite abhängen, wobei ursprünglich lokale

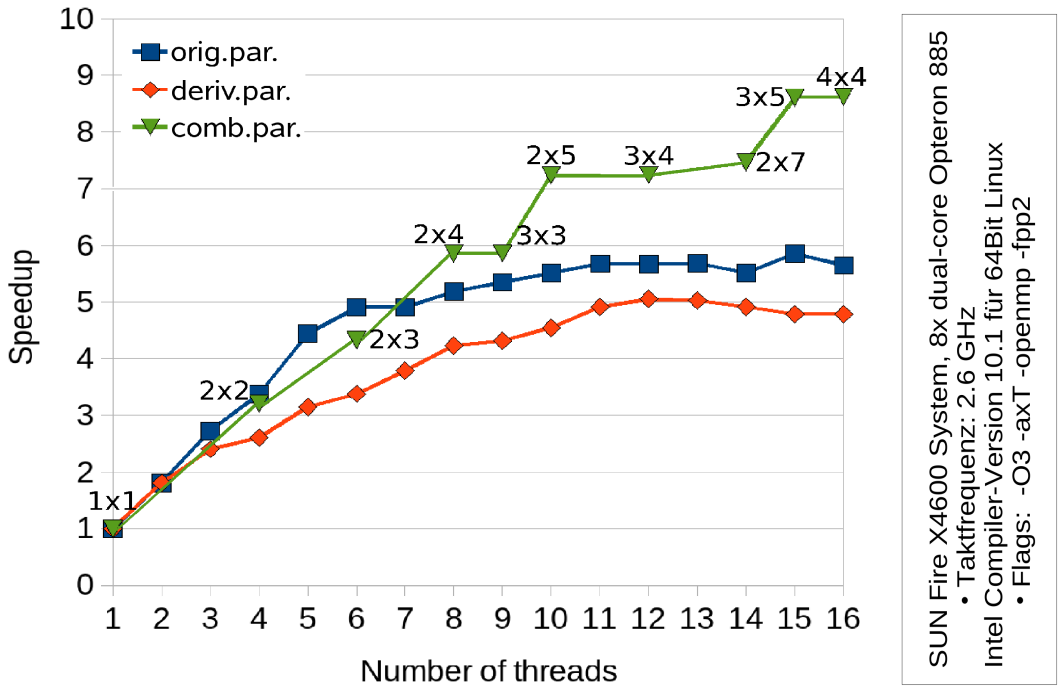


Abbildung 54: *Quelle*: aus [114]. Beschleunigungsfaktor (engl. „speedup“) einer „älteren“ *Skalar-Gradienten*-Modus Parallelisierung auf SMP-Architektur mit 16 Prozessorkernen. Die X-Achse zeigt die Anzahl der verwendeten Unterprozesse (engl. „number of threads“). Der Programmcode basiert auf einer (älteren) Version mit teilweise „lokaler“ Speicherreservierung und teilweise mit höher-dimensionalen Datenobjekten. Numerische und physikalische Modellbeschreibung siehe Anhang C.1.

Datenobjekte bevorzugt wurden, um eine möglichst gute Datenplatzierung auf ccNUMA-Architekturen zu erhalten. Auf der anderen Seite wurden viele Datenobjekte global mit einer erhöhten Dimensionierung eingesetzt, um die Implementation für die Gradientenparallelisierung zu erleichtern (siehe Vergleich zwischen äußerer und innerer lokaler Speicherreservierung in Kapitel 5.2.1). Der Nachteil der oft benötigten lokalen Speicherreservierung ist auf dieser „echten“ SMP-Architektur noch kein Problem, mehr dazu und über ScaleMP-Systeme in Kapitel 6.2.1.

5.3 Parallelisierung der stochastischen Versuche

Bei den stochastischen Verfahren benötigt man oft, wie schon erwähnt wurde, eine große Menge von unterschiedlich gestörten Modell-Simulationen. Das bedeutet, dass nicht wie bei den Ableitungsberechnungen ein transformierter Programmcode verwendet wird, sondern der fast unveränderte Programmcode der originalen Modell-Simulationssoftware. (Einmalige Vorbereitungen für AD und die zusätzlichen stochastischen Initialisierungen werden hier ignoriert.)

Im Wesentlichen muss dazu vor dem Beginn einer jeden Vorwärtsberechnung auf den Initialisierungsdaten eine systematische Störung abhängig vom stochastischen Verfahren aufgebracht werden. Im Folgenden soll die etwaige Weiterverarbeitung für eine Unsicherheitsanalyse oder Parameterschätzung wie schon zuvor vernachlässigt werden, da angenommen wird, dass der Hauptanteil des Berechnungsaufwands in der mehrfachen Vorwärtssimulation liegt. Die jeweilige Initialisierungsstörung der Daten kann zur Vereinfachung als Bestandteil der originalen Modell-Simulation angesehen werden und muss deshalb im Weiteren nicht gesondert berücksichtigt werden.

5.3.1 Basismöglichkeiten

Dementsprechend gestalten sich die in Abbildung 55 dargestellten Basismöglichkeiten für die Datenstrukturen sehr ähnlich zu denen von der Ableitungsberechnung bekannten (siehe Kapitel 5.2.1). Damit ergeben sich je nach Implementation die üblichen Unterschie-

Abbildung 55: Datenstruktur-Varianten für die Berechnung mehrerer stochastischer Versuche. Links: Mehrfachnutzung zu unterschiedlichen Zeitpunkten; Mitte: jeweils einzelne Datenobjekte; Rechts: höher-dimensionales Datenobjekt.

de bezüglich einer ständigen lokalen Speicherreservierung (mittleres Schema) oder den Fehlaufteilungs-Effekten beim Datenzugriff für das rechte Schema. Die dazu gehörigen unterschiedlichen Speicheranforderungen können völlig analog zum *Skalar-Gradienten*-Modus (nur ohne Ableitungsobjekte) betrachtet werden. Der serielle Berechnungsaufwand richtet sich nach der Anzahl der stochastischen Versuche V und ist deshalb mit **cpu** $= \mathrm{V} \cdot T_o(\mathrm{N}, 1)$ für alle gleich.

Unter Berücksichtigung der originalen Parallelisierung lässt sich das in Abbildung 56 dargestellte Programmbeispiel am besten mit dem *Skalar-Gradienten*-Modus der Ableitungsberechnung (aus Kapitel 5.2.1) vergleichen. Die hier ausgewählte Variante entspricht

```
      module s_caller_shared
        ! N : vector size
        integer, parameter :: N=100
        ! V : number of stochastic runs
        integer, parameter :: V=10
        ! x,y : global data vectors
        real x(N)
        real y(N)
      end module s_caller_shared
```

```
      subroutine omp_vdiv(N,α,x,y)
        ... ! local declaration
c$omp   parallel
        call vdiv(N,α,x,y)
c$omp   end parallel
      end subroutine omp_vdiv
```

```
      ! SV caller routine
      !   (äußere Speicherallokierung)
      use s_caller_shared
      real α
      do j = 1, V
        ... ! stochastic init. [j] &
            ! in x(1:N),α
        ...
        call omp_vdiv(N,α,x,y)
        ...
        ... ! evaluate [j] from y(1:N)
      end do
```

```
      ! original function
      subroutine vdiv(N,α,x,y)
        ... ! local declaration
c$omp   do
        do i = 1, N
          y(i) = α/x(i)
        end do
c$omp   end do
      end subroutine vdiv
```

```
mem: M_o(N)
cpu: V·T_o(N,t_i);  mit t_i|N
```

Abbildung 56: Stochastische Versucheberechnung für die Routine aus Abbildung 38 und für V = 10 stochastische Versuche.
Innere parallele Region; Wichtige Kennzeichnung; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

dem linken Datenstruktur-Schema (Abbildung 55) mit einem Speicherverbrauch von nur

mem $= M_o(\mathrm{N})$ und einem maximalen Berechnungsaufwand für die einzelnen Unterprozesse von **cpu** $= \mathrm{V} \cdot T_o(\mathrm{N}, \mathbf{t}_\mathrm{i})$ bei Verwendung von $\mathbf{t}_\mathrm{i}$ Unterprozessen.

Die Varianten der Versucheparallelisierung

Man betrachtet den schematischen Ablaufplan für die multiple Berechnung stochastischer Versuche in Abbildung 57. Wichtig ist hier die eingezeichnete blaue Markierung begin-

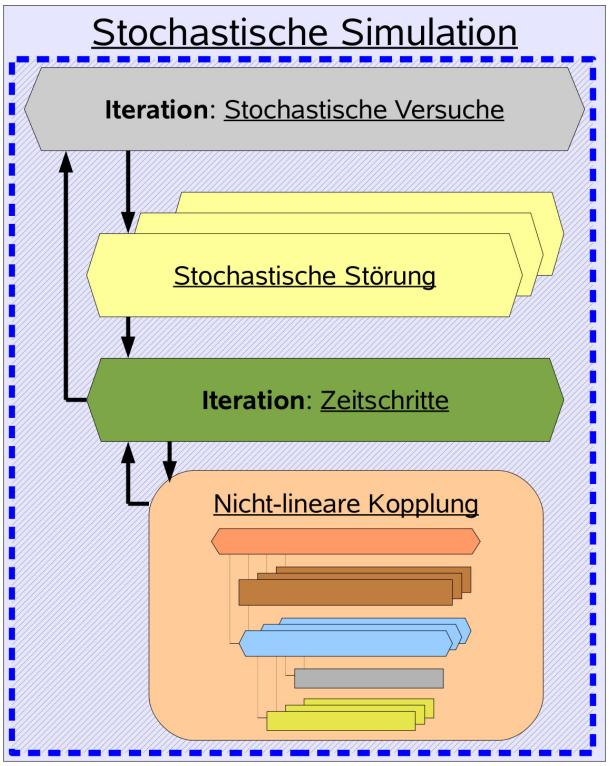


Abbildung 57: Schematischer Ablauf für die stochastischen Simulation. Die parallele Region ist durch den blau-gestrichelten Bereich gekennzeichnet.

nend mit der Iterationsschleife „**Iteration**: Stochastische Versuche“ über die Anzahl der Versuche. Während bei der Ableitungsberechnung viele verschiedene Strategien denkbar sind, bleibt hier nur noch der intuitive Ansatz übrig (abgesehen von den Parallelisierungsmöglichkeiten der originalen Modell-Simulation). Der einfachste Ansatz ist es, die verschiedenen Versuche von je einem einzelnen Unterprozess berechnen zu lassen, dargestellt im Anhang B.4 (Abbildung 72). Das ist verhältnismäßig einfach und entspricht hier dem Vorgehen wie für die Gradientenparallelisierung im *Skalar-Gradienten*-Modus. Im Weiteren wird die Parallelisierung über der Menge der Versuche (analog zur Gradientenparallelisierung) auch als „Versucheparallelisierung“ bezeichnet.

Im Allgemeinen ergibt sich aus Sicht der Parallelisierung ein Vorteil, denn die Anzahl der benötigten stochastischen Versuche ist oft sehr viel größer als die Anzahl benötigter Richtungsableitungen. Da die einzelnen Vorwärtsberechnungen rechnerisch fast vollkommen unabhängig von einander sind (ausgenommen die stochastisch abhängigen Störungen und Initialisierungen), lohnt es sich in solchen Fällen meist nicht, eine kombinierte Parallelisierung anzustreben. Dies gilt besonders dann, wenn die „originale Parallelisierung“ durch die Modelleigenschaften Beschränkungen aufweist und die Anzahl der Versuche (einige Hundert bis Tausende) ein deutliches Vielfaches der zur Verfügung stehenden Prozessorkerne ist.

Allerdings zeigen Erfahrungen gerade in Hinsicht auf stochastische Verfahren zur Parameterschätzung (z.B. ENKF [52, 53, 3, 54]), dass kleine Mengen von 50 bis 150 stochastisch verteilter Versuche durchaus schon ausreichend sein können. Bei diesen kleineren Mengen

gelangt man deswegen schnell in einen Bereich, in dem möglicherweise wieder mehr Prozessorkerne als Versuche vorliegen. In anderen Fällen kann aber auch ein zu hoher Speicherverbrauch (skalierend mit der Anzahl der parallel rechnenden Versuche) gegen eine ausschließliche Versucheparallelisierung sprechen. Für solche Fälle ist es dann wieder sehr wichtig, die nachfolgend ausgeführte Kombination aus mehreren Parallelisierungsebenen zu ermöglichen.

5.3.2 Kombinierte geschachtelte Parallelisierung

Bei einem direkten Vergleich der in Abbildung 58 gezeigten resultierenden Parallelisierung für die Kombination von originaler und Versucheparallelisierung mit der geschachtelten Parallelisierung zur Ableitungsberechnung (*Skalar-Gradienten*-Modus mit höherdimensionalen Datenobjekten in Anhang B.3; Abbildung 71) fällt eine starke Übereinstimmung auf.

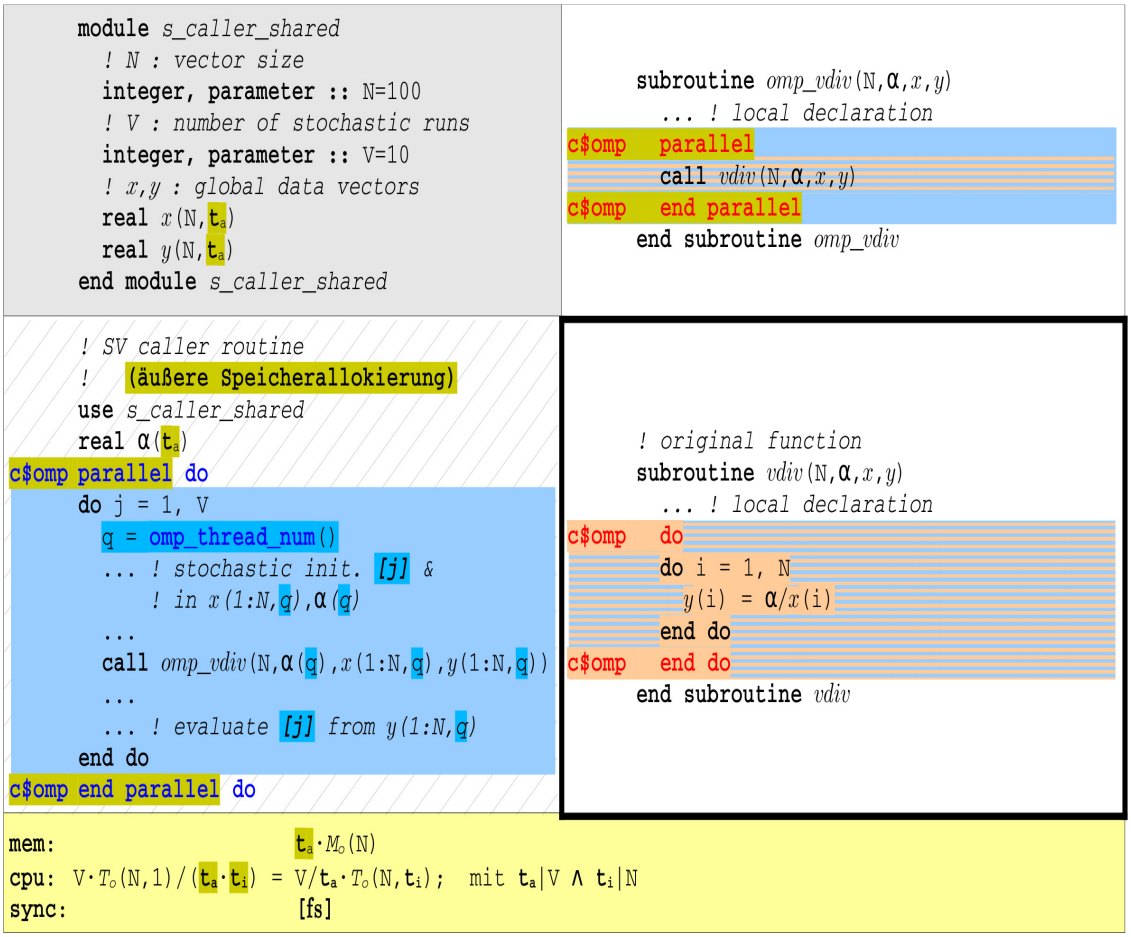


Abbildung 58: Geschachtelte Parallelisierung mit äußerer Speicherreservierung (höherdimensionale Datenobjekte) und effizienter Speichernutzung für $\mathbf{t}_\mathrm{a}$ Unterprozesse.
Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Wichtige Kennzeichnung für innere parallele Region; Geschachtelte parallele Region; Wichtige Kennzeichnung für geschachtelte Region; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

Selbst bei einer genaueren Untersuchung in Bezug auf den Berechnungsaufwand, dem Speicherverbrauch und dem Synchronisationsanteil lassen sich keine wesentlichen Unterschiede identifizieren. Nur absolut gesehen gibt es den bekannten Unterschied durch den Mehraufwand für die zusätzliche Ableitungsberechnung. Die geschachtelten Parallelisierungen mit *Skalar-Gradienten*-Modus (*Skalare*-Variante 1, 2 und 3) sind mit denen der stochastischen Versuche vergleichbar und erfordern auf dem originalen Programmcode der

Modell-Simulation die gleichen Vorbereitungen bezüglich der Datenstrukturen. Im Detail sind auch die Maßnahmen mit der automatischen Ableitungsparallelisierung („OpenMP-hiding“) kompatibel.

5.3.3 Bewertung und Vergleich

Abschließend zeigt die Abbildung 59 einen einfachen synthetischen Vergleichstest auf einem ScaleMP-System für eine Parallelisierung mit höher-dimensionalen Datenobjekten. Die Zeitmessungen wurden sukzessiv auf 1 bis 6 Rechenknoten mit jeweils vier unterschied-

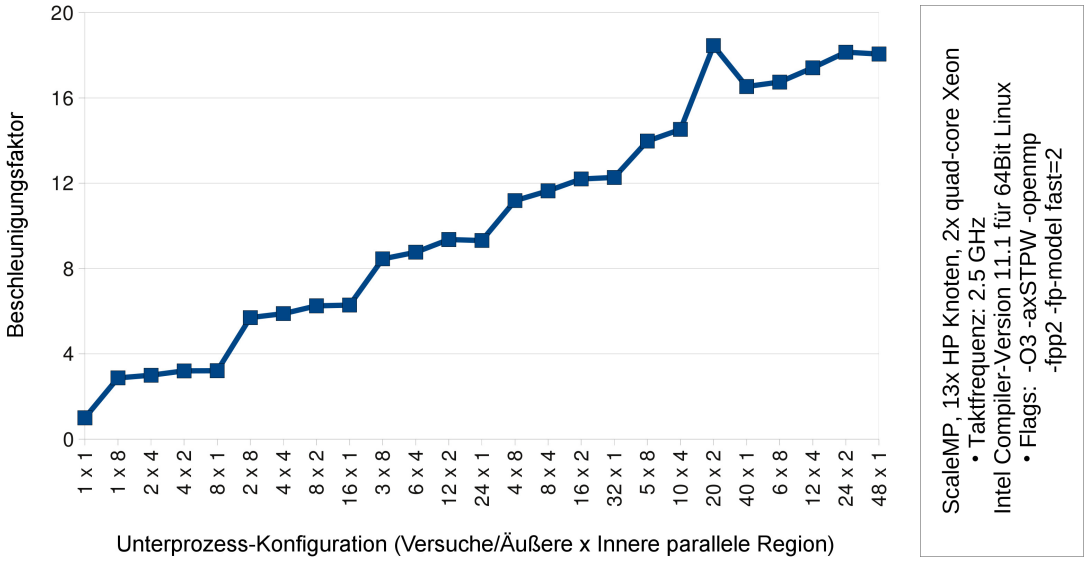


Abbildung 59: Beschleunigungsfaktor für geschachtelte Parallelisierung mit höher-dimensionalen Datenobjekten auf bis zu 48 Prozessorkerne (6 vSMP-Knoten). Hier wurde eine neuere Programmcode-Version mit fast ausschließlich höher-dimensionalen Datenobjekten verwendet. Das Test-Modell (Beschreibung im Anhang C.2) basiert auf der Diskretisierung mit $330 \times 75 \times 60$ Gitterpunkten und 320 nicht gestörten (identischen) Versuchen.

lichen Konfigurationen durchgeführt. Auf der X-Achse werden jeweils die Unterprozess-Konfigurationen in Form von $(\mathbf{t}_\mathrm{a} \times \mathbf{t}_\mathrm{i})$-Paaren aufgeführt. Die Konfigurationen unterscheiden sich in der Größe $\mathbf{t}_\mathrm{i}$ der „inneren“ Gruppe (8, 4, 2 und 1 Unterprozess(e) pro Gruppe). Die Anzahl $\mathbf{t}_\mathrm{a}$ für die „äußere“ Versucheparallelisierung entspricht dabei passend zu $\mathbf{t}_\mathrm{i}$ genau immer dem Faktor, bei dem die jeweilig beteiligten Rechenknoten voll ausgenutzt werden (Ausnahme 1×1).

Alle Messungen wurden im laufenden Regelbetrieb einmalig durchgeführt, während im Hintergrund auf den anderen 7 Rechenknoten des Gesamtsystems unspezifiziert viele (andere) Rechenläufe aktiv waren. Der blaue Graph zeigt eindrucksvoll wie gut die geschachtelte Parallelisierung für 320 (nicht gestörte) Versuche mit der Anzahl der verwendeten Rechenknoten nahezu linear skaliert. Zu beachten ist hierbei, dass einige Zeitmessungen benachteiligt sind, wenn insgesamt nur 24 oder 48 Unterprozesse verwendet werden. Da die Gesamtmenge von 320 Versuchen nicht ganzzahlig durch 24 oder 48 teilbar ist, liegt bei ihnen in der letzten Rechenphase keine gleichmäßige Lastverteilung mehr vor. Abgesehen von den sonstigen Nachteilen durch einen unspezifizierten Regelbetrieb und dem allgemeinem Systemverhalten, kann es so aber auch zu positiven Ausnahmen (z.B. 20×2-Konfiguration) kommen.

Für den Beschleunigungsfaktor (engl. „speedup“) liegt die (1×1)-Konfiguration als serielle Basiszeit zugrunde. Für einen einzelnen Rechenknoten ergibt sich aufgrund der Limitierung der Hauptspeicherbandbreite ein maximaler Beschleunigungsfaktor von $3,2$ (bester Wert mit der (8×1)-Konfiguration). Entsprechend kann man theoretisch maximal das 6-fache (also $19,2$) für 6 Knoten erwarten. Die Abbildung 59 zeigt mit einem maxima-

len Beschleunigungsfaktor von $18,4$ eindrucksvoll das Potential unter diesen nur teilweise synthetischen Bedingungen.

6 Gesamtlösungs-Konzept

Globale Bewertung der Parallelisierungsstrategien

Wichtig für einen abschließenden Vergleich der in den letzten Kapiteln aufgeführten Parallelisierungsstrategien ist auch eine mögliche Übereinstimmung zwischen der Gradientenparallelisierung und der Versucheparallelisierung. Denn die augenscheinliche Ähnlichkeit in der Parallelisierungsstrategie lässt sich auch direkt bei der Implementierung der Parallelisierung wieder finden. Das hat zur Folge, dass man die grundlegende Implementationsarbeit für die Gradientenparallelisierung auch für die Parallelisierung über die stochastischen Versuche wieder verwenden kann (*und umgekehrt*). Hieraus entsteht ein im Sinne der Erweiterungsproduktivität zusätzlicher entscheidender Vorteil bei der Verwendung der auf den *Skalar-Gradienten-Modus* beruhenden Parallelisierungsstrategien gegenüber allen anderen beschriebenen Varianten der Gradientenparallelisierung. Damit steht eindeutig fest, dass die geschachtelten Parallelisierungen (*Skalare*-Variante 2 und 3) durch ihre zahlreichen Vorteile für ein verallgemeinertes Gesamtlösungs-Konzept zu bevorzugen sind!

6.1 Anpassung der Strategie für moderne Rechner-Architekturen

In diesem Abschnitt werden die resultierenden Probleme und Lösungen bezüglich neuere Rechnersysteme vorgestellt. Abbildung 60 stellt das Beziehungsgeflecht zwischen Rechnerhardware, Parallelisierung und Simulationssoftware dar. Bei den Anforderungen der

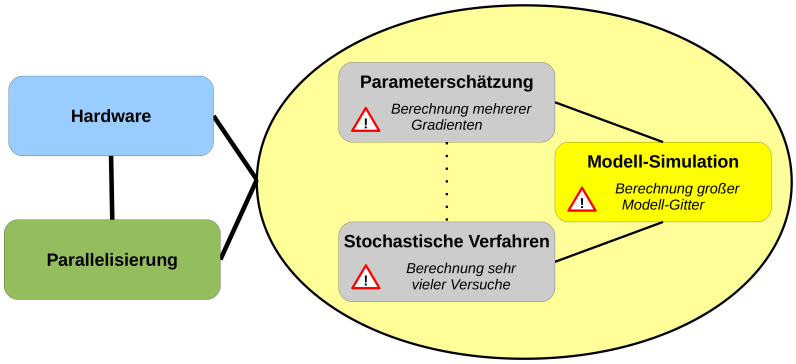


Abbildung 60: Übersicht der Abhängigkeiten zwischen Rechnerhardware, Parallelisierung und Simulationssoftware.

Simulationssoftware in Hinblick auf die Rechnerhardware handelt es sich um einen hohen Speicherverbrauch und einen hohen Berechnungsaufwand. Gleichzeitig muss die auf der Simulationssoftware implementierte Parallelisierung auf dem Rechnersystem möglichst gut skalieren, da sonst wertvolle Ressourcen verloren gehen.

Ein weiteres Mittel für eine kurze Rechenleistung und effiziente Ressourcennutzung kann der schon erwähnte Lastausgleich sein. Im weiteren Sinn kann man auch die geschachtelte Parallelisierung als Instrument zur Steigerung der Ressourcennutzung ansehen. Denn wie bereits erläutert, verbessert die geschachtelte Parallelisierung insgesamt auch die maximale Anzahl von nutzbaren Prozessorkernen. Das ist um so wichtiger, wenn jede einzelne Parallelisierungsebene für sich nur eine stark begrenzte Anzahl von Prozessorkernen unterstützt.

Ein anderes Mittel zur besseren Ressourcennutzung bezieht sich auf den Speicherverbrauch. Für die bisher als „äußere“ Parallelisierung bezeichnete Strategie geht man im Allgemeinen von einem skalierenden Speicherverbrauch bezüglich der Anzahl der einzusetzenden Prozessorkerne aus. Das muss aber nicht auch für die originale „innere“ Parallelisierung gelten. Auch in der hier zugrunde liegenden Simulationssoftware gilt, dass die originale Parallelisierung deutlich speichereffizienter ist, um eine ganze Größenordnung. Daraus ergibt sich die Möglichkeit, abhängig von dem verfügbaren Hauptspeicher des

Rechnersystems, die Konfiguration für die geschachtelte Parallelisierung auf die konkrete Modell-Anforderung (z.B. die Diskretisierung) abzustimmen. Mit dieser Konfiguration ist im Wesentlichen das Verhältnis gemeint, wie viel äußere Unterprozesse $\mathbf{t}_a$ es geben soll und mit welcher Anzahl $\mathbf{t}_i$ an inneren Unterprozessen pro Gruppe. Eine ausgewogene Konfiguration macht manche Modelle erst in angemessener Zeit berechenbar.

6.1.1 Unterstützung der ScaleMP-Plattform

Davon abgesehen sind oft Besonderheiten moderner Rechner-Architekturen zu berücksichtigen. Im Detail müssen daher zukunftsweisende Architekturen wie ScaleMP speziell unterstützt werden, um die Vorteile von Preis und Leistung nicht zu verlieren. Der Hauptunterschied zwischen „echten“ ccNUMA-Systemen und solch virtuellen wie ScaleMP ist, dass viele der system-nahen Prozesse durch Software und über ein herkömmliches (vergleichsweise langsames) Netzwerk abgebildet werden. Erfahrungen zeigen insbesondere die Bedeutung der Speicherreservierung auf [115]. Speziell implementierte Leistungstests zeigten, dass die Erstinitialisierung von über 100 Vektoren mit insgesamt 80 GByte Speicherplatzbedarf bis zu 25 Minuten dauerte. Das entsprach nicht den Erwartungen von moderner Rechnerhardware und führte zu einer Eingrenzung bei den zu verwendenden Parallelisierungsstrategien.

Demnach sind die vielen sonst für ccNUMA so günstigen lokal angelegten Datenobjekte (mit optimaler ccNUMA Speicherplatzierung) auf der ScaleMP-Plattform der Grund für starke Einbrüche in der Rechenleistung. Dieses Wissen wurde dann so genutzt, dass die frühere Strategie-Mischung aus lokaler Speicherreservierung und globalen Datenobjekten aufgehoben wurde. In einer verbesserten Variante wurde danach möglichst vollständig auf die globalen Datenobjekte mit erhöhter Dimensionierung umgeschwenkt. Der Vorteil entsteht nun dadurch, dass nur noch zu Beginn einmalig in der Initialisierungsphase alle größeren Datenobjekte initialisiert werden.

Abbildung 61 zeigt auf der ScaleMP-Plattform genau diesen erhofften Vorteil anhand eines synthetischen Vergleichstests für die Berechnung von 240 verschiedenen Richtungsableitungen. Dabei besteht die erste (ältere) Parallelisierung (blaue Quadrate) aus der be-

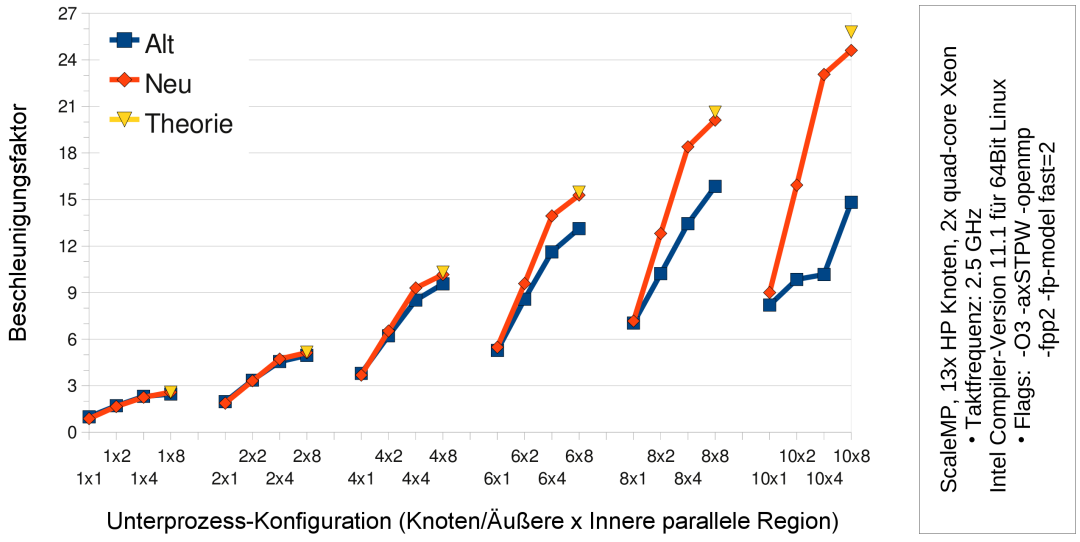


Abbildung 61: Beschleunigungsfaktor der Parallelisierungsstrategien: Alt und Neu auf ScaleMP-Architektur mit bis zu 80 Prozessorkernen (10 vSMP-Knoten). Die Erstinitialisierung der globalen Datenobjekte für Neu wird vernachlässigt und ist nicht in den Messungen enthalten. Das Test-Modell (Beschreibung im Anhang C.3) basiert auf der Diskretisierung mit $330 \times 75 \times 60$ Gitterpunkten und 240 Richtungsableitungen.

reits erwähnten Mischung von lokalen und globalen Datenobjekten, beide in Kapitel 5.2.5 beschrieben. Darüber hinaus zeigt hier der Beschleunigungsfaktor für die neue verbesserte

Variante (rote Karos), dass sie nahezu vollständig mit der Anzahl der Knoten skaliert. Auch hier liegt wieder der Bestimmung des Beschleunigungsfaktors die jeweilige serielle Rechenzeit aus der (1×1)-Konfiguration zugrunde.

Zum Vergleich ist mit den gelben Dreiecken das theoretische Skalierungsverhalten eingezeichnet. Es basiert auf der besten Rechenzeit innerhalb eines Knotens (limitierte Speicherbandbreite). Der resultierende Beschleunigungsfaktor wurde dann einfach mit der Anzahl der Knoten multipliziert. Diese theoretischen Vergleichswerte gehen also vom idealen Fall mit einer Erhöhung der Rechenleistung entsprechend der Hinzunahme von weiteren Rechenknoten aus. Die in Abbildung 61 gezeigten Ergebnisse belegen die bisherigen positiven Schätzungen in den Performance-Modellen für den Synchronisationsaufwand der Strategien mit höher-dimensionalen Datenobjekten. Auch hier sind auf der X-Achse wieder die üblichen Unterprozess-Konfigurationen in Form von $(\mathbf{t}_\mathrm{a} \times \mathbf{t}_\mathrm{i})$-Paaren angegeben, siehe dazu Kapitel 5.3.3.

Unterstützung von ccNUMA

Eine allgemeinere Unterstützung von ccNUMA-Systemen besteht darin, die schon erwähnte Erstinitialisierung von einzelnen Variablen (mehrdimensionale) so parallelisiert vorzunehmen, dass deren verschiedene Speicherblöcke verteilt auf unterschiedliche prozessornahe Speicher angelegt werden. Diese systematische Initialisierung muss natürlich zu der späteren Verwendung passen. Angenommen, man hat zwei gleichwertige Prozessoren mit jeweils eigenen prozessornahen Hauptspeicher. Weiterhin soll auf einen Vektor so zugegriffen werden, dass der erste Prozessor die erste Hälfte bearbeitet und der andere Prozessor die zweite Hälfte. Dann kann man bei entsprechender Hard- und Software (vom Betriebssystem abhängig) die erste Initialisierung so vornehmen, dass der erste Prozessor die erste Hälfte initialisiert und der Zweite die andere Hälfte. Als Folge kann dann sichergestellt werden, dass möglicherweise der größte Teil der ersten Hälfte des Vektors auch tatsächlich im lokalen Hauptspeicher des ersten Prozessors liegt und die zweite Hälfte weitestgehend im Speicher des Zweiten.

Das funktioniert umso besser, solange man jeweils ein gutes Vielfache der Standard-Speicherseitengröße (oft 4 KByte) verwendet, denn eine Speicherseite selbst kann nicht in der Mitte aufgeteilt bei verschiedenen Prozessoren lokal vorliegen. Die parallelisierte Datenplatzierung ist eine wichtige Maßnahme und die „Erstplatzierung“ der Speicherseiten (engl. auch unter dem Begriff „first touch“) oder ähnliche Mechanismen müssen auf einem ccNUMA-System gewährleistet werden, um die wichtige Speicherbandbreite ausschöpfen zu können. Die Beschreibung anderer Mechanismen wird in der Literatur unter „next touch“ oder „automatic page migration“ diskutiert.

6.1.2 Ergänzende konzeptuelle Anmerkungen

Neben den bisherigen Erläuterungen über die Verwendung von höher-dimensionalen Datenobjekten, soll ergänzend zu den Richtlinien 5.1 und 5.2 noch die nachfolgenden Anmerkungen für Klarheit sorgen.

- Die dieser Arbeit zugrunde liegenden Anwendungsszenarien erlauben eine große Gruppe von Berechnungen, die weitestgehend unabhängig von einander sind. Sie werden im Wesentlichen in einer parallelisierbaren „äußeren“ Schleife zusammengefasst. Alle Berechnungen bzw. die Programmcodes müssen dazu Eigenschaften erfüllen, die unter dem engl. Fachbegriff „thread-save“ bekannt sind. Diese Eigenschaften und damit die rechnerische Unabhängigkeit erhält man z.B. durch eine von Nebeneffekten freie und vor allem datenstrukturelle Unabhängigkeit. Erreicht werden kann dies unter anderem durch die Arbeit auf eigenen Kopien der Datenobjekte.

- Dies darf allerdings nicht dazu führen, dass ständig neue lokale Datenobjekte angelegt und aufgelöst werden (siehe Problematik Kapitel. 6.1.1). Vielmehr ist es aber

akzeptabel, dass einmal pro „äußerem“ Unterprozess alle lokal benötigten Datenobjekte angelegt werden. Das kann man durch eine „echte“ lokale Speicherreservierung zu Beginn der „äußeren“ parallelen Region erreichen. Andererseits kann man statt dessen auch außerhalb der parallelen Region die lokal benötigten Datenobjekte in Form von höher-dimensionalen *shared*-Datenobjekten anlegen. Im letzten Fall muss man dann eine parallelisierte Erstinitialisierung (oder Vergleichbares) vornehmen, um die prozessornahe Datenlokalität zu gewährleisten.

- Um den Speicherverbrauch nicht unnötig zu erhöhen, ist eine genaue Analyse über die Verwendung der einzelnen Datenobjekte hilfreich. Eine Besonderheit sind beispielsweise Daten, die zu Beginn und außerhalb jeglicher Parallelisierung nur einmal eingelesen oder initialisiert werden. Viele dieser Daten sind oft statisch, d.h. sie werden danach und speziell während den parallelen Berechnungen nicht mehr verändert, sondern ihre Werte werden in den Unterprozessen nur gelesen. Hier hat man im Allgemeinen die Wahl, ob man für eine bessere Rechenleistung lokale Kopien anlegt oder eine einzige Version für alle Unterprozesse global (*shared*) zur Verfügung stellt. Das kann viel Speicher sparen. Allerdings sollte man speziell auf einem ScaleMP-System bedenken, dass hier durchaus schon automatisch lokale Kopien je nach Bedarf erzeugt werden. Das beinhaltet manchmal auch nur die „wirklich notwendigen“ Teilbereiche der Datenobjekte. Bei dem dieser Arbeit zugrunde liegendem Suite-Projekt schien dieser Effekt recht effizient zu sein und führt damit zu einem Vorteil, da dadurch ein Teil des nötigen Programmier- oder Wartungsaufwands wegfallen kann.

- Explizit soll an dieser Stelle nochmal darauf hingewiesen werden, dass alle anderen Datenobjekte, die während der parallelen Ausführung von den Unterprozessen modifiziert werden, am besten in Form einer eigenen Kopie (oder Teilbereichs) vorliegen. Ansonsten führt das fast immer zu Synchronisationen, die es zu vermeiden gilt.

6.2 Skalierbarkeit der Konzeptlösung - Parallelisierung auf ScaleMP

Abschließend werden vergleichende Messungen auf unterschiedlichen Rechnersystemen vorgestellt. Es werden dazu Ergebnisse von Vergleichstests vorgestellt, die auf modernen bzw. jüngst veröffentlichten Rechnerplattformen ausgeführt wurden. Dass soll die generelle Eignung der hier vorgestellten Konzeptlösung bestätigen und das Potential aufzeigen, welches man auch auf zukunftsweisenden Rechner-Architekturen (allg. Kapitel 2.5), wie ScaleMP (spez. Kapitel 2.5.3), oder aktuellen „echten“ SMP-Maschinen erzielen kann.

6.2.1 Vergleichstest - synthetische Modelle

Als erstes werden zwei ScaleMP-Systeme verglichen, die sich in den verwendeten Rechenknoten unterscheiden. Das ursprüngliche System wird, genauso wie bisher auch, einfach als ScaleMP-System bezeichnet und besteht aus 13 Rechenknoten, wobei jeder mit zwei 4Kern-Prozessoren Intel-Xeon (Harpertown) bestückt ist. Hierbei handelt es sich jeweils um ein System mit *echten* „gemeinsamen Speicher“. Aufgrund der geteilten Speicherbandbreite erreicht man einen maximalen Beschleunigungsfaktor von nur $3,2$ für den Simulationscode und für 1 bis 8 Unterprozesse.

Das moderne Vergleichssystem basiert auf 16 Rechenknoten mit je zwei Nehalem 4Kern-Prozessoren der Firma *Intel*™ [116] und wird im Folgenden als ScaleMP-N bezeichnet. Außerdem handelt es sich bei jedem Knoten des ScaleMP-N Systems selbst um ein kleines ccNUMA-System. Die entsprechend höheren Erwartungen konnten für den Simulationscode mit einem maximalen Beschleunigungsfaktor von $7,5$ (1 bis 8 Unterprozess) erfüllt werden.

Die Abbildung 62 (veröffentlicht in [115]) zeigt für beide Systeme und je für drei verschiedene Unterprozess-Konfigurationen das Skalierungsverhalten. Die in der Legende und in Klammern spezifizierte Unterprozess-Konfiguration ($\mathbf{t}_\mathrm{a} \times \mathbf{t}_\mathrm{i}$-Notation) bezieht sich

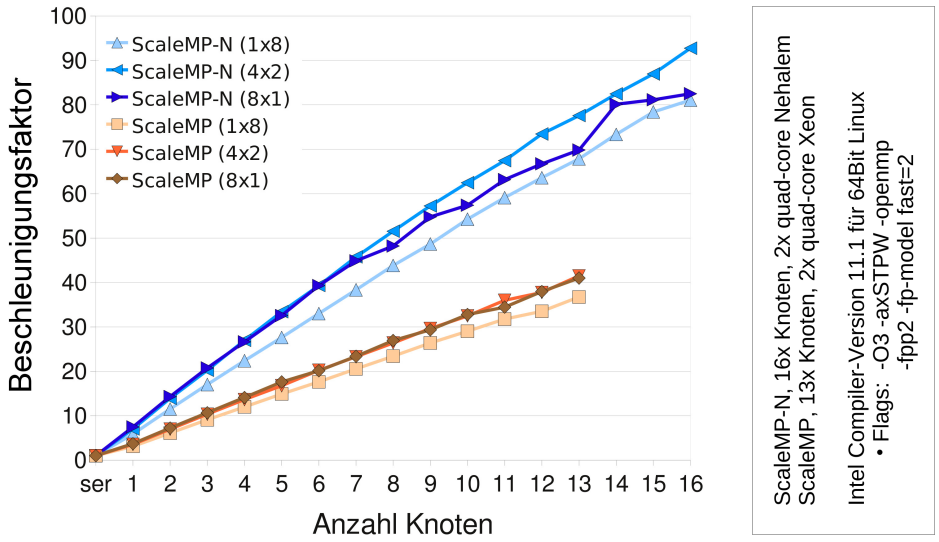


Abbildung 62: Beschleunigungsfaktor für verschiedene Unterprozess-Konfigurationen auf zwei vSMP-Systemen. Modell-Diskretisierung mit $330 \times 75 \times 60$ Gitterpunkten und 320 Richtungsableitungen, Modellbeschreibung siehe Anhang C.4. *Quelle*: eigene Bearbeitung nach [115].

dabei auf jeden einzelnen verwendeten Rechenknoten. Für drei Knoten mit einer (4×2)-Konfiguration sind beispielsweise insgesamt $3 \cdot 4 = 12$ äußere Unterprozesse aktiv, die jeweils eine Gruppe mit 2 inneren Unterprozessen zur Berechnung eröffnen. Entsprechend wurde systematisch das selbe Testmodell unter Verwendung von 1 bis 13 (16) Rechenknoten berechnet.

Für beide Systeme sieht man, nahezu unabhängig von der Unterprozess-Konfiguration, einen sehr starken linearen Zusammenhang mit der Anzahl der zur Testmodell-Berechnung eingesetzten Rechenknoten. Das deutet auf ein sehr gutes Skalierungsverhalten für diese Art von modernen Architekturen an und bestätigt damit die bisherigen Vorbetrachtungen.

Für ein ccNUMA basierend auf einem System mit echten „gemeinsamen Speicher“ sollte man analog einen ähnlich linearen Effekt, skalierend mit der Anzahl der Prozessoren (bzw. Prozessorsockel, engl. „socket“), beobachten können. Aus der Reihe der aktuellen und erst kürzlich verfügbar gewordenen Systeme stammt die Nehalem-EX Plattform mit vier 8Kern-Prozessoren (8 Rechenkerne). Hier stehen somit insgesamt $4 \times 8 = 32$ Prozessorkerne zur Verfügung.

Den jeweiligen Beschleunigungsfaktor für verschiedene Unterprozess-Konfigurationen ($\mathbf{t}_\mathrm{a} \times \mathbf{t}_\mathrm{i}$-Notation) zeigt der blaue Graph in der Abbildung 63. Zum Vergleich befinden sich hier zusätzlich auch rot-markierte Werte für die Beschleunigungsfaktoren eines wesentlich kleineren Systems („gemeinsamer Speicher“), bestückt mit zwei 4Kern-Prozessoren (ähnlich zu einem Knoten der ScaleMP-N Plattform). Die Abbildung 63 erlaubt zwei wichtige Beobachtungen. Einerseits skaliert auch hier der Beschleunigungsfaktor linear mit der Anzahl der verwendeten Prozessoren (bzw. Sockel). Dazu betrachtet man am besten nur die Werte, die bezüglich eines Prozessors (8 Kerne) die selbe Unterprozess-Konfiguration aufweisen. Für die Konfiguration (8×1) wäre das z.B. die Reihe (8×1), (16×1), (24×1) und (32×1), welche mit schwarzen Kreisen in der Abbildung 63 hervorgehoben ist.

Andererseits sieht man in dieser Abbildung auch Konfigurationen, in denen nicht nur alle Unterprozesse in der äußeren Gradientenparallelisierung rechnen. Stattdessen sieht man auch gegensätzliche Fälle, bei denen alle Unterprozesse nur für die innere originale Parallelisierung verwendet werden. Während für die gänzlich Äußere (32×1), mit 320 Richtungsableitungen, genug unabhängige Berechnungsblöcke zur Verfügung stehen, gilt dies hier für die Innere nicht. Das Testmodell weist mit 1,485 Mio. Gitterpunkten nicht zu jedem Zeitpunkt genug unabhängige Teilaufgaben auf, um idealerweise alle Unterprozesse

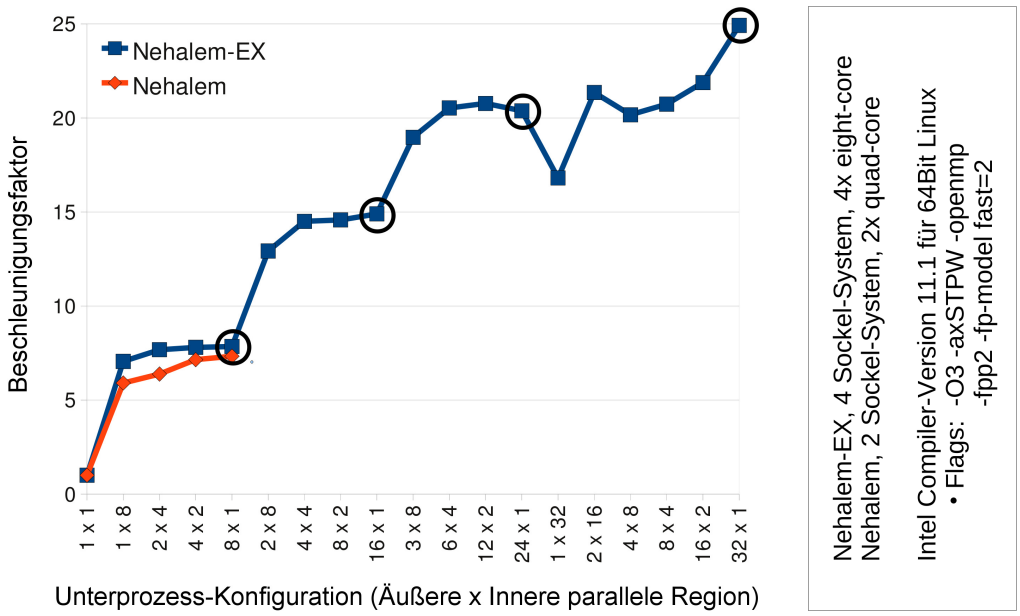


Abbildung 63: Beschleunigungsfaktor für verschiedene Unterprozess-Konfigurationen auf zwei Systemen mit „gemeinsamen Speicher“. Die schwarzen Kreise markieren die Prozessor-Unterprozess-Konfiguration (8×1). Modell-Diskretisierung mit $330 \times 75 \times 60$ Gitterpunkten und 320 Richtungsableitungen, Modellbeschreibung siehe Anhang C.4.

unter Last zu halten (modellabhängige Limitierung der inneren Parallelisierungsstufe). Deswegen ist der Beschleunigungsfaktor für (1×32) deutlich schlechter als für (32×1).

Allerdings sieht man hier auch, dass die Beschleunigungsfaktoren für die Unterprozess-Konfigurationen (2×16) und (16×2) nahezu gleichwertig sind. An dieser Stelle lohnt sich also eine genaue Abwägung, welche Konfiguration man verwenden sollte. Im Detail benötigt man bei diesem Beispiel für die (2×16)-Konfiguration ca. nur 1/8 soviel Hauptspeicher wie für die (16×2)-Konfiguration.

6.2.2 Vergleichstest - realitätsnahe Modelle und Anwendung

Die Erfahrungen und die daraus resultierenden Richtlinien konnten erfolgreich in der dieser Arbeit zugrunde liegenden Simulationssoftware umgesetzt werden, obwohl es sich dabei um einen ständig fortlaufenden Entwicklungsprozess handelte. Diese hohe Erweiterungsproduktivität ermöglichte einen Leistungsvergleich an großen, realitätsnahen Modellen.

Entsprechend werden im Nachfolgenden für drei wissenschaftlich relevante Modelle Zeitmessungen diskutiert. Sie zeigen auf, wie sich die Erwartungen aus den synthetischen Vergleichstests auf die realen Anwendungen: Parameterschätzung und stochastische Simulation, übertragen lassen. Dabei wurden alle drei Beispielanwendungen auf der hauptsächlich zur Verfügung stehenden ScaleMP-Plattform (13 Rechenknoten mit je zwei 4Kern-Prozessoren) im laufenden Betrieb berechnet. Allerdings standen dabei immer zwei Knoten nicht dem regulären Rechnerbetrieb zur Verfügung.

Bei den drei realitätsnahen Anwendungen handelt es sich im Detail um zwei Parameterschätzungen und eine stochastische Simulation. Die stochastische Simulation nutzte auf Grund des hohen zu erwartenden Berechnungsaufwands alle 11 verfügbaren Rechenknoten und lief damit mehr oder weniger ungestört (ähnlich den synthetischen Bedingungen). Dagegen beanspruchten die Parameterschätzungen jeweils (zu verschiedenen Zeitpunkten) nur 6 der 11 Rechenknoten. Auf den anderen 5 Knoten liefen während dessen unregelmäßig und nicht weiter spezifizierbare Anwendungen (reale Bedingungen). Weiterführende geophysikalische und numerische Details zu allen drei Anwendungsbeispielen findet man im Anhang D ab Seite 135.

Modell DIPL_S3D_NC_2_3B_3 (kurz D.1)

Bei der ersten Parameterschätzung D.1 handelt es sich, wie bei den anderen auch, um einen Pumpversuch einer „Hot Dry Rock“ (HDR) Versuchseinrichtung des Soultz-Reservoirs (siehe [117, 118, 119]). Hierbei ist D.1 von allen drei nachfolgenden Beispielen das am feinsten diskretisierte Modell mit der kleinsten Anzahl an simulierten Zeitschritten. In dem D.1-Beispiel wurden 11 Inversionsiterationen (Parameterschätzungen) für P = 6 Gesteinsparameter berechnet. Für eine einzelne Parameterschätzung (einzelne Inversionsiteration) wurden genau 6 Richtungsableitungen (Ableitungen nach den Gesteinsparametern) berechnet und anschließend einem Optimierungsalgorithmus übergeben. Danach folgte, noch innerhalb der Inversionsiteration, eine Art „line-search“, welche im Wesentlichen die (parallelisierte) Auswertung mehrerer einfacher Modell-Simulationen zur Folge hat.

Insgesamt wurde für die vollständige Berechnung aller 11 Iterationen von D.1 und für die Unterprozess-Konfiguration (6×8) nur 5659 *min* aufgewendet. An dieser Stelle benötigt man einen Vergleichswert, um zu beurteilen, wie stark man von einer effizienten Parallelisierung profitiert. Dazu wurde (analog auch für die anderen Beispiele) die erste Inversionsiteration durch eine serielle Berechnung nachgestellt, allerdings nur für eine aktive Richtungsableitung (statt 6). Die Berechnung einer einzelnen Iteration (inklusive Optimierungsfunktion, siehe Kapitel 2.2) für eine ausschließliche Parameterschätzung nach dem ersten Gesteinsparameter erforderte ca. 1000 *min* auf einem einzelnen Prozessorkern (1×1-Konfiguration). Das soll als Vergleichswert auch deshalb genügen, weil eine komplette serielle Nachberechnung über mehr als einen Monat hinweg auf Grund von regelmäßigen etwa alle zwei Wochen stattfindenden Wartungszyklen ohnehin nicht am Stück durchführbar gewesen wäre.

Der Einfachheit halber kann man zusätzlich grob schätzen, dass für die gleichzeitige Inversion nach 6 Gesteinsparametern (bei jeder Iteration) auch ungefähr die 6-fache Rechenleistung benötigt wird. Für dieses Modell ist außerdem bekannt, dass die erste Inversionsiteration nach 410 *min* abgeschlossen war. Das ergibt dann einen geschätzten Beschleunigungsfaktor von $6 \cdot 1000\ min/410\ min \approx 14,6$. Nach dem synthetischen Vergleichstest hätte man für 6 Knoten und einer (6×8)-Konfiguration ca. $17,6$ (vergleiche Abbildung 62; ScaleMP (1×8) auf 6 Knoten) erwarten können. Dieser erste Vergleich zwischen einem synthetischen und realitätsnahem Beispiel zeigt, wie gut die Erwartungen umgesetzt bzw. erfüllt werden können, obwohl das reale Anwendungsbeispiel sehr viel ungünstigere Modelleigenschaften (gröbere Diskretisierung, weniger Richtungsableitungen) für die Parallelisierung aufweist.

Darüber hinaus kann man für eine Betrachtung über alle 11 Iterationen einmal zusätzlich annehmen, dass für jede Iteration in etwa gleich viel berechnet werden muss. Das führt zu einer groben Schätzung von $6 \cdot 11 \cdot 1000\ min = 66000\ min \approx 1100\ h \approx 46\ Tage \approx 1,5\ Monate$ für die vollständige serielle Parameterschätzung. Dem steht die zuvor genannte parallele Rechenzeit von $5659\ min \approx 4\ Tage \approx 1/2\ Woche$ gegenüber, was in etwa einem Beschleunigungsfaktor von $11,7$ entspricht. Dies erscheint noch weniger effizient als $17,6$, ist aber auf Grund der sehr groben Annahmen immer noch beeindruckend hoch für ein realitätsnahes Anwendungsbeispiel. Insbesondere konnte wegen der Verkürzung der Gesamtrechenzeit von geschätzten $1,5$ Monaten auf nur eine halbe Woche deutlich schneller auf Modell-Fehler reagiert werden oder andere Anpassungen ausprobiert werden.

Modell DIPL_S3D_NC_4Q_10 (kurz D.2)

Bei dieser zweiten Parameterschätzung handelt es sich um ein ungefähr halb so feindiskretisiertes Modell mit doppelt so vielen Zeitschritten. Für die gleichzeitige Parameterschätzung von $P = 12$ Gesteinseigenschaften durch $D = 8$ Inversionsiterationen benötigte man hier 5664 *min*. Dabei wurde zur Berechnung auf 6 Rechenknoten die Unterprozess-Konfiguration (12×3) gewählt. Eine serielle Nachberechnung der ersten Inversionsiteration mit nur einer Richtungsableitung erforderte hier 1003 *min*. Eine Schät-

zung des Beschleunigungsfaktors B_f kann man durch

$$B_f = \frac{P \cdot D \cdot t_{ser}}{t_{par}}$$

berechnen, wobei gilt:

P : Anzahl der Parameter (12 für Modell D.2),

D : Anzahl der Inversionsiterationen (8 für Modell D.2),

t_{ser} : serielle Rechenzeit für $P = 1$ und $D = 1$ (1003 min) und

t_{par} : parallele Rechenzeit (5664 min für P und D entsprechend Modell D.2).

Eingesetzt erhält man mit $B_f = (12 \cdot 8 \cdot 1003\ min)/5664\ min$ insgesamt einen Faktor von 17 für die D.2-Parameterschätzung.

Ein nicht ganz vergleichbarer, aber naheliegender Wert (für das synthetische Modell) ist der Beschleunigungsfaktor für die (12×4)-Konfiguration auf 6 Knoten. Er ist nicht in Abbildung 62 dargestellt, Messungen ergaben aber $18,7$ für das dort verwendete synthetische Testmodell. Der kleine Unterschied zwischen 17 (realen) und $18,7$ (synthetischen) Faktor ist fast vernachlässigbar und unterstützt so die hohen Erwartungen an die Skalierbarkeit für die geschachtelte Parallelisierung bei der allgemeinen Ableitungsberechnung.

Modell S3D_STOCH_COARSE_0009 (kurz D.3)

Im Gegensatz zu den ersten beiden Beispielen müssen für dieses D.3-Modell keine Ableitungen berechnet werden. Statt dessen wurden 10000 Modell-Simulationen (Versuche) berechnet, um sie stochastisch auswerten zu können. Die Modell-Diskretisierung ist dabei ähnlich fein gewählt, wie für D.2. Ausgehend von einer Simulation mit nur 3000 Zeitschritten (pro Versuch) benötigte man für die serielle Berechnung (1×1-Konfiguration) nur 51 min für eine einzelne Modell-Simulation. Hierbei ist immer auch die stochastisch relevante Initialisierungsstörung enthalten. Sie wird für jeden Versuch mit Hilfe eines externen Aufrufs der SGSIM-Bibliothek [120] durchgeführt und ist von dem Berechnungsaufwand her gegenüber der eigentlichen Modell-Simulation vernachlässigbar.

Rechnet man die Laufzeit dieser Einzelsimulation auf die 10000 Versuche hoch, ergibt das insgesamt $10000 \cdot 51\ min = 510000\ min = 8500\ h \approx 354\ Tage \approx 1\ Jahr$. Sicherlich kann man die stochastisch numerischen Abhängigkeiten für die Initialisierungsstörung auch anders gewährleisten als innerhalb eines Programmaufrufs. Beispielsweise könnte man die Gesamtmenge aller Berechnungen generell auf unterschiedliche Programmstarts aufteilen und wäre nicht gezwungen, ein Jahr ununterbrochen durchrechnen zu müssen. Durch einen einzelnen Programmstart ist man aber in der Lage, Dateizugriffe und im gewissen Umfang auch einen erhöhten Speicherverbrauch oder anderen Rechenoverhead effektiver vermeiden zu können.

Das D.3-Beispiel wurde auf dem ScaleMP-System mit einer (88×1)-Konfiguration auf 11 Rechenknoten in 12260 $min \approx 204\ h \approx 8,5\ Tage$ berechnet. Das entspricht einem geschätzten Beschleunigungsfaktor von $510000\ min/12260\ min \approx 41,6$. Da als Vergleichswert kein Beschleunigungsfaktor auf 11 Knoten für das synthetische Testmodell (vergleiche Abbildung 59) vorliegt, soll hier eine einfache Schätzung ausgehend von dem maximalen Wert $18,5$ genügen. Wie in Kapitel 5.3.3 schon ausgeführt wurde, entspricht dieser maximal erreichte Beschleunigungsfaktor in etwa halb so vielen Rechenknoten wie für D.3, was für die Verwendung von doppelt so vielen Rechenknoten einen Beschleunigungsfaktor von $2 \cdot 18,5 = 37,0$ vermuten lässt.

Man liegt hier mit $41,6$ (für D.3) deutlich darüber und dies lässt sich auch nicht dadurch ausgleichen, dass man die grob angenommene Verdoppelung der Rechenknoten genauer einbezieht. Vielmehr sind die Ursachen in den unterschiedlich langen Berechnungszeiten der Versuche und dem schon oft erwähnten positiven Lastausgleich zu suchen. Der

Vorteil der Verkürzung der theoretischen Berechnungszeit von ca. einem Jahr auf nur etwas mehr als eine (reale) Woche ist aus Sicht des wissenschaftlichen Arbeitens enorm.

Auch wenn diese Parallelisierung trivial erscheint, hat sie doch ein paar entscheidende Vorteile gegenüber einer mehr oder weniger externen Lösung mit mehreren Programmstarts. Die Erfahrungen zeigten, dass nicht nur der Programmcode ohne diese zusätzliche Unterstützung (externe Lösung) etwas einfacher ist, sondern auch der mehrfache Programmstart mit ständig wiederkehrenden Speicherreservierungen einher geht, die sehr nachteilig auf der ScaleMP-Plattform sind. Insgesamt ist somit eine Einmalstart-Lösung gerade auf den vSMP-Systemen sehr effizient und schont wichtige Ressourcen. Konkret verbraucht in diesem Beispiel die serielle Berechnung für eine einzelne Modell-Simulation ca. $14,5\ MByte$ und die Einmalstart-Lösung mit 88 Unterprozessen ca. $923\ MByte$. Dagegen würden 88 einzelne parallel laufende Simulationen mit $88 \cdot 14,5\ MByte = 1276\ MByte$ rund 30% mehr verbrauchen als die Einmalstart-Lösung, was bei deutlich größeren Modellen ausschlaggebend sein könnte.

6.3 Ausblick auf Werkzeugunterstützung, höhere Ableitungen und Verallgemeinerung

Im Folgenden sollen drei Aspekte diskutiert werden, wie sich die in dieser Arbeit vorgeschlagenen und erarbeiteten Konzepte leichter auf andere Projekte übertragen lassen.

6.3.1 Werkzeugunterstützung

Als erstes könnte zukünftig eine *erweiterte* Unterstützung für Parallelisierung in den AD-Werkzeugen in Betracht gezogen werden. Generell gibt es schon einfache Unterstützungen von Parallelisierung in einigen der AD-Werkzeuge (z.B. OpenMP bei TAF), diese sind aber oft stark limitiert und konzentrieren sich normalerweise auf nur eine rudimentäre Parallelisierungsebene.

Wie in dieser Arbeit gezeigt wurde, sind jedoch keine ausgefeilten Analysetechniken seitens der Werkzeuge vonnöten, um mehrstufige Parallelisierungsebenen zu unterstützen. Hiermit ist gemeint, dass nicht vom AD-Werkzeug verlangt wird eine Art *Autoparallelisierung* für die Gradientenparallelisierung einzuführen, sondern statt dessen eine vorgegebene Parallelisierung zu erweitern. Gerade die Fähigkeit zur Nutzung zusätzlicher Parallelisierungsebenen macht diesen Ansatz dann besonders interessant für große Anwendungen.

Zusammenfassend benötigt man dafür in den Werkzeugen vier wichtige Fähigkeiten, die teilweise in Grundzügen schon vorhanden sind. Die ersten beiden beziehen sich auf die Einhaltung des Programmkontextes für die originalen OpenMP-Direktiven und die Einhaltung des Deklarationskontextes für die (neuen) Datenobjekte (siehe Kapitel 4.4). Hier verhalten sich die AD-Werkzeuge bereits entweder genau richtig oder machen etwas zu viel, was oft einfach abschaltbar ist.

Bei der dritten Fähigkeit handelt es sich im Wesentlichen um eine echte Erweiterung zum automatischen Umgang mit den in den Direktiven und Klauseln versteckten Rechenoperationen (siehe *reduction*-Klausel Kapitel4.5). Hierfür lassen sich aber bei genauerer Betrachtung oft einfache Regeln erstellen. Notfalls kann man auch, wie in dieser Arbeit vorgestellt, durch einen (automatisierten) Vorverarbeitungsschritt diese Rechenoperationen explizit herausarbeiten. Das ist dann wohl auch ein gangbarer Weg, wenn es bei der AD-Umgebung mittels „Ersetzen der Rechenoperatoren“ zu Problemen kommt.

Die vierte und letzte Anpassung in einem Quellcode transformierenden AD-Werkzeug kann nötig sein, weil man im Allgemeinen nicht von einem Datensichtbarkeitsbereichs-Attribute freien Programmcode ausgehen kann, wie er im Kapitel 4.2 beschrieben und in dieser Konzeptlösung vorausgesetzt wird. Die Vorteile bezüglich der Erweiterungsproduktivität sind klar, müssen aber nicht anderen Projekten aufgezwungen werden. So gesehen muss diese letzte Anpassung nur so viel beinhalten, dass sie die Datensichtbarkeitsbereichs-Attribute korrekt erweitert, bzw. die möglichen Folgen auf neue Datenobjekte berücksich-

tigt. Insbesondere muss hier nicht die in dieser Arbeit vorgeschlagene vollständige Eliminierung dieser Attribute durchgeführt werden, da eine solche Umstrukturierung möglicherweise auch sehr aufwändig zu implementieren wäre. Ein anderer Weg wäre auch denkbar, indem man die sogenannten „autoscoping“-Fähigkeiten [87] für das Werkzeug durch eine Art Zusammenarbeit mit Compilern mit entsprechender Fähigkeit herstellt.

6.3.2 Unterstützung höherer Ableitungen

Für viele Optimierungsalgorithmen und Anwendungen empfiehlt es sich, höhere Ableitungen (z.B. zweite Ableitung - Hesse-Matrix) einzubeziehen. Dazu gibt es in mehreren AD-Werkzeugen eine entsprechende Unterstützung. Üblicherweise kann das auf drei verschiedenen Wegen erreicht werden.

1. Kombination aus AD im Vorwärts-Modus und Rückwärts-Modus, siehe [121].

2. Eine direkte AD-Transformation für die zweite Ableitung bzw. die Verwendung von effizienten Hesse-Modulen (siehe [122]).

3. Die mehrmalige AD-Anwendung im Vorwärts-Modus.

Da die in dieser Arbeit vorgeschlagenen Techniken sich ausschließlich auf AD im Vorwärts-Modus beziehen, können hier keine Aussagen über die Nutzbarkeit im Zusammenhang mit dem Rückwärts-Modus getroffen werden. Für die Übertragbarkeit der in dieser Arbeit vorgestellten Techniken auf die obige 1. Möglichkeit fehlen deshalb weitergehende Untersuchungen.

Darüber hinaus kann man in zukünftigen Arbeiten untersuchen, in wieweit bei AD-Werkzeugen, die direkte zweite Ableitungen mit Hilfe von Hesse-Modulen (2. Möglichkeit) generieren können, Probleme auftreten können. Da hier Ähnlichkeiten mit der ersten Ableitung im Vorwärts-Modus bestehen, kann man für die Übertragbarkeit möglicherweise mit weniger Problemen rechnen als für den Rückwärts-Modus.

Für spezielle Anwendungsfälle kann man Ableitungscode höherer Ordnung erzeugen, in dem man eine mehrmalige Anwendung des AD-Werkzeugs im Sinne der 3. Möglichkeit durchführt. Da hier üblicherweise die zweite Ableitung analog zur Ersten durchgeführt wird, sollte die hier besprochene Unterstützung der Parallelisierung mit kleinen Anpassungen ebenfalls durchführbar sein. Weitergehende Untersuchungen könnten dann klären, ob eine mehrstufige Parallelisierung für jede zusätzliche Ableitungsstufe sinnvoll ist oder nur für die Letzte.

6.3.3 Verallgemeinerung auf C und C++

Ein weiterer wichtiger Aspekt für die Übertragbarkeit auf andere Projekte ist die Machbarkeit im Zusammenhang mit anderen Programmiersprachen wie C oder C++. Als erstes soll es deshalb um AD-Werkzeuge mit Unterstützung für den hier vorgeschlagenen Parallelisierungsansatz gehen.

Handelt es sich um ein Werkzeug bei dem die Rechenoperatoren „ersetzt“ werden (engl. „operator overloading“), sind im Wesentlichen nur analoge Probleme wie bei Fortran zu erwarten.

Bei Werkzeugen mit Programmtransformation gilt es zu beachten, dass es für C und C++ eine von Fortran abweichende Datensichtbarkeitsbereich-Voreinstellung für OpenMP gibt. An dieser Stelle wäre auch denkbar, dass sich diese Voreinstellungen in einem neueren OpenMP-Standard (Version > 2.5) auch für Fortran ändert. Dementsprechend müssen nicht nur Anpassungen für C und C++ entwickelt werden, sondern gegebenenfalls auch für Fortran.

Darüber hinaus müssten auch Erfahrungen erarbeitet werden, inwieweit Codeoptimierungen durch die Programmtransformation auf noch unbekannte Art und Weise die

Richtigkeit der OpenMP-Direktiven und Klauseln für C und C++ (Programmcode) beeinflussen. Diese und andere weiterführende Untersuchungen sind ein interessantes Ziel für zukünftige Projekte. Da der Fokus in dieser Arbeit auf Fortran liegt, wird hier kein Anspruch auf eine allgemeine Aussagekräftigkeit bezüglich anderer Programmiersprachen erhoben.

Abschließend kann man noch für die letztendlich bevorzugten Parallelisierungsstrategien sagen, dass sie wahrscheinlich auch mit denselben Vorteilen für C oder C++ anwendbar sind. Allgemein, auch in Fortran stehen dem oft sogenannte „projektabhängige Programmierstile“ oder Richtlinien entgegen. Diese können beispielsweise die Verwendung von globalen Datenobjekten untersagen oder nur mit Ausnahmen zulassen. Ein damit verbundener Zwang zu möglichst lokalen Datenobjekten kann deshalb allgemein auf den vSMP-Plattformen zu Nachteilen in der Rechenleistung wegen erhöhter Speicherreservierung führen. In solchen Fällen empfiehlt es sich, die Programmierrichtlinien für einen „guten“ Programmierstil mit effizienten Datenstrukturen zu revidieren.

7 Abschließende Bemerkungen

Ausgehend von einem bereits etablierten Simulationswerkzeug (SHEMAT) sollte eine Neuentwicklung zur Erreichung neuer wissenschaftlicher Zielsetzungen durchgeführt werden. Das ursprüngliche SHEMAT wurde zur ausschließlichen Vorwärtsberechnung einer hydrogeothermalen Simulation entwickelt. Dagegen ist die Neuentwicklung (Simulations-Suite) jetzt ein wertvolles Werkzeug zur Beantwortung inverser Problemstellungen (Parameterschätzung) und ein hochwertiges Hilfsmittel zur Durchführung von stochastischen Unsicherheitsanalysen. Die im Rahmen dieser Forschungsarbeit mitentwickelte Simulations-Suite profitiert daher maßgeblich von den folgenden neuen Eigenschaften (gegenüber der ursprünglichen SHEMAT-Software).

- Verfügbarkeit der stochastischen Simulation für die Unsicherheitsanalyse.
- Verfügbarkeit stochastischer und deterministischer Methoden für die Parameterschätzung.
- Verwendung exakter Ableitungen für die deterministischen Methoden zur Parameterschätzung.
- Eine mehrstufige Parallelisierung mit einer vorteilhaften Verteilung der Rechenarbeit auf mehrere Prozessoren.
- Effiziente Nutzung der Rechnerkapazitäten moderner Rechner-Architekturen bezüglich der Anzahl der Prozessoren und des Hauptspeichers.

Dazu wurde ein Gesamtlösungs-Konzept bestehend aus mehreren Teilbereichen vorgestellt. Dieses wurde auf dem umfangreichen Programmcode innerhalb eines Software-Projektes (Simulations-Suite) vollständig umgesetzt und die praxisnahe Tauglichkeit in einen mehrjährigen Entwicklungszeitraum unter Beweis gestellt. Abschließend wurde für wissenschaftlich relevante Anwendungsszenarien basierend auf geophysikalischen Modell-Simulationen die Fähigkeiten für das Hochleistungsrechnen erfolgreich überprüft.

Der größte Teilbereich des Gesamtlösungs-Konzeptes besteht aus einer Reihe von Richtlinien und Maßnahmen, um eine „erweiterte“ Parallelisierung auf dem sogenannten Ableitungscode zu erhalten. Aus Sicht des AD-Werkzeugs wird dazu die originale Parallelisierung des ursprünglichen Programmcodes ausgeblendet. Im Detail wurde gezeigt, wie ein parallelisierungs-spezifischer Datensichtbarkeitsbereich-Automatismus (engl. „data scoping“) ausgenutzt werden kann, um einen Teil der Parallelisierungsanweisungen überflüssig zu machen. Dies und weitere Richtlinien wurden ausgeführt, um sicher zu stellen, dass die ursprüngliche Parallelisierung gültig bleibt. Danach wurde veranschaulicht, wie man durch Umgehung bestimmter problematischer Parallelisierungsanweisungen die Parallelisierung auf die zusätzlichen Ableitungsanweisungen ausdehnt. Abschließend wurde dargelegt, wie durch Einhaltung einer bestimmten Aufrufstruktur und Funktionskapselungen diese Softwaretechnik unterstützt wird.

Bei der Übertragung auf das zugrunde liegende Software-Projekt konnte erreicht werden, dass die verwendeten AD-Werkzeuge (Adifor2, Tapenade, TAF) die originale (ursprüngliche) Parallelisierung nicht speziell unterstützen müssen. Diese originale Parallelisierung wurde darüber hinaus erst nach und nach bis zu ihrer jetzigen Vollständigkeit (entsprechend den definierten Richtlinien) erweitert. Diese reibungslose Vervollständigung über einen längeren Entwicklungszeitraum belegt damit die weitestgehende Unabhängigkeit gegenüber den anderen definierten physikalischen und numerisch funktionellen Erweiterungen (siehe Kapitel 2.3) und somit eine leistungsfähige Erweiterungsproduktivität.

Darüber hinaus konnte der Prozess zur Ableitungscodegenerierung effizient umgesetzt werden, da er überwiegend automatisiert abläuft. Diese Automatisierung konnte erst durch die hier eingesetzte und vorgestellte „OpenMP-hiding“-Technik ermöglicht werden, da die

AD-Werkzeuge eine OpenMP-Parallelisierung gar nicht oder für die hier geforderten Zwecke unzureichend unterstützten. Von den einmaligen Vorbereitungsmaßnahmen zur Automatisierung abgesehen, entfiel später weitestgehend eine manuelle Vor- und Nachbearbeitung, die üblicherweise nach jeder Änderung am Programmcode der Vorwärtssimulation anfallen würde.

Gleichzeitig zielt die hier eingesetzte Technik auch darauf ab, den Parallelisierungsaufwand in Form von weniger Direktiven und Klauseln übersichtlicher zu gestalten und zu verringern. Das verbesserte wesentlich die Wartbarkeit und damit die Erweiterungsproduktivität.

Da eine umfassende Konzeptlösung Gegenstand der wissenschaftlichen Zielausrichtung war, wurde kontinuierlich zur „OpenMP-hiding“-Technik auch der zweite große Teilbereich dieser Konzeptstudie über eine möglichst leistungsfähige Parallelisierungsstrategie erforscht. Diese Studien bauen auf bestehende Forschungsarbeiten aus dem vergleichbaren Umfeld der Gradientenparallelisierung mit MPI und ersten Ansätzen mit OpenMP auf. Weiterführend wurde im Rahmen dieser Arbeit dann systematisch untersucht, wie sich die ersten Ansätze mit OpenMP soweit verfeinern lassen, dass sie allgemeiner einsetzbar und in größeren wissenschaftlich-relevanten Projekten von Nutzen sind.

Gezeigt wurde dazu, wie die Ableitungsberechnung von einer erweiterten und mehrstufigen Parallelisierung profitieren kann. Hierfür wurden Richtlinien für die Datenstrukturen und Maßnahmen entwickelt, um zu gewährleisten, dass die innere Parallelisierungsebene unabhängig von der äußeren ist. Im Detail gelingt dies fast ohne Nebeneffekte, was in Hinblick auf eine Verallgemeinerbarkeit wichtig ist.

Ein wichtiges Ergebnis der systematischen Verbesserungen und deren Vergleiche ist die Empfehlung einer finalen Parallelisierungsstrategie für die Ableitungsberechnung. Hierbei zeigte sich, dass die auf den skalaren Gradienten-Modus beruhenden Möglichkeiten (*Skalare*-Variante 2 und 3) das größte Potential bei mehrfach limitierten Anwendungsszenarien haben.

Unabhängig davon war es für die Varianten im skalaren Gradienten-Modus auch leichter, die Besonderheiten moderner und insbesondere höhergradiger ccNUMA-Architekturen zu berücksichtigen. Das führte zu der Erkenntnis, immer wiederkehrende Speicherreservierungen in ihrer Anzahl stark beschränken zu müssen. Insbesondere führte das zu Umwandlungen von Datenobjekten mit lokaler Speicherreservierung (*private*-Datenobjekte) zu höher-dimensionalen und damit globalen Datenobjekten. Die Anwendbarkeit und Leistungsfähigkeit der in dieser Arbeit definierten und eingesetzten Techniken und ihrer Richtlinien wurde durch die erfolgreiche Übertragung in das dieser Arbeit zugrunde liegende Software-Projekt belegt [100, 115].

Ein erstes positives Ergebnis der mehrstufigen Parallelisierung zeigte sich dabei an einem synthetischen Modell-Beispiel. Dieser Vergleichstest bestätigte die generelle Fähigkeit, mit einer limitierten Modellgröße (grobe Diskretisierung) und einer beschränkten Anzahl von benötigten Richtungsableitungen effizient umgehen zu können. Da die Parallelisierung über der Menge der Richtungsableitungen bestimmte günstige Eigenschaften aufweist, konnte sie analog auch als äußere Parallelisierung über der Menge der stochastischen Versuche ausgenutzt werden. Für beide Anwendungen erbrachten spätere Zeitmessungen auf vSMP-Systemen ein nahezu lineares Skalierungsverhalten mit einer steigenden Anzahl von verwendeten Rechenknoten. Das entsprach den erwünschten (und vorausgesagten) Erwartungen und konnte dementsprechend auch auf mehrere reale Anwendungsszenarien transportiert werden.

Diese liefen im laufenden Regelbetrieb unter teilweise realen Bedingungen. Im Einzelnen bestätigten die Performance-Modelle für serielle Vergleichslaufzeiten, dass für die hier gemachten Parameterschätzungen die prognostizierte Rechenzeit von weit über einem Monat auf ca. eine halbe Woche (im parallelen Betrieb) reduziert werden konnte. Für eine stochastische Versuchsreihe konnte die prognostizierte Rechenzeit von fast einem

Jahr sogar auf etwas mehr als nur eine reale Woche reduziert werden. Insbesondere konnte wegen diesen drastischen Verkürzungen der Gesamtrechenzeiten deutlich schneller auf Modell-Fehler reagiert werden und andere Anpassungen und Modellvarianten ausprobiert werden. Die darauf gründenden geowissenschaftlichen Forschungsarbeiten wurden damit enorm unterstützt und die hier vorgestellte konzeptuelle Gesamtlösung wurde hinsichtlich ihrer hochleistungsfähigen Eigenschaften nochmals bestätigt.

Ein weiterer wichtiger Teilbereich dieser Konzeptstudie ist die Eignung für moderne und zukunftsweisende Rechner-Architekturen. Diese Fähigkeit konnte ebenfalls erfolgreich vorbereitet werden. Ausgangsbasis sind hier neuere Systeme mit einem gemeinsamen Speicher (engl. „shared memory“), die sich in ihrer modernen Form als flache ccNUMA-Systeme mit einer hohen Anzahl von Prozessorkernen auszeichnen. Die derzeitigen Tendenzen lassen darauf schließen, dass solche Systeme bald noch viel stärker zunehmen werden. Auf einem erst vor kurzem verfügbar gewordenen Vertreter, einem vier-Sockel System mit Nehalem-EX Prozessoren (insgesamt 32 leistungsfähige Rechenkerne), konnte die Eignung für das Hochleistungsrechnen mit dem hier vorgestellten Gesamtlösungs-Konzept aufgezeigt werden.

Solche Systeme können auch als Möglichkeit zur Vorhersage dienen, ob das Gesamtlösungs-Konzept auch für die zukünftigen vSMP-Generationen geeignet ist. Denn die vier-Sockel Systeme und größere könnten als Basisknoten in den kommenden vSMP-Systemen eingesetzt werden. Da sowohl für die jetzigen vSMP-Systeme als auch für die zu erwartenden neuen Basisrechenknoten (bzw. einem Stellvertreter) die positiven Skalierungseigenschaften erfolgreich aufgezeigt werden konnten, kann man auf eine generelle Eignung für die in naher Zukunft kommenden Hochleistungssysteme (dieser Art) schließen.

Hiermit wird das Teilziel, den besonderen Anforderungen von neuen Rechner-Architekturen gerecht zu werden, als erreicht betrachtet. Damit ist gezeigt, dass die Vorteile von preiswerten und potentiell sehr leistungsfähigen Rechnersystemen verbunden mit geringem Programmieraufwand gleichzeitig erfüllbar sind.

Zusammenfassend sind die einzelnen bis hierhin erfolgreich umgesetzten Teilaspekte und Konzeptlösungen das Ergebnis des Konzeptfindungsprozesses. Dieser stellt, wie in der Einleitung dieser Arbeit angemerkt wurde, hinsichtlich der Zusammenstellung der „allgemeinen Vergleichskriterien“ eine Neuheit dar. Die nachfolgende Übersicht zeigt alle letztendlich entscheidenden Vergleichskriterien.

1. Hohe Erweiterungsproduktivität durch Automatisierung und unabhängige Erweiterbarkeit

2. Nutzung exakter Ableitungen mit automatisierter Generierung des Ableitungscodes

3. Notwendigkeit zur Unterstützung seitens des AD-Werkzeugs oder eines zusätzlichen Transformations-Werkzeugs

4. Mehrstufige Parallelisierung mit guten Synchronisationseigenschaften und angemessenem Berechnungsaufwand (und Speicherverbrauch)

5. Kompatibilität der Parallelisierungen für stochastische Versuche und Richtungsableitungen

6. Hohe Rechenleistung auf modernen Rechner-Architekturen (insbesondere neue SMP- und derzeit verfügbare vSMP-Systeme)

Dabei sind insbesondere die Punkte 3. und 5. neue Ergebnisse dieser mehrjährigen Forschungsarbeit, da sich diese beiden Fragestellungen und ihre Antworten erst während der Projektarbeit ergaben.

Im Rahmen der geowissenschaftlichen Forschungsarbeiten wurde die Simulations-Suite umfangreich und weitestgehend unabhängig für verschiedenste geophysikalische und gekoppelte Zustandsvariablen im porösen Medium erweitert. Die für die Parameterschätzung spezifizierbaren physikalischen Größen beziehen sich mittlerweile auf alle 13 im Vorwärtscode definierten Gesteinsparameter (für alle Gesteinsschichten). Darüber hinaus werden auch zeitabhängige Randwerte unterstützt und damit erste Analysen und numerische Versuche durchgeführt. Eine erste Implementation zur Lösung inverser Probleme mit exakten Ableitungen wurde in [69] ausgeführt. Weitere Veröffentlichungen zu darauf aufbauenden Anwendungen sind derzeit noch in Arbeit. Frühe Implementationen der stochastische Methoden wurden für eine Reihe von stochastischen Unsicherheitsanalysen in [123, 124, 125] belegt.

Die hier zugrunde liegende Entwicklungsarbeit kennzeichnet sich durch eine in Großprojekten übliche Praxisumgebung mit einer permanent voranschreitenden Weiterentwicklung aus. Im Laufe des bisher zurück gelegten Entwicklungszeitraums für die Simulations-Suite konnten wegen der verwendeten Konzeptlösungen weit über 30 physikalisch und numerisch funktionelle Erweiterungen implementiert werden. Alle diese Erweiterungen erfolgten in relevanten und schon zu Beginn der Projektphase definierten Bereichen. Eine kurze Gesamtübersicht aller bisher erfolgten Erweiterungen findet man in Tabelle 4 im Anhang E.

Im Zusammenhang mit einer generellen Verallgemeinerbarkeit bzw. Übertragbarkeit auf andere Projekte wurde im Kapitel 6.3 ein Ausblick gegeben, inwieweit dazu noch weiterführende Untersuchungen notwendig sind. Davon unabhängig lässt sich sagen, dass das vorgestellte Gesamtlösungs-Konzept sich zumindest auf fast alle Fortran77/95 und OpenMP (bis Version 2.5) basierende Projekte übertragen lassen können sollte.

Damit sollte diese Arbeit als Unterstützung im wissenschaftlichen wie auch im industriellen Softwareentwicklungsprozess nutzbar sein. Die jeweiligen Entwickler und Forscher können sich damit besser und unabhängiger auf ihre speziellen, sehr facettenreichen Forschungsaspekte bei der Softwareentwicklung konzentrieren.

Literatur

[1] K. Mosegaard and M. Sambridge. Monte Carlo analysis of inverse problems. *Inverse Problems*, 18:29–54, 2002.

[2] A. Tarantola. *Inverse Problem Theory: and Methods for Model Parameter Estimation.* SIAM, Philadelphia, PA, 2005.

[3] G. Evensen. The Ensemble Kalman Filter: theoretical formulation and practical implementation. *Ocean Dynamics*, 53:343–367, 2003.

[4] P. C. Hansen. *Discrete Inverse Problems: Insight and Algorithms.* SIAM, Philadelphia, PA, 2010.

[5] H. Engl, M. Hanke, and A. Neubauer. Regularization of inverse problems, 1996.

[6] C. R. Vogel. *Computational Methods for Inverse Problems.* SIAM, Philadelphia, PA, 2002.

[7] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. D. Hovland. ADIFOR: Generating Derivative Codes from Fortran Programs. *Scientific Programming*, 1(1):11–29, 1992.

[8] R. Giering, T. Kaminski, and T. Slawig. Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.

[9] L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, September 2004. URL: `http://www.inria.fr/rrrt/rt-0300.html`.

[10] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[11] H. A. van der Vorst. BI-CGSTAB : A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[12] P. S. Huyakorn and G. F. Pinder. *Computational Methods in Subsurface Flow.* Academic Press, 1983.

[13] B. Chapman, G. Jost, R. van der Pas, and D. J. Kuck. Using OpenMP: Portable Shared Memory Parallel Programming, 2007.

[14] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP.* Morgan Kaufman Publishers, San Mateo, CA, 2000.

[15] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI: The Complete Reference (Vol.1). *The MPI Core*, 1, 1998.

[17] High Performance Computing Virtualization — Virtual SMP — ScaleMP, 2010. Available online at `http://www.scalemp.com/`; visited on August 1st 2010.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, München, 1995.

[19] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, München, 1994.

[20] C. Clauser, editor. *Numerical Simulation of Reactive Flow in Hot Aquifers. SHEMAT and Processing SHEMAT.* Springer, New York, 2003.

[21] L. C. Evans. *Partial Differential Equations. In: Graduate Studies in Mathematics*, Vol. 19. American Mathematical Society, Providence, Rhode Island, 1999.

[22] M. Renardy and R. C. Rogers. *TEXTS IN APPLIED MATHEMATICS: An Introduction to Partial Differential Equations*, Vol. 13, *AMS.* Springer-Verlag 2004, Berlin/Heidelberg/New York, 1998.

[23] D. Gilbarg and N. S. Trudinger. *Elliptic Partial Differential Equations of Second Order. In: Grundlehren der mathematischen Wissenschaften*, Vol. 224. Springer-Verlag, Berlin/Heidelberg/New York, 1977.

[24] G. de Marsily. *Quantitative Hydrogeology.* Academic Press, 1986.

[25] Michele Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418 – 477, 2002.

[26] O. Axelsson. A survey of preconditioned iterative methods for linear systems of algebraic equations. *BIT*, 25(1):166–187, 1985.

[27] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, third edition, 1999.

[28] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193:357–397, 2004.

[29] C. T. Kelley. *Solving nonlinear equations with Newtons method.* Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2003.

[30] M. Pernice and H. F. Walker. NITSOL: A Newton Iterative Solver For Nonlinear Systems. *SIAM Journal on Scientific and Statistical Computing*, 19(1):302–318, 1998.

[31] P. C. Hansen. *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion.* SIAM, Philadelphia, PA, 1998.

[32] M. Bertero, C. De Mol, and E. R. Pike. Linear inverse problems with discrete data: II. Stability and regularisation. *Inverse Problems*, 4, 1988.

[33] A. Kirsch. An introduction to the mathematical theory of inverse problems. *Applied Mathematical Science*, 120, 1996.

[34] J. Baumeister. Stable Solution of Inverse Problems, 1986.

[35] V. B. Glasko. Inverse problems of mathematical physics. *American Institute of Physics, New York*, 1984.

[36] G. Anger. *Inverse Problems in Differential Equations.* Akademie-Verlag, Berlin, 1990.

[37] H. W. Engl. Regularization methods for the stable solution of inverse problems. *Surveys on Mathematics for Industry*, 3:71–143, 1993.

[38] C. W. Groetsch. The Theory of Tikhonov Regularization for Fredholm Equations of the First Kind. In *Research Notes in Mathematics Series*. Pitman, Boston, MA, Vol. 105, pp. 104, 1984.

[39] C. D. Rodgers. Retrieval of athmospheric temperature and composition from remote measurements of thermal radiation. *Reviews of Geophysics and Space Physics*, 14:609–624, 1976.

[40] A. Tarantola and B. Valette. Inverse problems = Quest for information. *Journal of Geophysics*, 50:159–170, 1982.

[41] A. Tarantola and B. Valette. Generalized nonlinear inverse problems solved using the least-squares criterion. *Reviews of Geophysics and Space Physics*, 20:219–232, 1982.

[42] Å. Björk. *Numerical Methods For Least Squares Problems*. SIAM, Philadelphia, PA, 1996.

[43] C. D. Rodgers. *Inverse Methods for atmospheric sounding*. World Scientific, Singapore, 2000.

[44] J. C. Gilbert and J. Nocedal. Global convergence properties of conjugate gradients methods. *SIAM Journal on Scientific and Statistical Computing*, 2:21–42, 1992.

[45] J. Nocedal. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 35:773–782, 1980.

[46] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[47] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. No. 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[48] M. B. Giles. On the Iterative Solution of Adjoint Equations. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[49] H. M. Bücker, A. Rasch, E. Slusanschi, and C. H. Bischof. Delayed propagation of derivatives in a two-dimensional aircraft design optimization problem. In D. Sénéchal, editor, *17th Annual International Symposium on High Performance Computing Systems and Applications and OSCAR Symposium*, Sherbrooke, Québec, Canada, May 11–14 2003. NRC Research Press, Ottawa, pp. 123–126, 2003.

[50] J. C. O. Mello and M. V. F. Pereira. Evaluation of reliability worth in composite systems based on pseudo-sequential Monte Carlo simulation. *IEEE Transactions on Power Systems*, 9(3):1318—1326, 1994.

[51] J. R. Ubeda and R. N. Allan. Reliability assessment of composite hydrothermal generation and transmission systems using sequential simulation. *IEE Proceedings - Generation, Transmission and Distribution*, 141(4):257–262, 1994.

[52] Y. Chen and D. Zhang. Data assimilation for transient flow in geological formations via ensemble Kalman filter. *Advances in Water Resources*, 29:1107–1122, 2006.

[53] G. Evensen. Sequential data assimilation with a nonlinear quasigeostrophic model using Monte Carlo methods to forcast error statistics. *Journal of Geophysical Research*, 99:10143–10162, 1994.

[54] M. V. Krymskaya, R. G. Hanea, and M. Verlaan. An iterative ensemble Kalman Filter for reservoir engineering applications. *Computational Geosciences*, 2008.

[55] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, Vol. 120, *Lecture Notes in Computer Science*. Springer, Berlin, 1981.

[56] RWTH Aachen University, Institute for Scientific Computing and the Center for Computing and Communication. *Community Portal for Automatic Differentiation*, 7 2010. Available online at `http://www.autodiff.org/`; visited on July 1st 2010.

[57] J. J. Moré. Automatic Differentiation Tools in Optimization Software. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[58] M. Berz, C. H. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA, 1996.

[59] H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, Vol. 50, *Lecture Notes in Computational Science and Engineering*. Springer, New York, 2005.

[60] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[61] A. Griewank and G. Corliss. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, PA, 1991.

[62] C. G. Page. *The professional programmers guide to Fortran 77*. Pitman, London, 1988.

[63] M. Metcalf and J. K. Reid. *Fortran 90/95 Explained (second edition)*. Oxford University Press, Oxford, 1999.

[64] TOUGH2, 2009. Available online at `http://esd.lbl.gov/TOUGH2/`; visited on June 1st 2010.

[65] K. Pruess, C. Oldenburg, and G. Moridis. TOUGH2 User's Guide, Version 2.0. Technical Report LBNL-43134, Lawrence Berkeley National Laboratory, Berkley, CA, 1999.

[66] K. Zhang, Y.-S. Wu, and K. Pruess. User's Guide for TOUGH2-MP — A Massively Parallel Version of the TOUGH2 Code. Technical Report LBNL-315E, Lawrence Berkeley National Laboratory, Berkley, CA, 2008.

[67] S. Finsterle. iTOUGH2 User's Guide. Technical Report LBNL-40040, Lawrence Berkeley National Laboratory, Berkley, CA, 2007.

[68] S. Finsterle. iTOUGH2 Command Reference. Technical Report LBNL-40041, Lawrence Berkeley National Laboratory, Berkley, CA, 2007.

[69] V. Rath, A. Wolf, and H. M. Bücker. Joint three-dimensional inversion of coupled groundwater flow and heat transfer based on automatic differentiation: Sensitivity calculation, verification, and synthetic examples. *Geophysical Journal International*, 167(1):453–466, 2006.

[70] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994. URL: `http://www.netlib.org/pvm3/book/pvm-book.html`.

[71] S. Finsterle. Parallelization of iTOUGH2 Using PVM. Technical Report LBNL-42261, Lawrence Berkeley National Laboratory, Berkley, CA, 1998.

[72] iTOUGH2 Home Page - Multiphase Inverse Modeling, 2007. Available online at `http://esd.lbl.gov/ITOUGH2/`; visited on August 1st 2010.

[73] F. Thorns. *Das Virtualisierungs-Buch: Konzepte, Techniken und Lösungen.* Computer & Literatur, Böblingen, 2008.

[74] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[75] A. Vasilevsky. Linux virtualization on Virtual Iron VFe. *Linux Symposium, Ottawa, Ontario Canada*, 2:235–250, 2005.

[76] K. Kaneda. Virtual machine monitor for providing a single system image, 2006. Available online at `http://web.yl.is.s.u-tokyo.ac.jp/ ~kaneda/dvm/`; visited on August 1st 2010.

[77] The Versatile SMP (vSMP) architecture and solutions based on vSMP Foundation, 2008. ScaleMP White Paper. Available online at `http://www.cray.com/Assets/PDF/products/cx1/TheVersatileSMPArchitecture.pdf`; visited on August 1st 2010.

[78] D. Schmidl, C. Terboven, D. an Mey, and H. M. Bücker. Binding nested OpenMP programs on hierarchical memory architectures. In M. Sato, T. Hanawa, M. S. Mueller, and B. R. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and more, Proceedings of the International Workshop IWOMP 2010*, Tsukuba, Japan, 2010. Lecture Notes in Computer Science, Springer, Berlin, Vol. 6132, pp. 29–42, 2010.

[79] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 2.5*, 2005. Available online at `http://www.openmp.org/specs/`; visited on July 1st 2010.

[80] S. W. Ambler and R. Jeffries. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* John Wiley & Sons, Ltd., New York, USA, 2002.

[81] K. Beck. *Extreme Programming – das Manifest: Die revolutionäre Methode für Softwareentwicklung in kleinen Teams.* Addison-Wesley, München, 2000.

[82] W.-G. Bleek and H. Wolf. *Agile Softwareentwicklung: Werte, Konzepte und Methoden.* dpunkt.verlag, Heidelberg, 2008.

[83] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices.* Prentice Hall, Englewood Cliffs, NJ, 2002.

[84] MathWorks - MATLAB and Simulink for Technical Computing, 2010. Available online at `http://www.mathworks.com/`; visited on August 1st 2010.

[85] C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[86] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.

[87] Y. Lin, C. Terboven, D. an Mey, and N. Copty. Automatic Scoping of Variables in Parallel Regions of an OpenMP Program. In B. M. Chapman, editor, *Shared Memory Parallel Programming with Open MP: 5th International Workshop on Open MP Applications and Tools (WOMPAT 2004)*, Houston, TX, USA, May 17–18 2005. Lecture Notes in Computer Science, Springer, Berlin, Vol. 3349, pp. 83–97, 2005.

[88] J.-H. Chow, L. E. Lyon, and V. Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In M. A. Bauer, K. Bennet, W. M. Gentleman, J. H. Johnson, K. A. Lyons, and J. Slonim, editors, *1996 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, November 12–14 1996. CASCON, IBM, Toronto, pp. 76–89, 1996.

[89] C. Brunschen and M. Brorsson. OdinMP/CCp–A Portable Implementation of OpenMP for C. In G. C. Fox and L. Moreau, editors, *the First European Workshop on OpenMP (EWOMP)*, Lund, Sweden, September 30 – October 1st 1999. Concurrency and Computation: Practice and Experience, John Wiley & Sons Ltd., New York, USA, Vol. 12, pp. 21–26, 2000.

[90] H. Wehnes. *FORTRAN 77: Strukturierte Programmierung mit FORTRAN 77.* Hanser, München; Wien, 1985.

[91] RRZN. *FORTRAN 95: Nachschlagewerk für ANSI-Fortran / DIN-Fortran / ISO/IEC 1539-1.* Regionales Rechenzentrum für Niedersachsen (RRZN), Universität Hannover, Hannover, 2001.

[92] C. H. Bischof, N. Guertler, A. Kowarz, and A. Walther. Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, Bonn, Germany, August 11–15 2008. Lecture Notes in Computational Science and Engineering, Springer, Berlin, Vol. 64, pp. 163–173, 2008.

[93] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.

[94] U. Naumann. Reducing the Memory Requirement in Reverse Mode Automatic Differentiation by Solving TBR Flow Equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002.* Lecture Notes in Computer Science, Springer, Berlin, Vol. 2330, pp. 1039–1048, 2002.

[95] C. Faure and U. Naumann. The taping problem in automatic differentiation. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Springer, New York, 2002.

[96] A. Griewank, C. H. Bischof, G. F. Corliss, A. Carle, and K. Williamson. Derivative Convergence for Iterative Equation Solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[97] Y. Saad and M. H. Schultz. GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[98] M. B. Giles. Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, Bonn, Germany, August 11–15 2008. Lecture Notes in Computational Science and Engineering, Springer, Berlin, Vol. 64, pp. 35–44, 2008.

[99] M. Tadjouddine, S. A. Forth, and A. J. Keane. Adjoint Differentiation of a Structural Dynamics Solver. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*. Lecture Notes in Computational Science and Engineering, Springer, Berlin, pp. 309–319, 2005.

[100] A. Wolf, V. Rath, and H. M. Bücker. Parallelisation of a Geothermal Simulation Package: A Case Study on Four Multicore Architectures. In C. H. Bischof, H. M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, and F. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*. Advances in Parallel Computing, IOS Press, Amsterdam, The Netherlands, Vol. 15, pp. 451–458, 2008.

[101] G. Radicati di Brozolo and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse non-symmetric systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.

[102] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*. Cambridge University Press, Cambridge, pp. 111–197, 1993.

[103] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT*, 24:623–633, 1984.

[104] G. Meurant. Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2. Technical Report LBL-18023, University of California, Berkley, CA, 1984.

[105] H. A. van der Vorst. The performance of Fortran implementations for preconditioned conjugate gradients on vector computers. *Parallel Computing*, 3:49–58, 1986.

[106] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[107] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–32, 1988.

[108] H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof. Bringing Together Automatic Differentiation and OpenMP. In *15th International Conference on Supercomputing (ICS '01)*, Sorrento, Italy, June 17–21 2001. ACM Press, New York, NY, USA, pp. 246–251, 2001.

[109] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey. Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. In J. N. Almhana and V. C. Bhavsar, editors, *16th Annual International Symposium on High Performance Computing Systems and Applications*, Moncton, NB, Canada, June 16–19 2002. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 121–126, 2002.

[110] H. M. Bücker, B. Lang, D. an Mey, A. Rasch, and C. H. Bischof. Exploiting OpenMP's Memory Management for Parallel Automatic Differentiation. Preprint of the Institute for Scientific Computing RWTH–CS–SC–01–07, RWTH Aachen University, Aachen, 2001.

[111] H. M. Bücker, A. Rasch, and A. Wolf. A Class of OpenMP Applications Involving Nested Parallelism. In *19th ACM Symposium on Applied Computing*, Nicosia, Cyprus, March 14–17 2004. ACM Press, New York, NY, USA, Vol. 1, pp. 220–224, 2004.

[112] C. H. Bischof, L. Green, K. Haigler, and T. Knauff. Parallel Calculation of Sensitivity Derivatives for Aircraft Design Using Automatic Differentiation. In *AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, (AIAA 94-4261)*. American Institute of Aeronautics and Astronautics, USA, pp. 73–84, 1994.

[113] C. H. Bischof, A. Bouaricha, P. Khademi, and J. J. Moré. Computing Gradients in Large-Scale Optimization Using Automatic Differentiation. *INFORMS Journal on Computing*, 9:185–194, 1997.

[114] H. M. Bücker, A. Rasch, V. Rath, and A. Wolf. Semi-automatic Parallelization of Direct and Inverse Problems for Geothermal Simulation. In *24th ACM Symposium on Applied Computing*, Honolulu, Hawaii, USA, March 8–12 2009. ACM Press, New York, Vol. 2, pp. 971–975, 2009.

[115] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. H. Bischof. How to scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. *IEEE International Conference on Cluster Computing*, 0:29–37, 2010.

[116] Laptop, Notebook, Desktop, Server and Embedded Processor Technology - Intel, 2010. Available online at `http://www.intel.com/?de_DE_03`; visited on August 1st 2010.

[117] J. C. Breesee, editor. *Geothermal Energy in Europe: The Soultz hot dry rock project*. Gordon and Breach, Philadelphia, PA, 1992.

[118] D. Pribnow and C. Clauser. Heat and fluid flow at the Soultz hot dry rock system in the Rhein graben. In E. Iglesias, D. Blackwell, T. Hunt, J. Lund, and S. Tamanyu, editors, *World Geothermal Conference 2000*, Kyushu-Tohoku, Japan, 2000. International Geothermal Association, Auckland New Zealand, pp. 3835–3840, 2000.

[119] R. Schellschmidt and C. Clauser. The thermal regime of the Upper Rhine graben and the anomaly of Soultz. *Zeitschrift für Angewandte Geologie*, 42(1):40–44, 1996.

[120] C. V. Deutsch and A. G. Journel. *GSLIB: Geostatistical Software Library and User's Guide*. Oxford University Press, New York, 1998.

[121] M. Martinelli and L. Hascoët. Tangent-on-Tangent vs. Tangent-on-Reverse for Second Differentiation of Constrained Functionals. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, Bonn, Germany, August 11–15 2008. Lecture Notes in Computational Science and Engineering, Springer, Berlin, Vol. 64, pp. 151–161, 2008.

[122] J. Abate, C. H. Bischof, A. Carle, and L. Roh. Algorithms and Design for a Second-Order Automatic Differentiation Module. In *International Symposium on Symbolic and Algebraic Computing (ISSAC)*, Kihei, Maui, Hawaii, United States, July 21–23 1997. ACM, New York, NY, USA, pp. 149–155, 1997.

[123] C. Vogt, D. Mottaghy, A. Wolf, V. Rath, and C. Clauser. Reducing Temperature Uncertainties by Stochastic Geothermal Reservoir Modeling. *Geophysical Journal International*, 181:321–333, 2010.

[124] C. Vogt, D. Mottaghy, V. Rath, A. Wolf, R. Pechnig, and C. Clauser. Quantifying uncertainties in geothermal energy exploration. In *Geophysical Research Abstracts of the European Geoscience Union General Assembly (EGU 2009)*, Vienna, Austria, April 19–24 2009. EGU, Vol. 11, pp. EGU2009–1434, 2009.

[125] C. Vogt, D. Mottaghy, V. Rath, A. Wolf, R. Pechnig, and C. Clauser. Quantifying Uncertainty in Geothermal Reservoir Modeling. In R. Horne, editor, *World Geothermal Congress 2010 (WGC2010)*, Bali, Indonesia, April 25–29 2010. International Geothermal Association, Bochum, Germany, 2010.

[126] S. Pearson. NAG Fortran Library: Description, 2010. Available online at `http://www.nag.com/numeric/FL/FLdescription.asp`; visited on August 1st 2010.

A Kurzbeschreibungen der OpenMP 2.5 Direktiven und Klauseln

Direktive	Kurzbeschreibung
parallel	Bereich zwischen *parallel* und *end parallel* ist eine parallele Region. Eine Gruppe von Unterprozessen berechnet parallel alle hierin definierten Berechnungsblöcke (gesteuert durch Arbeitsverteilungs-Direktiven).
do	Aktiviert die Arbeitsverteilung für die nachfolgende Schleife. Der Schleifenkörper definiert die unabhängigen Berechnungsblöcke (die verschiedenen Schleifen-Iterationen).
sections	Aktiviert die Arbeitsverteilung für die mit *section* definierten Berechnungsblöcke.
section	Definiert den Beginn der einzelnen Berechnungsblöcke innerhalb des *sections*-Bereichs.
single	Nachfolgender Berechnungsblock wird von nur einem Unterprozess ausgeführt.
master	Nachfolgender Berechnungsblock wird nur von dem Haupt-Unterprozess der Gruppe ausgeführt.
workshare	Aktiviert die Arbeitsverteilung auf den Vektor-Rechenoperationen des nachfolgenden Bereichs.
barrier	Alle Unterprozesse müssen an dieser Stelle auf einander warten. Es muss eine einheitliche Daten-Sicht hergestellt werden (durch gegenseitigen Daten-Austausch bzw. Konsistenz-Abgleich über den Hauptspeicher). Danach erst dürfen die Unterprozesse weiter rechnen.
critical	Nachfolgender Bereich darf zeitgleich von nur einem Unterprozess ausgeführt bzw. betreten werden.
atomic	Nachfolgende Rechenoperation (kein Bereich) darf zeitgleich von nur einem Unterprozess ausgeführt werden.
flush	Für die spezifizierten Variablen muss an dieser Stelle eine einheitliche Daten-Sicht (durch Konsistenz-Abgleich über den Hauptspeicher) hergestellt werden.
ordered	Nachfolgender Bereich darf nur in der durch die Schleifen-Iterationen festgelegte Reihenfolge von den einzelnen Unterprozessen ausgeführt bzw. betreten werden.
threadprivate	Für die spezifizierten global definierten Variablen werden lokale Kopien angelegt. Die Verwendung erfolgt dann analog zu Variablen mit dem *private*-Datensichtbarkeitsbereichs-Attribut.

Tabelle 2: Direktiven für den OpenMP-Standard in Version 2.5. Mehr Details zur genauen Verwendung und Einschränkungen finden sich im Standard [79].

Klauseln	Kurzbeschreibung
default	Definiert die Voreinstellungen bezüglich der Datensichtbarkeitsbereichs-Attribute.
shared	Die spezifizierten Variablen werden mit dem Daten-Attribut für einen geteilten Zugriff versehen. Alle Unterprozesse können gemeinsam auf diese Variablen zugreifen.
private	Die spezifizierten Variablen werden mit dem Daten-Attribut für einen geschützten Zugriff versehen. Jeder Unterprozess arbeitet auf seiner eigenen lokalen Variablen-Kopie.
firstprivate	Die spezifizierten Variablen werden mit dem Daten-Attribut *private* versehen, wobei jede Kopie mit dem Wert von vor Eintritt in den Direktiven-Bereich initialisiert wird.
lastprivate	Die spezifizierten Variablen werden mit dem Daten-Attribut *private* versehen, wobei der Wert der letzten Kopie beim Austritt aus dem Direktiven-Bereich übernommen wird.
reduction	Die spezifizierten Variablen werden mit dem Daten-Attribut *private* innerhalb der nachfolgenden Schleife versehen. Beim Austritt aus der Schleife wird die spezifizierte Rechenoperation global auf alle lokalen Kopien angewendet und so ein gemeinsames globales Ergebnis für alle Unterprozesse berechnet.
copyin	Der Wert der spezifizierten *threadprivate*-Variablen des Haupt-Unterprozesses wird zu Beginn des Direktiven-Bereichs für alle anderen Unterprozesse der Gruppe übernommen.
copyprivate	Der Wert der spezifizierten lokalen Variablen eines Unterprozesses wird für alle anderen Unterprozesse der Gruppe übernommen.

Tabelle 3: Klauseln für den OpenMP-Standard in Version 2.5. Mehr Details zur genauen Verwendung und Einschränkungen finden sich im Standard [79]. Die Klauseln *schedule*, *nowait* und *num_threads* werden aufgrund vernachlässigbarer Nebeneffekte nicht explizit in dieser Arbeit besprochen.

B Programmbeispiele Parallelisierung

B.1 Originale Parallelisierung

```
module g_caller_shared
  ! N : vector size
  integer, parameter :: N=100
  ! P : number of gradients
  integer, parameter :: P=10
  ! x,y : global data vectors
  real,allocatable :: x(:), g_x(:)
  real,allocatable :: y(:), g_y(:)
end module g_caller_shared
```

```
! AD caller routine,SG-Mode(innen private)
use g_caller_shared
real α, g_α
do j = 1, P
  allocate(x(N), y(N))
  allocate(g_x(N), g_y(N))
  ... ! seeding [j] in g_x(1:N),g_α
  ...
  call g_omp_vdiv(N,α,g_α,x,g_x,y,g_y)
  ...
  ... ! evaluate [j] from g_y(1:N)
  deallocate(g_x, g_y)
  deallocate(x, y)
end do
```

```
mem: M_o(N)+M_o(N) = 2·M_o(N)
cpu:               P·(1+c)·T_o(N,t_i);  mit t_i|N
```

Abbildung 64: Fortran-Modul und Gradientenschleife (originale Parallelisierung) mit innerer lokaler Speicherreservierung.
Kennzeichnung für „lokale“ Speichernutzung; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Performance-Modelle

```
module g_caller_shared
  ! N : vector size
  integer, parameter :: N=100
  ! P : number of gradients
  integer, parameter :: P=10
  ! x,y : global data vectors
  real x(N,P), g_x(N,P)
  real y(N,P), g_y(N,P)
end module g_caller_shared
```

```
! AD caller routine,SG-Mode(höherdimensional)
use g_caller_shared
real α(P), g_α(P)
do j = 1, P
  ... ! seeding [j] in g_x(1:N,j),g_α(j)
  ...
  call g_omp_vdiv(N,α(j),g_α(j), &
    x(1:N,j),g_x(1:N,j),y(1:N,j),g_y(1:N,j))
  ...
  ... ! evaluate [j] from g_y(1:N,j)
end do
```

```
mem: 2·P·M_o(N)
cpu: P·(1+c)·T_o(N,t_i);  mit t_i|N
```

Abbildung 65: Fortran-Modul und Gradientenschleife (originale Parallelisierung) mit äußerer Speicherreservierung und höher-dimensionalen Datenobjekten.
Kennzeichnung für höher-dimensionale Datenobjekte; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Performance-Modelle

B.2 Gradientenparallelisierung

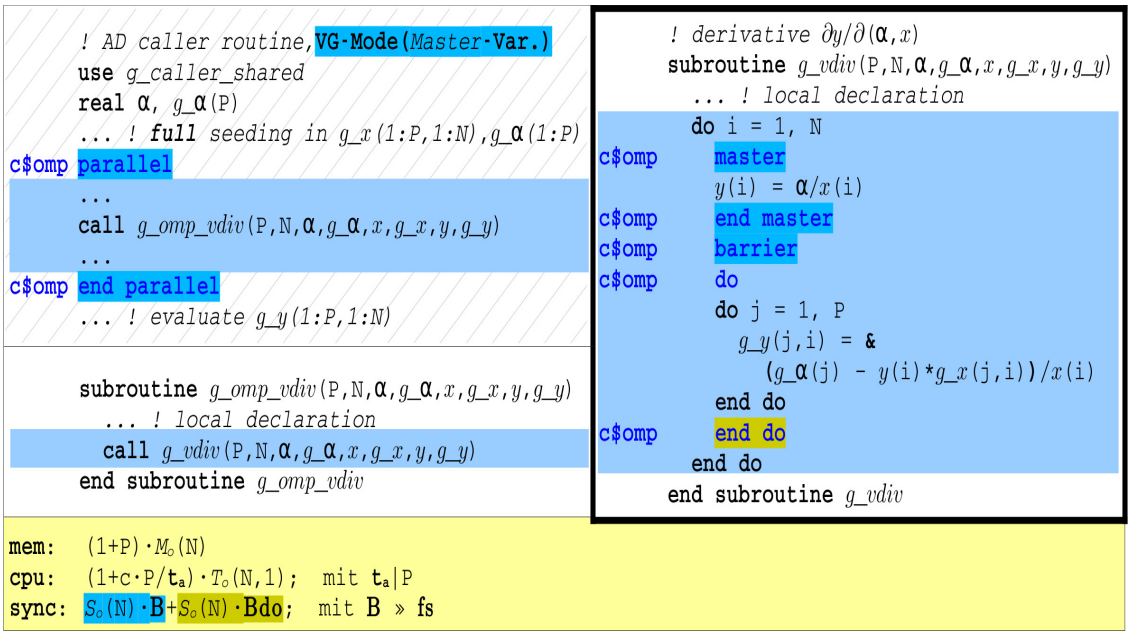


Abbildung 66: Gradientenaufruf, Zwischenroutine und Ableitungsfunktion *g_vdiv* als *Master*-Variante und mit Gradientenparallelisierung.
Äußere parallele Region; Kennzeichnung für *Master*-Parallelisierung; Kennzeichnung der impliziten Synchronisation; Hauptabschnitt Funktionsaufruf; Berechnungsroutine; Performance-Modelle

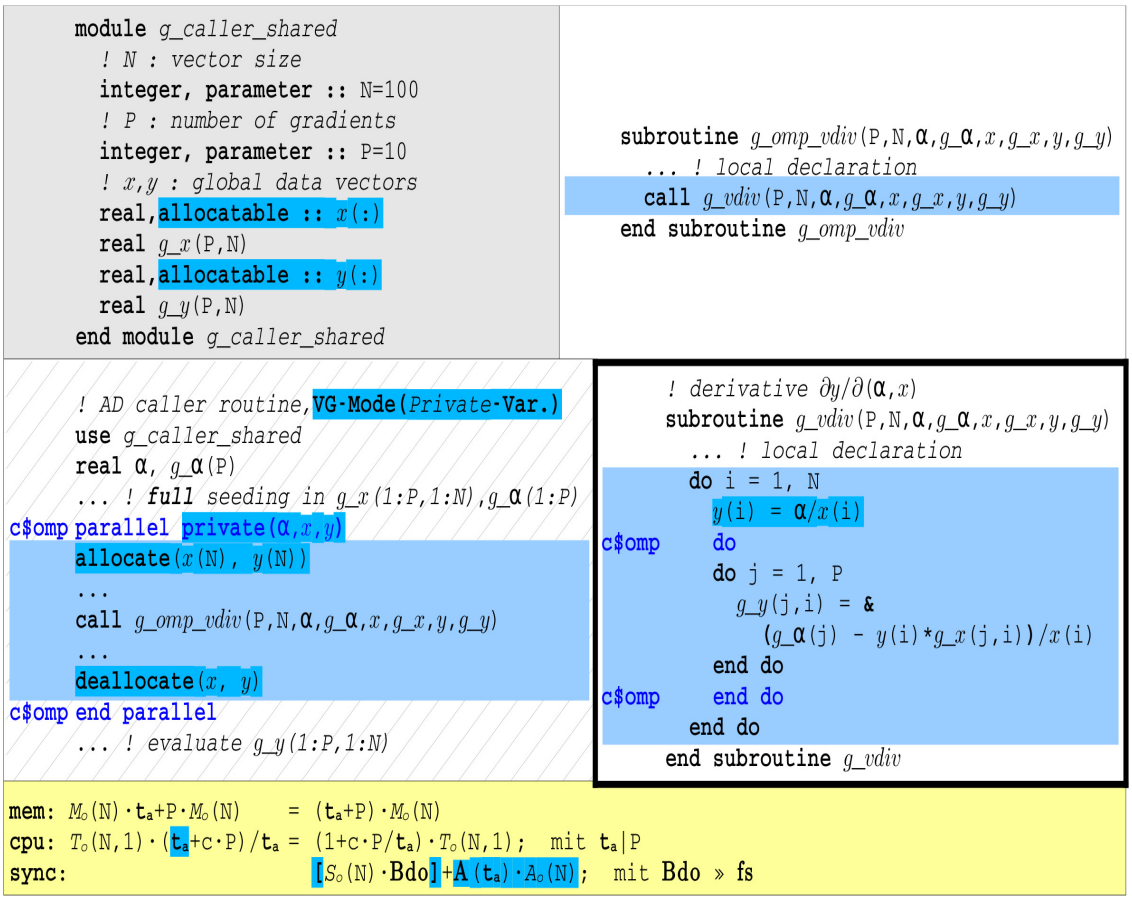


Abbildung 67: Komplettes Programmbeispiel für die *Private*-Variante und mit Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung der *Private*-Variante; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

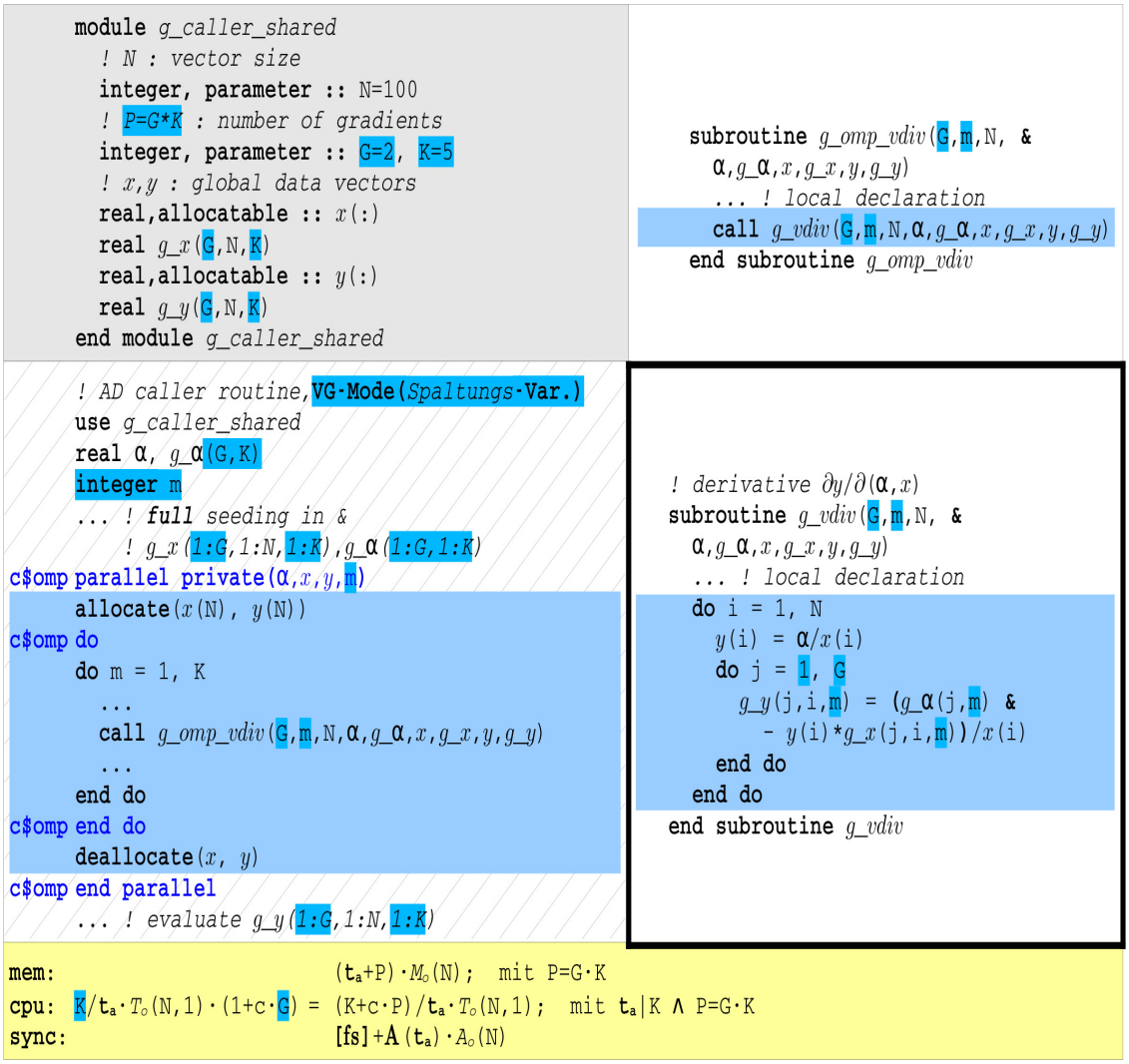


Abbildung 68: Komplettes Programmbeispiel für die *Spaltungs*-Variante und mit Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung der *Spaltungs*-Variante; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

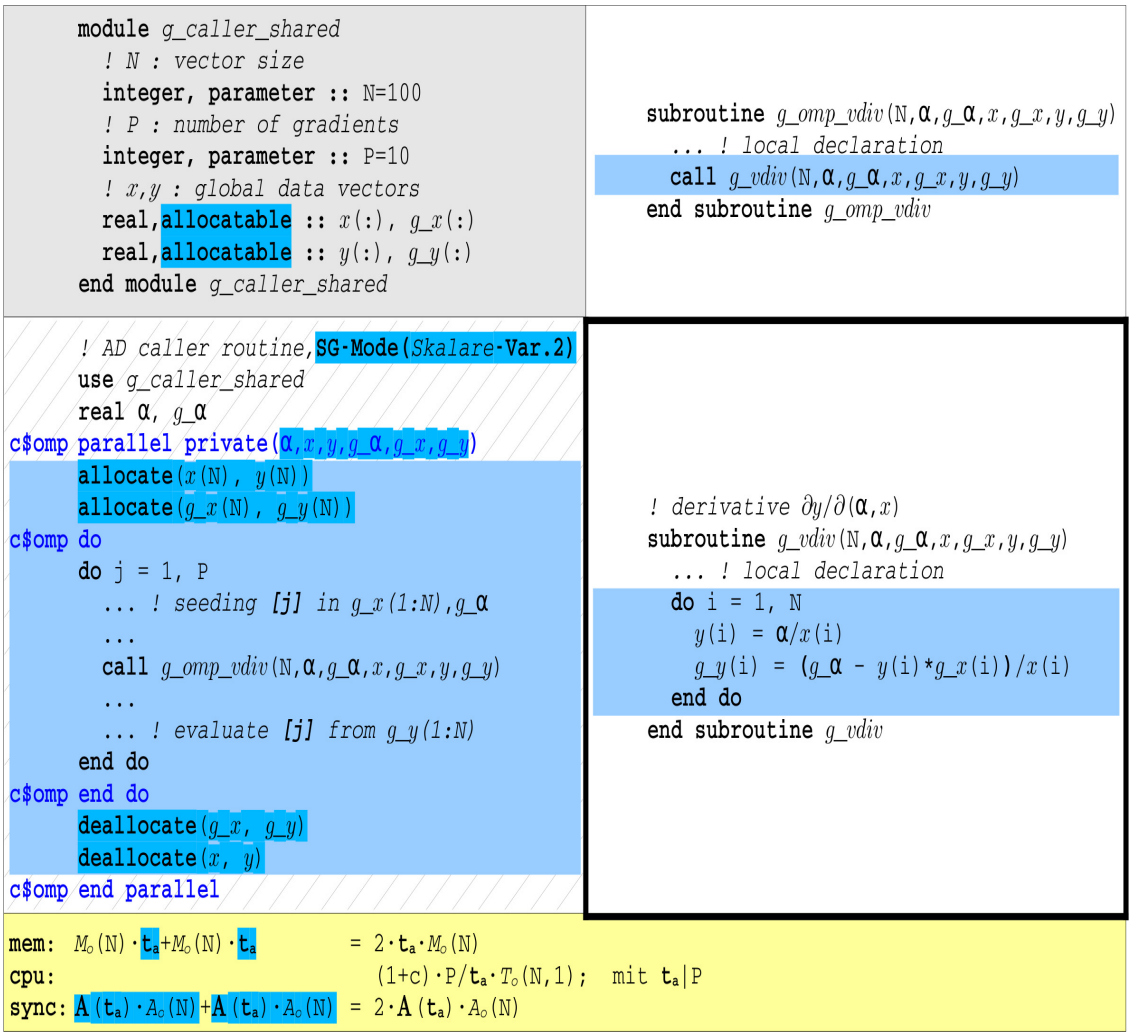


Abbildung 69: Komplettes Programmbeispiel für die *Skalare*-Variante 2 mit lokaler Speicherreservierung und Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung der *Skalare*-Variante 2; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

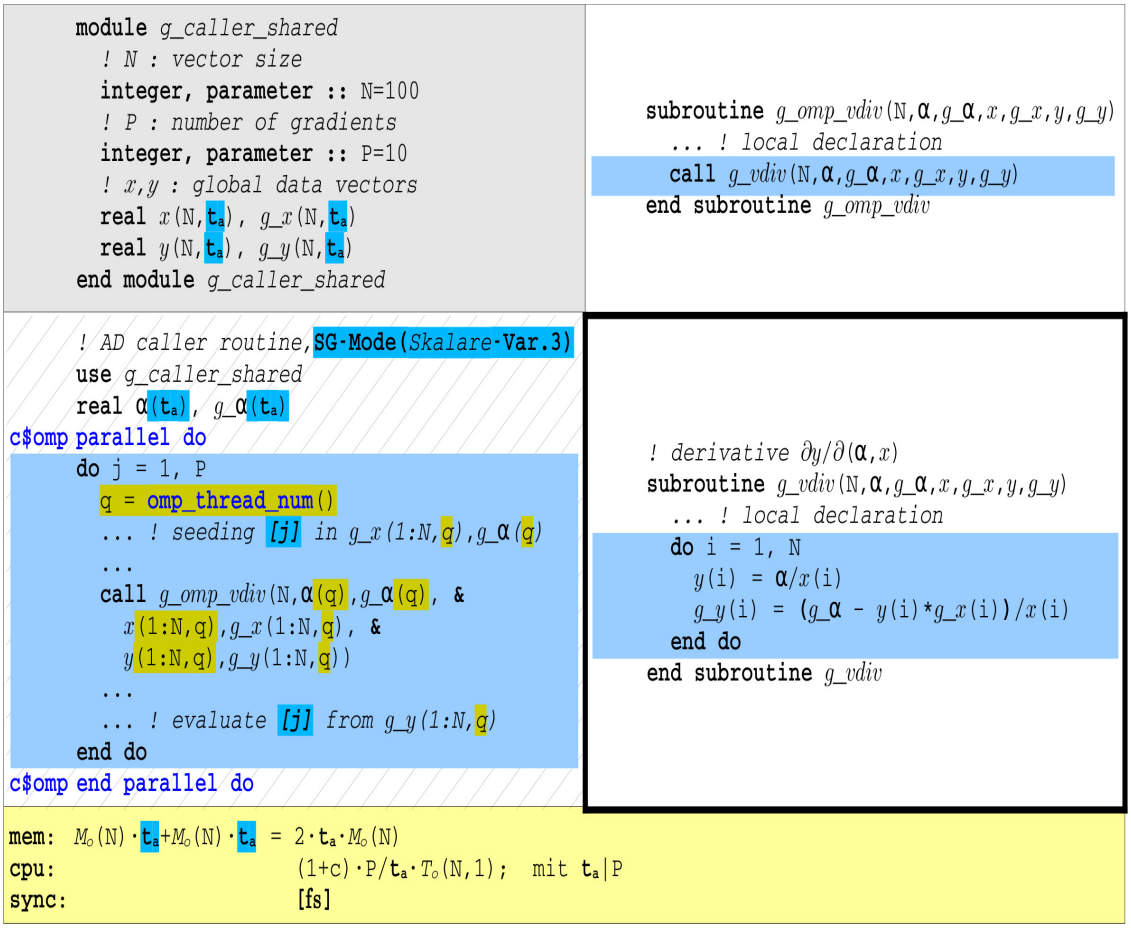


Abbildung 70: Komplettes Programmbeispiel für die *Skalare*-Variante 3 mit höher-dimensionalen Datenobjekten, effiziente Variante mit ein um Faktor $\mathbf{t}_a$ höheren Speicherverbrauch für jedes Datenobjekt und Gradientenparallelisierung.
Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Wichtige Kennzeichnung für höher-dimensionale Datenobjekte; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

B.3 Geschachtelte Parallelisierung für Gradienten

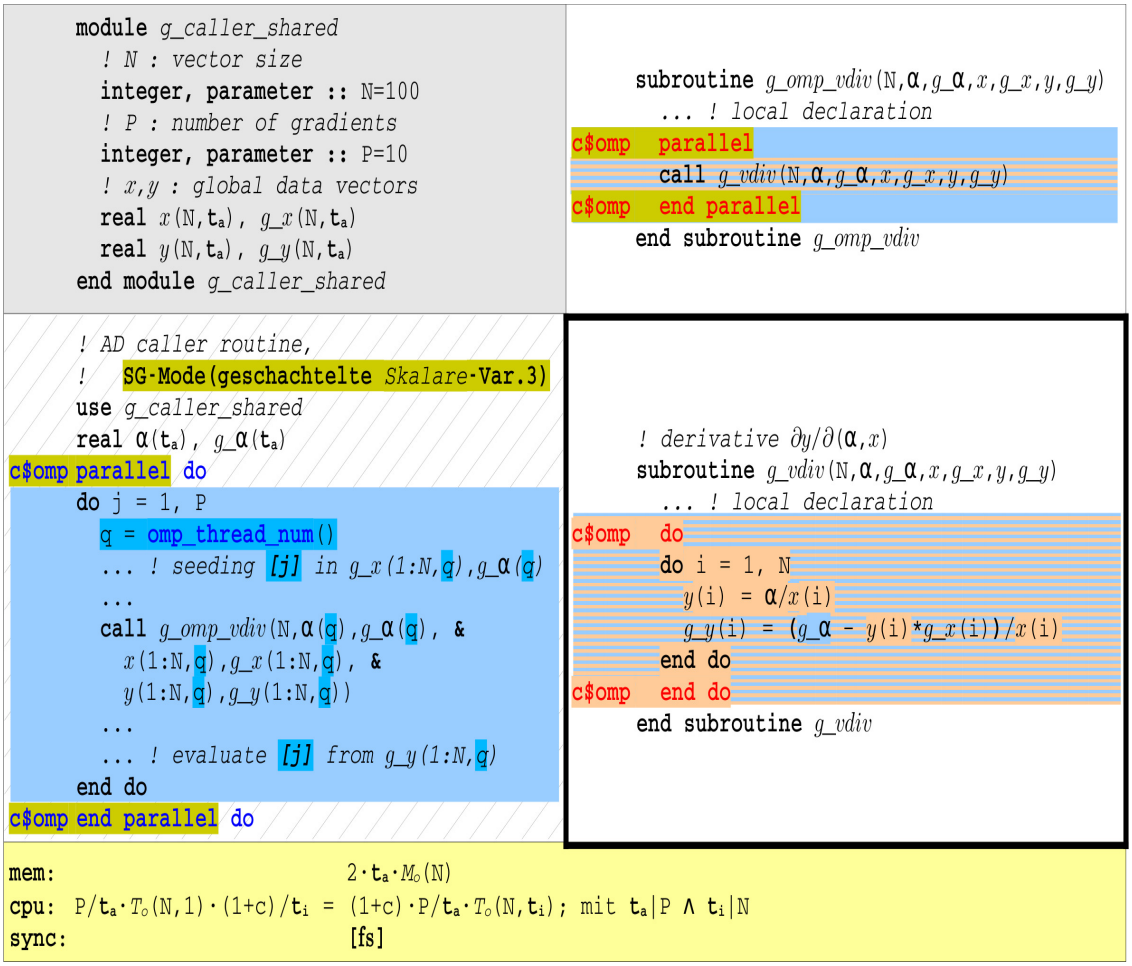


Abbildung 71: Komplettes Programmbeispiel für die geschachtelte Parallelisierung, aufbauend auf *Skalare*-Variante 3 mit höher-dimensionalen Datenobjekten.
Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Wichtige Kennzeichnung für innere parallele Region; Geschachtelte parallele Region; Wichtige Kennzeichnung für geschachtelte Region; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

B.4 Versucheparallelisierung

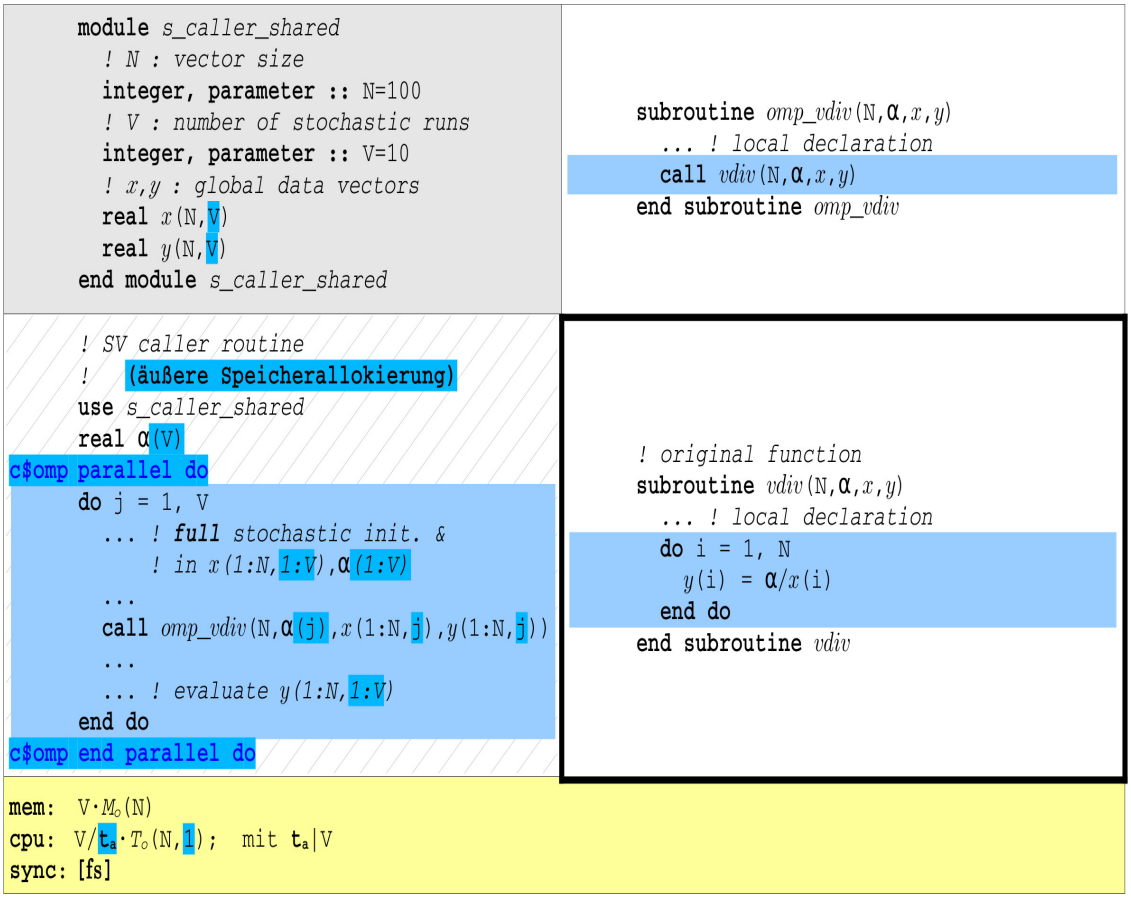


Abbildung 72: Komplettes Programmbeispiel für die Versucheparallelisierung mit äußere Speicherreservierung (höher-dimensionale Datenobjekte).
Äußere parallele Region; Wichtige Kennzeichnung für äußere Region; Fortran-Modul Definition; Hauptabschnitt Funktionsaufruf; Zwischenroutine; Berechnungsroutine; Performance-Modelle

C Modellbeispiele, synthetische Vergleichstests

C.1 Parameterschätzung: bench_33MB

Autor: Dipl.-Inform. Andreas Wolf

Modellbeschreibung: Ein 3-Schichten Modell im stationären Zustand, unter Berücksichtigung von dem hydraulischen Potential und der Temperatur. Abbruch der nichtlinearen Konvergenz nach der ersten Iteration.

Charakteristika: Diskretisierung $30 \times 30 \times 30$ Gitterpunkte; kein Injektor; kein Producer; keine Messdaten; Re-Injektionscode=„none“; Kopplung=„bas“

Parameterschätzung: Inversion von der Porosität, der Permeabilität, der Wärmeleitfähigkeit und der Wärmeproduktion für alle 3 Schichten (12 aktive Parameter); Optimiert für Vergleichstests

Berechnung: verschiedene Unterprozess-Konfigurationen (äußere Unterprozesse × innere Unterprozesse); 1 Inversionsiteration; Zeitmessung nur über Ableitungsberechnung

Rechnersystem: SMP-System mit 8×2Kern-Prozessoren („dual-core“, Opteron 885)

C.2 Stochastische Versuche: bench_1288MB

Autor: Dipl.-Inform. Andreas Wolf

Modellbeschreibung: Ein 160-Schichten Modell im stationären Zustand, unter Berücksichtigung von dem hydraulischen Potential und der Temperatur. Abbruch der nichtlinearen Konvergenz nach der ersten Iteration.

Charakteristika: Diskretisierung $330 \times 75 \times 60$ Gitterpunkte; kein Injektor; kein Producer; keine Messdaten; Re-Injektionscode=„none“; Kopplung=„bas“

Simulation: 320 nicht-stochastisch gestörte identische Versuche; Optimiert für Vergleichstests

Berechnung: verschiedene Unterprozess-Konfigurationen (äußere Unterprozesse × innere Unterprozesse); Zeitmessung nur über Versucheberechnung

Rechnersystem: ScaleMP (13 Rechenknoten mit je 2×4Kern-Prozessoren)

C.3 Parameterschätzung: bench_XXL_small

Autor: Dipl.-Inform. Andreas Wolf

Modellbeschreibung: Ein 100-Schichten Modell im stationären Zustand, unter Berücksichtigung von dem hydraulischen Potential und der Temperatur. Abbruch der nichtlinearen Konvergenz nach der ersten Iteration.

Charakteristika: Diskretisierung $330 \times 75 \times 60$ Gitterpunkte; kein Injektor; kein Producer; keine Messdaten; Re-Injektionscode=„none“; Kopplung=„bas“

Parameterschätzung: Inversion von der anisotropen Permeabilität in X,Y und Z Richtung für 80 (von 100) Schichten (240 aktive Parameter); Optimiert für Vergleichstests

Berechnung: verschiedene Unterprozess-Konfigurationen (äußere Unterprozesse × innere Unterprozesse); 1 Inversionsiteration; Zeitmessung nur über Ableitungsberechnung

Rechnersystem: ScaleMP (13 Rechenknoten mit je 2×4Kern-Prozessoren)

C.4 Parameterschätzung: bench_1288MB

Autor: Dipl.-Inform. Andreas Wolf

Modellbeschreibung: Ein 160-Schichten Modell im stationären Zustand, unter Berücksichtigung von dem hydraulischen Potential und der Temperatur. Abbruch der nichtlinearen Konvergenz nach der ersten Iteration.

Charakteristika: Diskretisierung $330 \times 75 \times 60$ Gitterpunkte; kein Injektor; kein Producer; keine Messdaten; Re-Injektionscode=„none“; Kopplung=„bas“

Parameterschätzung: Inversion von der anisotropen Permeabilität in X und Y Richtung für alle 160 Schichten (320 aktive Parameter); Optimiert für Vergleichstests

Berechnung: verschiedene Unterprozess-Konfigurationen (äußere Unterprozesse $\times$ innere Unterprozesse); 1 Inversionsiteration; Zeitmessung nur über Ableitungsberechnung

Rechnersystem: verschiedene ScaleMP- und SMP-Systeme (siehe Kapitel 6.2.1)

D Modellbeispiele, reale Anwendung

D.1 Parameterschätzung: DIPL_S3D_NC_2_3B_3

Quelle: E.ON ERC, Applied Geophysics and Geothermal Energy

Autoren: Dipl.-Geol. Christian Kosack, Dr. Volker Rath

Modellbeschreibung: Ein zeitabhängiger Pumpversuch für Hot Dry Rock (HDR) Versuchseinrichtung in Soultz [117, 118, 119], unter Berücksichtigung von dem hydraulischen Potential, der Temperatur und der Tracer-Konzentration.

Charakteristika: Diskretisierung $29 \times 60 \times 29$ Gitterpunkte; 1×Injektor; 2×Producer; 2×Bohrungen (284 Messdaten); 2000 Zeitschritte;
Re-Injektionscode=„wells3dN_CK3B“; Kopplung=„const“

Parameterschätzung: Inversion von der Porosität, der Permeabilität und der Diffusivität für je 2 Schichten (6 aktive Parameter); Optimiert für Tracerverlauf

Berechnung: im laufenden Betrieb auf 6 Rechenknoten mit (6×8)-Konfiguration (äußere Unterprozesse × innere Unterprozesse); 11 Inversionsiterationen; Laufzeit = 5659 *min*

Rechnersystem: ScaleMP (13 Rechenknoten mit je 2×4Kern-Prozessoren)

D.2 Parameterschätzung: DIPL_S3D_NC_4Q_10

Quelle: E.ON ERC, Applied Geophysics and Geothermal Energy

Autoren: Dipl.-Geol. Christian Kosack, Dr. Volker Rath

Modellbeschreibung: Ein zeitabhängiger Pumpversuch für Hot Dry Rock (HDR) Versuchseinrichtung in Soultz [117, 118, 119], unter Berücksichtigung von dem hydraulischen Potential, der Temperatur und der Tracer-Konzentration.

Charakteristika: Diskretisierung $29 \times 36 \times 29$ Gitterpunkte; 1×Injektor; 1×Producer; 1×Bohrungen (142 Messdaten); 4000 Zeitschritte;
Re-Injektionscode=„wells3dN_CK“; Kopplung=„const“

Parameterschätzung: Inversion von der Porosität, der Permeabilität und der Diffusivität für je 4 Schichten (12 aktive Parameter); Optimiert für Tracerverlauf

Berechnung: im laufenden Betrieb auf 6 Rechenknoten mit (12×3)-Konfiguration (äußere Unterprozesse × innere Unterprozesse); 8 Inversionsiterationen; Laufzeit= 5664 *min*

Rechnersystem: ScaleMP (13 Rechenknoten mit je 2×4Kern-Prozessoren)

D.3 Stochastische Versuche: S3D_STOCH_COARSE_0009

Quelle: E.ON ERC, Applied Geophysics and Geothermal Energy

Autoren: Dipl.-Phys. Chrsitian Vogt, Dr. Gabriele Marquart, Dipl.-Geol. Christian Kosack

Modellbeschreibung: Ein zeitabhängiger Pumpversuch für Hot Dry Rock (HDR) Versuchseinrichtung in Soultz [117, 118, 119], unter Berücksichtigung von dem hydraulischen Potential, der Temperatur und der Tracer-Konzentration.

Charakteristika: Diskretisierung $21 \times 36 \times 21$ Gitterpunkte; 1×Injektor; 1×Producer; 3000 Zeitschritte; Re-Injektionscode=„wells3d_stoch“, Kopplung=„bas“

Simulation: 10000 mit SGSIM [120] stochastisch gestörte Versuche

Berechnung: exklusiv auf 11 Rechenknoten mit (88×1)-Konfiguration (äußere Unterprozesse × innere Unterprozesse); Laufzeit= 12251 *min* ($204h$)

Rechnersystem: ScaleMP (13 Rechenknoten mit je 2×4Kern-Prozessoren)

E Aufzählung der bisher im Suite-Projekt insgesamt implementierten Erweiterungen und Module

Modul	**Anzahl**	**Aufzählung**
physikalische Prozesse	5	hydraulisches Potential, Wärmetransport, Stofftransport, elektrisches Potential, Druck[1]
physikalische Kopplungen	11	IAPWS, bas, basc, bascl, const, conv, frac, freezing, ice, kola, soultz
nicht-lineare Systemlöser	1(+2)	Picard-Fixpunktiteration [12], (*zusätzlich nur für Testzwecke:* zwei Alternativen basierend auf Newton-Methode [29], davon eine mit NITSOL [30])
lineare Gleichungs-systemlöser	4	CG [10], BiCGStab [11]; *extern:* LAPACK [27], NAG [126]
Vorkonditionierer (Varianten)	4	*keine*, Diagonal, SSOR-Eisenstadt, ILU-0
Optimierer (deterministisch)	5	„bayessche“-Ansatz [39, 40, 41], dessen Daten- und Parameterraum Formulierungen [43], LBFGS [45, 46], NLCG [44]
stochastische Initialstörungen	2	SGSIM[120], VISIM
Optimierer (stochastisch)	1	ENKF [52, 53, 3, 54]

Tabelle 4: Flexible Modul-Schnittstellen (Klassifizierung), mit entsprechender Aufzählung insgesamt bisher implementierter Varianten und Erweiterungen; [1]Es kann nicht gleichzeitig basierend auf dem hydraulischen Potential und dem Druck gerechnet werden.

Lebenslauf

Persönliche Daten

Name:	Wolf
Vorname:	Andreas
Geburtstag:	03.10.1975
Geburtsort:	Berlin-Lichtenberg
Staatsangehörigkeit:	deutsch

Qualifikationen

1995:	Abitur, Weinberg-Gymnasium, Kleinmachnow
ab 1996 bis 2002	Studium der Informatik (Nebenfach Elektrotechnik) an der RWTH Aachen University, Abschluss: Diplom-Informatiker
ab 2002 bis 2011	Promotionsstudium im Fachbereich Informatik an der RWTH Aachen University